AD-A230 293

DTIC FILE COPY

Technical Document 1837
June 1990

Advanced Numerical Techniques of Performance Evaluation

Volume II

University of Washington

DTIC
ELECTE
DEC 28 1990
D

Approved for public release; distribution is unlimited.

The views and conclusions contained in this report are those of the contractors and should not be interpreted as representing the official policies, either expressed or implied, of the Naval Ocean Systems Center or the U.S. Government.

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

Contract N66001-87-D-0136 was carried out by University of Washington, Department of Computer Sciences, Seattle, WA 98195, under the technical coordination of T. Sterrett, Computer Systems Software and Technology Branch, Code 411, Naval Ocean Systems Center, San Diego, CA 92152-5000.

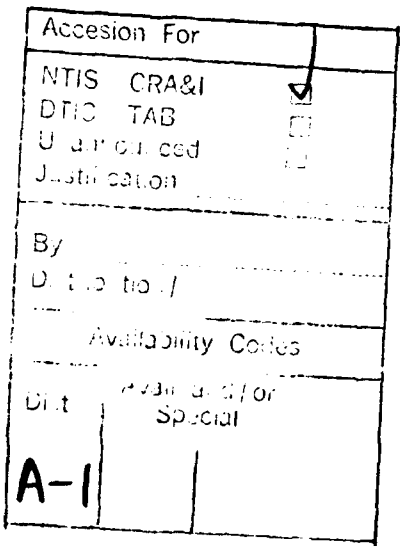
Released by
R. A. Wasilausky, Head
Computer Systems Software
and Technology Branch

Under authority of
A. G. Justice, Head
Information Processing and
Displaying Division

FS

Parallel Discrete-Event Simulation





Techniques for Efficient Shared-Memory Parallel Simulation

David B. Wagner, Edward D. Lazowska, and Brian N. Bershad
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

The conservative (Bryant-Chandy-Misra) parallel simulation paradigm was originally developed in a distributed system setting. However, the recent widespread availability of relatively inexpensive, medium-scale shared-memory multiprocessors encourages a re-examination of traditional approaches to its implementation. Many of the obstacles to good performance, such as communication delay, sub-optimal model partitioning, artificial blocking, and the high cost of deadlock avoidance, deadlock detection, and deadlock recovery, can be reduced or even eliminated in a shared-memory implementation.

This paper presents a number of techniques for efficient shared-memory parallel simulation. All of these techniques have been implemented in *Synapse*, an integrated programming environment/run-time system that supports efficient parallel discrete-event simulation on shared-memory multiprocessors. A new technique for artificial blocking and deadlock avoidance, *lazy blocking avoidance*, is built directly into Synapse's multi-threaded run-time kernel. The Synapse run-time system also contains mechanisms for *efficiently detecting* and breaking the few deadlocks that are not prevented by lazy blocking avoidance.

Our performance results are substantially better than those reported in previous empirical studies in this area, enough to challenge their negative conclusions about the feasibility of the conservative approach to parallel simulation.

1 Introduction

Discrete event simulation is computationally expensive. A promising approach to speeding up large simulations is to execute different parts of them in parallel.

A widely used model for describing parallelism in discrete event simulations was proposed independently by Bryant [6] and by Chandy and Misra [8]. The physical system to be simulated is partitioned into a set of component entities called *physical processes* (PPs) which interact only at discrete times (e.g., through the scheduling of events). This system of PPs can then be simulated as a collection of *logical processes* (LPs) that communicate via the sending and receiving of timestamped messages. The scheduling of an event for PP_x at time t in the physical system is simulated by sending a message with timestamp t to LP_x.

Each LP has its own local clock, which indicates how long the LP has executed in *simulation time*. In effect, the global event list and global clock of a sequential simulation have been done away with; their counterparts in a parallel simulation are the set of LP input message queues and the set of local clocks, respectively. In order to correctly simulate a PP, the corresponding LP must process messages in timestamp order, as opposed to their real-time arrival order. The major challenge to implementing such a parallel simulation is the synchronization of LPs to ensure that event causality is maintained, without resorting to lock-step (i.e., time-driven) execution.

Two general approaches to synchronization in parallel discrete event simulation have been proposed in the literature: optimistic and conservative. In the optimistic approach [12], an LP can process every message

Our work is supported by the National Science Foundation (Grants No. CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, US WEST Advanced Technologies, the Washington Technology Center, Digital Equipment Corporation (the Systems Research Center and the External Research Program), Hewlett-Packard, and the USENIX Association.

as soon as it arrives; however, if a message with an earlier timestamp subsequently arrives, the LP must roll back its state to the time of the earlier message and re-execute from that point. This may require the cancellation of messages that the LP has sent, which can cause rollback at other LPs. Optimistic parallel simulation requires substantial low-level support in order to be efficient [11].

In the conservative approach [6, 8, 14], an LP does not accept a message for processing until it is certain that *no* message with an earlier timestamp can ever arrive. Thus, an LP can block even though it has pending input messages. Very often, it turns out that much of this blocking is unnecessary. Even worse, the simulation is susceptible to deadlock even if the physical system is deadlock-free. We discuss these problems in greater detail in the next section.

The early work on parallel simulation took place in a distributed system setting. While this is perhaps desirable from the standpoint of generality and scalability, the lack of shared memory in such systems complicates synchronization. We believe that algorithms for parallel simulation ought to be re-evaluated in light of the recent availability of shared memory multiprocessors such as the Sequent [2, 23] and Firefly [22]. Although the use of shared memory can make traditional synchronization algorithms run much more efficiently, completely different approaches can and should be considered in this environment.

Our concern in this paper is the efficient support of conservative parallel simulation in a shared-memory multiprocessor environment. Section 2 describes the difficulties associated with distributed implementations of conservative parallel simulation, and how these difficulties can be reduced or eliminated by the availability of shared memory. Section 3 presents techniques for high performance shared-memory parallel simulation in more detail. These techniques have been implemented in the *Synapse* system; Section 4 examines the performance of Synapse on a number of benchmarks. Finally, our conclusions and directions for future research are discussed in Section 5.

2 Conservative Parallel Simulation

2.1 Problems with the Conservative Approach

In the conservative approach to parallel simulation [6, 8, 14], an LP does not accept a message for processing until it is certain that *no* message with an earlier timestamp can ever arrive. Since LPs may have to wait for other LPs to produce messages before they can proceed, deadlock can occur even if the system being modeled is deadlock-free.

As an example of how deadlock can occur in a conservative parallel simulation, consider Figure 1. This simulation models a simple computer system consisting of a cpu and a disk with a feedback loop. The figure depicts the state of the simulation after LP_{source} has produced all of its messages and halted. LP_{cpu} cannot accept the first message from LP_{source} because it does not know if LP_{disk} is going to send it a message with a smaller timestamp. Since LP_{cpu} is waiting for LP_{disk} and vice-versa, the simulation is deadlocked. The deadlock in this example is the result of a cycle in the communication graph of the simulation. Deadlock is likely to recur periodically, its exact frequency depending upon the output branching probabilities at LP_{disk} and the relative service times of LP_{cpu} and LP_{disk}.

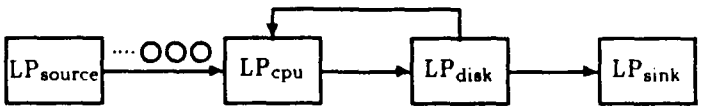


Figure 1: A deadlocked simulation.

One solution to the deadlock problem is to allow the simulation to deadlock, detect it, and then recover. The simulation thus consists of a sequence of phases performing useful computation (hopefully) in parallel, separated by *phase interfaces*, wherein a computation takes place to to break the deadlock and allow various LPs to proceed [8]. Two drawbacks to this approach are immediately apparent. First, the simulation is

making no progress during the phase interfaces. Second, this approach contributes nothing to the efficiency of the parallel computation phases of the simulation.

At least two phenomena are responsible for performance degradation of the parallel computation phases. The first is that the simulation may encounter ***partial deadlocks***, i.e., deadlocks involving only a subset of the LPs. When this happens there may be long periods of time during which the number of runnable LPs is much smaller than the number of processors, but is still non-zero. In the extreme case of a system with very little interaction between some LPs, the simulation may have only one runnable LP for long periods of time. (This was illustrated in Figure 1, in which a partial deadlock involving LP_{cpu} and LP_{disk} endured for as long as LP_{source} was producing messages.) Second, even in the absence of deadlock (partial or total), performance degradation may occur due to ***artificial blocking***. Artificial blocking (we use the terminology of Reynolds [18]) arises when a blocked LP has fallen far enough behind in simulation time that its choice of next message to accept cannot be affected by subsequent message arrivals. In contrast to a partial deadlock, it may be the case that all of the blocked LP's message sources are progressing in simulation time, but the LP has no way of knowing this. In this case there is no reason for the LP to remain blocked.

An alternative to deadlock detection and recovery that also addresses partial deadlock and artificial blocking is *deadlock avoidance*. A typical deadlock avoidance scheme has each LP periodically sending *null messages* to its "downstream" LPs to inform them that the sender will not be sending any real messages before a certain time. Unfortunately, this approach can be prohibitively expensive, especially when the degree of branching in the communication graph is high.

2.2 Distributed vs. Shared-Memory Implementation

By a *distributed* parallel simulation, we mean a parallel simulation implemented on a collection of computers (hosts) communicating on a relatively low bandwidth, relatively high latency network. Each host is assigned the responsibility of executing one or more LPs. Communication between LPs that do not reside on the same host results in a physical message being sent on the network. Since inter-host communication in a distributed system is so much more expensive than intra-host communication, it is desirable to assign LPs to hosts in a way that minimizes the number of messages sent between LPs on different hosts. A second constraint on the placement of LPs is the desirability of keeping the processing load balanced among all the hosts in the network. The resolution of these two often-conflicting constraints is known as the *partitioning problem*.

One obvious benefit of shared memory is that it makes message passing very fast. In fact, except for the overhead of locking and unlocking shared data structures, sending and receiving a message in a shared-memory parallel simulator need be no more expensive than queueing and de-queueing an event in a sequential simulator (assuming that the sending and receiving processes are both active). Thus, the partitioning problem reduces to a load-balancing problem. The use of a global scheduling policy can eliminate this problem entirely, by allowing any LP can run on any processor; this maximizes processor utilization, and thus efficiency.

A shared memory implementation also reduces the cost of deadlock detection and recovery. In a distributed system, deadlock detection protocols typically require $O(e)$ messages, where e is the number of of edges in the communication graph [7]. The deadlock recovery algorithm proposed in [8] requires $n \cdot e$ messages, where n is the number of LPs and e is the number of edges in the communication graph. In a shared-memory environment, much cheaper, centralized algorithms can be employed.

Reed et al. [16, 17] implemented a conservative parallel simulator on a shared-memory multiprocessor in order to investigate the effect of reduced communication costs on performance. The implementation was essentially the same as it would have been in a distributed environment, except that a central controller process was used to detect deadlock. The results were disappointing.

The experience of Reed et al. points out that communication costs are not the only obstacle to acceptable performance of conservative parallel simulation. The availability of shared memory provides an opportunity to re-examine traditional approaches, with an eye towards increasing the degree of parallelism during the periods in which the simulation is not deadlocked. The ability of any component in the system to use the state information of any other component, which is clearly infeasible in a distributed implementation, clearly

ought to be exploited. Also, the run-time system can be optimized for performing parallel simulation, thus reducing the overhead of synchronization. For example, explicit null messages are not needed to perform deadlock avoidance; instead, the sending of a null message can be implied by the modification of certain shared variables [9, 10]. However, there are other costs to consider when implementing a deadlock avoidance scheme: the synchronization necessary to schedule blocked LPs, and the extra processor utilization resulting from unnecessary context switches. A new technique for deadlock avoidance, *lazy blocking avoidance*, minimizes these costs by being implemented as part of the run-time kernel.

A detailed discussion of many of these techniques is the topic of the next section.

3 Techniques for Exploiting Shared Memory

3.1 Logical Process Synchronization

The basic algorithm run by each LP in a distributed parallel simulation (i.e., one without any shared memory) is shown in Figure 2(a) [14]. For each pair i, j such that there is a communication link from LP_i to LP_j, the variable t_{ij} contains the timestamp of the last message sent on that link. (Initially, $t_{ij} = 0$ for all such i, j.) Because of the requirement that messages sent on each link must be in non-decreasing timestamp order, LP_j knows that LP_i cannot send it a message with timestamp less than t_{ij}. The variable H_j is maintained as the minimum of the t_{ij}, and thus LP_j knows that it cannot subsequently receive a message with timestamp less than H_j from *any* of its message sources. Hence, LP_j can safely consume all received messages that have timestamps no larger than H_j, allowing it to simulate PP_j up to at least time H_j.

In [14], H_j is called the *clock value* of LP_j. However, this does not agree with the intuitive notion that an LP's clock value should be an accurate measure of its progress in the simulation: it is not always true that LP_i has simulated PP_i up to time H_i, only that it is *allowed* to do so. For this reason, we introduce another variable, C_j, which is defined as the minimum timestamp of any message that LP_j can generate (whereas H_j is an upper bound on the timestamps of messages than LP_j is allowed to consume). When a message m is consumed, C_j is set to the maximum of C_j and $m.time$. This is because if $C_j < m.time$, it must be the case that the state of PP_j does not change in the interval between C_j and $m.time$. Thus C_j is always at least as great as the timestamp of the message that was most recently consumed by LP_j[1]. In our nomenclature, C_j is called the clock value, since it more accurately reflects the progress of LP_j than does H_j, which is called the *message acceptance horizon*. This distinction will become important in a moment.

In the basic distributed simulation algorithm, LP_j uses t_{ij} as an estimate of how far LP_i has advanced in the simulation. An obvious drawback to this is that LP_i may have advanced arbitrarily far beyond t_{ij}, but LP_j has no way of knowing this in the absence of message traffic from LP_i. This can give rise to LP_j being artificially blocked. Given the availability of shared memory, there is no reason to use t_{ij} as an estimate of LP_i's progress, when the exact value (C_i) is available for inspection. This is the strategy of the algorithm shown in Figure 2(b). There are two important differences between the original and the modified algorithms; these are identified by a number in the right margin of Figure 2(b):

(1) The reason for taking the maximum of C_i and t_{ij} in the calculation of H_j is that LP_i may be able to predict that the next message it will send to LP_j will be at time $t_{ij} > C_i$, even though it cannot guarantee that it will not send an earlier message to some other LP (and hence is unable to advance C_i). Thus, the t_{ij} values may not be redundant.

(2) In addition to waiting for messages to arrive whenever H_j cannot be advanced, LP_j must also be alert for changes to any of the C_i values of its message sources.

The last modification is the most difficult one to implement efficiently. If there are more processors than LPs, busy waiting is the obvious solution. Otherwise, the waiting LP must block and rely on some other entity to awaken it at an appropriate time. If a blocked LP is awakened too soon, computational resources

[1] It is possible for C_j to be advanced by LP_j in the course of simulating an event, if that event represents a non-preemtible action of PP_j. A simple example of this is the service of a customer by a FIFO server.

```
H_j := 0;  C_j := 0;
while not finished do
    m := earliest available message;
    while m.time ≤ H_j do
        consume(m);
        C_j := max(C_j, m.time);
        simulate(m.event);
        m := earliest available message;
    endwhile
    H_j := min_i{t_{i,j}};
    while m.time > H_j do
        await(message arrival);
        H_j := min_i{t_{i,j}};
    endwhile
endwhile
```

(a)

```
H_j := 0;  C_j := 0;
while not finished do
    m := earliest available message;
    while m.time ≤ H_j do
        consume(m);
        C_j := max(C_j, m.time);
        simulate(m.event);
        m := earliest available message;
    endwhile
    H_j := min_i{max(C_i, t_{i,j})};                    (1)
    while m.time > H_j do
        await(message arrival or
              {ΔC_i for some i});                       (2)
        H_j := min_i{max(C_i, t_{i,j})};                (1)
    endwhile
endwhile
```

(b)

Figure 2: Algorithm for LP_j in a distributed parallel simulation (a) and in a shared-memory parallel simulation (b).

will be wasted; on the other hand, an LP that remains blocked even though its message acceptance horizon has advanced far enough to enable consumption of a message is *artificially blocked*.

3.2 Artificial Blocking Avoidance

There are actually two separate issues in the implementation of artificial blocking avoidance, namely: *who* should awaken a blocked LP, and *when*?

One answer to question of *who* is: any LP_i that is a source of messages for the waiting LP. This suggests several alternatives for *when*: whenever C_i changes; whenever LP_i sends a message to any of its possible destinations; or whenever LP_i is about to block. (In the last two cases, wakeups would only be done if C_i had changed since the last round of wakeups.) Note that this solution is the shared-memory analog of null-message based deadlock avoidance. No null messages are actually sent, since the information they convey would be redundant in the shared-memory environment. Nevertheless, there are still several drawbacks to this scheme.

First, LPs that are doing useful work are required to spend time waking up other LPs. Doing the wakeups only when an LP is about to block solves this problem, but is likely to increase the amount of time that the waiting LPs spend blocked. (Each choice of *when* makes a different tradeoff between the burden placed on the LPs that are doing the wakeups, and the amount of artificial blocking.) Second, it may be the case that the LP that is awakened is still unable to consume any messages; in this case, awakening it may be delaying the scheduling of LPs that have real work to do.

Reynolds [18] proposed the following solution to this problem. Suppose LP_j wishes to consume a message with timestamp t, but some of its source LPs have not progressed that far. For each such source LP_i, LP_j inserts a tuple of the form (j, t) in a data structure belonging to LP_i. Each time LP_i advances its clock value, it checks its data structure to see if any other LPs need to be notified. In this way, the amount of extra work done by the running LP and the number of unnecessary wakeups are reduced (but not eliminated!).

To avoid placing any burden whatsoever on the LPs that are doing useful work, the responsibility for waking blocked LPs can be delegated to the run-time kernel. A blocked LP could be awakened at regular intervals; this is logically equivalent to having the blocked LP poll its message sources. But this still doesn't solve the problem of unnecessary wakeups. A better solution would be to have the kernel use an idle processor to recompute the message acceptance horizon of a blocked LP, *without* performing a context switch. LP_j would be awakened if the new value of H_j were greater than or equal to the timestamp of its earliest pending

message. This strategy has the advantage of not causing any unnecessary wakeups. At the same time, the potential amount of delay experienced by a blocked LP that has useful work to do is minimized, since there are no context switches taking place until such an LP is identified. Finally, it has the property that the amount of computational effort devoted to avoiding unnecessary blocking increases with the number of blocked LPs! When there are no idle processors, it doesn't matter if LPs are artificially blocked, and no effort is made to unblock them; but as the number of blocked LPs increases, so will the number of idle processors, and hence the rate at which blocked LPs are examined.

We call this mechanism *lazy blocking avoidance*, since no work is done unless some physical processor has nothing better to do. In contrast, the use of null messages (or its shared-memory analog) can be viewed as an *eager* approach to artificial blocking and deadlock avoidance, since it consumes processor resources on a regular basis (e.g., each time an LP's clock advances, or each time an LP blocks). Lazy blocking avoidance is executed by processors, which are physical resources, rather than LPs, which are logical ones. In the remainder of this paper, we will use the term *eager blocking avoidance* to mean "deadlock and artificial blocking avoidance based on null messages" to emphasize this distinction and to avoid ambiguous use of the term "deadlock avoidance".

3.3 Detecting and Breaking Deadlocks

Unfortunately, it is not always possible to avoid deadlocks in a parallel simulation. In order for deadlock avoidance to work perfectly, the simulation cannot contain a cycle of LPs, all having a lower bound message processing time of zero. (For a proof of this, refer to Peacock, Wong, and Manning [15]; an intuitive explanation is that if such a cycle existed, then a message could traverse the cycle infinitely many times without advancing any clock values, and the LPs in the cycle could never know if it were safe to proceed.) For this reason, some mechanism for detecting and breaking deadlocks is almost always required.

As mentioned previously, algorithms for detecting deadlock in a distributed system are expensive. Because of this expense, the algorithm may only be run periodically, and a significant amount of time may elapse between the time a deadlock occurs and the initiation of the deadlock detection algorithm. In contrast, detecting deadlock in a shared-memory environment is trivial: deadlock has occured when all processors are idle and there are no LPs ready to run. A simple check can be performed each time a processor goes idle, ensuring that the deadlock will be detected immediately after the last running LP blocks.

In order to break deadlock, the scheduler must determine $\hat{t}$, the earliest simulation time at which any message will be simulated by any LP, assuming no further messages arrive. Let $\hat{m}_i$ be the earliest pending message at LP_i, with timestamp $\hat{m}_i.time = \hat{t}_i$. Assuming that no further messages arrive at LP_i, $\hat{m}_i$ will be the next message simulated by LP_i; furthermore, doing so will cause C_i to be set to $\max(\hat{t}_i, C_i) = \hat{C}_i$[2]. Then $\hat{t}$ is simply $\min_i\{\hat{C}_i\}$. Since $\hat{t}$ is the earliest time at which any message in the simulation will be simulated, it is clearly a lower bound on the earliest time that any message will be sent. Thus, the message horizon of every LP in the simulation can be advanced to $\hat{t}$, and every LP_i for which $\hat{t}_i < \hat{t}$ can be awakened.

The efficiency of this procedure hinges on the algorithm used to compute $\hat{t}$. The most straightforward algorithm would be to inspect every LP in the simulation. This could be expensive when the number of LPs is large.

An alternative would be to maintain a priority queue of blocked LPs, ordered by increasing values of C_i. The computation of $\hat{t}$ could then be done merely by inspecting the head of the queue. Additionally, the LPs having pending messages timestamped earlier than $\hat{t}$ would be likely to be near the head of the queue[3]. The drawback to this method is the cost of maintaining the priority queue. Although this may seem wasteful, especially if deadlocks are infrequent, this overhead is not as large as it first appears. First of all, there are many well-known priority queue structures for which the complexity of insertion and deletion is logarithmic in the size of the queue [1]. Second, note that LPs never need to be removed from the queue, and for some priority queue implementations, such as the heap, repositioning an object is much cheaper than deleting and re-inserting it. More importantly, *an LP only needs to be repositioned when it is about to block*. This is

[2] If there are no pending messages at LP_i, $\hat{C}_i = \hat{t}_i = \infty$.

[3] Note, however, that any such LP could be arbitrarily far back in the queue if $C_i \gg \hat{t}_i$.

because the queue order needs to be correct only at the time a deadlock occurs, but so long as any LP is still running, the simulation cannot be deadlocked.

There is one "catch" to this scheme: in order to guarantee that the queue is correctly ordered when a deadlock occurs, a blocked LP must be awakened whenever its $\hat{C}$ value changes. This can only happen when the LP receives a message m such that $m_i.time < \hat{t}_i$, in which case it should be awakened by the LP that sent the message.

3.4 Summary of Techniques

We have identified four techniques for speeding up a parallel simulation in a shared-memory environment:

1. Exact measures of progress of all LPs in the system can be obtained by inspecting shared variables, instead of relying on (possibly stale) information supplied by received messages.
2. Explicit sending of null messages is unnecessary for the implementation of "standard" deadlock avoidance.
3. The lazy blocking avoidance technique may significantly reduce the overhead of deadlock and artificial blocking avoidance.
4. A centralized scheduler can make deadlock detection trivial and deadlock breaking inexpensive.

4 Performance Measurements

The techniques discussed in the previous section have been implemented in the *Synapse* system. Synapse uses the conservative approach to LP synchronization, but exploits the availability of shared memory to provide efficient solutions to the problems of deadlock and artificial blocking.

Synapse is a combination of an object-oriented programming environment and a run-time system. As a programming environment, Synapse provides abstract types, or *classes*, that represent logical processes (the *LogicalProcess* class), simplex communication channels (the *Link* class), and timestamped messages (the *Message* class). As a run-time system, Synapse provides a separate thread of control for each LogicalProcess instance, and a scheduler that works together with the fundamental classes to ensure a correct and efficient parallel execution of the simulation. By using Synapse, a simulation programmer is relieved of the burden of synchronizing LPs and handling deadlocks.

Synapse currently is implemented as an extension of the PRESTO parallel programming environment [3, 4, 5]. PRESTO is a set of tools for building parallel programming systems on shared-memory multiprocessors. PRESTO's object-oriented design allows easy extension, not only at the level of the interface to the applications programmer, but also at the level of the run-time kernel. Synapse has exploited this flexibility by customizing specific parts of the run-time kernel (such as the scheduler) without having to change other components of the system. PRESTO has proven to be an excellent implementation base for the rapid construction of an efficient parallel simulation environment.

4.1 The Benchmarks

Empirical studies of the conservative (Bryant-Chandy-Misra) approach to parallel simulation have been reported by Reed, Malony, and McCredie [17], Reed and Malony [16], and Fujimoto [9, 10]. All of these studies were conducted on shared-memory multiprocessors: a 20-processor Balance* 21000 in the case of Reed et al., and a 17-processor Butterfly† in the case of Fujimoto. We used these studies as a source of benchmarks, supplemented by some of our own. Each of the benchmarks used in our study is a closed queueing network with a fixed customer population; they are described next.

*Balance is a trademark of Sequent Computer Systems, Inc.

†Butterfly is a trademark of Bolt, Beranek, and Newman, Inc.

Reed, Malony, and McCredie used a variety of queueing network models as benchmarks. Two of these, the cyclic network and the central server network, are shown in Figures 3-4. In their study, each server of the queueing network is a separate logical process, and each customer in the network is represented by a message. In addition, the fork and merge points in the network are represented as separate logical processes, distinct from the server LPs[4]. Thus, a server LP simply sends each received messages on its (single) output link, changing the message's timestamp to reflect the service demand at that node. Fork and Merge nodes do nothing except forward the messages they receive, without modifying their timestamps (of course, Fork nodes also choose a destination for each message). Note that although Synapse does not distinguish between routing and service nodes, we implemented this separation of functionality for the two benchmarks shown in order to make our results comparable to those of Reed et al.

The ratio of communication to computation in a simulation of a queueing network model is large, and the separation of service and routing increases it even more. This suggests that it should be difficult to obtain good parallel speedup. In fact, their initial work led Reed et al. to conclude that, with rare exceptions, the conservative approach is not a viable one for the parallel simulation of queueing network models. Subsequently, Reed and Malony conjectured that the conservative approach might be more attractive for problems with a lower ratio of communication to computation. They extended [17] by having each logical process representing a server spin for a specified amount of *real* time, simulating extra computational effort in the simulation, each time a message was processed[5]. However, once again the authors concluded that the conservative technique is not viable for the simulation of queueing network models.

Fujimoto's study took a similar approach to that of Reed and Malony, in that it, too, inserted artificial computational delays in the servers. His study primarily used another benchmark, the torus network, but also included the central server network used by Reed et al. A 4x4 torus network is shown in Figure 5. Note that there is no separation of service and routing functions in the torus network benchmark. Also note that each of the edges in the illustration represents a pair of communication channels, one in each direction. *Fujimoto met with generally better success than Reed et al.*

Finally, we introduce one additional benchmark in this paper, the fully-interconnected network. In this model, each node can send messages to and receive messages from any other node. Because of the high degree of connectivity in this model ($k(k-1)$ communication links in a k-node network), it furnishes a stress test for simulation strategies that rely on null messages to avoid deadlock and artificial blocking.

4.2 Our Hardware Testbed

Synapse was benchmarked on two versions of a Sequent Computer Systems shared-memory multiprocessor. We were fortunate (or cursed, depending on your point of view) to have our Sequent upgraded from a Balance 21000 to a Symmetry* 81 during the course of this research. The most important attributes of these systems are summarized in Table 1 [20, 21]. (Our Balance system was configured almost identically to the one used by Reed et al. Note that our Symmetry system was an early model that still utilized write-through caches. The next release of Symmetry will have write-back caches. This will reduce bus traffic considerably. Sequent estimates that system performance will increase by 25-40% when the write-back cache is added.)

4.3 Balance Experiments

The 4 node cyclic network and the 5 node central server network were benchmarked on the Balance implementation of Synapse. Since we had 9 processors available, we were able to use one processor per LP for each of these benchmarks. Separate runs were made using all three of the algorithms described in Section 3: deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance. (The eager blocking avoidance implementation used explicit null messages, these message were sent only just before an LP was

[4] This is because the authors were using the RESQ [19] scheme for queueing network specification.

[5] It should be stressed that the amount of *real* time consumed by the logical process is completely independent of the amount of *simulated* time consumed by the server.

*Symmetry is a trademark of Sequent Computer Systems, Inc.

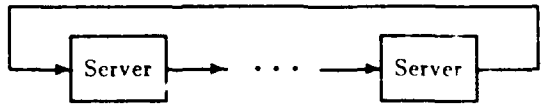


Figure 3: Cyclic network.

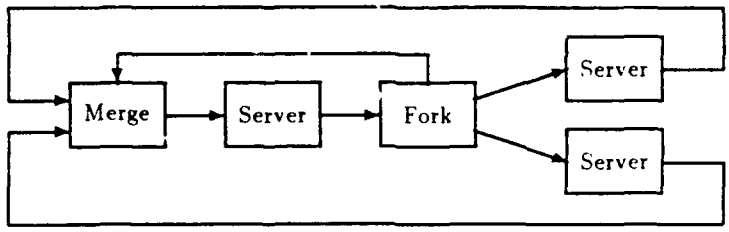


Figure 4: Central server network.

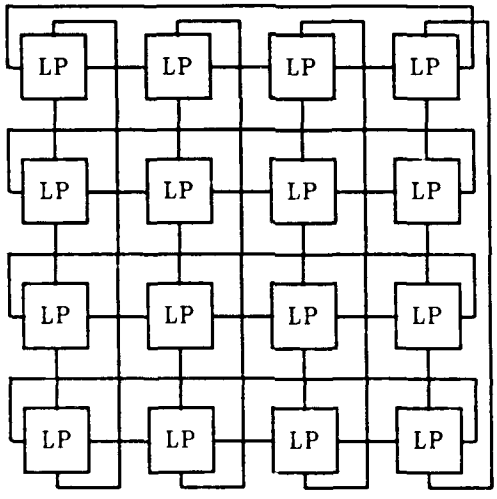


Figure 5: Torus network (4x4).

Attribute		Balance	Symmetry
processor subsystem	quantity	10	10
	cpu	NS32032	i80386
	clock rate	10MHz	16 MHz
	fpu	NS32031	i80387, WTL1167
cache	size	8 Kbytes	64 Kbytes
	discipline	write-through	write-through
eff. bus bandwidth		26.7 Mb/s	53.3 Mb/s
memory size		32 Mbytes	32 Mbytes
acquire & release lock		22 μsec.	6.2 μsec.
null procedure call		15 μsec.	4.9 μsec.

Table 1: Comparison of hardware testbeds.

about to block, as opposed to each time its clock advanced.) In all cases, the service time distributions for all LPs were deterministic with identical means.

Following the methodology of Reed, et al., a 1-processor "parallel" simulation using deadlock detection and recovery was used as the base value for all speedup calculations. (Deadlock detection and recovery was used because it provides the best performance in the 1-processor case.) The 1-processor parallel implementation was almost certainly not as efficient as a good sequential implementation would have been. Although this tends to exaggerate the absolute speedup values, we used this approach in order to make our results directly comparable to those of Reed et al. Since our main goal was to illustrate the relative worth of the various deadlock strategies, and in particular, to validate lazy blocking avoidance as a useful alternative to traditional approaches, the choice of single processor algorithm was irrelevant.

4.3.1 The Cyclic Network

Note that the absence of any merge points in this network means that each LP can always process messages immediately upon arrival, which in turn means that there can never be any deadlocks. In addition, the absence of branching in the network eliminates the need to send any null messages. Under these conditions, perfect speedup should be achievable no matter which of the parallel simulation algorithms is used. This benchmark can thus be considered a "minimal competency" test for a parallel simulator, since if it could not do well under such ideal circumstances, there would be little hope for reasonable performance on more complicated problems! Figure 6 shows that Synapse can indeed achieve perfect speedup on this benchmark using all three simulation algorithms[6].

These results demonstrate the general efficiency of Synapse's multi-threaded run-time kernel. Our speedup figures are approximately twice as good as those reported by Reed et al. [17], in which a 1-processor parallel implementation was also used as the basis for speedup calculations. (Also note that our Balance system was configured almost identically to Reed's.)

4.3.2 The Central Server Network

The results for this network are shown in Figure 7. The first graph compares the speedup obtained with the deadlock detection and recovery, lazy blocking avoidance, and eager blocking avoidance algorithms. Lazy blocking avoidance outperforms deadlock detection and recovery across the board. Eager blocking avoidance performs poorly at low network populations due to the fact that LPs are spending most of their time propagating null messages. However, at populations greater than 2, eager blocking avoidance appears to be the top performer. All three methods are equally good once the population reaches a certain level

[6] The apparent super-linear speedup for population 8 was an anomaly caused by context switches; the details are unimportant.

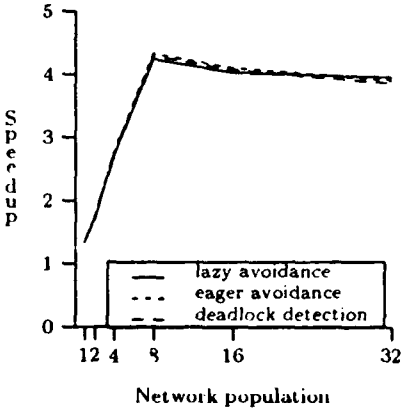


Figure 6: Speedup for 4-node cyclic network.

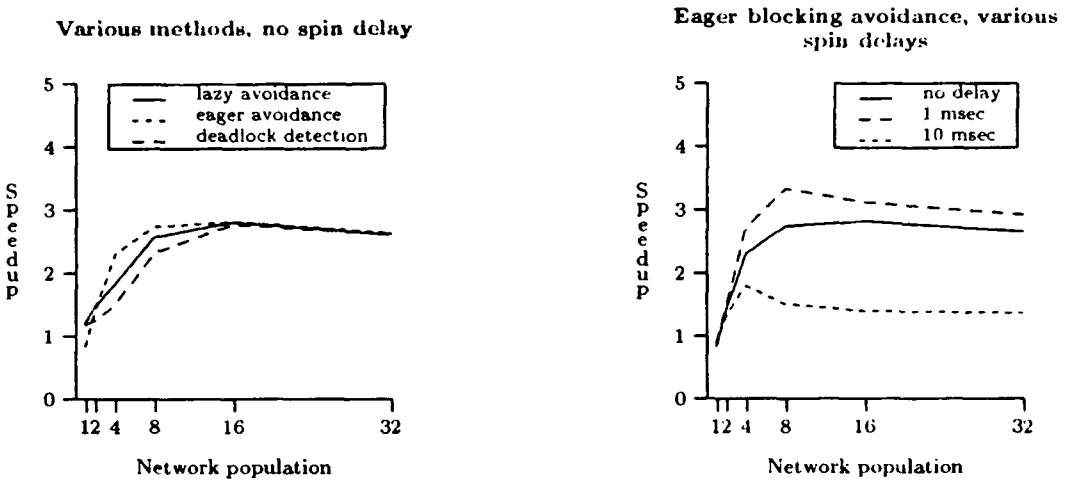


Figure 7: Speedup for central server network.

(approximately 16), since there are enough messages in the network so that deadlocks rarely occur. In all cases, we were able to exceed the speedups reported in [17].

The second graph in Figure 7 shows the effects of inserting real-time spin delays of 1 and 10 milliseconds (at LPs corresponding to servers only) on the eager blocking avoidance algorithm. (Similar, but smaller, variations in performance were obtained for the other two algorithms.) In each case, the delay was inserted at a point after a message is received but *before* any messages are sent out. This corresponds to the methodology used in [9, 10, 16]. In all three approaches, the inclusion of a small delay improves speedup slightly, but a larger delay has a much stronger, negative effect. It would seem that as the delay gets longer, processor utilization ought to increase, leading to better speedup. But in fact, with the 10 millisecond delay speedup is essentially a constant that is only slightly larger than unity. We will now argue that this is a limitation inherent to the model, rather than of the implementation.

Consider a queueing network model of the simulation itself. By examining the topology of the network we can compute upper bounds on the utilizations of each LP. Without the spin delay, we assume that the service times at each LP (i.e. the *real time cost* of processing a message) are identical. In this case, the asymptotic utilizations of the merge node, the fork node, and the central server node are each unity, and the utilizations of the peripheral server nodes are each 0.33. Thus, the total utilization in the network, which is an upper bound on speedup, is only 3.67.

The addition of a real time spin delay to the server nodes but not to the fork and merge nodes changes the relative service times of the nodes and hence will affect utilizations. As the service times at the server nodes grow large compared to the service times at the fork and merge nodes, the total utilization in the network approaches 1.67. This explains the poor performance of the simulation when a 10 millisecond spin delay is used.

4.3.3 Summary of Balance Results

For the cyclic network benchmark, we were able to achieve perfect speedup, relative to a 1-processor "parallel" implementation. (The apparent super-linear speedup for this benchmark was an anomaly caused by context switches.)

For the central server benchmark, we could not achieve perfect speedup; however, we presented an analytical argument that shows that speedup for this benchmark cannot exceed 3.67. This argument also predicted poor speedup when large computational delays were inserted at the server LPs. In both cases, our measured speedups were a substantial fraction of the predicted maximum speedups.

4.4 Symmetry Experiments

A 4x4 torus networks and a 16 node fully-interconnected network were benchmarked on the Symmetry version of Synapse. Both benchmarks were simulated using various numbers of processors, with a maximum of 9.

As we wanted to compare our results for the torus network to those obtained by Fujimoto [9, 10], we made three changes to our experimental methodology and one change to our implementation:

- Speedup calculations were based on results obtained from a true sequential simulator. This means that (a) a global event list was maintained, eliminating deadlocks, and (b) there was no synchronization overhead due to locking. However, in order to conform to the process/message paradigm of Synapse, rather than doing a procedure call to process events as they are removed the event list, a co-routine switch to a separate thread of control was performed[7].
- Fujimoto's research showed that the timestamp increment (i.e. service time) distributions in the LPs had a profound effect on performance. Thus, we ran each benchmark using a variety of timestamp increment distributions.
- The amount of cpu time that LPs spent processing each message was bounded from below by an exponentially distributed random variable with a mean of 1 millisecond.
- The implementation of eager blocking avoidance was modified to eliminate the explicit sending of null messages.

4.4.1 The Torus Network

We benchmarked a 4x4 torus network using 9 processors. Figure 8 compares the performance of the three synchronization methods on this benchmark with a network population of 64 (4 messages per node). The first graph shows speedup as a function of the number of processors, using a deterministic timestamp increment function with mean 1 (the optimal case). The second graph shows the result of using an exponentially distributed timestamp increment function with mean 10.5 and minimum value 1. These two graphs convincingly demonstrate the importance of the timestamp increment function on performance. In both cases, our results for the deadlock detection and recovery and eager blocking avoidance methods compare favorably with those obtained by Fujimoto.

Note that lazy blocking avoidance is a large improvement over deadlock detection and recovery, bearing out the hypothesis that a significant amount of parallelism is lost due to artificial blocking. For this benchmark, however, eager blocking avoidance is the top performer.

4.4.2 The Fully-Interconnected Network

We devised the fully-interconnected network benchmark specifically to tax the eager blocking avoidance strategy. Since there is so much branching in this network, null messages multiply extremely rapidly.

Figure 9 shows the performance of Synapse on a 16 node fully-interconnected network with a message population of 240[8]. As before, the first graph shows the results obtained when the timestamp increment function is deterministic with mean 1, and the second graph shows the results obtained when it is exponentially distributed with mean 10.5 and minimum value 1. In both cases, lazy blocking avoidance is superior

[7] It should be noted that, thanks in part to the computational delay inserted after the processing of each message, the overhead of context switches was never more than 5%. Regardless, it would be an unfair to compare Synapse to a procedure call-based sequential simulator, since Synapse could itself be implemented in such a fashion (i.e., by using a squad of "worker threads", one per processor, to animate the LPs.) However, this would require a change in the programming paradigm, since an LP's state would have to be explicitly saved by the programmer each time the LP blocked.

[8] This population may seem rather high, but 240 messages is only one message per link (each LP is connected to 15 other LPs). So on a per-link basis, the population density for this benchmark is the same as for the torus benchmark in the previous section.

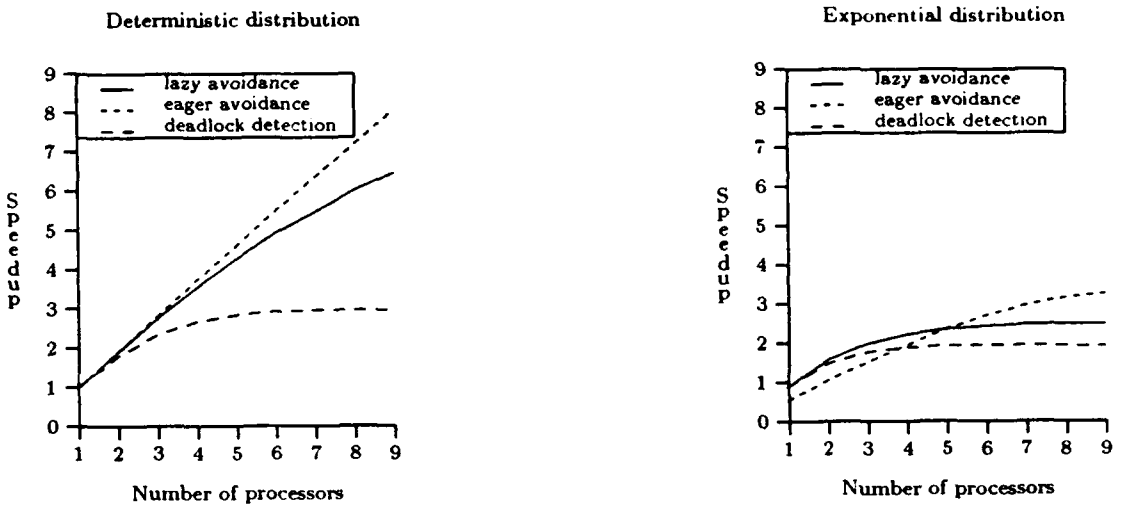


Figure 8: Speedup vs. number of processors for 4x4 torus, 4 messages per node.

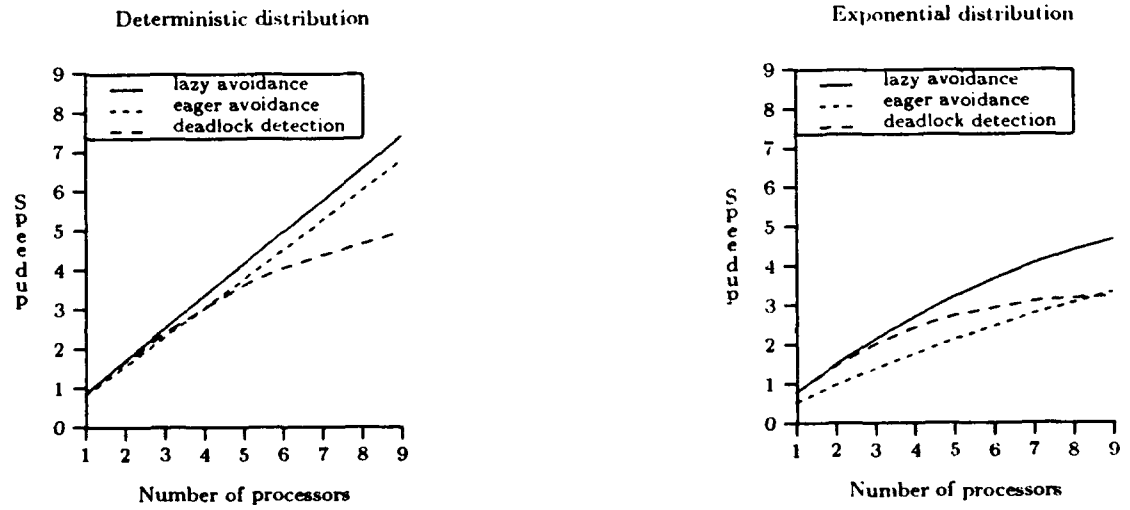


Figure 9: Speedup vs. number of processors for 16 node fully-interconnected network, 15 messages per node.

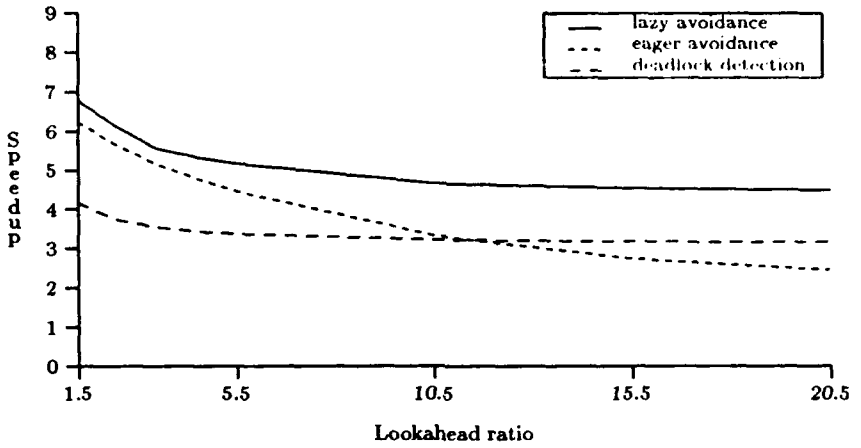


Figure 10: Speedup as a function of lookahead ratio (exponentially distributed timestamp increments).

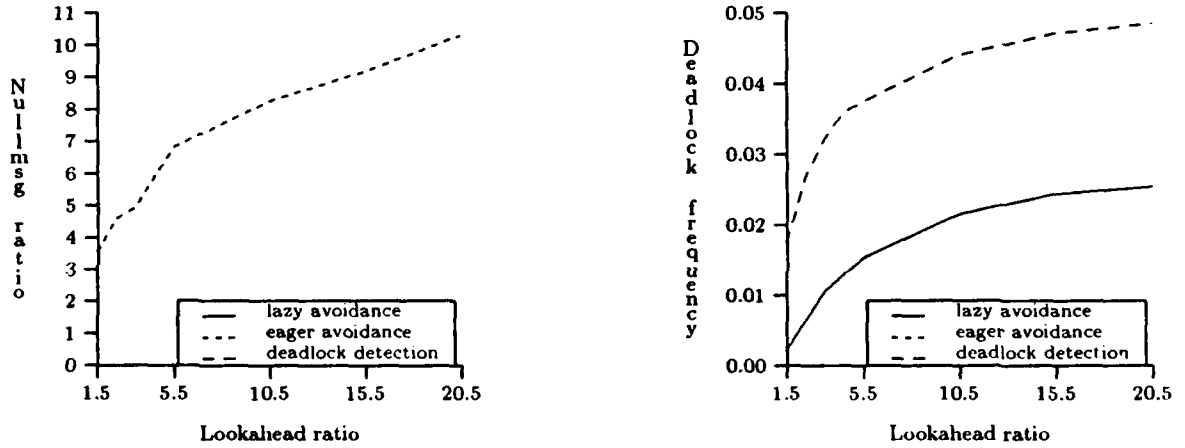


Figure 11: Null message ratio and deadlock frequency as a function of lookahead ratio (exponentially distributed timestamp increments).

to the other two methods.

Although all three protocols are affected by changes in the timestamp increment function, eager blocking avoidance is by far the most sensitive in this respect. Figure 10 shows speedup (using 9 processors) as a function of the *lookahead ratio* of the timestamp increment function. The lookahead ratio, as defined by Fujimoto, is the ratio of the mean of the timestamp increment function to its minimum value, or *lookahead*. (In all cases, the distribution of timestamp increments was exponential. Also, the distribution had a lower bound of 1, so the lookahead ratio was equal to the mean of the distribution.)

The large degradation in performance of the eager blocking avoidance protocol is due to the circulation of (virtual) null messages in the network. At each LP, the timestamp of the "null message" (i.e., the lower bound on the timestamp of future messages produced by the LP) increases by the value of the lookahead. As the lookahead ratio increases, a "null message" will have to circulate for a much longer period of time, on the average, before some LP's clock will have advanced far enough to enable it to process a real message. The first graph in Figure 11 shows that the *null message ratio* (the average number of null messages "sent" per real message) is a monotonically increasing function of lookahead ratio.

On the other hand, the performance of deadlock detection and recovery and lazy blocking avoidance are much more robust with respect to changes in lookahead ratio. The second graph in Figure 11 shows *deadlock*

frequency (the average number of deadlocks per message) as a function of lookahead ratio. In contrast to the null message ratio, the deadlock frequency appears to be bounded from above. Even more interesting is the fact that the average number of LPs awakened each time deadlock is broken (not shown here) is nearly a constant, with a value of approximately 6.3 for both algorithms. This suggests that, even when LPs have extremely poor lookahead characteristics, if the network is large enough it may be profitable to simulate it in a quasi-synchronous fashion. By this we mean an implementation in which all LPs are periodically forced to block, at which point the deadlock breaking algorithm would be run[9]. An approach similar to this has been used for parallel Monte Carlo simulation by Lubachevsky [13].

Finally, note that in all cases the frequency of deadlocks is much lower using lazy blocking avoidance than using deadlock detection and recovery.

4.5 Summary of Performance Measurements

We are encouraged by these preliminary results of Synapse's performance on the benchmarks. They demonstrate that the conservative approach to parallel simulation, when tailored to a shared memory environment, can yield good performance. This is in sharp contrast to the conclusions reached by Reed et al. [16, 17]. We believe there may be several reasons for this:

- Some of their benchmarks contained inherent limitations on obtainable speedup (Section 4.3.2).
- Their implementation failed to take advantage of the lookahead characteristics of FIFO servers [9, 10]. This may explain their lackluster performance results for the cyclic network, which in principle should yield perfect speed up, and does so in our experience (Section 4.3.1).
- They did not exploit the accessibility of shared state to reduce null message traffic.

The outlook for conservative shared memory parallel simulation painted by Fujimoto was considerably brighter [9, 10]. However, Fujimoto was only able to achieve good speedup using eager blocking avoidance. Our results indicate that lazy blocking avoidance can outperform eager blocking avoidance when the network is highly interconnected. Also, the performance of lazy blocking avoidance (as well as deadlock detection and recovery) seems to be more robust than that of eager blocking avoidance with respect to the timestamp increment distribution used by the LPs.

Finally, we note that the average number of LPs awakened per invocation of the deadlock breaking algorithm appears to be independent of the lookahead ratio. This suggests a quasi-synchronous strategy for simulating large networks containing LPs with poor lookahead characteristics.

5 Conclusions

The preliminary performance results of the Synapse system demonstrate that the conservative approach to parallel simulation, when tailored to a shared memory environment, can yield good speedup. Synapse's novel approach to the problems of deadlock and artificial blocking, *lazy blocking avoidance*, greatly reduces deadlock frequency and improves speedup as compared to deadlock detection and recovery. Our results also indicate that lazy blocking avoidance can outperform eager blocking avoidance (i.e., null messages) when the network is highly interconnected, *especially* when LPs have poor lookahead characteristics.

Synapse is a programming environment as well as a run-time system. The object-oriented design and inheritance mechanisms of the Synapse implementation make it easy for a programmer to construct correct, efficient parallel simulations, without having to concern himself with issues such as synchronization and deadlock.

We plan to continue this research along the following lines:

[9] This is *not* equivalent to using a centralized event list and simulataneously processing all events with the exact same timestamp. In the approach suggested here, events with different timestamps may still be processed in parallel.

- General performance tuning of the simulator. In particular, we plan to benchmark Synapse on a Symmetry system with more than 10 processors to identify any performance bottlenecks as the system is scaled up.
- Investigation of methods for inferring lookahead characteristics of LPs without requiring that such knowledge be provided explicitly by the programmer.
- Development of a parallel simulation "toolkit" consisting of various sub-classes of the LogicalProcess, each one optimized for performing a particular function (e.g. processor sharing servers, multiple class servers, flow-controlled servers, etc.).

Acknowledgments

A colleague of ours, Jason Lin, first pointed out the problem with inserting spin delays in the central server network. We also would like to thank our friends at Sequent Computer Systems: Bob Beck, for his assistance in porting the PRESTO kernel to the Symmetry, and Shreekant Thakkar, for his insights into some of our early performance bottlenecks.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] B. Beck and B. Kasten. VLSI Assist in Building a Multiprocessor UNIX System. In *Proc. USENIX Summer Conference*, 1985.

[3] B.N. Bershad. The PRESTO User's Manual. Technical Report 88-01-04, Department of Computer Science, University of Washington, January 1988.

[4] B.N. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report 87-09-01, Department of Computer Science, University of Washington, September 1987. To appear, *Software Practice and Experience.*

[5] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D.B. Wagner. An Open Environment for Building Parallel Programming Systems. In *ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988.

[6] R.E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT,LCS,TR-188, Massachussetts Insitute of Technology, Cambridge, Mass., 1977.

[7] K.M. Chandy, L.M. Haas, and J. Misra. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[8] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[9] R.M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. Technical Report UUCS-87-026a, Department of Computer Science, University of Utah, November 1987.

[10] R.M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 14–20, San Diego, CA, February 1988. Society for Computer Simulation International.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Time Warp Operating System. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93, Austin, TX, November 1987. ACM.

[12] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[13] B.D. Lubachevsky. Efficient Distributed Event Driven Simulation of Multiple-Loop Networks. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 12–21. ACM, May 1988.

[14] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–60, March 1986.

[15] J.K. Peacock, J.W. Wong, and E.G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, March 1979.

[16] D.A. Reed and A.D. Malony. Parallel Discrete Event Simulation: the Chandy-Misra Approach. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 8–13, San Diego, CA, February 1988. Society for Computer Simulation International.

[17] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[18] P.F. Reynolds. A Shared Resource Algorithm for Distributed Simulation. In *Proc. 9th International Symposium on Computer Architecture*, pages 259–266, Austin, TX, 1982. IEEE.

[19] C.H. Sauer, E.A. MacNair, and S. Salza. A Language for Extended Queueing Networks. *IBM Journal of Research and Development*, 24(6):747–755, November 1980.

[20] Sequent Computer Systems, Inc., Portland, OR. *Balance Technical Summary*, November 1986.

[21] Sequent Computer Systems, Inc., Portland, OR. *Symmetry Technical Summary*, February 1987.

[22] C.P. Thacker and L.C. Stewart. The Firefly Multiprocessor Workstation. In *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Palo Alto, CA, October 1987. ACM.

[23] S.S. Thakkar, P.R. Gifford, and G.F. Fieland. Balance: A Shared Memory Multiprocessor System. In *Proc. 2nd International Conference on Supercomputing*, May 1987.

Parallel Simulation of Queueing Networks: Limitations and Potentials

David B. Wagner and Edward D. Lazowska
Department of Computer Science
University of Washington
Seattle, WA 98195

September 12, 1988

Abstract

This paper concerns the parallel simulation of queueing network models (QNMs) using the conservative (Chandy-Misra) paradigm. Most empirical studies of conservative parallel simulation have used QNMs as benchmarks. For the most part, these studies concluded that the conservative paradigm is unsuitable for speeding up the simulation of QNMs, or that it is only suitable for simulating a very limited subclass of these models (e.g., those containing only FCFS servers). In this paper we argue that these are unnecessarily pessimistic conclusions. On the one hand, we show that the structure of some QNMs inherently limits the attainable simulation speedup. On the other hand, we show that QNMs without such limitations can be efficiently simulated using some recently introduced implementation techniques.

We present an analytic method for determining an upper bound on speedup, and use this method to identify QNM structures that will exhibit poor simulation performance. We then survey a number of promising implementation techniques, some of which are quite general in nature and others of which apply specifically to QNMs. We show how to extend the latter to a larger class of service disciplines than had been considered previously.

1 Introduction

Queueing network models (QNMs) are important tools for studying the performance of computer systems [8]. Although QNMs are usually viewed as an analytic alternative to computationally expensive simulation techniques, there are in fact several reasons why performance analysts might want to simulate the performance of QNMs:

- A QNM may not be analytically tractable.
- An approximate analytic solution technique may need to be validated.

Since simulation is so computationally expensive, there recently has been a great deal of interest in bringing the power of parallel processing to bear on the task. Most studies of the conservative

Our work is supported by the National Science Foundation (Grants No. CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

parallel simulation paradigm [3, 12] used QNMs as benchmarks [5, 6, 14, 16, 17, 21]. Many of these studies concluded that the conservative paradigm is unsuitable for simulating QNMs, or that it is only suitable for simulating a very limited subclass of these models (e.g., those containing only FCFS servers). For example, the poor performance results obtained by Reed et al. [16, 17] led those authors to conclude that "the Chandy-Misra technique is *not a viable approach* to parallel simulation of queueing network models." (Emphasis theirs.) Fujimoto [5, 6] later demonstrated that one reason for the poor results obtained by Reed et al. was that their implementation was too general – it did not take advantage of the *lookahead* characteristics of FCFS servers[1]. He concluded that even QNMs containing cycles can achieve good performance if server semantics are exploited to provide as much lookahead as possible. He did postulate, however, that the conservative approach would be ill-suited for simulating certain types of servers (e.g., prioritized queues), due to their "inherently" poor lookahead properties.

In this paper we argue that these are overly pessimistic conclusions. Our argument is broken into three parts, presented in Sections 3, 4, and 5.

In Section 3, we show that certain QNMs can be guaranteed to yield poor speedups when simulated in parallel, because their structure limits this speedup even though the systems they represent may contain a great deal of parallelism. We present a simple analytic method for determining an *a priori* upper bound on the degree of parallelism of a QNM parallel simulation. This method can be used both to identify QNMs that cannot benefit from parallel simulation, and as an aid in deciding how to partition a QNM into units of execution in the parallel simulation. This partitioning can yield improved speedups.

In Section 4, we briefly survey a number of recent techniques that yield substantial improvements in parallel simulation speedups for general problems. These techniques can be characterized as aggressively exploiting shared memory. As an example, Fujimoto compared two synchronization schemes: deadlock detection and recovery, and null message-based deadlock avoidance. Deadlock avoidance was superior, but he tested only a single benchmark extensively (the torus network). In [21], we (along with B. Bershad) introduced a new synchronization method, *lazy blocking avoidance*, that outperformed null message-based deadlock avoidance in networks with a high degree of connectivity. In addition, we found that the performance of lazy blocking avoidance was less sensitive to *lookahead ratio*[2] than was null message-based deadlock avoidance.

In Section 5, we briefly survey and then extend techniques that are specifically oriented towards the parallel simulation of QNMs. For example, recently Nicol [14] introduced an improved technique for computing the lookahead of QN servers, called the *future list*. Whereas previous work concentrated on methods for *propagating* lookahead values, Nicol was the first to treat the *computation* of lookahead as a separate concern. His technique appeared to be very promising, but its application was limited to FCFS servers. We will show that the future list technique can be extended to a much wider class of service disciplines.

The remainder of this paper is organized as follows. Section 2 briefly reviews the conservative paradigm for parallel simulation. Sections 3, 4, and 5 are as described above. Finally, Section 6 presents conclusions and directions for future research.

[1] Lookahead characterizes the ability of a logical process to predict future messages that it will send; FCFS servers have excellent lookahead.

[2] The lookahead ratio of a server is defined as the mean of the server's timestamp increment (i.e., service time) distribution divided by the server's lookahead.

2 Conservative Parallel Simulation

A widely used model for describing parallelism in discrete event simulations was proposed independently by Bryant [2] and by Chandy and Misra [3]. The physical system to be simulated is partitioned into a set of component entities called *physical processes* (PPs) which interact only at discrete times (e.g., through the scheduling of events). This system of PPs can then be simulated as a collection of *logical processes* (LPs) that communicate via the sending and receiving of time-stamped messages. The scheduling of an event for PP_x at time t in the physical system is simulated by sending a message with timestamp t to LP_x. The LPs are scheduled on the physical processors.

Each LP has its own local clock, which indicates how long the LP has executed in *simulation time*. In effect, the global event list and global clock of a sequential simulation have been done away with; their counterparts in a parallel simulation are the set of LP input message queues and the set of local clocks, respectively. In order to correctly simulate a PP, the corresponding LP must process messages in timestamp order, as opposed to their real-time arrival order. The major challenge to implementing such a parallel simulation is the synchronization of LPs to ensure that event causality is maintained, without resorting to lock-step (i.e., time-driven) execution.

Two general approaches to synchronization in parallel discrete event simulation have been proposed in the literature: optimistic and conservative. In the optimistic approach [8], an LP can process every message as soon as it arrives; however, if a message with an earlier timestamp subsequently arrives, the LP must roll back its state to the time of the earlier message and re-execute from that point. This may require the cancellation of messages that the LP has sent, which can cause rollback at other LPs. Optimistic parallel simulation requires substantial low-level support in order to be efficient [7].

In the conservative approach [2, 3, 12], an LP does not accept a message for processing until it is certain that no message with an earlier timestamp can ever arrive. Since LPs may have to wait for other LPs to produce messages before they can proceed, deadlock can occur even if the system being modeled is deadlock-free. Also, it is very often the case that an LP is blocked waiting for a message from some other LP whose clock has already exceeded the pending message that the first LP wishes to process. In this case, using the terminology of Reynolds [18], we say that the LP is *artificially blocked.*

One solution to the deadlock problem is to allow the simulation to deadlock, detect it, and then recover. The simulation thus consists of a sequence of phases performing useful computation (hopefully) in parallel, separated by *phase interfaces*, wherein a computation takes place to to break the deadlock and allow various LPs to proceed [3]. Two drawbacks to this approach are immediately apparent. First, the simulation is making no progress during the phase interfaces. Second, this approach does nothing to reduce the amount of artificial blocking in the simulation.

An alternative to deadlock detection and recovery that also addresses artificial blocking is *deadlock avoidance.* A typical deadlock avoidance scheme has each LP periodically sending *null messages* to its "downstream" LPs to inform them that the sender will not be sending any real messages before a certain time. Unfortunately, this approach can be prohibitively expensive, especially when the degree of branching in the communication graph is high. (In Section 4 we survey a number of shared-memory techniques for reducing the overhead of deadlock avoidance.)

Unfortunately, experience shows that solutions to the problems of deadlock and artificial blocking are necessary, but not sufficient, in order to achieve good speedup in a parallel simulation. In the next section, we show that even under the very optimistic assumption that the frequency of

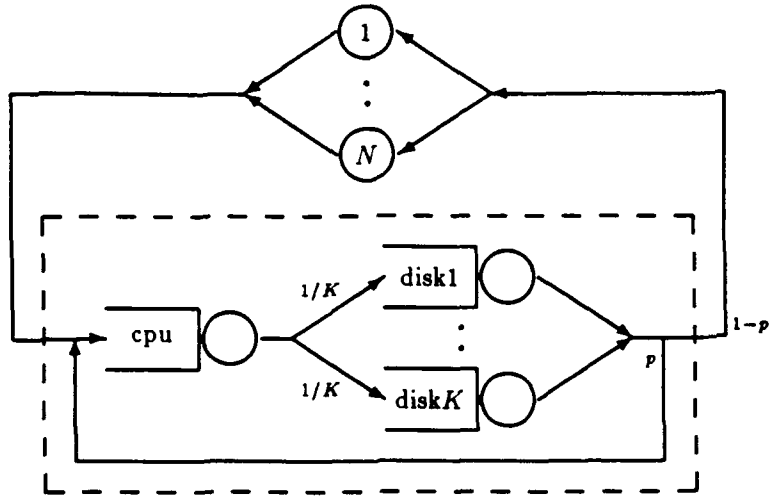


Figure 1: Example of a QNM with poor parallel simulation characteristics.

deadlocks and artificial blocking is negligible, it is still not possible to achieve good speedup for some QNM topologies.

3 Bounding the Speedup of QNM Simulations

A QNM consists of a collection of service centers (servers). In order to simulate the QNM in parallel, these servers must be mapped into a collection of logical processes (LPs). The most straightforward mapping would be to use a separate LP to simulate each server in the QNM. (For the moment, we ignore the question of whether or not such a mapping is optimal.) This strategy can lead to unrealistic expectations about the degree of parallelism in the simulation. This is best illustrated by example.

Figure 1 shows a QNM model of a timesharing computer system with N terminals, a single cpu, and K disks. For the moment, consider only the subnetwork inside the dashed box. There is the potential for a great deal of parallelism in this subnetwork, since $K + 1$ customers can be in service at the same time (one customer at the cpu and one customer at each of the disks). This parallelism is realized because the service time per visit at the cpu is orders of magnitude smaller than the service time at each of the disks.

However, the presence of parallelism in the system being modeled does not imply the presence of the same degree of parallelism in the simulation of that system. To a first approximation, the computation required by an LP is proportional to the number of messages processed by that LP. Since the total message throughput of all of the disk LPs must equal the message throughput of the cpu LP, the total of the utilizations of the disk LPs will equal the utilization of the cpu LP, which cannot exceed one. In other words, the cpu LP is a bottleneck. Therefore, the average level of parallelism in a simulation of the indicated subnetwork, and hence speedup, cannot exceed two. We refer to this bound as the *speedup potential.*

By the same reasoning, the total utilization of all the LPs modeling the terminals cannot exceed the utilization of the cpu. In fact, because of the feedback loop within the central subnetwork, the

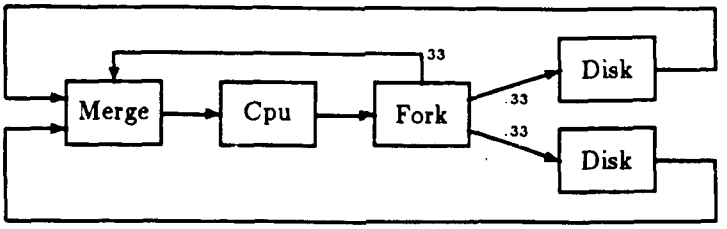


Figure 2: A central server model with explicit fork and merge nodes.

total utilization of the terminal LPs will be only $1 - p$ times the utilization of the cpu LP. Thus, the speedup potential of this entire QNM is $3 - p$. Note that this analysis ignores the possibility of deadlocks, and the associated degradation in performance. Thus, even a speedup of $3 - p$ is probably unachievable for this network!

More generally, the technique we propose is to consider the network of LPs as a queueing network model of the simulation itself. The customers in this model are the messages in the logical system, and the servers are the LPs. From the matrix of routing probabilities we can compute the relative throughputs at each of the servers, and hence their relative utilizations. (Since the service demand of a customer at a server is the *real* time needed by the server LP to process a message, we assume that all service demands are the same.) Next, we normalize the relative utilizations so that the largest of them is one. Making the optimistic assumption that every server with normalized utilization one is 100% utilized (as might happen if the message population in the network is so large that deadlocks are eliminated), the sum of the normalized utilizations is the speedup potential. (This is, of course, inspired by *aysmptotic bound analysis* [13].) Because of the assumptions involved, this quantity will only be an upper bound on speedup. Nevertheless, it is useful for identifying QNMs that are inherently unsuitable for simulation in parallel.

As a concrete example of this technique, consider the simulation benchmark shown in Figure 2, which was originally introduced by Reed et al. [17]. This benchmark represents a QNM similar to the central subsystem of Figure 1 with $K = 2$, except that there is an additional loop connecting the output of the cpu LP to its input. (The presence of the fork and merge LPs is an artifact of the RESQ [19] specification language used by Reed et al.) Despite the fact that Reed et al. used their parallel simulator for their single as well as multiple-processor measurements, they were unable to achieve a speedup greater than 1.25 for this benchmark. Both Fujimoto [5] and ourselves [21], using highly optimized simulation implementations, found that the maximum ahievable speedup for this benchmark was just under 3, despite the fact that the benchmark contains 5 LPs.

This is easily explained by noticing that message throughput is identical at the merge, cpu, and fork LPs, but that the throughput at each disk is only one-third of that amount. Thus, in a parallel simulation, the utilization of each disk LP will be only one-third the utilization of the merge, cpu, or fork LPs. Under the generous assumption that the latter three LPs will be 100% utilized, the total utilization in the network, and hence the speedup potential, is only 3.67. Therefore, as a percentage of the theoretical maximum, the speedups obtained by Fujimoto and by ourselves for this benchmark are actually quite respectable. (Note that this speaks well for the efficiency of the implementations, although it is a negative result in terms of the potential for speeding up the simulation of this particular benchmark.)

In [16], Reed and Malony conjectured that a simulation with a high ratio of computation to communication ought to have improved performance. In order to substantiate this, they simulated this benchmark with various spin delays inserted in the LPs representing the cpu and disks. However, their results were inconclusive (in no case did their speedups exceed 1.25).

Contrary to Reed and Malony's intuition, we found that speedup was actually reduced as spin delays were increased, until it approached a value only slightly greater than one. Again, this is easily explained by appealing to our method. As the spin delays are increased at the cpu and disk LPs, the utilizations of these LPs increase relative to those of the fork and merge LPs. Thus, the speedup potential in the network is asymptotically 1.67.

From the method proposed here, together with some experience, we have learned the following lessons:

- Servers (or subnetworks) that are connected in parallel between a fork and a merge point may not contribute to parallelism.
- Explicit fork and merge nodes, which are required by certain specification languages (e.g., RESQ), probably will not contribute to parallelism, especially if their service times are small relative to the "real" LPs.
- It may be difficult to obtain parallelism between different subnetworks of a simulation, if the relative visit counts at the different subnetworks vary a great deal.

Although these results may seem to be largely negative, it is not our intent to discourage research into the parallel simulation of QNMs. Rather, we hope that the identification of these common pitfalls will help avoid wasting time on "lost causes".

Also, the identification of redundant (with respect to speedup) servers can aid in the partitioning of the QNM into logical processes. There are many cases in which redundant servers can be coalesced into a single LP (cf. Figure 1), and there are several reasons why doing so may improve performance:

- Reducing the number of LPs will reduce the number of context switches.
- Reducing the number of communication channels will reduce the overhead of deadlock avoidance, especially if it is based on null messages.
- Coalescing LPs in closed (sub)networks can improve the performance of the future list technique (Section 5).

4 General Optimization Techniques

The early work on parallel simulation took place in a distributed system setting. The lack of shared memory in such systems not only increases communication costs, it also complicates synchronization.

Reed et al. [16, 17] implemented a conservative parallel simulator on a shared-memory multiprocessor in order to investigate the effect of reduced communication costs on performance. (The algorithms used were essentially the same as they would have been in a distributed environment, except that a central controller process was used to detect deadlock.) The results were disappointing.

The experience of Reed et al. points out that communication costs are not the only obstacle to acceptable performance of conservative parallel simulation. However, the availability of shared memory provides an opportunity to re-examine traditional approaches, with an eye towards increasing the degree of parallelism during the periods in which the simulation is not deadlocked. In this section we briefly survey some shared-memory optimizations that are applicable to any conservative parallel simulation.

4.1 Deadlock Detection and Recovery

In [21], we showed that deadlock detection and recovery can be made very efficient if it is built into a customized run-time kernel. The use of a centralized scheduler makes deadlock detection trivial: deadlock has occurred when all processors are idle and there are no LPs ready to run. When deadlock does occur, recovery is done in constant time, at the expense of a small cost (logarithmic in the number of LPs) to be paid each time an LP blocks.

4.2 Eager Blocking Avoidance

Fujimoto [5, 6] observed that, in a shared memory implementation, deadlock avoidance can be accomplished without explicitly sending null messages. This is because only the *most recently sent* null message on any given communication channel is important – it conveys at least as much information to the receiver as any of the earlier messages. Therefore, a streamlined mechanism can be employed in which the sending of a null message is implied by setting certain shared variables and then notifying the receiver.

The salient feature of this technique is that, each time a certain type of event occurs at an LP, that LP is responsible for notifying all of its outputs (i.e., "sending" null messages). The event in question is usually one of the following, arranged in decreasing order of frequency: a change in the LP's clock value; the sending of a message on any of the LP's output channels; or blocking of the LP. No matter which choice is used, the burden of synchronization is placed on the LP at which the event occurs. For this reason, we have chosen to call this method *eager blocking avoidance*: the LPs are "eager" to run the protocol.

A disadvantage of this protocol is that many of the notifications are unnecessary. For example, the LP doing the notification may not have advanced far enough to allow the LP being notified to proceed. Even if this is not the case, the LP being notified may still be awaiting notification of progress from other LPs.

4.3 Appointments

The *appointment protocol* was introduced by Nicol [14]. An appointment is a promise not to send a message before a certain time; thus, an appointment is semantically equivalent to a null message. The difference here is that the scheduling of appointments is *demand-driven*: when an LP is unable to receive a message because the timestamp of the message exceeds the appointment time of one or more of the LP's sources, the LP requests a new appointment from said sources. Thus, appointments are not made unnecessarily, which hopefully reduces the overhead of the protocol relative to eager blocking avoidance.

4.4 Lazy Blocking Avoidance

We introduced a technique called *lazy blocking avoidance* [21], which places the burden of deadlock avoidance on the run-time kernel. Whenever a physical processor find that there are no LPs on the ready queue, it chooses a blocked LP according to some strategy and checks to see if the LP is artificially blocked. (Recall that an LP is artificially blocked if it is awaiting a message from an LP whose clock has already exceeded the timestamp of the earliest pending message at the waiting LP.) This check is inexpensive because all clock values reside in shared memory. If the LP is artificially blocked, it is awakened and run on the processor that unblocked it.

Lazy blocking avoidance has the advantage of not causing any unnecessary wakeups. At the same time, the potential amount of delay experienced by a blocked LP that has useful work to do is minimized, since there are no context switches taking place until such an LP is identified. Finally, it has the property that the amount of computational effort devoted to avoiding unnecessary blocking increases with the number of blocked LPs! When there are no idle processors, it doesn't matter if LPs are artificially blocked, and no effort is made to unblock them; but as the number of blocked LPs increases, so will the number of idle processors, and hence the rate at which blocked LPs are examined.

Lazy blocking avoidance has been shown to outperform eager blocking avoidance when the degree of interconnection in the network is high. Also, the performance of lazy blocking avoidance seems to be less sensitive to the lookahead ratio of the service time distribution than does the performance of eager blocking avoidance [21].

4.5 Some Examples

In [21], we showed by means of an implementation that good speedups can be achieved for a wide variety of parallel simulation benchmarks if the availability of shared memory is aggressively exploited. As an example, Figure 3(a) compares the performance of an early version of our simulator, which used explicit null messages, to a more refined version employing the optimized eager blocking avoidance discussed in Section 4.2. This graph shows speedups for a 4x4 torus network. (Speedups are reported relative to a sequential implementation of the simulator.) Even this simple optimization yields a factor of two improvement in speedup.

Figure 3(b) shows the improvement over eager blocking avoidance that can be achieved using lazy blocking avoidance (Section 4.4) in some cases. This graph shows speedups for a 16 node fully interconnected network. Because of the high connectivity of this benchmark, it is a stress test for eager blocking avoidance.

5 QNM-Specific Optimization Techniques

Experience has shown that general-purpose conservative parallel simulators do not perform as well as simulators that are tailored to a particular application [5, 6, 14, 16, 17, 21]. As an extreme example, Lin, Baer, and Lazowska [11] showed that even a parallel simulator designed specifically for trace-driven simulation of multiprocessor cache coherence protocols could not perform as well as a simulator tailored to a *particular* cache coherence protocol. Thus, research in conservative parallel simulation needs to be directed towards techniques that are specific to particular problem domains.

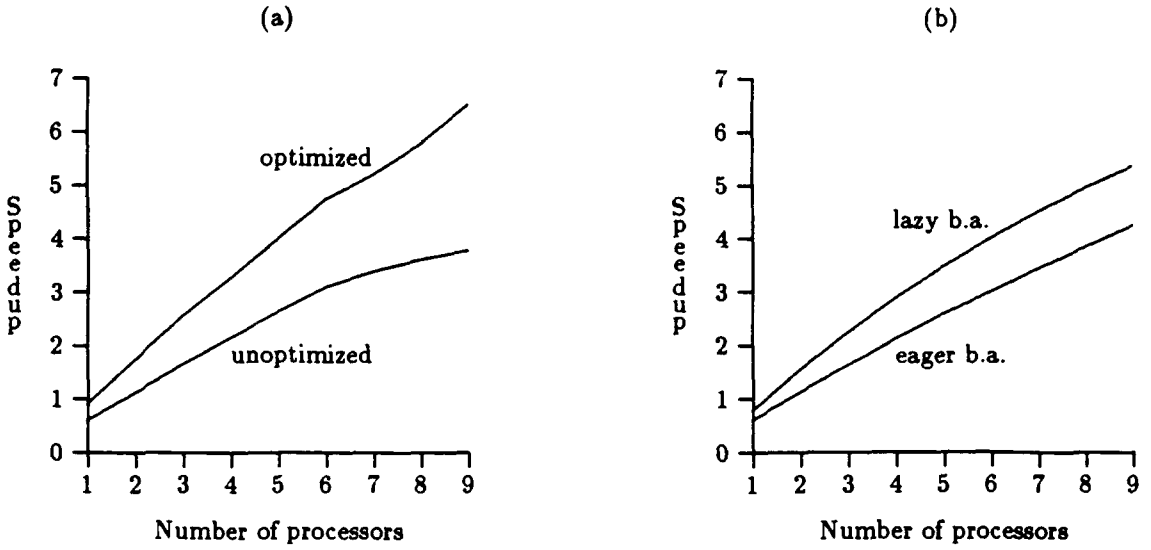


Figure 3: (a) Comparison of explicit null message-based eager blocking avoidance with shared-memory optimized eager blocking avoidance. (b) Comparison of optimized eager blocking avoidance with lazy blocking avoidance.

In this section we explore methods for calculating lookahead of LPs simulating QNM servers with various types of service disciplines. We begin by motivating the study of lookahead calculation, followed by a brief description of the seminal work in this area, Nicol's *future list* technique [14]. The remainder of the section is devoted to extensions of the future list technique to a wider class of service disciplines.

5.1 The Importance of Lookahead

Lookahead characterizes the ability of a logical process to predict future messages that it will send. More precisely, if an LP can predict at time t that it will send no message before time t', then the lookahead at time t is $L(t) = t' - t$. From the definition, it is easy to show that the function defined by $t + L(t)$ is monotonically non-decreasing, even though $L(t)$ may not be.

Fujimoto [4, 5] demonstrated the importance of lookahead by comparing two implementations of FCFS servers: an *eager server*, which sends a message as soon as its departure time is computed, and a *lazy server*, which doesn't send a message until all inbound messages with timestamps later than the computed departure have arrived. Figure 4 [4], which compares the performance of these two implementations on Reed's central server benchmark, convincingly demonstrates the importance of lookahead.

In general, Fujimoto's work demonstrated a strong correlation between the performance of conservative parallel simulation of QNMs and the lookahead ratio of the LPs in the simulation. (Recall that lookahead ratio is defined as the ratio of the mean of the LP's timestamp increment function (i.e., the server's service time distribution) to the LP's lookahead value.) Fujimoto found that, as lookahead ratio increased, performance suffered[3].

[3]Note that the lookahead ratio of the lazy server described in the preceding paragraph is infinite, since it has a lookahead of zero.

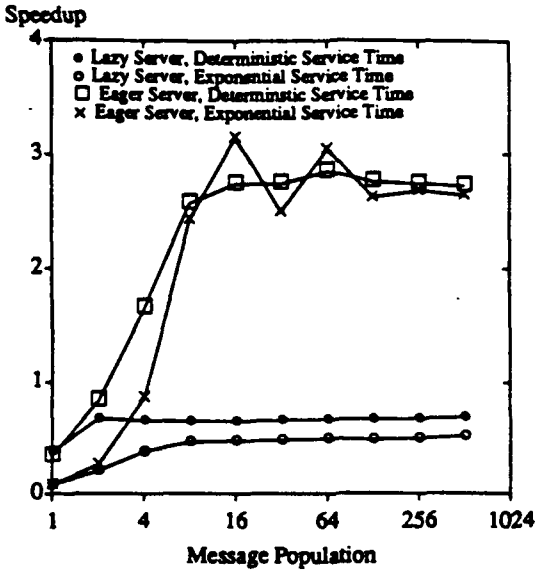


Figure 4: Speedup of the central server benchmark using lazy and eager servers. (Reproduced from [4] by permission of R.M. Fujimoto.)

This phenomenon is easy to understand. Intuitively, lookahead represents knowledge by an LP that it will not generate a message before a certain amount of (simulated) time has elapsed. For example, if LP_1 can send messages to LP_2, and LP_1's local clock value is 10 but its lookahead is 40, then LP_1 knows that it will not generate a message to LP_2 until at least time 50. If LP_2 is privvy to this information, it can use it to avoid unnecessary synchronization delays.

Although "bigger is better" where lookahead is concerned, large lookahead values do not in and of themselves guarantee good performance. The lookahead must be comparable to the expected amount of clock skew between neighboring LPs. Since LPs affect each others' clocks by exchanging messages, this implies that lookahead values need to be comparable to the average message timestamp increment in order to be useful. Thus, lookahead ratio, which is the ratio of the mean of the timestamp increment distribution to the mean of the lookahead, is a good predictor of performance.

Lookahead ratio is especially important when some form of deadlock avoidance is being used. This is because all deadlock avoidance protocols work by propagating clock values through the simulation, in the absence of any real messages. Each time the protocol allows some LP to advance its local clock, the LP recalculates its lookahead, adds the new lookahead value to its new clock value, and propagates this information to its outputs. If the simulation contains a cycle of blocked LPs, the protocol loops around this cycle until some LP's clock value exceeds the timestamp of a message it is waiting to receive. If lookahead ratio is extremely poor, the protocol will loop many times before any real message can be accepted for processing. In fact, unless every cycle contains at least one logical process with strictly positive lookahead, null message-based deadlock avoidance is not guaranteed to work [15]. For this reason, most implementations require that service time distributions have lower bounds that are strictly positive.

Lookahead ratio is also important to the performance of deadlock detection and recovery. Smaller lookahead ratio means fewer deadlocks, and when a deadlock does occur, allows more logical processes to be awakened, on the average, during recovery from the deadlock. This is because the deadlock recovery algorithm will be able to determine that a larger number of LPs cannot causally affect each other, and hence may be scheduled for execution in parallel.

The preceding remarks should help to motivate the exploration of lookahead calculation.

5.2 FCFS Servers: the Future List Technique

Typically, a logical process calculates lookahead based on messages that have already been processed by the LP. However, Nicol [14] pointed out that by pre-sampling an LP's service time and routing distributions, information about messages that have yet to arrive at the LP can be used in the lookahead calculation. The statistical integrity of the simulation is preserved by keeping these samples in a queue called the *future list.* Then, when a message is received by the LP, its service time and destination are taken from the head of the future list. Nicol's implementation showed excellent performance across a range of network topologies and service time distributions. Whereas previous work concentrated on methods for *propagating* lookahead values, Nicol was the first to treat the *computation* of lookahead as a separate concern.

If a customer is in service at an LP using the FCFS service discipline, then no departure can take place until said customer completes service. Therefore, the lookahead is the residual service time of the customer in service. If there are no customers in service, however, then lookahead is equal to a, the lower bound on the service time distribution (which may be zero).

On the other hand, if the future list technique is used, the service time distribution can be pre-sampled to determine the service time s of the *next customer to arrive.* In this case, the LP knows that no messages will be sent until at least time $t + s$, where t is the current local clock value of the LP. Hence, the LP's lookahead is s. By using this technique, there is no need for the service time distribution of the LP to be bounded away from zero. Moreover, even if such a bound existed, the future list would improve performance, since the minimum value of the service time distribution might be very small compared to its mean (i.e., the server might have a large lookahead ratio). Another way of putting this is that the future list improves performance because it yields an *expected lookahead ratio* of unity, regardless of the distribution being used [10].

For a FCFS server with only a single destination, the future list is a single entry, since the service time of the next job to arrive is the only factor that can affect lookahead. In order to extend this technique to servers with a choice of destinations, it is necessary to compute (*service time, destination*) pairs until the sampled destination matches the one for which a lookahead value is being computed. Once the future list contains at least one message for every possible destination, further pre-sampling is unnecessary.

In the remainder of this section we extend the future list technique to other common service disciplines, and we calculate the expected lookahead that results. We assume that there is only one possible destination for a message sent by the server being considered; it is straightforward to extend these techniques to servers with a choice of destinations.

5.3 Multiple class servers with priority scheduling

Suppose there are C customer classes in the QNM, numbered $1 \ldots C$. A priority scheduling server schedules customers FCFS within each class, satisfying the condition that all queued customers belonging to class i are served before any customers belonging to class j, if $i < j$. In addition, if the server is preemptive, a customer belonging to class j will be preempted from service if a customer belonging to class i, $i < j$, arrives while the class j customer is in service. (There is also a choice of whether to restart or resume the preempted customer when the server becomes available again; for our purposes, it does not matter.)

Suppose that a customer belonging to class j is in service at an MCP server, and its residual service time is r_j. If the server is non-preemptive, then clearly the next customer to depart from

the server will be the one in service, and hence the lookahead is r_j. On the other hand, if the server is preemptive, the lookahead is

$$L_{MCP}(j) = \min(\{r_j\} \cup \{a_i : i < j\}) \tag{1}$$

where a_i is the minimum service time for customers belonging to class i. To see this, note that the next customer to depart from the server might not be the one that is currently being served, since a higher priority customer could arrive in less than r_j time units. The worst case scenario, in terms of lookahead, would be if a higher priority customer, say, a class i customer, arrived immediately. Since the service time of such a customer is unknown, a departure could take place in as little as a_i time units. Thus, the lookahead, which must be a lower bound on the time until the next departure, is given by (1).

If there is no customer currently in service, then Equation (1) still holds if we define j and r_j to be $C + 1$ and ∞, respectively.

Lookahead for an MCP server can be improved by implementing a *future array* that is indexed by customer class. The future array always contains one sample from the service time distribution of each customer class. If we denote the contents of the future array as $s_1 \ldots s_C$, then lookahead is given by

$$L_{MCP}(j) = \min(\{r_j\} \cup \{s_i : i < j\})$$

where j, r_j are defined as before. Each time a job arrives, its service time is taken from the proper location in the future array and another sample is generated to take its place.

We will now calculate $E[L_{MCP}(j)]$ for the case where service times for every class are exponentially distributed. We shall make use of the following result from the theory of statistics [20, sec. 1.5.2]: if $X_1, X_2, \ldots, X_n$ are independent random variables that are exponentially distributed with respective parameters $\lambda_1, \lambda_2, \ldots, \lambda_n$, then the statistic $V = \min\{X_1, X_2, \ldots, X_n\}$ is exponentially distributed with parameter $(\lambda_1 + \lambda_2 + \ldots + \lambda_n)$. Thus,

$$E[\min\{X_1, X_2, \ldots, X_n\}] = \frac{1}{\lambda_1 + \lambda_2 + \ldots + \lambda_n} \tag{2}$$

Let the service time for class i be exponentially distributed with parameter λ_i. Note that, because of the memoryless property of the exponential distribution, the residual service time of the customer in service (if there is one) has the same distribution as the service time for that customer. Since the exponential distribution has a lower bound of zero, then without using the future array technique $E[L_{MCP}]$ is obviously zero. On the other hand, using the future array we obtain

$$E[L_{MCP}(j)] = \begin{cases} \frac{1}{\lambda_1 + \ldots + \lambda_j} & 1 \le j \le C \\ \frac{1}{\lambda_1 + \ldots + \lambda_C} & \text{otherwise (i.e., no customer in service)} \end{cases}$$

Note that the quantity we have calculated is the expected lookahead given that a class j customer is in service. In order to evaluate $E[L_{MCP}]$, the *unconditional* expected lookahead, we would need to know the probability that a customer of any given class (or no customer at all) is in service at a given time. This might be accomplished by introducing assumptions about the arrival distributions for each customer class.

5.4 Last-come first-served servers

If a server is LCFS without pre-emption then its lookahead is the same as for a FCFS server. Thus, we will concern ourselves only with preemptive LCFS servers[4].

Under LCFS, lookahead is equal to a, the minimum value of the service time distribution. The situation is not improved even by using the future list technique, since no matter how many samples are kept in the future list, there is always the possibility that more than this number of customers will arrive.

However, the future list can be of some utility if there is a population constraint N at the server. This is a fairly common situation in QNMs. For example, if the QNM is closed, there is a fixed customer population in the entire network, and that population can be used for N. Even if the QNM is not closed, it may be the case that the server is part of a subnetwork with a fixed population (e.g., a cpu in a memory-constrained system).

Suppose that there are n customers at the server with residual service times $r_1, \ldots, r_n$. Since the customer population at the server is bounded by N, there can be no more than $N - n$ arrivals before some customer will depart. Therefore, the future list needs to contain $N - n$ entries. Let s_i be the service time of the i-th "customer" in the future list. Then the lookahead is given by

$$L^N_{LCFS}(n) = \min\left(\{r_i : 1 \leq i \leq n\} \cup \{s_i : 1 \leq i \leq N - n\}\right)$$

For example, when service times are exponentially distributed with parameter λ, the residual service times are also exponentially distributed with parameter λ, and the expected value of $L^N_{LCFS}(n)$ is (applying 2)

$$E[L^N_{LCFS}(n)] = \frac{1}{N\lambda}.$$

Thus, expected lookahead ratio for a population-constrained LCFS server is N; note that this quantity is independent of the mean of the service time distribution.

5.5 Processor sharing servers

Processor sharing (PS) is an idealization of round robin (RR) scheduling. Under RR, each job receives a *quantum* of service before it must relinquish control to the next job in the queue, rejoining the queue at its tail. PS is defined as the limiting case of the RR algorithm as the quantum goes to zero, so that control of the processor circulates infinitely rapidly among all jobs. The effect is that jobs are served simultaneously, but each of the n jobs in service receives only $1/n$-th of the full power of the processor. PS often is appropriate to model cpu scheduling in systems where some form of RR scheduling actually is employed [9].

Suppose that, at local time t, a single customer is in service at a PS server, and that the customer's residual service time is r. Then it cannot be concluded that the next message sent by the server LP will be at time $t + r$. The reason for this is that another customer may arrive before time $t + r$, effectively "stealing service" from the original customer, and thus delaying its departure. Even worse, the newly arrived customer may have a service time so small that it will depart even *earlier* than time $t + r$.

As an example, suppose that the current local clock value is 100, and the single customer in service is scheduled to depart at time 150 ($r = 50$). If a customer arrives with a service demand

[4] As was the case for MCP servers, for our purposes it is irrelevant whether preempted jobs are resumed or restarted.

of only 10, then in the absence of further arrivals, the second customer will complete service (and thus cause a message to be sent) at time 120, and the original customer will complete service at time 160.

Although it may seem hopeless, lookahead can be computed for PS servers. If there are n customers in service with residual service times $r_1 \ldots r_n$, lookahead is given by

$$L_{PS}(n) = \min(\{nr_i : 1 \leq i \leq n\} \cup \{(n+1)a\}) \tag{3}$$

where a is a lower bound on the service time distribution of the LP. To prove this, there are two cases to consider:

Case 1. No new customers arrive before one of the current customers completes service. Then the next customer to depart will be the one with residual service time $\min\{r_i : 1 \leq i \leq n\}$. Since it must share the processor with $n-1$ other customers, it will require $n \cdot \min\{r_i\} = \min\{nr_i\}$ time units to complete service.

Case 2. A new customer arrives before any departure takes place. In the worst case, the new customer will arrive immediately. If the new customer's service time is less than the residual service time of any of the customers already in service, then the new customer will be the next to depart. Since there are now a total of $(n+1)$ customers receiving service, The new customer will require at least $(n+1)a$ time units to complete service.

Therefore, the minimum time that must elapse until any customer could finish service, which is the lookahead of the server, is given by Equation (3).

The future list technique can improve the lookahead of a PS server, if there is either (a) a population constraint at the server, or (b) a lower bound on the service time distribution of the server. As discussed in the previous section, case (a) is fairly common in QNMs.

We consider this case first, and we denote the population constraint by N. As before, if no customers arrive before any current customer completes service, the next departure will take place in $\min\{nr_i : 1 \leq i \leq n\}$ time units. Suppose now that one or more arrivals occur before any such departure. The worst case scenario would be if the arrivals occurred immediately. Let s_i be the service time of the i-th "customer" in the future list. If a single customer were to arrive, then it could depart in no less than $(n+1)s_1$ time units, since it would have to share the processor with n other customers. Similarly, if two customers were to arrive, the earliest that either of them could depart is $\min\{(n+2)s_1, (n+2)s_2\}$. Thus, if either one or two customers should arrive, the earliest departure time for either of the new arrivals is $\min\{(n+1)s_1, (n+2)s_1, (n+2)s_2\} = \min\{(n+1)s_1, (n+2)s_2\}$.

In general, if m customers should arrive, the soonest that any of the new arrivals could depart would be in $\min\{(n+1)s_1, (n+2)s_2, \ldots, (n+m)s_m\}$ time units. Since there can be at most N customers in service simultaneously, no more than $N-n$ customers can arrive before a departure must take place. Therefore, the time until the earliest possible departure, i.e., the lookahead, is

$$L^N_{PS}(n) = \min(\{nr_i : 1 \leq i \leq n\} \cup \{(n+i)s_i : 1 \leq i \leq N-n\}) \tag{4}$$

Next we consider the case in which the customer population is unbounded, but there is a minimum service time of a. In this case, the number of samples to include in the future list calculation, which we will call m, is a free variable. By combining the reasoning that led to (3) and (4), we obtain the following expression for lookahead as a function of n and m:

$$L^\infty_{PS}(n,m) = \min(\{nr_i : 1 \leq i \leq n\} \cup \{(n+i)s_i : 1 \leq i \leq m\} \cup \{(n+m+1)a\}) \tag{5}$$

Note that when $m = 0$, (5) reduces to (3). The first term in the minimum represents the n customers already in service; the second term represents the next m customers that might arrive, whose service times have been pre-sampled; and the last term represents the arrival of an $m+1$-st customer with an arbitrarily small service time.

In general, calculating the expected lookahead for a PS server is very difficult. One exception is if the service times at the server are exponentially distributed. Let the parameter of the service time distribution be λ. The following result is proved in Appendix A:

$$E[L_{PS}^N(n)] = \begin{cases} \frac{1/\lambda}{\sum_{i=1}^{N}(1/i)} & \text{if } n = 0 \\ \frac{1/\lambda}{1+\sum_{i=n+1}^{N}(1/i)} & \text{if } 1 \le n < N \\ 1/\lambda & \text{if } n = N \end{cases}$$

In order to calculate the expected lookahead independent of n, we would need to know the distribution of the number of customers in service at any given time, which would require us to make additional assumptions. The important thing to note is that, for a given n and N, $E[L_{PS}^N(n)]$ is a constant times $1/\lambda$, the mean of the service time distribution. Thus, $E[L]$, which is a linear combination of $E[L_{PS}^N(n)]$, $0 \le n \le N$, will also be linear in $1/\lambda$. Therefore, expected lookahead ratio $= (1/\lambda)/E[L]$ will be *independent of the mean of the service time distribution.*

It is interesting to examine the behavior of $E[L_{PS}^N(n)]$ as N gets large. The summation in the denominator is a subsequence of the *harmonic series* $\sum_{i=1}^{\infty}(1/i)$, which diverges. Thus, for $0 \le n < N$,

$$\lim_{N\to\infty} E[L_{PS}^N(n)] = 0.$$

As lookahead decreases, lookahead ratio increases, and performance is expected to suffer. However, it is a well known fact that the partial sums of the harmonic series grow logarithmically; hence, $E[L_{PS}^N(n)]$ decreases to zero very slowly. Thus, even for "large" customer populations $L_{PS}^N(n)$ is expected to be substantially better than having no lookahead at all. Figure 5, which shows $L_{PS}^N(n)$ as a function of N for various values of n, clearly illustrates this behavior.

Given that expected lookahead has no positive lower bound as N increases and n remains fixed, it seems counter-intuitive that it should be a constant equal the mean of the service time distribution when $n = N$. The explanation of this fact is as follows. As $n = N$ increases without bound, the minimum of the residual service times will tend towards zero. However, because the number of customers in service is becoming infinite, each customer only gets an infinitesimal share of the server's attention. Thus, the expected time until the next departure, which is the ratio of these two quantities, is finite and non-zero.

Finally, we would like to point out another counter-intuitive fact: there do exist service time distributions for which

$$\lim_{N\to\infty} E[L_{PS}^N(n)] > 0 \tag{6}$$

for all values of N. We claim that the Erlang-k distributions [1, sec. 3.2.5] for $k > 1$ have this property. (The Erlang-1 distribution is the same as the exponential distribution, and so obviously does not have property (6).) The proof of this claim is in Appendix B.

Although we were unable to obtain a closed-form solution for $E[L_{PS}^N(n)]$ when service times have Erlang-k distributions, we did use Monte Carlo simulation to estimate this quantity. Figure 6

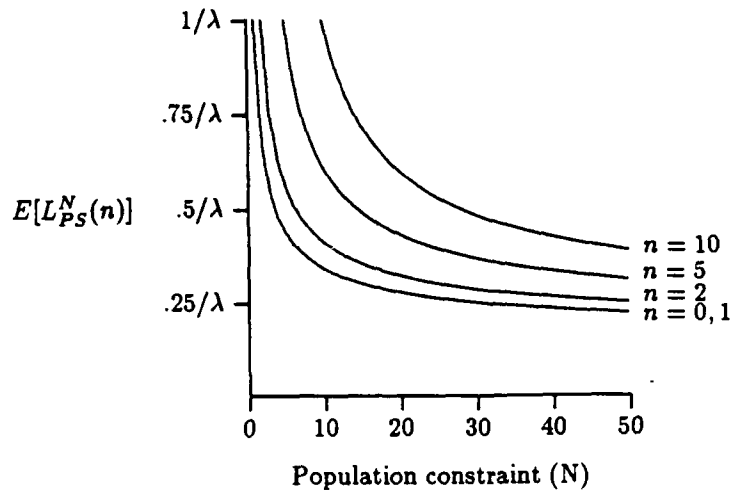


Figure 5: Expected lookahead as a function of population constraint for a processor sharing server with exponential service times.

shows $\overline{L}^N_{PS}(0)$ (average lookahead when the server is idle) as a function of population constraint, for service times that have the exponential, Erlang-2, and Erlang-3 distributions.

5.6 Applications to Model Partitioning

As shown in the previous sections, expected lookahead is improved for service disciplines such as LCFS with preemption and PS if there is a population constraint at the server. This provides a motivation for the identification of redundant (w.r.t. performance) servers as outlined in Section 3. By coalescing such servers into a single LP, their expected lookahead can be improved. The reason for this is quite simple: when calculating the lookahead for a particular server S, the LP knows that customers that are currently in service at the other servers it is simulating cannot arrive at S any sooner than the time at which they depart from those servers. It is straightforward to modify the lookahead calculation to take this into account.

5.7 Theory vs. Practice

We conclude this section with some pragmatic considerations and a cautionary note.

Lookahead for LCFS and PS service disciplines, which takes into account every customer that might possibly arrive, is not necessarily expensive to calculate. The future list can be thought of as a pipeline: once it has been filled (by sampling the service time distribution N times), the service time distribution is then sampled only once per departure from the server. The dominating cost is the calculation of the minimum over all of the customers at the server (both those already in service and those represented by entries in the future list). If the population constraint is not too large, a naive approach should work well. If this is not the case, more efficient procedures will be required. We are currently refining algorithms for these calculations.

Finally, a caveat: it is important to understand that the expected lookahead ($E[L]$) must not be used as a lookahead value! At any given moment, the computed lookahead may be smaller or larger than $E[L]$. Using too large a value for lookahead can lead to incorrect results.

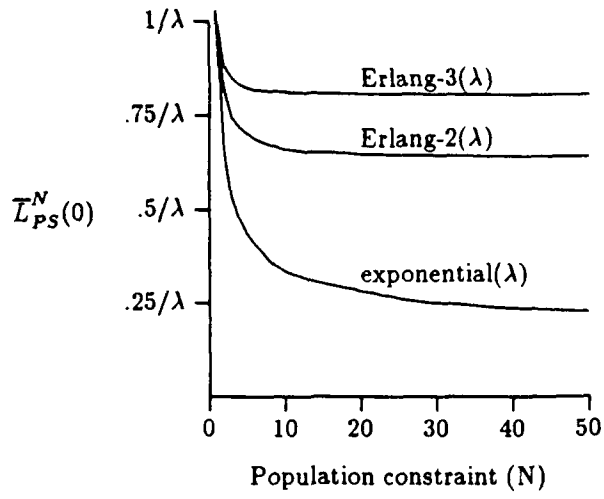


Figure 6: Average lookahead as a function of population constraint for an idle ($n = 0$) processor sharing server with various service time distributions.

6 Conclusions

We believe that conservative parallel simulation can be effective for queueing network models if both the presence of shared memory and the semantics of the QNM are aggressively exploited. In particular, we have shown the following:

- It is possible to determine *a priori* that some QNMs contain inherent limitations to parallel simulation speedup. We have presented a simple analytic technique for computing the *speedup potential* of a QNM, and have used this technique to identify some common QNM features that may limit performance.
- There are a number of performance-enhancing implementation techniques that are generally applicable to any shared-memory conservative simulation implementation.
- No implementation can expect to do a good job on all problems; thus, specific knowledge about the problem being simulated should be exploited. An example of this is the future list technique, which is specifically applicable to QNMs.
- It is possible to extend the future list technique to improve the expected lookahead of servers with non-FCFS service disciplines, and thus improve the speedup of a parallel simulation of such servers. In particular:
  - Lookahead for MCP servers can always be improved.
  - Lookahead for LCFS servers can be improved if there is a population constraint at the server.
  - Lookahead for PS servers can be improved if either (a) there is a population constraint at the server or (b) there is a non-zero lower bound on service times.
- Under the assumption that service times are exponentially distributed, we have obtained the following results:

  - The expected lookahead ratio for an MCP server is proportional to the number of customer classes served by the server.
  - The expected lookahead ratio for a LCFS server with population constraint N is proportional to N.
  - The expected lookahead ratio for a PS server with population constraint N is proportional to $\log N$.

- Under the assumption that service times are Erlang-k distributed, $k > 1$, the expected lookahead ratio for a PS server with population constraint N is bounded by a constant independent of N.

- The fact that lookahead for certain service disciplines is improved when the customer population at the server is constrained implies that it may be useful to simulate more than one server with a single logical process. This is especially true if the servers in question have been identified (using the techniques from Section 3) as contributing little or nothing to the speedup of the simulation.

Acknowledgments

We would like to thank Jason Lin and Richard Anderson for their useful technical comments. We also extend our appreciation to Richard Fujimoto for allowing us to reproduce some of his results in Figure 4.

A Expected Lookahead for PS Servers with Exponential Service Times

Recall that lookahead at a population-constrained PS server is given by (Equation (4)):

$$L^N_{PS}(n) = \min(\{nr_i : 1 \leq i \leq n\} \cup \{(n+i)s_i : 1 \leq i \leq N-n\})$$

where n is the number of customers in service, N is the population constraint, r_i is the residual service time of the i-th customer in service, and s_i is the i-th entry in the future list (i.e., the service time of the i-th next customer to arrive).

Suppose the service time distribution is exponential with parameter λ. Define new random variables $u_i = nr_i$ and $v_i = (n+i)s_i$. Then each random variable u_i is exponentially distributed (because of the memoryless property) with mean n/λ, and each random variable v_i is exponentially distributed with mean $(n+i)/\lambda$. Applying Equation (2), we obtain

$$\begin{aligned} E[L^N_{PS}(n)] &= E[\min(\{u_i : 1 \leq i \leq n\} \cup \{v_i : 1 \leq i \leq N-n\})] \\ &= \frac{1}{\sum_{i=1}^{n}(\lambda/n) + \sum_{i=n+1}^{N}(} \end{aligned}$$

$$= \begin{cases} \frac{1/\lambda}{\sum_{i=1}^{N}(1/i)} & \text{if } n = 0 \\ \frac{1/\lambda}{1+\sum_{i=n+1}^{N}(1/i)} & \text{if } 1 \le n < N \\ 1/\lambda & \text{if } n = N \end{cases}$$

which is the desired result.

B Expected Lookahead for PS Servers with Erlang-k Service Times

The Erlang-k probability distribution is given by [1]

$$F_k(x) = 1 - e^{-k\lambda x}\left[1 + \frac{k\lambda x}{1!} + \frac{(k\lambda x)^2}{2!} + \cdots + \frac{(k\lambda x)^{k-1}}{(k-1)!}\right] \tag{7}$$

Our goal in this section is to prove the following theorem:

Theorem B.1 *Suppose that service times at a PS server are distributed according to the Erlang-k probability distribution for some $k \ge 2$. Then for any $n \ge 0$,*

$$\lim_{N\to\infty} E[L_{PS}^{N}(n)] > 0$$

To prove this, we need the following two lemmas.

Lemma B.1 *If $X_1, X_2, \ldots, X_m$ are i.i.d. Erlang-2 random variables with parameter λ, n is a non-negative integer, and c is a real number such that $0 < c \le (n+1)/2\lambda$, then*

$$\lim_{m\to\infty} \Pr(\min\{jX_{j-n} : n+1 \le j \le n+m\} > c) > 0 \tag{8}$$

Proof.

$$\begin{aligned} P &= \Pr(\min\{jX_{j-n} : n+1 \le j \le n+m\} > c) \\ &= \Pr\left[\bigwedge_{j=n+1}^{n+m}(jX_{j-n} > c)\right] \\ &= \prod_{j=n+1}^{n+m}\Pr(jX_{j-n} > c) \\ &= \prod_{j=n+1}^{n+m}\Pr(X_{j-n} > c/j) \\ &= \prod_{j=n+1}^{n+m}(1 - F(c/j)) \\ &= \prod_{j=n+1}^{n+m} e^{-2\lambda c/j}(1 + 2\lambda c/j) \end{aligned}$$

Taking the natural logarithm of both sides of this expression,

$$\begin{aligned}\log P &= \sum_{j=n+1}^{n+m} \log\left[e^{-2\lambda c/j}(1+2\lambda c/j)\right] \\ &= \sum_{j=n+1}^{n+m} [\log(1+2\lambda c/j) - 2\lambda c/j]\end{aligned}$$

Note that since $j \geq n+1$ and $c \leq (n+1)/2\lambda$, $2\lambda c/j \leq 1$. Making use the fact that $\log(1+x) \geq x - x^2/2$ for $0 \leq x \leq 1$, we have

$$\begin{aligned}\log P &\geq \sum_{j=n+1}^{n+m} \left[\left(2\lambda c/j - \frac{(2\lambda c/j)^2}{2}\right) - 2\lambda c/j\right] \\ &= -\frac{1}{2}\sum_{j=n+1}^{n+m}\left(\frac{2\lambda c}{j}\right)^2\end{aligned}$$

Therefore,

$$\lim_{m\to\infty} \log P \geq -\frac{1}{2}\sum_{j=n+1}^{\infty}\left(\frac{2\lambda c}{j}\right)^2$$

Since $2\lambda c/j < 1$, this infinite series converges, and thus

$$\lim_{N\to\infty} \log P > -\infty$$

$$\lim_{N\to\infty} P > 0$$

Lemma B.2 *For any $k \geq 2$, if $X_1, X_2, \ldots, X_m$ are i.i.d. Erlang-k random variables, Equation (8) holds.*

Proof. Let $Y_1, Y_2, \ldots, Y_m$ be i.i.d. Erlang-2 random variables. Let $F_X(X_i)$ and $F_Y(Y_i)$ be the distribution functions of the X_i and Y_i, respectively. From (7), $1 - F_X(X_i) \geq 1 - F_Y(Y_i)$, i.e., for any $c \geq 0$,

$$\Pr(X_i > c) \geq Pr(Y_i > c)$$

The result follows directly from the proof of Lemma B.1.

Proof of Theorem B.1. Let $R_1, \ldots, R_n$ be the random variables representing the residual service times of the n customers in service and let $S_1, \ldots, S_m$ be the random variables representing the $m = N - n$ entries in the future list. Then for $c > 0$,

$$\begin{aligned}\Pr\left[L_{PS}^N(n) > c\right] &= \Pr[\min(\{nR_i : 1 \leq i \leq n\} \cup \{(n+i)S_i : 1 \leq i \leq N-n\}) > c] \\ &= \Pr[(\min\{nR_i : 1 \leq i \leq n\} > c) \wedge (\min\{(n+i)S_i : 1 \leq i \leq N-n\} > c)] \\ &= \Pr[\min\{nR_i : 1 \leq i \leq n\} > c] \cdot \Pr[\min\{(n+i)S_i : 1 \leq i \leq N-n\} > c]\end{aligned}$$

Therefore, since n is fixed,

$$\lim_{N\to\infty} \Pr\left[L_{PS}^N(n) > c\right] =$$
$$\Pr[\min\{nR_i : 1 \le i \le n\} > c] \cdot \lim_{N\to\infty} \Pr[\min\{(n+i)S_i : 1 \le i \le N-n\} > c] \qquad (9)$$

Since residual service times cannot be identically zero, $\Pr[\min\{nR_i : 1 \le i \le n\} > c]$ is the product of a finite number of non-zero terms, and hence is non-zero. Furthermore, if $c \le (n+1)/2\lambda$, the limit on the right side of Equation (9) is non-zero by Lemma B.1. Therefore,

$$\lim_{N\to\infty} \Pr\left[L_{PS}^N(n) > c\right] > 0$$

which implies that

$$E[L_{PS}^N(n)] > 0$$

References

[1] A.O. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications.* Academic Press, New York, NY, 1978.

[2] R.E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT,LCS,TR-188, Massachussetts Insitute of Technology, Cambridge, Mass., 1977.

[3] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[4] R.M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. Technical Report UUCS-87-026a, Department of Computer Science, University of Utah, November 1987.

[5] R.M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. In *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1988.

[6] R.M. Fujimoto. Performance Measurements of Distributed Simulation Strategies. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 14–20, San Diego, CA, February 1988. Society for Computer Simulation International.

[7] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Time Warp Operating System. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93, Austin, TX, November 1987. ACM.

[8] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[9] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[10] Y-B. Lin, August 1988. Personal communication.

[11] Y-B. Lin, J-L. Baer, and E.D. Lazowska. Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols. Technical Report 88-03-02, Department of Computer Science, University of Washington, March 1988. Revised August 1988.

[12] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–60, March 1986.

[13] R.R. Muntz and J.W. Wong. Asymptotic Properties of Closed Queueing Network Models. In *Proc. 8th Princeton Conference on Information Sciences and Systems*, 1974.

[14] D.M. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. In *Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[15] J.K. Peacock, J.W. Wong, and E.G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, March 1979.

[16] D.A. Reed and A.D. Malony. Parallel Discrete Event Simulation: the Chandy-Misra Approach. In *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 8–13, San Diego, CA, February 1988. Society for Computer Simulation International.

[17] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[18] P.F. Reynolds. A Shared Resource Algorithm for Distributed Simulation. In *Proc. 9th International Symposium on Computer Architecture*, pages 259–266, Austin, TX, 1982. IEEE.

[19] C.H. Sauer, E.A. MacNair, and S. Salza. A Language for Extended Queueing Networks. *IBM Journal of Research and Development*, 24(6):747–755, November 1980.

[20] H.M. Taylor and S. Karlin. *An Introduction to Stochastic Modeling.* Academic Press, Orlando, FL, 1984.

[21] D.B. Wagner, E.D. Lazowska, and B.N. Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. Technical Report 88-04-05, Department of Computer Science, University of Washington, August 1988.

Conservative Parallel Discrete-Event Simulation: Principles and Practice

by David Barry Wagner

Simulation is one of the most important computational technologies in use today. Unfortunately, its importance is matched by its appetite for computational resources. These factors make parallel simulation a topic with far-reaching consequences in all fields of science and engineering. This thesis is concerned with one approach to this problem, *conservative loose event-driven parallel simulation*, the objective of which is to apply multiple processors to a single simulation run in an effort to reduce its time-to-completion.

There are several factors that make parallel simulation difficult. First, the fact that a physical system has a high degree of concurrency does not necessarily mean that a simulation of that system will benefit from parallelism. We introduce two simple analytic techniques that can be used to bound from above the speedup potential of parallel simulations.

Second, a parallel simulation requires synchronization to ensure that the results obtained are equivalent to those of a sequential simulation of the problem. We argue that

the availability of inexpensive, medium-scale shared-memory multiprocessors mandates a re-examination of synchronization algorithms for conservative loose event-driven parallel simulation. Our investigations lead to a novel synchronization technique called *lazy blocking avoidance*. Our measurements show that lazy blocking avoidance performs at least as well as, and often substantially better than, two other synchronization methods that have been widely discussed in the literature (deadlock detection and recovery, and eager blocking avoidance).

Third, experience has shown that subtle changes in modeling techniques can lead to dramatic differences in performance. We argue that the semantics of the specific simulation problem can and should be exploited to improve performance, and we present several new techniques for doing so. We show analytically that these techniques improve *expected lookahead*, which has been shown to be strongly correlated with performance, and we validate these predictions with measurement data from our parallel simulator. We demonstrate the real-world applicability of our techniques by applying them to a case study taken from the literature.

We have implemented all of the techniques presented in this dissertation in a prototype conservative parallel simulator called Synapse. Synapse is useful both as a simulation toolkit and as a laboratory for comparing the performance of various synchronization mechanisms. Synapse is able to achieve significant speedups on a wide variety of simulation problems.

Table of Contents

List of Figures vi

List of Tables x

Chapter 1: Introduction 1

1.1 Problem Motivation 1

1.1.1 Discrete-Event Simulation is Important 2

1.1.2 Discrete-Event Simulation Has Large Computational Requirements 3

1.1.3 Discrete-Event Simulation is Notoriously Difficult to Parallelize . . 5

1.2 Parallelizing Individual Simulation Runs 6

1.3 Thesis Statement 8

1.4 Overview of the Dissertation 9

Chapter 2: Parallel Discrete-Event Simulation 11

2.1 Sequential Discrete-Event Simulation 11

2.2 Model Function Distribution 13

2.3 Loose Event-Driven Parallel Simulation 14

2.4 Conservative Loose Event-Driven Parallel Simulation 16

2.5 Deadlock in Conservative Loose Event-Driven Parallel Simulation 19

2.6 Other Model Function Distribution Strategies 24

2.6.1 Optimistic Loose Event-Driven Parallel Simulation 24

2.6.2 Conservative Phased Event-Driven Simulation 27

2.6.3 Simulation Space-Time 29
2.7 Alternatives to Model Function Distribution 30
2.7.1 Application-Level Distribution 30
2.7.2 Support Function Distribution 36
2.8 Chapter Summary 36

Chapter 3: Predicting Performance 38
3.1 Previous Work 39
3.2 Asymptotic Bound Analysis 41
3.3 A Tighter Bound on LP Utilization 47
3.4 Chapter Summary 52

Chapter 4: Exploiting Shared Memory 54
4.1 Shared Memory vs. Distributed Parallel Simulation 54
4.2 A New LP Algorithm for Shared Memory 56
4.3 Shared-Memory Blocking Avoidance Techniques 59
4.3.1 Eager Blocking Avoidance 59
4.3.2 SRADS 60
4.3.3 Lazy Blocking Avoidance 61
4.4 Efficient Deadlock Detection and Recovery 62
4.5 Summary of Techniques 64
4.6 Performance of the Algorithms 64
4.6.1 The Hardware Base 64
4.6.2 The Sequential Simulator 65
4.6.3 The Benchmark 66
4.6.4 Results 68
4.6.5 Conclusions 78
4.7 Chapter Summary 79

Chapter 5: Exploiting Model Semantics 81
5.1 Lookahead 81
5.2 First-Come-First-Served Servers 85
5.3 Extensions of the Future List Technique 92
5.3.1 Multiple Class Preemptive Priority Servers 92
5.3.2 Processor Sharing Servers 95
5.3.3 FCFS Multiple-Server Centers 104
5.4 Chapter Summary 106

Chapter 6: System and Language Support 108
6.1 Presto 109
6.2 Synapse 110
6.2.1 The LogicalProcess and Message Classes 110
6.2.2 The SimScheduler Class 111
6.2.3 Performance Tuning 114
6.2.4 Overall Synapse Structure 116
6.3 Chapter Summary 121

Chapter 7: Case Study 123
7.1 Description of the Problem 123
7.2 The No-Feedback Model 125
7.3 The Feedback Model 127
7.4 Chapter Summary 130

Chapter 8: Conclusions and Future Directions 132

Bibliography 138

Appendix A: The Tradeoff Between Inter- and Intra-Replication Parallelism for Stochastic Simulations 151

Appendix B: Expected Lookahead for PS Servers with Erlang-k Service Times . . 154

Acknowledgements

I would like to thank the members of my supervisory committee, Ed Lazowska, John Zahorjan, Richard Anderson, and Jean-Loup Baer, for all their help and encouragement. Their feedback greatly improved the quality of this dissertation. Special thanks are due to my advisor, Ed Lazowska, for taking an interest in me at a critical point in my graduate studies, and for the countless hours he has spent rescuing me from situations that I talked my way into.

I'd also like to thank Jerre Noe, under whose guidance I first cut my teeth on computer science research.

My life in the graduate student "trenches" was made much more enjoyable by having a classmate like Kevin Jeffay down there with me. Kevin always knew exactly what I was going through, because he was usually going through it at the very same time! His excellent sense of humor was always welcome; I borrowed from it frequently.

I am also indebted to the gang in 429 Sieg, who never stopped encouraging me to graduate, who were always there with a solution when I didn't know what to do next, and who gave me an appropriate send-off.

Cecelia Buchanan, Michael Cohen, and Mike Schwartz each got me jobs when I needed them.

A largely unsung heroine in our organization is Margie Ramsdell, who was always available to help on a moment's notice, and whose ability to handle any administrative emergency with aplomb never ceased to amaze me.

Last but not least, I would like to thank the Hewlett-Packard Corporation for supporting me under their Faculty Development Program, and the Microsoft Corporation for awarding me an ARCS Fellowship.

Chapter 1

Introduction

This dissertation concerns the use of parallel computing for discrete-event simulation. In this introductory chapter we will motivate the problem, describe some traditional approaches to it, state our thesis, and give an overview of the remainder of the dissertation.

1.1 Problem Motivation

Why is the use of parallel computing for discrete-event simulation an important and interesting problem to study? In this section we will argue the following points:

- Discrete-event simulation is an important, general purpose modeling technique.
- It frequently has daunting computational requirements.
- It appears to be a natural candidate for parallel processing, but attempts at this have been generally disappointing.

These points have made parallel discrete-event simulation the subject of intense research interest.

1.1.1 Discrete-Event Simulation is Important

Simulation is a general technique for studying the behavior of a system by exploring a *model* of that system. We shall call the system being simulated the *physical system*, and the model employed by the simulation the *logical system*. Like other modeling techniques, simulation is widely used in science and engineering because it enables practitioners to study the behavior of systems for which direct experimentation would be impractical, dangerous, or even impossible. Its generality allows the modeling of systems that are too complex for analytic techniques. Simulation is one of the most important computational technologies in use today.

There are two broad classes of simulation models, *static* and *dynamic*. A static simulation model is a representation of a system at a particular time, whereas a dynamic simulation model is a representation of a system as it evolves over time [Law & Kelton 82].

Monte Carlo methods are traditionally used to estimate the probabilities of a static model's states through sample-driven experimentation. Monte Carlo simulation makes no causal assumptions about a model; only statistical correlations between input and output values are explored [Kreutzer 86].

Simulations of dynamic models rely on some sequencing variable to measure progress. The successive values taken on by the sequencing variable represent the passage of time in the physical system; thus, the sequencing variable is called the *simulation clock*. Dynamic simulations can generally be broken down into two types, *continuous* and *discrete-event*, depending on how the model changes state over time.

Continuous simulation concerns the modeling of a system by a representation in which the state variables change continuously over time [Law & Kelton 82]. Continuous models typically involve difference equations that give relationships for the rates of change of the state variables with respect to time. Continuous simulation finds a wide range of applications in fields such as engineering physics (fluid flow, heat transfer), ecology (predator/prey models, epidemic models), economics (market analysis), and the social

sciences (demographic models) [Braun 75, Naylor 71, Roberts et al. 83].

Discrete-event simulation is used to model the behavior of systems in which phenomena of interest change value or state only at discrete, irregular points in time [Fishman 78]. In this type of simulation, the simulation clock jumps from one cluster of relevant state transitions, called an *event*, to the next.[1] Examples of such systems occur in the fields of industrial engineering (manufacturing and inventory control), operations research (scheduling, resource management), military science (weapons systems effectiveness, wargames), traffic analysis (vehicular as well as communications), and of course, computer science (system performance modeling, logic circuit functional verification) [Ferrari 78, Hillier & Lieberman 86, Poole & Szymankiewicz 77, Reitman 81, Sauer & Chandy 81, Thesen 78].

A detailed discussion of the major simulation modeling paradigms can be found in [Kreutzer 86, Law & Kelton 82]. The remainder of this dissertation is concerned solely with discrete-event simulation.

1.1.2 Discrete-Event Simulation Has Large Computational Requirements

Discrete event simulation is one of the most computationally intensive tasks to which computers are applied. This is due mainly to two factors:

- Individual simulation runs can take a long time.
- Simulation is often a "subroutine" of some larger modeling process. Thus, many individual simulation runs may be necessary in order to arrive at a result.

A good example of an application for which even a single simulation run is expensive is the trace-driven simulation of cache memory behavior in computer systems. Since there is no widely accepted model of program memory reference behavior, cache simulations

[1] An alternative time-advance mechanism, *fixed-increment time advance* [Law & Kelton 82], is a special case of the *next-event time advance* mechanism discussed here.

are usually driven by traces of actual program memory references. The granularity of time in these traces is extremely small (on the order of a single instruction) and the traces need to represent a reasonably long interval of time (at least several seconds of program execution) in order to give a representative picture of program behavior [Eggers et al. 89]. This gives rise to enormous trace files, requiring large amounts of computer time to simulate.

Other factors contribute to the length of individual simulation runs. For example, many simulations rely on random processes to describe behaviors that are too detailed (or not well enough understood) to model exactly. This implies that the sequence of output values of the simulation is a stochastic process. Therefore, it is necessary to make each simulation run long enough so that the outputs of the simulation are representative of the steady-state behavior of the system [Heidelberger & Welch 83, Kelton & Law 84, Law 83].

The stochastic nature of many simulations has another ramification: in order to assess the statistical uncertainty of the results, it is standard practice to use multiple *independent replications* (runs of the simulation with different random number seeds) to derive confidence intervals. Thus, stochastic simulations not only require long individual simulation runs, but also may require multiple runs for statistical reasons.[2]

An example of a process that uses simulation as a "subroutine" is VLSI circuit validation. In a typical VLSI design cycle, simulation is used to validate changes to the circuit as they are made; it is considered unacceptably risky to fabricate a chip without simulating it first. As chip designs increase in complexity, simulation is becoming a bottleneck for the entire design process [Perry 89].

Another example of the simulation-as-subroutine paradigm is that simulation is frequently used to analyze the sensitivity of a system to changes in certain parameters. In this case, many runs of the simulation may be required, each with slightly differ-

[2]Actually, the need for multiple independent replications of a stochastic simulation is an argument *against* parallelizing individual simulation runs [Heidelberger 86]. An analysis of the tradeoffs involved is contained in Section 2.7.1.

ent parameter settings. While it might be possible to speed up this process by simply running many sequential simulations simultaneously, it is often the case that the parameter settings for run n depend upon the results of run $n-1$. In circumstances such as these, it is more efficient to "navigate" through the parameter space rather than take a "scattershot" approach.

The overall message is that applying parallelism to individual simulation runs is an important approach to dealing with the large computational requirements of discrete-event simulation.

1.1.3 Discrete-Event Simulation is Notoriously Difficult to Parallelize

Discrete-event simulation seems to be an ideal candidate for parallel processing because the systems being simulated typically contain a great deal of concurrent activity. For example, a computer system may have several active entities, such as CPUs, disks, channels, and terminals, that all operate concurrently. However, attempts to realize this potential for parallelism have met with only mixed success.

Early attempts at parallel simulation were conducted in a distributed system setting, and were successful on only a limited class of models [Seethalakshmi 79]. The advent of low-cost, shared memory multiprocessor computers sparked a renewed interest in the problem, in the hopes that removing the overhead of distribution would improve performance. Experiments by Reed et al. showed that communication overhead was not the only factor responsible for poor performance [Reed & Malony 88, Reed et al. 88]. It has become increasingly clear that, even in the absence of communication overhead, there are at least three important factors that make achieving good speedup difficult.

First, it turns out that the degree of concurrency in the system being modeled is not always a good indicator of the speedup potential of a simulation of that system. This is because there may be little relationship between the real time needed to process an event in the simulation and the amount of simulated time represented by this event; therefore, there may be bottlenecks in the simulation that are not present in the physical system.

Second, it is necessary to identify events as being *causally independent* before they can be executed in parallel, or else the parallel simulation may not produce the same results as a sequential simulation. What this means in practice is that some synchronization mechanism is required to keep processors from executing events too soon. This synchronization turns out to be a major performance impediment.

Third, experience with a variety of problems has shown that performance is highly sensitive to the specific model under consideration, and that subtle changes in modeling techniques can lead to dramatic differences in performance [Fujimoto 88a, Fujimoto 88b].

We will address each of these problems in this dissertation.

1.2 Parallelizing Individual Simulation Runs

We pointed out in the previous section that not only is simulation important and computationally intensive, but that in many cases it is necessary to parallelize individual simulation runs in order to reduce the time-to-completion of the overall modeling process. In this section we outline the dominant approaches to this problem; a full discussion of these approaches is deferred to Chapter 2.

To begin with, we must define what it means for a simulation to be correct. We say that an event e_2 is causally dependent on another event e_1 if the execution of e_2 can be affected, directly or through some chain of intermediate events, by the outcome of the execution of e_1. In order for a simulation to be correct it is necessary that all pairs of causally related events be executed in the correct relative order. Corresponding to our intuitive notion that time increases in the direction of causality in the real world, we take as an axiom that an event can be causally dependent only on events with strictly smaller timestamps. Thus, the sequential simulation strategy of executing all events in non-decreasing[3] timestamp order is guaranteed to preserve causal relationships and produce a correct execution. (This is proved in [Misra 86].)

[3]We assume that events scheduled to occur at the exact same simulation time can be executed in any order without changing the outcome of the simulation.

On the other hand, unless two events have identical timestamps (in which case they are causally independent by assumption), it is nontrivial to determine whether or not they are causally related. One style of parallel simulation, called *tight event-driven* [Kaudel 87], finesses this problem by executing in parallel only those events that have identical timestamps. Unfortunately, few systems have the requisite degree of event simultaneity to benefit from this approach. In order to achieve reasonable speedups on a wide variety of problems it is necessary to identify and to execute in parallel causally independent events having *different* timestamps. This approach is called *loose event-driven.*

The basic strategy of loose event-driven parallel simulation is to partition the simulation into a collection of processes that communicate only through message passing. Each message carries a timestamp corresponding to the simulation time of the event that the message represents. Since there is no shared state, causal dependencies can arise only through the transmission of information contained in the messages. Thus, the problem reduces to ensuring that events (messages) are executed in timestamp order *locally* at each process [Chandy & Misra 79, Jefferson 85]. Unfortunately, a process has no way of knowing, in general, whether or not the unprocessed message with the (currently) smallest timestamp at that process will in fact be the next message that should be processed, or if another message with an even smaller timestamp might still arrive (a so-called "straggler message"). This discrepancy between simulation time (the timestamp values of messages) and physical time (the actual order of message arrivals) arises because of the use of multiple processors and the absence of any global simulation clock.

Loose event-driven parallel simulation algorithms can be partitioned into two classes, the *optimistic* algorithms and the *conservative* algorithms. These algorithms differ in the way in which they deal with the problem of straggler messages. An optimistic algorithm allows a process to execute an event as soon as the message representing that event arrives. If a straggler message arrives, it is handled by rolling back the state of the process to a point in simulation time no later than the timestamp of the straggler, and

then re-executing from that point as necessary. A conservative algorithm *prevents* out of order event executions by forcing a process to block whenever there is insufficient information about the progress of other processes to ensure that a straggler message cannot arrive.

1.3 Thesis Statement

Both optimistic and conservative loose event-driven algorithms present significant implementation challenges. Both classes of algorithms have received a great deal of attention in the literature in recent years [Reynolds 85, Unger & Jefferson 88, Unger & Fujimoto 89]. Conservative algorithms are the focus of this dissertation. Our thesis is that conservative loose event-driven parallel simulation can achieve good speedup on a wide variety of problems. More specifically, our thesis is the following:

1. The conclusions of some previous studies of conservative loose event-driven parallel simulation were unnecessarily negative.

2. The aggressive use of shared memory eliminates or alleviates most of the major obstacles to good performance.

3. Exploiting model semantics on a case-by-case basis can have a dramatic effect on speedup.

4. Point (3) seems to be contrary to the idea that "transparent is better" where parallel programming is concerned. Consequently, a parallel simulator should provide orthogonal mechanisms for specification of model behavior and performance tuning of model execution.

5. Conservative loose event-driven parallel simulation is an effective way to reduce the time-to-completion of simulations of a wide variety of systems.

1.4 Overview of the Dissertation

In Chapter 2 we present the historical development of the conservative loose-event driven approach to parallel simulation. We then survey the literature in the area of parallel simulation, which includes not only conservative loose event-driven algorithms but also other approaches such as optimistic, time-driven, and conservative phased event-driven algorithms. All of these approaches fall into the general category of model function distribution (MFD), which is the strategy of trying to speed up the time to completion of a single simulation run by executing more than one event at a time.

We also discuss in Chapter 2 an alternative to MFD called application-level distribution (ALD), which involves running multiple sequential simulations in parallel. This approach is often advocated when the simulation is used to explore some parameter space, or when the stochastic nature of the simulation outputs necessitates the use of multiple independent replications to calculate confidence intervals. We argue that ALD complements, rather than obviates, MFD.

In Chapter 3 we examine the problem of predicting the speedup of a parallel simulation prior to its implementation. It is highly desirable to be able to do this because, as we shall see, some simulation problems having an apparently high level of logical parallelism do not have significant potential for being sped up.

Following a survey of past approaches to this problem, we present two new models of the performance of parallel simulation. These models are applicable to any style of parallel simulation, as they assume that blocking is negligible and predict upper bounds on speedup.

A central tenet of ours is that taking full advantage of shared memory requires a re-examination of traditional approaches to conservative parallel simulation. In Chapter 4 we identify four techniques for efficiently implementing a conservative parallel simulation in a shared memory environment. At least one of these techniques, *lazy blocking avoidance*, is completely novel. Measurements from our prototype parallel simulator *Synapse* validate the efficacy of these techniques, and reveal certain factors that are crucial to

good performance.

Another tenet is that the semantics of the problem being simulated can and should be exploited to improve performance. We explore this idea in Chapter 5. We argue that *lookahead* is an important factor in performance, and support this argument with empirical evidence in the form of measurement data from Synapse. The bulk of the chapter is then devoted to an investigation of techniques for improving the lookahead of various types of queueing system servers.

Chapter 6 describes the design goals and some implementation details of our prototype parallel simulator, Synapse. Synapse is a custom-tailored programming environment designed specifically for the efficient implementation of the techniques for exploiting shared memory and model semantics that were presented in Chapters 4 and 5. As demonstrated in those chapters, Synapse is able to achieve significant speedups on a wide variety of simulation problems. The development of Synapse as a laboratory and a tool is another contribution of this dissertation.

Chapter 7 demonstrates the real-world applicability of the techniques of Chapters 4 and 5, and the overall efficiency of Synapse, on a case study taken from the literature.

Finally, Chapter 8 summarizes the contributions of this dissertation and suggests avenues for further research in this area.

Chapter 2

Parallel Discrete-Event Simulation

In this chapter we present the approach to parallel simulation that is studied in this dissertation, *conservative loose event-driven parallel simulation.* We contrast it with other approaches to speeding up the time-to-completion of parallel simulations, such as optimistic, time-driven, and conservative phased event-driven simulations.

At the end of the chapter we also describe methods for speeding up the entire simulation modeling process (as opposed to a single simulation run), and argue that the use of intra-run and inter-run parallelism complement each other.

2.1 Sequential Discrete-Event Simulation

In the following discussion, we refer to the system being simulated as the *physical system*, and the model employed in the simulation as the *logical system.*

Any discrete-event simulator must correctly sequence the execution of events in simulation time. As we pointed out in the previous chapter, the sequential algorithm for discrete-event simulation achieves correctness by ensuring that all events are executed in monotonically non-decreasing simulation-time order.

```
while not finished do
    e = eventlist.min();
    clock = e.timestamp;
    e.action();
endwhile
```

Figure 2.1: An event routine-based sequential simulation algorithm.

```
while not finished do
    e = eventlist.min();
    clock = e.timestamp;
    switch_to(e.owner);
endwhile
```

Figure 2.2: Logical process algorithm in a process-oriented simulation.

A generic algorithm for a sequential discrete-event simulation is shown in Figure 2.1. `Eventlist` can be any type of priority queue data structure, ordered by event time-stamps. The variable `clock` represents a lower bound on the time up to which all state changes in the physical system have been simulated by the logical system. In the type of algorithm shown here, each event contains a reference to an *event routine*, which is a procedure that contains the actions associated with a particular type of event. These actions can include modifying the simulation state variables, collecting statistics, and creating and inserting additional events into the event list.

The observation that the physical system being simulated often contains a great deal of concurrency has led to a simulation style called the *process-oriented style*, which is typified by the SIMULA language [Birtwistle et al. 79]. In this style of simulation, each independent entity in the physical system (which we will call a *physical process*, or *PP*) is represented by a separate thread of execution (which we will call a *logical process*, or *LP*). Conceptually, the LPs are executing concurrently, although languages such as SIMULA usually provide only the illusion of concurrent execution by time-slicing the LPs on a single processor. A generic algorithm for a process-oriented simulation executive is shown in Figure 2.2. The simulation executive repeatedly removes the earliest event from the event list, determines the LP that is the owner of the next event (message) to be executed, advances the simulation clock appropriately, and transfers control to that LP.

(The LP's thread of execution, by assumption, is suspended waiting for the next event to be simulated by that LP.) The LP then decides upon the appropriate actions to take, which may include all those mentioned for event routines, plus waiting for additional events to arrive (which transfers control of the processor back to the executive).

Unless otherwise noted, what we mean by parallel simulation is any strategy for concurrently processing events at different LPs. This type of parallelization of a simulation is sometimes called *model function distribution (MFD)* [Kaudel 87].

2.2 Model Function Distribution

One MFD strategy is to modify a sequential algorithm to remove simultaneously from the event list all events that have the minimal timestamp, and to process them in parallel. This is termed *tight event-driven parallel simulation* [Peacock et al. 79]. This approach has been applied to the simulation of bus-based multiprocessor architectures [Wilson 86, Wilson 87].

There are two problems with the tight event-driven approach. First, it does not yield a great deal of parallelism unless the physical system is inherently synchronous, i.e., one in which multiple state changes typically occur at exactly the same time. Even VLSI circuit simulation, which would seem to contain an enormous amount of truly concurrent activity, often does not meet this criterion [Bailey & Snyder 89]. Second, if the system did exhibit a high level of synchronous activity, parallel accesses to the centralized event list could become a bottleneck. For systems with such characteristics a time-stepped approach is often more appropriate [Yu et al. 89].

Fortunately, the criterion that an event's timestamp must be minimal in order for the event to be executed is unnecessarily restrictive. Let us consider exactly under what circumstances the execution of one event can affect the execution of another. Clearly, event e_i affects the execution of event e_j if the execution of e_i creates or cancels e_j. Also, e_i affects e_j if the execution of e_j reads or writes state information that was created or altered by the execution of e_i. In either case we assume that the timestamp of e_i is

strictly less than the timestamp of e_j. We say that an event *a causally affects* an event *b* (or that *b* is *causally dependent on a*) if there is some chain of events $a = e_0, e_1, \ldots, e_n = b$ such that for each pair e_i and e_{i+1}, the execution of e_i affects the execution of e_{i+1}. (In other words, the *causally affects* relation is simply the transitive closure of the *affects* relation.)

For some pairs of events *a* and *b* it may be the case that neither *a* causally affects *b* nor *b* causally affects *a*; in this case we say that *a* and *b* are *causally independent* events. (In particular, note that any two events with exactly the same timestamp are causally independent by assumption.) Thus, the *causally affects* relation defines a *partial order* on the events in a simulation.

This definition of causal dependence is very similar to the notion of event ordering (in *real* time) in a distributed system developed by Lamport [Lamport 78]. This relationship has been explored in detail by Jefferson [Jefferson 85].

From the definition, it follows immediately that as long as every pair of causally dependent events are executed *in the correct order*, then (a) the portion of the system state examined by any particular event will be the same as it would have been if all events had been executed in some non-decreasing timestamp order,[1] and (b) exactly the same events will be executed. Thus, a simulation in which events are executed in any order that is consistent with the partial order defined by causal dependence is indistinguishable from some sequential execution of the simulation.

Strategies that attempt to extract parallelism from a simulation by allowing concurrent execution of causally independent events (with possibly different timestamps) are called *loose event-driven* parallel simulation strategies [Peacock et al. 79].

2.3 Loose Event-Driven Parallel Simulation

Consider a process-oriented simulation with the following characteristics:

[1] We use the phrase "some non-decreasing timestamp order" because event timestamps are not necessarily unique.

- Every LP has its own local clock.
- There are no shared variables between LPs.
- LPs communicate future events to each other by sending timestamped messages. The timestamp on a message is equal to the simulation time at which the event is scheduled to take place, and is always greater than the value of the sending LP's clock at the point when the message is sent.
- An LP cancels a future event at another LP by sending a message that is timestamped strictly less than the simulation time of the event being canceled, and is always greater than the value of the sending LP's clock at the point when the message is sent.
- Messages are buffered at the receiving LP; consuming a buffered message causes the LP's clock to be advanced to the timestamp of the message and causes the event corresponding to the message to be executed.

Note that since the event list is effectively distributed among the LPs, there is no simple way to determine the globally "next" event. However, the following theorem provides the basis for an algorithm for loose event-driven parallel simulation.

Theorem 2.1 *If each LP consumes messages in non-decreasing timestamp order, then the execution of the simulation is correct.*

Proof. We shall show that for every pair of events e_i and e_j such that e_i *affects* e_j, e_i is executed before e_j. Therefore, by transitivity every pair of causally dependent events is executed in order of the causal dependence, and the overall ordering of event execution will be consistent with the partial order defined by causal dependency.

For brevity, we denote the timestamp of an event e by $\mathrm{ts}(e)$. Note that e_i affects e_j implies $\mathrm{ts}(e_i) < \mathrm{ts}(e_j)$.

If e_i and e_j are executed by the same LP then e_i and e_j are executed in timestamp order by assumption, and so e_i is executed before e_j. Thus we need concern ourselves

only with pairs of causally dependent events that are executed at different LPs.

Consequently, assume that e_i is executed at LP_a and e_j is executed at LP_b. Since there are no variables shared between LPs, the only way for e_i to affect e_j is if e_i creates or cancels e_j:

1. If e_i creates e_j, then clearly LP_a must execute e_i before sending a message representing the occurrence of e_j to LP_b. Thus LP_b cannot execute e_j until LP_a has executed e_i.

2. If e_i cancels e_j, then we presume that LP_a sent e_j to LP_b before executing e_i; otherwise, there would be no reason to have sent e_j. Then LP_a will send LP_b a message with timestamp strictly less than $ts(e_j)$ to cancel e_j; since LP_b will consume this message before consuming e_j, e_j will never be executed. Thus, it is vacuously true that e_i will be executed before e_j. ∎

There are several strategies for guaranteeing that messages in a loose event-driven parallel simulation are consumed in non-decreasing order. The first one that we will describe, *conservative* loose event-driven parallel simulation, is the subject of this dissertation.

2.4 Conservative Loose Event-Driven Parallel Simulation

In the *conservative* approach to loose event-driven parallel simulation [Bryant 77, Chandy et al. 79, Chandy & Misra 79, Misra 86], LPs communicate over explicit unidirectional *channels*, and the following additional constraints are imposed on the basic loose event-driven scheme:

- The communication topology of the simulation must be statically defined.
- An LP must send messages on each of its outputs channels in non-decreasing timestamp order.

```
C_j := 0;
foreach i do t_ij = 0;
while not finished do
    while no buffered messages do
        await(message arrival);
    endwhile
    m := earliest buffered message;
    H_j := min_i{t_ij};
    while m.time > H_j do
        await(message arrival);
        m := earliest buffered message;
        H_j := min_i{t_ij};
    endwhile
    consume(m);
    C_j := max(C_j, m.time);
    simulate(m.event);
    /* may change C_j and/or */
    /* send messages to other LPs */
endwhile
```

Figure 2.3: Algorithm for LP_j in a conservative parallel simulation.

The second constraint implies that an LP cannot send any message until it is sure that the event represented by that message will not have to be canceled.

The basic algorithm run by each LP in a conservative loose event-driven parallel simulation is shown in Figure 2.3. For each pair i, j such that there is a channel from LP_i to LP_j, the variable t_{ij} contains the timestamp of the last message sent on that channel. t_{ij} is called the *channel clock value* of the channel [Misra 86]. (Initially, $t_{ij} = 0$ for all i, j.)

Theorem 2.2 *Conservative loose event-driven parallel simulation is correct.*

Proof. Because of the requirement that messages must be sent on each channel in non-decreasing timestamp order, at any given point in the execution of the simulation LP_j cannot receive a message from LP_i with timestamp less than t_{ij}. The variable H_j is maintained as the minimum of the t_{ij}; thus, when the timestamp of the earliest buffered message m is equal to H_j, LP_j cannot cannot subsequently receive a message with timestamp less than $m.time$ from *any* of its message sources. Hence, LP_j will consume mes-

sages in non-decreasing timestamp order, and the simulation is correct by Theorem 2.1. ∎

A detailed exposition of this approach to simulation is given by Misra [Misra 86]. In that paper, H_j is called the *clock value* of LP_j. However, this does not agree with the intuitive notion that an LP's clock value should be a measure of the progress of PP_j through simulated time: it is not always true that LP_j has simulated PP_j up to time H_j, only that it is *allowed* to do so. For this reason, we introduce another variable, C_j, which is defined as the minimum timestamp of any message that LP_j can generate (whereas H_j is an upper bound on the timestamps of messages than LP_j is allowed to consume). In our nomenclature, C_j is called the clock value, since it more accurately reflects the progress of PP_j than does H_j, which we call the *message acceptance horizon.*

Whenever a message m is consumed, C_j is set to the maximum of C_j and $m.time$. This is because if $C_j < m.time$, it must be the case that the state of PP_j does not change in the interval between C_j and $m.time$, and thus LP_j can produce no more messages until it processes m. On the other hand, if $C_j > m.time$, it means that C_j was advanced by LP_j in the course of simulating an earlier event. This implies that that event represented the initiation of a non-preemptible action of PP_j that ends no earlier than time C_j,[2] and consequently that LP_j cannot produce another message before time C_j. Thus C_j is always at least as great as the timestamp of the message that was most recently consumed by LP_j.

The approach described in this section has the appellation *conservative* because it assumes that if anything can go wrong, it will. If an LP has even a single input channel for which the channel clock value is less than the timestamp of the LP's earliest buffered message, it cannot consume that message, and must block instead. We call any such input channel a *blocking channel* for the LP.

[2] A simple example is the commencement of service at a FCFS server.

2.5 Deadlock in Conservative Loose Event-Driven Parallel Simulation

We divide the possible states of an LP into five categories, *halted*, *idle*, *blocked*, *deadlocked*, or *active*:

- An LP is *halted* if its state satisfies the termination condition for the simulation and it will produce no further messages.
- An LP that is not halted is *idle* if it has no messages buffered on any of its input channels.
- An LP that is not halted or idle is *blocked* if its message acceptance horizon is smaller than the timestamp of the its earliest buffered message.
- An LP is *deadlocked* if either (a) the LP is idle and all of its message sources are either deadlocked or halted, or (b) the LP is blocked and the sources of all its blocking input channels are either deadlocked or halted.
- An LP is *active* if it is neither halted, idle, blocked, nor deadlocked.

We say that a conservative parallel simulation is deadlocked if at least one LP is deadlocked and all other LPs in the simulation are either halted or deadlocked.

An example of deadlock is shown in Figure 2.4. (In the figure, rectangles represent LPs, numbers inside circles represent timestamped messages, and numbers in parentheses represent channel clock values of empty channels.) The figure depicts the state of the simulation after LPs A, B, and C have halted. LP D is blocked because the timestamp of its earliest buffered message is 12 but the channel clock value of the empty channel from LP B (and hence LP D's message acceptance horizon) is only 8. Intuitively, LP D cannot be sure that a message timestamped less than 12 will not arrive from LP B. Since LP B is halted, no more messages will be forthcoming from it, and LP D is deadlocked. Therefore, the simulation is deadlocked.

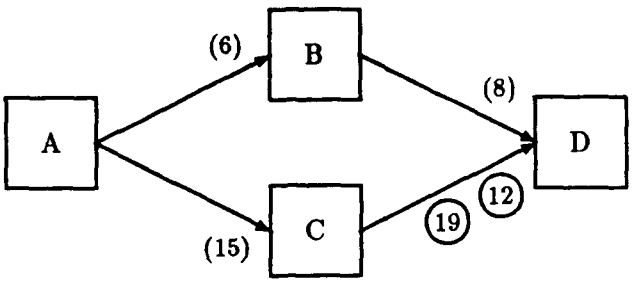


Figure 2.4: A deadlocked simulation.

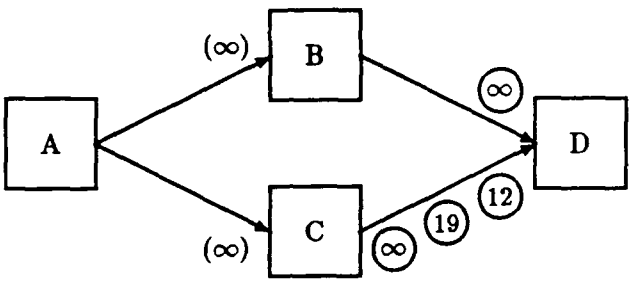


Figure 2.5: Prevention of deadlock by using end-of-simulation messages.

A simple way to prevent deadlock of the type shown in Figure 2.4 is as follows. Every LP that is about to halt should send a special *end-of-simulation* message, with a timestamp that is defined to be larger than any timestamp that can occur in the simulation, to all of its outputs. Because of the way in which the timestamp on the end-of-simulation message is defined, no LP will ever consume an end-of-simulation message unless every one of its input channels contains only an end-of-simulation message, which would imply that every one of its sources had already halted. Therefore, if an LP consumes an end-of-simulation message, then it should also halt (after sending end-of-simulation messages to its outputs, of course).

The effect of this strategy is shown in Figure 2.5, in which end-of-simulation messages are depicted as messages with the special timestamp "∞". Because LP D is no longer blocked (it has no empty input channels), it can consume the messages from LP C. Eventually, either LP D's state will satisfy the termination condition of the simulation or else LP D will consume an end-of-simulation message and halt.

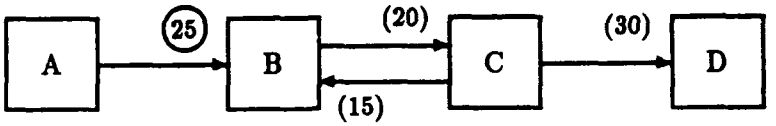


Figure 2.6: A deadlock resulting from a directed cycle of blocked LPs.

A qualitatively different type of deadlock is shown in Figure 2.6. In this example, none of the LPs in the simulation are halted; the reason for the deadlock is the presence of a directed cycle of blocked LPs. In this case, the fates of the deadlocked LPs B, C, and D[3] are independent of the state of LP A (which may be active or halted), since they are not waiting for messages from LP A.

One solution to this type of deadlock is to allow the simulation to deadlock, detect it, and then recover. The simulation thus consists of a sequence of phases performing useful computation (hopefully) in parallel, separated by *phase interfaces*, wherein a computation takes place to break the deadlock and allow various LPs to proceed [Chandy & Misra 81]. A straightforward way to recover from deadlock would be to identify the message with the smallest timestamp in the entire simulation, and allow that message to be consumed. In the example shown in Figure 2.6, this means that LP B would be allowed to consume the buffered message from LP A. (More sophisticated deadlock recovery algorithms will be presented in Section 4.4.)

Two drawbacks to deadlock detection and recovery are immediately apparent. First, the simulation is making no progress during the phase interfaces. Second, this approach contributes nothing to the level of parallelism during the computation phases of the simulation.

At least two phenomena are responsible for performance degradation of the parallel computation phases. The first is that the simulation may encounter *partial deadlocks*, i.e., deadlocks involving only a subset of the LPs. When this happens there may be a long period of time preceding each deadlock during which the number of runnable LPs

[3]Note that although LP D has no buffered messages, it is deadlocked according to our definition because its only message source (LP C) is deadlocked.

is much smaller than the number of processors, but is still non-zero. In the extreme case of a system with very little interaction between some LPs, the simulation may have only one runnable LP for long periods of time.

The second phenomenon that reduces the level of parallelism during the computation phases is called *artificial blocking.*[4] A blocked LP is artificially blocked if none of the sources of its blocking input channels will send any message to the LP timestamped earlier than the LP's earliest buffered message. In other words, the blocked LP has fallen far enough behind its message sources in simulation time that its choice of the next message to consume cannot be affected by subsequent message arrivals. In contrast to a deadlocked LP, it may be the case that the sources of an artificially blocked LP's blocking input channels are progressing in simulation time, but the LP is unaware of this. Thus, there is no reason for the LP to remain blocked.

An alternative to deadlock detection and recovery that also addresses partial deadlock and artificial blocking is *deadlock avoidance* [Chandy et al. 79, Chandy & Misra 79, Peacock et al. 79]. Deadlock avoidance has each LP periodically sending *null messages* to its "downstream" LPs. A null message has no effect on the simulation state variables; its only purpose is to convey a timestamp, to indicate that the sender will not be sending any real messages with smaller timestamps.

The end-of-simulation messages introduced earlier in this section were a special case of null messages. As a more general example, suppose that LP C in Figure 2.6 could predict, based on its current state, that it would not be sending any more messages to LP B before simulation time 30. Then it would send a null message with timestamp 30 to LP B, which would enable LP B to consume the buffered message from LP A (which has timestamp 25).

However, suppose it were not possible for LP C to make such a bold prediction. Suppose instead that the best that any LP x in this simulation could do was to predict that it would send no messages timestamped less than its message acceptance horizon

[4]This terminology was introduced by Reynolds [Reynolds 82].

Table 2.1: Avoiding deadlock with null messages.

H_B	H_C	Action
15	20	null(21) from C to B
21	20	null(22) from B to C
21	22	null(23) from C to B
23	22	null(24) from B to C
23	24	null(25) from C to B
25	24	B consumes message from A

H_x plus one. Then the sequence of null messages shown in Table 2.1 would have to be exchanged by LPs B and C before LP B could consume the message from LP A. As this example shows, the null message approach can be expensive, and the problem is exacerbated when the degree of branching in the communication graph is high. In extreme cases it might actually be more efficient to allow the simulation to deadlock and run the deadlock recovery protocol than to continue sending null messages.

Furthermore, deadlock avoidance cannot avoid deadlocks in all circumstances. In order for deadlock avoidance to work perfectly, the simulation cannot contain a cycle of LPs, all having a lower bound message processing time of zero.[5] In the previous example, if it were not possible for at least one of LPs B and C to make its predicted time of next message output *strictly greater* than its message acceptance horizon, then deadlock would have been unavoidable. In practice, therefore, some deadlock recovery mechanism must be available in case deadlock avoidance fails.

We will discuss the impact of shared memory on the implementation of conservative loose event-driven parallel simulation in Chapter 4.

[5]For a proof of this, refer to Peacock, Wong, and Manning [Peacock et al. 79]. An intuitive explanation is that if such a cycle existed, then a message could traverse the cycle infinitely many times without advancing any clock values, and the LPs in the cycle could never know if it were safe to proceed.

2.6 Other Model Function Distribution Strategies

For the sake of completeness, we now provide a brief description of other MFD strategies. The interested reader can find more details in the publications cited here.

2.6.1 Optimistic Loose Event-Driven Parallel Simulation

The optimistic approach to loose event-driven parallel simulation, developed by Jefferson and Sowizral [Jefferson & Sowizral 83, Jefferson 85], relies on the assumption that messages usually arrive in timestamp order. The optimistic strategy calls for an LP to process messages as fast as it can, without waiting to verify that this is the correct thing to do. In the case that this course of action turns out to be incorrect (i.e., a message arrives that is timestamped earlier than one that has already been consumed), the simulation relies on *rollback* of the LP's erroneous computation and *re-execution* from the point at which the late-arriving message should have been processed. Thus, all messages that are not rolled back will be eventually executed in non-decreasing timestamp order.

Note that the conservative scheme's requirements that the communication topology be static, that LPs send messages on each channel in non-decreasing timestamp order, and that LPs never attempt to revoke a message, are not necessary for correctness in the optimistic approach (although they may have an impact on efficiency).

The mechanism proposed by Jefferson and Sowizral to make rollback possible is called Time Warp [Jefferson & Sowizral 83]. We outline its operation here. First, each LP is required to *checkpoint* its state at regular intervals. Also, each LP must save its input messages, in order that they may be available for reprocessing if necessary, and must save one *negative message* (sometimes called an *antimessage*) corresponding to each output message generated by the LP.

When a message arrives in the *virtual past* of some LP (i.e., when a message arrives that has a timestamp smaller than the value of that LP's local clock), a rollback is triggered. The LP restores its state from the most recent checkpoint that is timestamped no later than the late-arriving message, so that it can perform the minimal amount of re-

execution that includes this message. Side-effect cancellation is accomplished by sending out all saved negative messages corresponding to positive messages that had been sent erroneously. The semantics of receiving a negative message are as follows: if the negative message arrives before the corresponding positive message has been consumed, it simply annihilates the positive message from the input queue of the destination LP. No further corrective action is needed because the erroneous message has not yet affected the LP. On the other hand, if the corresponding positive message has already been consumed, then in addition to the annihilation the negative message triggers a rollback of the destination LP, in order to undo the effects the erroneous message had on this LP. The rollback and re-execution of this LP may in turn cause the sending of more negative messages that trigger rollbacks at other LPs.

Periodically, a central controller initiates a distributed computation that determines a lower bound on the timestamp of the oldest unprocessed message in the entire simulation. This timestamp is called the *global virtual time (GVT)*. Since the simulation can never roll back to a point earlier than GVT [Jefferson 85], memory used by checkpointed state and saved messages timestamped earlier than GVT can be reclaimed.

The foregoing description should make it clear that there are several significant challenges to implementing the Time Warp mechanism efficiently: state checkpointing, negative message propagation, message flow control, GVT calculation, and garbage collection of old states and messages. Jefferson et al. have addressed these challenges with the *Time Warp Operating System (TWOS)*, a special purpose operating system in which Time Warp is the normal mechanism for process synchronization [Jefferson et al. 87].

One of the arguments in favor of the optimistic approach is that the amount of work wasted due to rollback is no greater than the amount that would have been wasted if the LP had instead blocked. In fact, Lin and Lazowska have shown that under certain assumptions, an optimistic strategy always performs at least as well as a conservative one [Lin & Lazowska 89]. A key assumption of the analysis is that there is no cost to state checkpointing or rollback other than the cost of re-executing erroneous computations; if

there is, Madisetti et al. have shown that the overall rate of GVT advancement in the presence of rollbacks is slower than the rate of advancement of the slowest LP's clock would be in the *absence* of rollbacks [Madisetti et al. 89b, Madisetti et al. 89c].

However, the assumption that there is no cost to rollback other than re-execution of erroneous computations does not hold if LPs do not checkpoint their state every time they process a message. For if this were the case, then a rollback could require the re-execution of *correct* messages whose timestamps happened to lie between the timestamp of the rollback-inducing message and the timestamp of the state checkpoint used for recovery. Thus, there is a tradeoff between the overhead of state saving and the overhead of re-execution after rollback. Fujimoto showed empirically that the average rollback distance is only a small number of messages, implying that if state is not checkpointed after every message then a significant fraction of rollbacks will suffer from this effect [Fujimoto 89]. Fujimoto's experiments also showed that performance degrades significantly as the size of the LP state vector is increased. As a result, he has suggested that the overhead of state checkpointing is one of the most serious challenges to the performance of the optimistic approach, and that the approach may be feasible only for applications in which the cost associated with processing a message is significantly larger that the cost of checkpointing an LP's state. To solve this problem, Fujimoto et al. [Fujimoto et al. 88] have proposed the use of special purpose hardware to reduce the overhead of state checkpointing and restoration.

Sokol et al. have attempted to reduce the number of rollbacks in optimistic parallel simulation by placing a limit on the temporal distance between any two LPs in the simulation [Sokol et al. 88]. This scheme is called the *Moving Time Window*. However, measurements taken so far by Hwang et al. have been inconclusive [Hwang et al. 89]. Similar studies undertaken using TWOS have not shown any demonstrable advantage to this strategy except under exceptional circumstances [Jefferson 89].

Abrams has observed that one of the major problems with the Time Warp mechanism is that negative messages end up "chasing" their positive counterparts through the

simulation [Abrams 88]. In order to avoid this problem, TWOS gives negative messages a higher priority than positive messages. Fujimoto has developed a shared memory algorithm called *direct cancellation* that can track down erroneous messages by a simple tree traversal [Fujimoto 89]. Since dereferencing a pointer is much faster than sending a message, the direct cancellation mechanism is presumably very effective at stopping erroneous side-effects from propagating. Direct cancellation has the additional benefit of eliminating the need to store negative copies of previously output messages. However, its use is limited to a shared memory environment.

Jefferson reports that one of the most difficult problems in implementing TWOS was message flow control [Jefferson 89]. It is fundamentally more complex than flow control in a conservative system for a variety of reasons. For example, in a conservative system flow control can be done on a channel-by-channel basis, whereas in a Time Warp system there is no concept of channel, so flow control must be done on a system-wide basis. In a conservative system, input messages can be deleted as soon as they have been processed; likewise, it is not necessary to keep negative copies of output messages as it is in Time Warp. These problems have been addressed in the Ph.D. thesis by Gafni [Gafni 85].

Additional studies of the performance of the optimistic parallel simulation strategy can be found in [Agre 89, Baezner et al. 89, Ebling et al. 89, Gates & Marti 88, Gilmer, Jr. 88, Hontalas et al. 89, Lomow et al. 88, Presley et al. 89, Wieland et al. 89].

2.6.2 Conservative Phased Event-Driven Simulation

Various conservative approaches to parallel simulation that can best be described as *phased event-driven* have been proposed independently by Najjar et al. [Najjar et al. 87], Lubachevsky [Lubachevsky 88], and Ayani [Ayani 89]. The basic idea in all approaches is that the simulator repeatedly iterates through two phases: first, a computation is performed to determine a subset of the existing future events that are *mutually* causally independent; second, all events in this subset are executed in parallel. A barrier synchronization is performed at the end of each phase to ensure that all LPs are synchronized

before beginning the next phase.

In all three approaches, the computation of mutually causally independent events is based on the concept of a *directional distance* between LPs. The directional distance from LP_1 to LP_2 is defined to be the minimum amount of elapsed simulation time between the execution of an event at LP_1 and the possible observance of any causal effect of that execution at LP_2. The directional distances are pre-computed based on the interconnection graph of the LPs and the characteristics of message timestamp increments at each LP.

In Najjar's approach, the first phase is performed by a central controller. This has the disadvantage that the central controller will become a bottleneck if there are a large number of LPs in the simulation. We are unaware of any implementation based on this approach.

Ayani's approach distributes the computation of the set of mutually causally independent events, removing this potential bottleneck. The computation is broken into two sub-phases. First, each LP marks the next event it would execute in the absence of any synchronization restrictions. Second, all LPs compare their marked event to the marked events of all other LPs, using directional distances to determine if events are causally dependent. (An additional barrier synchronization is required between these two sub-phases.) A modified algorithm that reduces the number of event comparisons required has also been formulated. Ayani reports on an implementation of this technique in [Ayani 89].

Lubachevsky's approach is very similar to Ayani's, but he has introduced the idea of *bounded lag*. That is, in Ayani's approach there is no *a priori* constraint on the difference between the timestamps of simultaneously executing events. Thus, many events in the simulation are candidates for simultaneous execution. Lubachevsky's scheme places a bound on the timestamp difference (the lag) between any two events that can be simulated in parallel. Combined with the restriction of a minimum message propagation delay (in simulation time) between any two LPs, this means each LP needs to check its

candidate event against candidate events belonging only to LPs that are within a small neighborhood (based on directional distance) around itself. Lubachevsky's approach also makes use of what he has termed *opaque periods*, i.e., periods of simulation time during which one LP "promises" not to send a message to another LP.[6]

On a simulation of an asynchronous Ising atomic spin model, Lubachevsky has obtained speedups of approximately 1900 on a 128x128 Connection Machine[7] [Lubachevsky 88]. To be fair, it should be pointed out that the Ising model contains an extraordinary degree of concurrent activity. Lubachevsky has also presented theoretical arguments that his approach is scalable to very large shared memory architectures (execution time increases only logarithmically with the number of processors) [Lubachevsky 89].

The major disadvantage to all of these schemes is that no useful work is being done during the event identification phases, thus limiting the potential parallel efficiency [Amdahl 67].

A slight variation on the phased event-driven approach has recently been proposed by Chandy and Sherman [Chandy & Sherman 89b]. They first outline a synchronous algorithm that is very similar to the ones already presented, and then propose modifications that eliminate the global synchronization points, thus making the protocol completely asynchronous. Their algorithm is based on the theory of global snapshots in a distributed system [Chandy & Misra 88]. However, they reportedly have implemented only the synchronous version of the algorithm.

2.6.3 Simulation Space-Time

Chandy and Sherman have proposed a unifying framework for parallel discrete-event simulation that they call *simulation space-time* [Chandy & Sherman 89a]. They view the entire history of a simulated system as existing *a priori*, with the task of the sim-

[6]The concept of the opaque period is identical to the *lookahead* concept referred to by other authors [Misra 86, Fujimoto 88a]. We defer a full discussion of lookahead to Chapter 5.

[7]Connection Machine is a registered trademark of Thinking Machines Corporation.

ulation being to evaluate every point in the space-time cross-product of the system's history. They assert that every method for simulation simply uses a different strategy for evaluating simulation space-time, and that what parallel simulation strategies all have in common is that they choose a particular decomposition of the simulation space-time among the processors. They claim that in principle it should be possible to partition simulation space-time arbitrarily, and to evaluate the values in each partition using a relaxation algorithm. Although this is an intriguing approach, its practicality remains to be demonstrated.

2.7 Alternatives to Model Function Distribution

The survey paper by Kaudel identifies two other strategies in addition to MFD [Kaudel 87]: *application-level distribution (ALD)* and *support function distribution (SFD)*. In this section we argue that neither of these techniques obviates MFD.

2.7.1 Application-Level Distribution

In Section 1.1.2 we pointed out that not only do individual simulation runs have large computational demands, but that many modeling processes require multiple simulation runs in order to arrive at a result. Given an application in which the latter is true, the most straightforward way to speed up the production of results would seem to be to run multiple sequential simulations in parallel. This approach is called *application-level distribution (ALD)* [Kaudel 87].

Examples of application domains in which simulation is used as a subroutine include:

- Design processes, in which simulation is used to validate a design, the results of which are used as input to the next stage of the design. An obvious example is VLSI circuit design.

- Parameterization studies, in which simulation is used to investigate the effects of certain parameters on the model, or to search for a parameterization that produces

a specified behavior. The latter case is characteristic of applications in which simulation is used to tune the performance of a system.

- Simulations of stochastic models, in which multiple replications of the simulation using different random number seeds are necessary in order to generate independent samples from which to calculate confidence intervals.

ALD is not a panacea in any of these cases, however. There are at least four situations in which ALD may not be able to fully utilize the available processing resources:

- There are many application domains in which ALD is not applicable because simulation is used as part of a feedback loop.

- In a parameterization study, ALD does not offer useful parallelism beyond a certain point; this point is reached when additional runs will not yield information allowing a more intelligent choice of future parameter values. Under these circumstances, the time-to-completion of each simulation run becomes the critical path of the entire process.

- In a stochastic simulation, ALD does not offer useful parallelism beyond the point at which the marginal increase in accuracy obtained by running additional replications is not as great as that obtained by increasing the run length of each replication.

- In general, ALD cannot fully exploit all available processors unless processors are the bottleneck resource. Limitations on resources other than processors may require the use of MFD to increase processor utilization.

In all of these situations, MFD becomes more desirable as the total number of processors increases. We now discuss each of these situations in turn.

Simulation as part of a feedback loop

An example of an application in which simulation is part of a feedback loop is VLSI design. A VLSI design cycle typically alternates between a design phase and a validation phase, with the outputs of the validation phase (the simulation) being required before the next design phase can begin. Thus, although multiple simulation runs may be required, they must occur sequentially. The simulation can become the bottleneck for the entire design cycle [Perry 89], and ALD does not offer a solution.

Simulation as part of a parameterization study

In a parameterization study, it may be possible to run multiple instances of the simulation (using different parameterizations) in parallel, reducing the need to parallelize individual simulation runs. However, it may also be the case that the analyst is trying find a parameterization that produces a particular model behavior; this is characteristic of applications in which simulation is used to tune the performance of a system. In such a case, the results of one run would be needed in order to guide the selection of parameters for the next run, and it would be more efficient to "navigate" through the parameter space rather than to take a "scattershot" approach. This is another case in which turnaround time for a single execution of the simulation is the overriding concern.

Stochastic simulations

For stochastic simulations ALD is always applicable. The reason for this is that the output observations of a stochastic simulation form a random sequence that is almost always autocorrelated [Fishman 78, Law & Kelton 82, Law 83]. The ramification of this is that the sample variance of observations from a single run is likely to be less than the true variance, leading to confidence intervals that are unrealistically small. Therefore, it is necessary to run multiple replications of the simulation using different random number seeds in order to generate truly independent samples from which to calculate confidence intervals.

Given the straightforwardness of ALD in the case of a stochastic simulation, what motivation is there for using MFD to parallelize the individual simulation runs as well? The main motivation is that ALD, by itself, cannot reduce the overall time necessary to produce a result below that which is necessary to run *each replication for a statistically* valid period of simulated time. That is, increasing the number of replications cannot make up for the loss of accuracy resulting from running individual replications for too short a period of simulated time.

The source of this inaccuracy is that the initial conditions of a simulation are never completely representative of the steady state behavior of the model,[8] so the distributions of the output observations change over time. Consequently, a sample mean based on only the first n observations from the simulation is actually a biased estimator of the true mean, and the coverage of the calculated confidence intervals may be less than expected [Law 83].

The implication of all this is that, for a given amount of processor resources, there is a tradeoff between the length of each replication and the total number of replications. As the total number of processors applied to the problem is increased, it will eventually be the case that the marginal benefit of additional replications is not as great as the benefit of increasing the run length (in simulation time) of each replication [Heidelberger 86]. (The number of processors required to make this true is dependent on statistical properties of the output variables.) When this point is reached, any additional processors should be used to speed up each individual replication.

Furthermore, no matter how many replications are used, it is still important to run each replication long enough (in simulated time) to reduce the initialization bias. (A precise determination of how long is "long enough" is still a subject for research [Heidelberger & Welch 83, Kelton & Law 84, Schruben 83].) If simulation run length is traded for a larger number of replications, there may be a significant probability of making erroneous inferences about the system under study. Thus, ALD by itself cannot reduce

[8] If they were, there would be no point to running the simulation!

the time-to-completion of the overall modeling process beyond a certain point without compromising the accuracy of the results. This suggests that whenever the number of processors available is greater than the number of replications required for generating confidence intervals, MFD should be used in conjunction with ALD to obtain the same level of statistical accuracy in less elapsed time.

A more detailed discussion of the tradeoff between the number of replications and the length of each replication is contained in Appendix A.

Yet another effect that reduces the efficiency of ALD when it is applied to stochastic simulation is that the completion time of individual replications is a random variable.[9] Grossly oversimplifying, it turns out that in order to obtain unbiased estimators of model outputs it is necessary to wait for all started replications to finish [Heidelberger 86]. As the number of replications run in parallel increases, an unbiased stopping rule leads to poor processor utilization near the end of the simulation run.

As a qualitative illustration of this effect, consider the following scenario. Suppose that the completion time for the i-th replication is some random variable C_i. Using P independent replications, the expected time-to-completion for the entire simulation run is $E[\max_{i=1}^{P} C_i]$. Now consider the following strategy: each replication is parallelized across all P processors, and the $i+1$-st replication is not started until the i-th replication has completed. If the average speedup of the parallel simulation algorithm on P processors is $s(P)$, then the expected time to completion when simulating in parallel is $E[\sum_{i=1}^{P} C_i]/s(P) = P \cdot E[C_i]/s(P)$. Therefore, as long as $s(P) > P \cdot E[C_i]/E[\max\{C_i\}]$, intra-replication parallelism is beneficial.[10] In fact, Heidelberger showed that, under certain distributional assumptions about the completion times C_i, as the number of

[9]The randomness of completion times may be a significant factor when the simulation is estimating *transient* behavior of the model, or when the *regenerative method* for obtaining quasi-independent samples is being used within each replication [Crane & Lemoine 77, Iglehart & Shedler 80].

[10]For loose event-driven parallel simulations, the lack of global control implies that the "tail utilization effect" also exists in the parallel case. However, the grain size of the computations involved is much smaller, hence so is the magnitude of the effect.

processors P increases, the speedup obtained is approximately proportional to $P/\log P$ rather than to P.

Resource limitations

A pragmatic consideration against ALD *in general* is that the resource requirements of simulation are not limited to processors. For example, the memory requirements of a sequential simulation may make it impossible (or counter-productive, due to excessive paging) to run as many sequential simulations as there are processors in the multiprocessor. On the other hand, the amount of memory required by a conservative parallel simulation should not be significantly greater than the amount required by a single sequential simulation.[11] Then MFD would be necessary to utilize the extra processors that ALD alone would have to leave idle.

Memory is not the only shared resource in a shared memory multiprocessor. Multiprogramming leads to contention for other resources as well, such as the memory bus or the memory interconnection network, and the I/O subsystem. Thus, it is unreasonable to expect that n simultaneously executing replications of a sequential simulation would finish in the same amount of elapsed time as a single replication running on an otherwise idle machine. In general, then, it may be advantageous to use MFD in addition to, or instead of, ALD in any situation in which processors are not the bottleneck resource.

Summary

Our conclusion is that ALD does not in general eliminate the need for MFD, for the following reasons:

- There are many application domains in which ALD is not applicable because simulation is used as part of a feedback loop.

[11] This claim is probably untrue for an optimistic parallel simulation, because of the memory required for state checkpoints [Jefferson et al. 87]. For an example, see [Wieland et al. 89].

- In a parameterization study, ALD does not offer useful parallelism beyond the point at which there is not enough information available to make any more intelligent choices of parameters and the time-to-completion of each simulation run becomes the critical path of the entire process.
- In a stochastic simulation, ALD does not offer useful parallelism beyond the point at which the marginal increase in accuracy obtained by running additional replications is not as great as that obtained by increasing the run length of each replication.
- ALD cannot fully exploit all available processors unless processors are the bottleneck resource. Limitations on resources other than processors may require the use of MFD to increase processor utilization.

In all cases, MFD becomes more desirable as the total number of processors increases.

2.7.2 Support Function Distribution

Support function distribution (SFD) distributes work by assigning support tasks such as random number generation, event list handling, statistics collection, and input/output, to additional processors. However, the amount of parallelism that can be obtained in this manner appears to be quite limited [Comfort 82, Comfort 83]. Furthermore, this technique is orthogonal to ALD and MFD, i.e., it can be used in conjunction with either of them equally well. Therefore, we do not consider SFD further.

2.8 Chapter Summary

We began this chapter by presenting the historical development of the conservative loose-event driven approach to parallel simulation. We then presented a survey of the literature in the area of parallel simulation, which includes not only the conservative approach but also other approaches such as optimistic, time-driven, and conservative phased event-driven simulations. All of these approaches fall into the general category of model func-

tion distribution (MFD), which is defined as trying to speed up the time to completion of a single simulation run by executing more than one event at a time.

We also discussed an alternative to MFD called application-level distribution (ALD), which involves running multiple sequential simulations in parallel. This approach is often advocated when the simulation is used to explore some parameter space, or when the simulation contains stochastic components that necessitate the use of multiple replications to calculate confidence intervals for simulation outputs.

We presented several arguments that ALD complements, rather than replaces, MFD, First, ALD is constrained by the presence of information feedback in the modeling process. Second, ALD does not offer useful parallelism beyond a certain point; this is true even in the case of stochastic simulations. Third, ALD cannot fully exploit all available processors unless processors are the bottleneck resource. In all cases, MFD becomes more desirable as the total number of processors increases.

Chapter 3

Predicting Performance

In Chapter 1 we stated our belief that the conclusions of previous studies of conservative parallel discrete-event simulation were unnecessarily negative. In this chapter we address one of the reasons for that belief, namely, that the degree of concurrency in the system being simulated is not always a good indicator of the speedup potential of the simulation of that system. We show that some of the benchmarks used in earlier studies had very little speedup potential, despite the fact that the benchmarks modeled highly concurrent systems. Although this is a negative result, in that it shows that some simulation problems cannot benefit from the application of parallelism, there is also a positive view. Previous studies that did not obtain good speedups on certain problems should not be taken as an indictment of the approach to parallel discrete-event simulation that was taken in those studies.

The contribution of this chapter is to present two techniques for calculating speedup potential. Both of our techniques are applicable to any style of parallel simulation, as they make the simplifying assumption that blocking is negligible. The first technique is an application of *asymptotic bound analysis* [Muntz & Wong 74] to obtain upper bounds on speedup. The second technique uses *mean value analysis* [Reiser & Lavenberg 80] to obtain tighter speedup bounds than those given by the first technique. These techniques have a great deal of practical value, because they can help simulation practitioners avoid

wasting time trying to parallelize simulation problems with poor speedup potential.

3.1 Previous Work

A very general technique for bounding the speedup of a parallel computation is *critical path analysis* [Even 79, sec. 6.5]. This technique involves an analysis of the *precedence graph* of the parallel computation. The precedence graph is a directed acyclic graph in which the nodes represent sub-tasks of the computation and the arcs represent precedence constraints. There are two distinguished nodes representing the beginning and end of the computation. The graph is labeled with weights as follows: each node is labeled with the amount of time needed to perform the sub-task represented by that node, and each arc is labeled with the amount of time needed to communicate synchronization information between the sub-tasks represented by the endpoints of the arc. When the graph has been labeled in this fashion, then the maximum weighted path from the start node to the end node, called the *critical path*, represents the minimum amount of time necessary to complete the parallel computation. Therefore, critical path analysis provides an upper bound on speedup.

Livny [Livny 85] and Berry and Jefferson [Berry & Jefferson 85, Berry 86] have applied critical path analysis to the problem of parallel simulation. The dependency graph is built from a post-mortem analysis of a log of all event executions in a simulation. The nodes in the graph represent events and the edges between the nodes represent direct (i.e., one-step) causal dependencies between events. (In other words, there is an edge from node e_1 to node e_2 if either (i) both events were executed by the same LP, event e_1 was executed before event e_2, and there were no intervening events at that LP, or (ii) event e_1 caused the scheduling of event e_2 at another LP.) Each node is labeled with the real execution time of the event and each edge is labeled with the real elapsed time between the completion of the event the edge originates from and the beginning of execution of the event the edge points to (representing the communication latency between LPs; this is taken to be zero for pairs of events at the same LP). As in the

general case, the maximum weighted path through the graph represents the minimum possible execution time of the simulation. Thus, it provides a standard of comparison by which to judge the efficiency of a parallel simulator.

The major drawback to critical path analysis is that it provides only *a posteriori* results. In order to use it one must already have programmed and run the simulation under consideration. Also, the speedup predicted is not necessarily representative of the simulation problem in general, but only of that particular simulation run, with the exact input parameters and random number seeds that were used.

If these arguments sound familiar, it is because they are also cited frequently as a case for using analytic modeling instead of benchmarking. Unfortunately, the analytic modeling of parallel simulation is still in its infancy. Very few results have appeared in the literature, and most of these are either qualitative in nature or deal only with very restricted special cases.

The effect of rollback on the progress of optimistic computation has been analyzed in [Jefferson & Witkowski 84, Mitra & Mitrani 87, Madisetti et al. 89a, Madisetti et al. 89b, Madisetti et al. 89c]. While each of these analyses is technically sound, there has recently been some debate as to whether or not rollback is a good predictor of the efficiency of an optimistic parallel simulation. The experience of Jefferson et al. in implementing and using the Time Warp Operating System suggests that state-saving overhead, not rollback, is the major obstacle to good performance [Jefferson et al. 87, Hontalas et al. 89, Presley et al. 89, Wieland et al. 89, Ebling et al. 89].

The only analyses of conservative algorithms of which we are aware deal with phased event-driven algorithms (Section 2.6.2), in which the measure of performance is taken to be the average number of events that are enabled for execution during each phase. However, it is not clear how this measure translates into actual speedup because of the cost of periodic global synchronization and the time spent identifying enabled events.

Lubachevsky analyzed the computational complexity of his *bounded lag* distributed simulation algorithm, and showed that, under certain assumptions about event "density"

in simulated time, the running time of the algorithm increases logarithmically in the size of the problem as the size and number of processors are simultaneously increased [Lubachevsky 89]. To achieve this behavior the algorithm uses parallel tree-structured computations for calculating minima and reaching barriers in logarithmic time.

Nicol has analyzed a variant of the conservative phased event-driven approach, in which it is assumed that the simulated duration time of an event at an LP can be computed before the message representing that event is sent to that LP [Nicol 89]. Under some fairly loose assumptions about event duration times, Nicol derived the expected number of events executed in each phase of the algorithm as a function of the average number of events that are occurring in the model at any given point in simulated time (which Nicol calls the *activity level* of the simulation model). His major result was that if the activity level is larger than the number of processors (P) squared, then the average number of enabled events in each processing phase is at least $P/3$. Furthermore, this quantity asymptotically approaches P as the activity level is increased relative to P.

In the remainder of this chapter we present two techniques for bounding the speedup of parallel simulation. The first technique is an application of asymptotic bound analysis to obtain upper bounds on speedup. The second technique uses mean value analysis to obtain tighter bounds than those given by the first technique. Both of these techniques make the simplifying assumption that blocking is negligible; their use is to demonstrate the speedup that can be attained under ideal conditions.

3.2 Asymptotic Bound Analysis

In order to run a simulation in parallel, the physical processes (PPs) in the simulated system must be mapped onto a collection of logical processes (LPs). The most straightforward mapping would be to use a separate LP to simulate each PP. This mapping can lead to unrealistic expectations about the degree of parallelism in the simulation. This is best illustrated by example.

Figure 3.1 shows a queueing network model of a timesharing computer system with

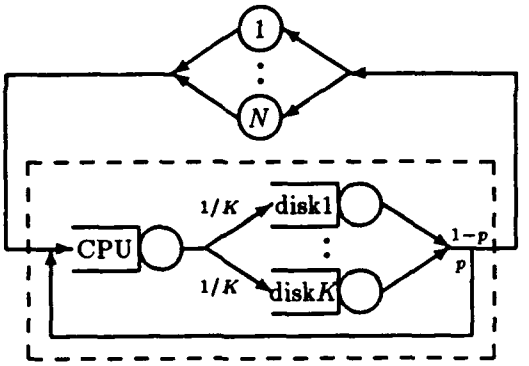


Figure 3.1: Example of a queueing network model with poor parallel simulation characteristics.

N terminals, a single CPU, and K disks. Such systems are designed to have a great deal of concurrent activity, with the service capacities of the various components of the system being carefully chosen to keep utilizations as balanced as possible.

For the moment, consider only the subnetwork inside the dashed box. There is the potential for a great deal of concurrency in this subnetwork, since $K + 1$ customers can be in service at the same time (one customer at the CPU and one customer at each of the disks). This concurrency is realized because, even though a job make many more visits to the CPU than to any one of the disks (K times as many, on average), the service time per visit at the CPU is typically orders of magnitude smaller than the service time per visit at any disk.

However, the presence of concurrency in the system being modeled does not imply the presence of the same degree of concurrency in the simulation of that system. Suppose that each of the servers in the model (CPU, disk, and terminal) is simulated by a different LP. To a first approximation, the computation required by an LP is proportional to the number of messages processed by that LP. Since the total message throughput of all of the disk LPs must equal the message throughput of the CPU LP, the total of the utilizations of the disk LPs will equal the utilization of the CPU LP (which cannot

exceed one):

$$\sum_{i=1}^{K} u_{\mathrm{LP}_{\mathrm{disk}i}} = u_{\mathrm{LP}_{\mathrm{CPU}}} \leq 1$$

In other words, the CPU LP is a bottleneck in the simulation. The average level of concurrency in a simulation of the indicated subnetwork, which is equal to the sum of the average utilizations of the LPs simulating the PPs in the subnetwork, cannot exceed two! Hence, speedup is bounded from above by two. We refer to this bound as the *speedup potential.*

By the same reasoning, the total utilization of all the LPs modeling the terminals cannot exceed the utilization of the CPU LP. In fact, because messages leaving any of the disk LPs are routed back to the CPU LP with probability p, the total utilization of the terminal LPs will be:

$$\sum_{i=1}^{N} u_{\mathrm{LP}_{\mathrm{term}i}} = (1-p) \cdot u_{\mathrm{LP}_{\mathrm{CPU}}} \leq 1 - p$$

Thus, the average parallelism in the simulation of this entire model is $(3-p) \cdot u_{\mathrm{LP}_{\mathrm{CPU}}}$, which is at most $3 - p$. Note that this analysis ignores the possibility of blocking, and the associated degradation in performance. Thus, even a speedup of $3 - p$ is probably unachievable for this model!

This intuitive introduction shows the power of *asymptotic bound analysis (ABA)* [Muntz & Wong 74]. ABA gives us an inexpensive technique for evaluating the suitability of a particular simulation model for execution in parallel. The basic idea is that if the speedup potential of a particular simulation problem is small, then it is probably not worth building a parallel simulation of that problem at all. Furthermore, the fact that a parallel simulation system fails to achieve significant speedup on such an example cannot be considered an indictment of that approach to parallel simulation. Instead, it should be viewed as a statement about the infeasibility of applying parallelism to a simulation of that particular problem.

In more detail, the technique we propose is to consider the simulation as a closed queueing system. The service centers in this system are the LPs in the simulation, and

the customers are the messages sent by the LPs. Denote the message throughput of LP_i by x_i, and the probability that LP_i forwards a message to LP_j by p_{ij}. Then since the flow of customers out of any server must equal the flow of customers into that server, we have:

$$x_j = \sum_i p_{ij} x_i \tag{3.1}$$

In other words, the x_i are the eigenvalues of the matrix $\mathbf{P} = [p_{ij}]$ (the *routing probability matrix*). Note that the system of equations given by (3.1) is linearly dependent, so its solution gives *relative* throughputs. The relative utilizations u_i of the servers can then be calculated as

$$u_i = d_i x_i \tag{3.2}$$

where d_i is the average cost (in real time) of processing a message at LP_i.[1]

We now make the generous assumption that the server with the largest relative utilization will in fact be 100% utilized; this corresponds to assuming that the LP it represents always has a message available for processing. Then an upper bound on the total utilization of all servers, and hence the speedup potential of the simulation, is given by

$$\frac{1}{\max_i u_i} \sum_i u_i \tag{3.3}$$

where the first term is simply a normalizing constant to ensure that no server's utilization exceeds unity.

Because of the assumptions involved, the quantity given by Equation (3.3) will be only an upper bound on speedup. Nevertheless, it is useful for identifying problems that are inherently unsuitable for simulation in parallel.

As a concrete example of this technique, consider the simulation benchmark shown in Figure 3.2, which was originally introduced in [Reed et al. 88]. This benchmark represents a queueing network model of a computer system similar to the central subsystem of Figure 3.1 with only two disks, except that there is an additional loop connecting

[1] Up to this point, we have assumed that the d_i are all equal.

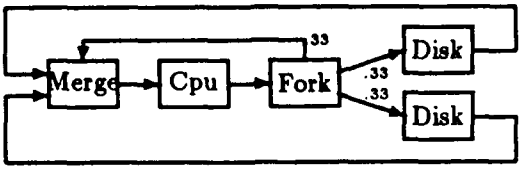


Figure 3.2: A central server model with explicit fork and merge nodes.

the output of the CPU LP to its input. (The presence of the fork and merge LPs is an artifact of the RESQ [Sauer et al. 80] specification language used by Reed et al.) Despite the fact that Reed et al. used their parallel simulator for their single as well as multiple processor measurements, they were unable to achieve a speedup greater than 1.25 for this benchmark. Both Fujimoto [Fujimoto 88a] and Wagner et al. [Wagner et al. 89], using highly optimized simulation implementations, achieved maximum speedups for this benchmark of just under 3, despite the fact that the benchmark contains 5 LPs.

This is easily explained by noticing that the message throughput is identical at the merge, CPU, and fork LPs, but that the throughput at each disk LP is only one-third of that amount. Assuming that the cost of processing a message is approximately the same at all LPs (i.e., that the d_i are all equal):

$$u_{\mathrm{LP}_{\mathbf{merge}}} = u_{\mathrm{LP}_{\mathbf{fork}}} = u_{\mathrm{LP}_{\mathrm{CPU}}}$$

$$u_{\mathrm{LP}_{\mathbf{disk1}}} = u_{\mathrm{LP}_{\mathbf{disk2}}} = \frac{1}{3} u_{\mathrm{LP}_{\mathrm{CPU}}}$$

Under the generous assumption that $u_{\mathrm{LP}_{\mathbf{merge}}} = u_{\mathrm{LP}_{\mathbf{fork}}} = u_{\mathrm{LP}_{\mathrm{CPU}}} = 1$, the total LP utilization in the simulation of this network, and hence the speedup potential, is only 3.67. Therefore, as a percentage of the theoretical maximum, the speedups obtained by Fujimoto and by Wagner et al. for this benchmark are actually quite respectable. (Note that this speaks well for the efficiency of the implementations, although it is a negative result in terms of the potential for speeding up a simulation of this particular benchmark.)

Reed and Malony conjectured that performance might be better for a simulation with a high ratio of computation to communication [Reed & Malony 88]. In order to

investigate this, they simulated this benchmark with various spin delays inserted in the LPs representing the CPU and disks. However, their results were inconclusive (in no case did their speedups exceed 1.25).

Contrary to Reed and Malony's intuition, Wagner et al. found experimentally that speedup was actually reduced as spin delays were increased, until it approached a value only slightly greater than 1 [Wagner et al. 89]. Again, this is easily explained by appealing to asymptotic bound analysis. As the spin delays are increased at the CPU and disk LPs, the ratios $u_{\mathrm{LP}_{\mathrm{fork}}}/u_{\mathrm{LP}_{\mathrm{CPU}}}$ and $u_{\mathrm{LP}_{\mathrm{merge}}}/u_{\mathrm{LP}_{\mathrm{CPU}}}$ tend towards zero. Thus, the speedup potential in the network, given by Equation (3.3), is asymptotically equal to

$$1 + \frac{u_{\mathrm{LP}_{\mathrm{disk1}}}}{u_{\mathrm{LP}_{\mathrm{CPU}}}} + \frac{u_{\mathrm{LP}_{\mathrm{disk2}}}}{u_{\mathrm{LP}_{\mathrm{CPU}}}} = 1.67$$

From the method proposed here, together with some experience, we have learned the following lessons:

- LPs (or subnetworks of LPs) that are connected in parallel between a fork and a merge point may not contribute to parallelism.
- Explicit fork and merge nodes probably will not contribute to parallelism, unless their service times are comparable to the service times of the "real" LPs.[2]
- It may be difficult to obtain parallelism between different subnetworks of a simulation, if the relative visit counts at the different subnetworks vary a great deal.

Although these results may seem to be largely negative, there is also a positive view: the identification of these common pitfalls will help avoid wasting time on "lost causes". Furthermore, we note that previous studies that did not obtain good speedups on such problems should not be considered an indictment of the approaches to parallel discrete-event simulation that were taken by those studies.

[2]This is just another way of saying that the amount of parallelism available through support function distribution (Section 2.7.2) is usually quite limited.

Also, the identification of redundant (with respect to speedup) LPs can aid in the mapping of PPs to LPs. There are many cases in which redundant LPs can be coalesced into a single LP (cf. Figure 3.1), and there are several reasons why doing so may improve performance:

- Reducing the number of LPs will reduce the number of context switches.
- Reducing the number of communication channels will reduce the overhead of blocking avoidance, especially if blocking avoidance is based on null messages (Chapter 4).
- Coalescing LPs in closed (sub)networks can improve the performance of lookahead calculations that are dependent on population constraints (Chapter 5).

3.3 A Tighter Bound on LP Utilization

For a simulation problem with a homogeneous topology (such as a ring, torus, or hypercube) message throughputs are identical at all LPs, and thus there is no bottleneck LP. Thus, the ABA method of the previous section, which assumes that the bottleneck LP is 100% utilized, predicts a speedup potential equal to the number of LPs in the simulation.

In actuality, since there are only a limited number of messages in the simulation it is impossible for any LP to be 100% utilized. Our approach in this section is to apply the method of mean value analysis (MVA) [Reiser & Lavenberg 80] to the problem.

Briefly, MVA is a method for obtaining performance measures such as throughputs, queue lengths, and customer residence times of service centers in closed queueing networks. In contrast to ABA, the performance measures given by MVA are based on the number of customers actually in the system.

The MVA algorithm is able to compute these performance measures for a given customer population n from the corresponding measures obtained when the population is $n-1$ by assuming that $A_k(n)$, the arrival instant queue length at the k-th service

center when there are n customers in the network, is equal to $Q_k(n-1)$, the steady-state average queue length at the service center when there are only $n-1$ customers in the network. This assumption turns out to be true if service times at all service centers are exponentially distributed [Lavenberg & Reiser 80, Sevcik & Mitrani 81]. This enables the algorithm to begin with the performance measures for a single customer and iteratively compute the performance measures for all populations up to and including the desired one in the following manner:

1. The customer residence times at each service center are calculated according to the equation

$$R_k(n) = D_k(Q_k(n-1)+1) \tag{3.4}$$

   where $R_k(n)$ is the residence time at service center k when there are n customers in the network and D_k is the service demand at service center k.

2. Little's law [Little 61] is applied to the network as a whole to obtain the throughput of customers in the network from the residence times at each of the service centers:

$$X(n) = \frac{n}{\sum_k R_k(n)} \tag{3.5}$$

3. Little's law is applied to each service center individually to obtain the new steady-state average queue lengths:

$$Q_k(n) = X(n)R_k(n) \tag{3.6}$$

Like ABA, MVA gives only an upper bound on speedup, but it is a tighter upper bound. ABA assumes that the simulation contains enough messages to keep the bottleneck LP 100% utilized. Our MVA approach represents the true message population of the simulation. However, it assumes that messages queued at an LP can always be processed — i.e., that blocking never occurs.

Unlike ABA, MVA is sensitive to the service time distributions at the service centers. Unfortunately, this is a problem for us, because we believe that exponential service

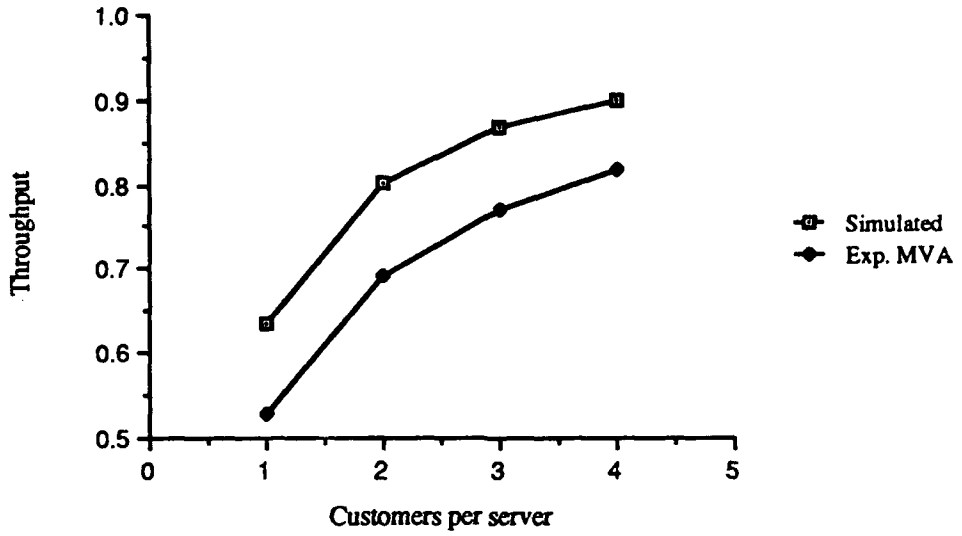


Figure 3.3: The inaccuracy introduced by the exponential service time assumption of MVA.

times are not realistic for our application. This is because there is unlikely to be much variability in the amount of time required to *simulate different* messages at the same LP; this time is likely to be dominated by such operations as receiving a message, generating random numbers, calculating statistics, and sending a message, the relative frequency of which ought not to vary a great deal from message to message. Therefore, we would like to obtain performance measures for queueing networks with *deterministic* service times. Since residence times at a service center with exponentially distributed service times are expected to be larger than at a service center with deterministic service times, the performance measures given by the standard MVA algorithm will tend to be somewhat pessimistic for our model.

Figure 3.3 illustrates the magnitude of this inaccuracy by comparing the measured throughputs of a simulation of a homogeneous queueing network (a 3x3 edge-connected mesh) with deterministic service times to the throughputs predicted by a mean value analysis of the same network.

We can accommodate deterministic LP service times within our MVA model by

modifying the residence time equation in the MVA algorithm (Equation (3.4)). The residence time equation depends upon the exponential assumption in two ways. First of all, it assumes that $A_k(n)$, the arrival instant queue length at service center k when there are n customers in the network, is equal to $Q_k(n-1)$, the average queue length at the service center when there are only $n-1$ customers in the network. We retain this assumption, although it does introduce a small amount of error.

The second way in which the residence time equation depends upon the exponential assumption is that it assumes that the expected residual service time of a customer already in service at an arrival instant is equal to the mean service time. The exponential distribution is the only continuous distribution having the property that expected residual life is equal to the mean of the distribution (the so-called "memoryless" property). Fortunately, this assumption can easily be corrected.

We can make the dependence on the memoryless property explicit by rewriting the residence time equation as follows:

$$\begin{aligned} R_k(n) &= D_k(Q_k(n-1)+1) \\ &= D_k(Q_k(n-1)+1+U_k(n-1)-U_k(n-1)) \\ &= D_k + D_k(Q_k(n-1)-U_k(n-1)) + D_kU_k(n-1) \end{aligned} \tag{3.7}$$

where $U_k(n-1)$ is the utilization of service center k with $n-1$ customers in the network. Note that since server utilization can be thought of as the probability of finding a customer in service at a random instant, $(Q_k(n-1)-U_k(n-1))$ can be interpreted as the expected number of customers at the LP that are *queued but not in service*. Thus, the first term in Equation (3.7) represents the expected service time of a customer arriving at the service center, the second term represents the expected total service time for the customers that are already queued but have not yet entered service, and the last term represents the expected remaining service time of the customer in service (if any) at the instant of arrival.

To handle non-exponential service time distributions we must modify the last term in Equation 3.7 to reflect the true expected remaining service time. Since the expected

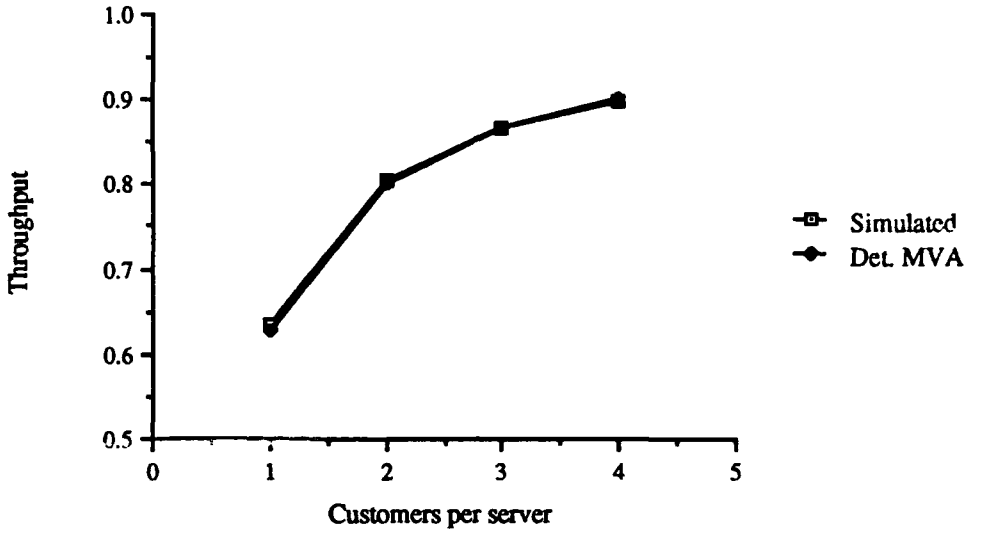


Figure 3.4: Comparison of the simulation model to MVA with the deterministic service time approximation.

residual life of a deterministic service time is just one-half of the mean service time, this leads to:

$$\begin{aligned} R_k(n) &= D_k + D_k(Q_k(n-1) - U_k(n-1)) + \frac{D_k}{2}U_k(n-1) \\ &= D_k\left(Q_k(n-1) + 1 - \frac{U_k(n-1)}{2}\right) \end{aligned} \tag{3.8}$$

Although the resulting algorithm is only approximate, since it is no longer true that $A_k(n) = Q_k(n-1)$, Figure 3.4 shows that the throughputs predicted by it agree very closely with those obtained by a simulation of the model.

Figure 3.5 shows the parallel efficiency predicted by this model for homogeneous networks of 4, 8, 16, and 32 LPs[3] as a function of the average number of messages per LP in the network. Because of the optimistic nature of the model, and the fact that there are no bottlenecks in the simulation, these numbers can be taken as hard upper bounds on the efficiency obtainable by simulating such networks in parallel.

[3]Since MVA is insensitive to all aspects of the queueing network topology except relative visit counts (which affect relative throughputs), our model gives the same results for any homogeneous topology of a given size.

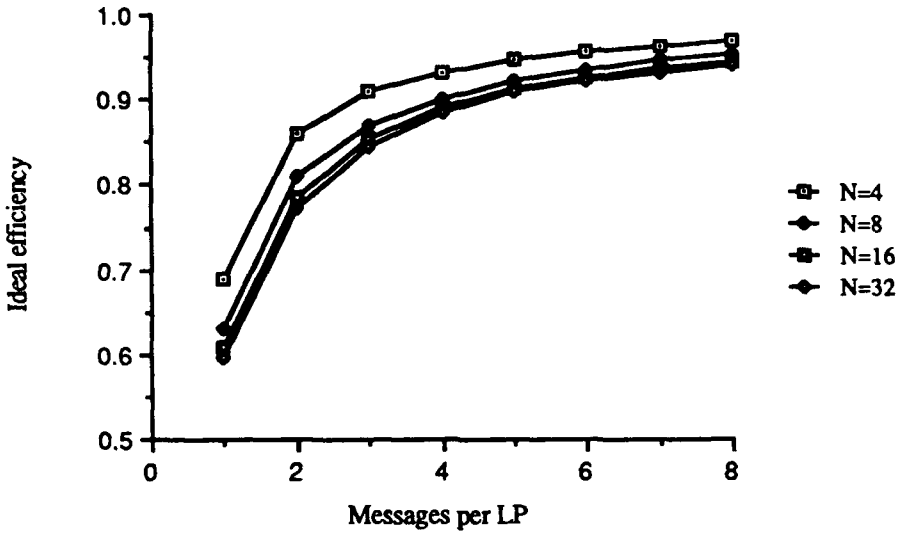


Figure 3.5: Predicted efficiency under ideal conditions.

The point that this model is intended to drive home is that even under ideal conditions, a parallel simulation cannot be expected to deliver anywhere near perfect efficiency unless the number of messages in the simulation is fairly high. The effect is stronger as the size of the simulation increases, due to the fact that with more LPs, there is a greater probability that some of the LPs will have no buffered messages at any given instant.

3.4 Chapter Summary

In this chapter we examined the problem of predicting the performance of a parallel simulation. We began by reviewing previous efforts to analyze the performance of parallel simulation, and observed that all of the analytic work that has been reported in the literature dealt with either the optimistic approach or the phased event-driven conservative approach. In all cases, speedup was not predicted directly; rather, it was implied by some auxiliary metric (absence of rollbacks in the optimistic case, and average number of events enabled for execution during each phase in the conservative phased event-driven case).

In the remainder of the chapter we presented two models of parallel simulation. These models are applicable to any style of parallel discrete-event simulation, since they deal only with throughput considerations and they assume that blocking is negligible. The first of these two techniques was based on asymptotic bound analysis, while the second was based on a modified mean value analysis algorithm. These models are useful for determining the speedup potential of specific simulation models. They show that a system possessing a high degree of concurrency does not necessarily yield a model with a high speedup potential. Although some problems are inherently unsuitable for simulation in parallel, previous studies that did not obtain good speedups on such problems should not be considered an indictment of the approaches to parallel discrete-event simulation that were taken by those studies.

Chapter 4

Exploiting Shared Memory

Our concern in this chapter is the efficient support of conservative parallel discrete-event simulation in a shared memory multiprocessor environment. First we describe the difficulties associated with distributed implementations of conservative parallel discrete-event simulation, and how these difficulties can be reduced or eliminated by the availability of shared memory. Next, we argue that although the use of shared memory can make traditional synchronization algorithms run much more efficiently, completely different approaches can and should be considered in this environment. We present four techniques for high performance shared memory conservative parallel simulation. These techniques have been implemented in the *Synapse* system, which is used as the basis of all performance measurements presented in this dissertation. (The Synapse user interface and some implementation aspects are discussed in Chapter 6.) Measurement data from Synapse demonstrates the efficacy of these techniques.

4.1 Shared Memory vs. Distributed Parallel Simulation

By a *distributed* parallel simulation, we mean a parallel simulation implemented on a collection of computers (hosts) communicating on a relatively low bandwidth, relatively high latency network. Each host is assigned the responsibility of executing one or more

logical processes (LPs). Communication between LPs that do not reside on the same host results in a message being sent on the network. Since inter-host communication in a distributed system is so much more expensive than intra-host communication, it is desirable to assign LPs to hosts in a way that minimizes the number of messages sent between LPs on different hosts. A second constraint on the placement of LPs is the desirability of keeping the processing load balanced among all the hosts in the network. The resolution of these two often-conflicting constraints is known as the *partitioning problem* [Nicol & Reynolds 85a, Nicol & Reynolds 85b].

One obvious benefit of shared memory is that it makes message passing very fast. In fact, if we ignore delays due to contention for shared data structures, sending and receiving a message in a shared memory parallel simulator can be less expensive than queueing and de-queueing an event in a sequential simulator. This is because a sequential simulator maintains all scheduled events in a single event list, and the best known event list implementations require $O(\log N)$ amortized time to insert and delete items (where N is the number of items in the event list) [Jones 86]. On the other hand, the data structure into which a message is enqueued in a loose event-driven parallel simulator typically contains only a fraction of the events (messages) in the simulation, specifically, only those events that affect the LP to which the message is being sent. Thus, the distribution of the event list among the LPs has the potential to reduce execution costs.[1]

Since the assignment of LPs to processors in a shared memory environment has very little effect on communication costs, the most important criterion in choosing an assignment is that computational load be balanced. Thus, the partitioning problem reduces to a computation load-balancing problem. Furthermore, the use of a global scheduling policy can eliminate this problem entirely, by allowing any LP to run on any processor; this maximizes processor utilization, and thus efficiency.

A shared memory implementation also reduces the cost of deadlock detection and

[1] In Section 4.6.4 we shall see an example in which running the parallel simulation algorithm on a single processor is *faster* than the standard, centralized event list-based sequential algorithm!

recovery. In a distributed system, deadlock detection protocols typically require $O(e)$ messages, where e is the number of edges in the communication graph [Chandy et al. 83]. The deadlock recovery algorithm proposed in [Chandy & Misra 81] requires $O(n \cdot e)$ messages, where n is the number of LPs and e is the number of edges in the communication graph. In a shared memory environment, much cheaper, centralized algorithms can be employed.

Reed et al. implemented a conservative parallel simulator on a shared memory multiprocessor in order to investigate the effect of reduced communication costs on performance [Reed et al. 88, Reed & Malony 88]. The implementation was essentially the same as it would have been in a distributed environment, except that a central controller process was used to detect deadlock. The results were disappointing, indicating that partitioning and communication costs are not the only obstacles to the acceptable performance of conservative parallel simulation. Degradation in parallelism due to artificial blocking is the major culprit.

The availability of shared memory provides an opportunity to re-examine traditional approaches to LP synchronization, with an eye towards increasing the average level of parallelism in the simulation by decreasing the amount of LP blocking. The ability of any component in the system to use the state information of any other component, which is infeasible in a distributed implementation, clearly ought to be exploited. Also, the run-time system can be optimized for performing parallel simulation, thus reducing the overhead of synchronization.

4.2 A New LP Algorithm for Shared Memory

The basic algorithm run by each LP in a distributed parallel simulation (i.e., one without any shared memory) was described in Section 2.4 and is reproduced here in Figure 4.1(a). Recall that t_{ij}, the channel clock value for the channel from LP_i to LP_j, is the timestamp of the last message sent on that channel; H_j, the message acceptance horizon of LP_j, is a lower bound on the timestamp of messages that can arrive at LP_j; and C_j, the local

```
C_j := 0;
foreach i do t_ij = 0;
while not finished do
    while no buffered messages do
        await(message arrival);
    endwhile
    m := earliest buffered message;
    H_j := min_i{t_ij};
    while m.time > H_j do
        await(message arrival);
        m := earliest buffered message;
        H_j := min_i{t_ij};
    endwhile
    consume(m);
    C_j := max(C_j, m.time);
    simulate(m.event);
    /* may change C_j and/or */
    /* send messages to other LPs */
endwhile
```

(a)

```
C_j := 0;
foreach i do t_ij = 0;
while not finished do
    while no buffered messages do
        await(message arrival);
    endwhile
    m := earliest buffered message;
    H_j := min_i{max(C_i, t_ij)};                (1)
    while m.time > H_j do
        await(message arrival
              or {ΔC_i for some i});             (2)
        m := earliest buffered message;
        H_j := min_i{max(C_i, t_ij)};            (1)
    endwhile
    consume(m);
    C_j := max(C_j, m.time);
    simulate(m.event);
    /* may change C_j and/or */
    /* send messages to other LPs */
endwhile
```

(b)

Figure 4.1: Algorithm for LP_j in a distributed parallel simulation (a) and in a shared memory parallel simulation (b).

simulation time at LP_j, is a lower bound on the timestamp of any message that can be sent by LP_j.

In the basic distributed simulation algorithm, LP_j uses t_{ij} as an estimate of how far LP_i has advanced in simulation time. An obvious drawback to this is that LP_i may have advanced arbitrarily far beyond t_{ij}, but LP_j has no way of knowing this in the absence of message traffic (real or null) from LP_i. This can give rise to LP_j being artificially blocked.

Given the availability of shared memory, there is no reason to use t_{ij} as an *estimate* of LP_i's progress, when the *exact* value (C_i) is available for inspection. This is the strategy of the algorithm shown in Figure 4.1(b). To accomplish this, there are two important differences between the original and the modified algorithms; these are identified by a number in the right margin of Figure 4.1(b):

(1) The reason for taking the maximum of C_i and t_{ij} in the calculation of H_j is that

LP_i may be able to predict that it will send no messages to LP_j timestamped earlier than some time $t > C_i$. Furthermore, this may be the case even though LP_i might still send a message to some LP_k, $k \neq j$, with a timestamp smaller than t, and is thus unable to advance C_i. If this is the case, LP_i can send a null message with timestamp t to LP_j, which will increase t_{ij} to t. Thus, the t_{ij} values may not be redundant.

(2) In addition to waiting for messages to arrive whenever H_j cannot be advanced, LP_j must also be alert for changes to any of the C_i values of its message sources. A change in one or more of the C_i may increase H_j enough to allow LP_j to consume a buffered message.

The second modification is the more difficult one to implement efficiently. If there are as many processors as there are LPs, busy waiting is the obvious solution. If not, LP_j might eventually be forced to relinquish its processor, with possibly unfortunate consequences: if the sources of LP_j's blocking input channels were to subsequently advance their clocks without sending any messages to LP_j, LP_j might become artificially blocked.

Null message-based deadlock avoidance would solve this problem, because eventually LP_i would send a null message to LP_j, which would cause it to increase t_{ij} (and possibly to notice that other message sources had increased their clock values as well). Unfortunately, null message-based deadlock avoidance can be very expensive (Section 2.4). In the next section we will investigate ways to exploit shared memory to avoid artificial blocking, and hence deadlock. We prefer the more general term *blocking avoidance* to describe these techniques, since the goal is not just to avoid deadlocks, but to increase the average level of parallelism during the periods in which the simulation is not deadlocked as well.

4.3 Shared-Memory Blocking Avoidance Techniques

The cost of blocking avoidance can be greatly reduced by exploiting the following observation [Misra 86]: since null messages convey no information other than a timestamp, and since the timestamps of all messages sent on a particular channel must be in non-decreasing order, the most recently sent message (real or null) on any given channel supersedes all previously sent null messages on that channel. The ramifications of this observation for a distributed memory implementation are that when a message arrives on some input channel, all unconsumed null messages in the channel queue can be thrown away.[2]

But in a shared memory implementation, it is unnecessary to use null messages at all. Instead, the channel clock value t_{ij} (which resides in shared memory) can be modified directly by LP_i. This optimization was first incorporated into an implementation by Fujimoto [Fujimoto 88a, Fujimoto 88b].

Direct modification of a channel clock value is so inexpensive that the actual awakening of the LP at the destination end of the channel becomes the major overhead of shared memory blocking avoidance. Because of this, there are two important issues in the implementation of shared memory blocking avoidance, namely: *who* should awaken a blocked LP, and *when*?

4.3.1 Eager Blocking Avoidance

One answer to question of *who* should awaken a blocked LP_j is: any LP_i that is a source of messages for LP_j. This suggests several alternatives for *when*: whenever $\max\{C_i, t_{ij}\}$ changes; whenever LP_i sends a message to any of its possible destinations; or whenever LP_i is about to block. (In the last two cases, wakeups would only be done if $\max\{C_i, t_{ij}\}$ had changed since the last round of wakeups.) There are several drawbacks to policies of this type:

[2]Refer to [Su & Seitz 89] for a comparison of the effect of various message buffering schemes on this optimization.

- The cost of the protocol is borne by the LPs that are *not* blocked, who might have other work to do. Doing the wakeups only when an LP is about to block solves this problem, but delays the awakening of LPs that could profit by it, thus increasing the amount of artificial blocking suffered by those LPs.

- It might be the case that the LP that is awakened is still blocked, due to some other blocking input channel; in this case, awakening it simply caused an unnecessary c .text switch. In this case, even if the wakeup were performed LP that was about to block, the wakeup might be delaying the scheduling of an *unblocked* LP that is waiting for a processor on which to run.

The salient feature of this type of blocking avoidance policy is that the wakeup mechanism is executed regardless of whether or not there is any need to do so at the moment, i.e., whether or not there are enough unblocked LPs to keep all of the processors busy. Because of this, we term this policy *eager blocking avoidance* (by way of analogy with the programming language concept called *eager evaluation*, in which expressions are evaluated even though their values may never be required).

4.3.2 SRADS

SRADS (a Shared Resource Algorithm for Distributed Simulation) is a blocking avoidance policy proposed by Reynolds to address the problems discussed in the previous section [Reynolds 82].

Suppose LP_j wishes to consume a message with timestamp t, but some of its source LPs have not progressed that far. For each such source LP_i, LP_j inserts a tuple of the form (j, t) in a data structure belonging to LP_i. Each time LP_i advances its clock value, it checks its data structure to see if any other LPs need to be notified. In this way, LP_i does not spend time notifying LPs that are not waiting for it, or for whom LP_i's clock has ɔt advanced far enough. Therefore, the amount of extra work done by the running LP is reduced, as is the number of unnecessary wakeups. However, there is still

no guarantee that any wakeups performed will actually enable a blocked LP to proceed, since it may be waiting on more than one other LP.

4.3.3 Lazy Blocking Avoidance

To avoid placing any burden whatsoever on the LPs that are doing useful work, we propose delegating the responsibility for waking blocked LPs to the run-time kernel.

A blocked LP could be awakened at regular intervals; this is logically equivalent to having the blocked LP poll its message sources. But this still doesn't solve the problem of unnecessary wakeups. A better solution would be to have the kernel use an idle processor to recompute the message acceptance horizon of a blocked LP, *without* performing a context switch. LP_j would be awakened only if the new value of H_j were greater than or equal to the timestamp of its earliest buffered message. This strategy has the advantage of not causing any unnecessary wakeups. At the same time, the potential amount of delay experienced by a blocked LP that has useful work to do is minimized, since there are no context switches taking place until such an LP is identified. Finally, it has the property that the amount of computational effort devoted to avoiding unnecessary blocking increases with the number of blocked LPs! When there are no idle processors, no effort is made to unblock LPs; but as the number of blocked LPs increases, so will the number of idle processors, and hence the rate at which blocked LPs are examined.

We call this mechanism *lazy blocking avoidance*, since, in contrast to eager blocking avoidance, no work is done unless some physical processor has nothing better to do. Lazy blocking avoidance is executed by processors, which are physical resources, rather than LPs, which are logical ones. In the remainder of this dissertation, we will use the terms "lazy blocking avoidance" and "eager blocking avoidance" to emphasize this distinction and to avoid the ambiguity of the terms "deadlock avoidance" and "blocking avoidance".

4.4 Efficient Deadlock Detection and Recovery

As we pointed out in Section 2.4, it is not always possible to avoid deadlocks in a parallel simulation. In order for deadlock avoidance to work perfectly, the simulation cannot contain a cycle of LPs, all having a lower bound message processing time of zero [Peacock et al. 79]. For this reason, some mechanism for detecting and breaking deadlocks is almost always required.

As mentioned previously, algorithms for detecting deadlock in a distributed system are expensive. Because of this expense, the algorithm may only be run periodically, and a significant amount of time may elapse between the time a deadlock occurs and the initiation of the deadlock detection algorithm. In contrast, detecting deadlock in a shared memory environment is trivial: deadlock has occurred when all processors are idle and there are no LPs ready to run. A simple check can be performed each time a processor goes idle, ensuring that the deadlock will be detected immediately after the last running LP blocks.

In order to break deadlock, the scheduler must determine $\hat{t}$, a lower bound on the earliest simulation time at which any message will be sent by any LP, assuming no further messages arrive. Let $\hat{m}_i$ be the earliest buffered message at LP_i, with timestamp $\hat{m}_i.time = \hat{t}_i$. Assuming that no further messages arrive at LP_i, $\hat{m}_i$ will be the next message simulated by LP_i; furthermore, doing so will cause C_i to be set to $\max(\hat{t}_i, C_i) = \hat{C}_i$.[3] Then $\hat{t}$ is simply $\min_i\{\hat{C}_i\}$. Since $\hat{t}$ is the earliest time at which any message in the simulation will be simulated, it is clearly a lower bound on the earliest time that any message will be sent.[4] Thus, the message acceptance horizon of every LP in the

[3] If there are no buffered messages at LP_i, $\hat{C}_i = \hat{t}_i = \infty$.

[4] With some help from the LPs, it is possible to do even better. Chandy and Misra proposed a scheme in which each LP_i computes a quantity called U_{ij} for every one of its output channels [Chandy & Misra 81]. U_{ij} is defined as the time of the next message output on the channel from LP_i to LP_j *assuming no further message arrivals* at LP_i. (The motivation for the definition of U_{ij} is from sequential simulation, in which each LP posts into the event list the time of its next message output assuming no further message inputs.) Then $\min_{i,j} U_{ij}$ can be used as a value for $\hat{t}$.

simulation can be advanced to $\hat{t}$, and every LP_i for which $\hat{t}_i < \hat{t}$ can be awakened.

The efficiency of this procedure hinges on the algorithm used to compute $\hat{t}$. The most straightforward algorithm would be to inspect every LP in the simulation. This could be expensive when the number of LPs is large.

An alternative would be to maintain a priority queue of blocked LPs, ordered by increasing values of $\hat{C}_i$. The computation of $\hat{t}$ could then be done merely by inspecting the head of the queue. Additionally, the LPs having buffered messages timestamped earlier than $\hat{t}$ would be likely to be near the head of the queue[5]. The drawback to this method is the cost of maintaining the priority queue. Although this may seem wasteful, especially if deadlocks are infrequent, this overhead is not as large as it first appears. First of all, there are many well-known priority queue structures for which the complexity of insertion and deletion is logarithmic in the size of the queue [Aho et al. 74, Gonnet 84]. Second, note that LPs never need to be removed from the queue, and for some priority queue implementations, such as the heap, repositioning an object is much cheaper, in practice, than deleting and re-inserting it. More importantly, *an LP only needs to be repositioned when it is about to block.* This is because the queue order needs to be correct only at the time a deadlock occurs, but so long as any LP is still running, the simulation cannot be deadlocked.

There is one "catch" to this scheme: in order to guarantee that the queue is correctly ordered when a deadlock occurs, a blocked LP must be awakened whenever its $\hat{C}$ value changes. This can only happen when the LP receives a message m such that $m_i.time < \hat{t}_i$, in which case it should be awakened by the LP that sent the message.

(As we shall see later in this chapter, either of the blocking avoidance schemes discussed previously reduces the frequency of deadlocks enough so that this extra complexity in the deadlock breaking algorithm is not justified.)

[5]Note, however, that any such LP could be arbitrarily far back in the queue if $C_i \gg \hat{t}_i$.

4.5 Summary of Techniques

We identified four techniques for efficiently implementing a conservative parallel simulation in a shared memory environment:

1. A centralized scheduler eliminates the partitioning problem, and also makes deadlock detection trivial and deadlock breaking inexpensive.
2. Exact measures of progress of all LPs in the system can be obtained by inspecting shared variables, instead of relying on (possibly stale) information supplied by received messages.
3. Direct modification of channel clock values eliminates the need for explicit null messages.
4. Idle processors can be put to work checking for artificially blocked LPs.

This last observation motivated the development of a new protocol for artificial blocking and deadlock avoidance, which we call lazy blocking avoidance.

4.6 Performance of the Algorithms

In this section we present some performance measurements of the Synapse implementation of deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance. (Refer to Chapter 6 for details on the design and implementation of Synapse.) Additional measurements will be presented in the next chapter.

4.6.1 The Hardware Base

All measurements were made on a 20 processor Sequent Symmetry.[6] Each processor in the Symmetry is a 16 MHz i80386 with a private 64 Kbyte write-back cache. Processors are connected to memory via a shared bus with a maximum sustainable data transfer rate of 53.3 Mbytes/second. Our Symmetry has 32 Mbytes of memory.

[6] *Symmetry* is a trademark of Sequent Computer Corporation.

4.6.2 The Sequential Simulator

In the interest of fairness, all speedups were computed relative to a sequential simulator. This sequential simulator is "interface-compatible" with Synapse, which means that the same simulation source code can be run on either simulator. Like Synapse, the sequential simulator is process-oriented (Section 2.1). It differs from Synapse in that it keeps all pending events in a centralized heap data structure [Aho et al. 74]. Each time an LP tries to receive an event, the event heap is checked. If the earliest event in the heap belongs to the LP in question, that event is returned to the LP. Otherwise, the receive primitive performs a coroutine switch to the event's proper owner, which is, by assumption, waiting for its invocation of the receive primitive to return.

A possible criticism of our approach is that an event routine-based sequential simulator (Section 2.1) might yield better performance than our process-oriented sequential simulator. We offer the following justifications for our approach:

- An event routine-based simulator does *not* implement the same algorithm as a process-oriented simulator. In other words, a process-oriented simulator presents an abstraction to the user — communicating sequential processes — that an event routine-based simulator cannot provide. (Whether or not this abstraction is a useful one is mostly a matter of personal preference, and is not relevant in the present context.)

- The fact that all parallel simulators reported to date have been process-oriented is more a consequence of historical accident[7] than of any fundamental implementation restriction. It is easy to envision a *work-crew* based parallel simulator, in which there is a unique *worker thread* per physical processor that performs simulation tasks via calls to user-defined event routines. (In fact, such a simulator would no doubt be much easier to implement than a process-oriented one.)

[7]The earliest widely-known work in this area [Chandy et al. 79, Chandy & Misra 79, Chandy & Misra 81] suggested communicating sequential processes as a natural framework for parallel simulation.

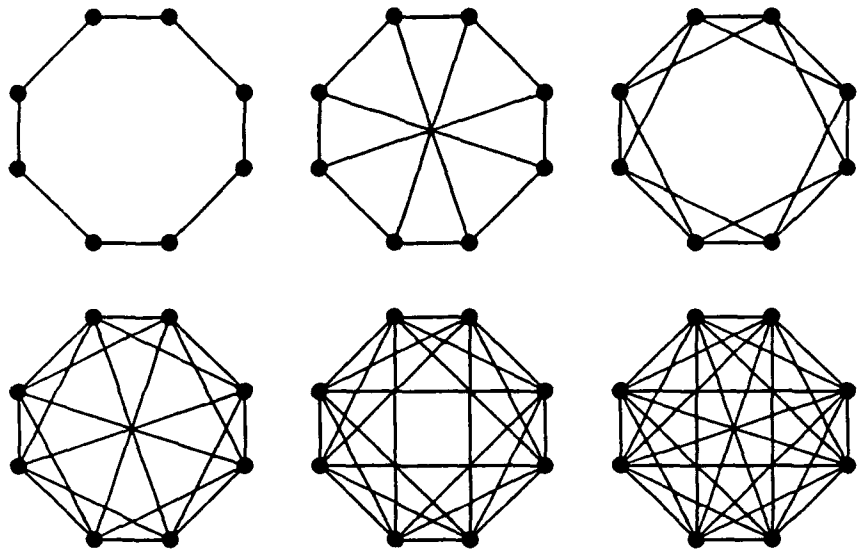


Figure 4.2: Possible topologies for an 8 server benchmark.

- Since our main goal in this section is to investigate the relative worth of various deadlock-handling and application-structuring strategies in the parallel setting, the choice of single-processor algorithm *is not critical.*

4.6.3 The Benchmark

The benchmark used for these measurements was a family of queueing network models of first-come-first-served (FCFS) servers, with each server simulated by a different LP. In all cases, the topology of the network and the routing of messages within the network were uniform, in order to avoid making any LP or group of LPs a bottleneck (Chapter 3). The topology was varied by changing the connectivity of the LPs, from a minimum of two up to a maximum of the number of LPs minus one.[8] For example, Figure 4.2 shows all topologies used for an 8 LP benchmark. Each line between pairs of LPs represents two communication channels, one in each direction.

A typical pseudo-code implementation for a FCFS server LP is shown in Figure 4.3. In this algorithm, **servicetime()** and **destination()** are functions that return pseudo-

[8]We did not simulate any networks with a connectivity of one because there would be no blocking in such a network.

```
while not finished do
    m = receive();
    case m.type of
        ARRIVAL:
            enqueue(m);
            if (idle)
                schedule_departure(clock+servicetime());
                idle = FALSE;
            endif
        DEPARTURE:
            m = dequeue();
            /* keep statistics as necessary */
            send(destination(), m, clock);
            if (customers queued)
                schedule_departure(clock+servicetime());
            else
                idle = TRUE;
            endif
    endcase
endwhile
```

Figure 4.3: A FCFS server algorithm.

random numbers according to some probability distribution.

While this is certainly a reasonable way to implement a FCFS server, it is far from optimal when used in a parallel simulation. The reason for this is that a higher level of parallelism can be obtained if customers are forwarded to their destinations as soon as their departure times are known. Since service at a FCFS server is non-preemptive, the server need only keep track of the next time at which it will become free. Then, an arriving customer's departure time can be determined by adding the customer's service time to the time at which the server will become free (which is the current clock value if the server is presently idle). Furthermore, the performance of deadlock recovery and/or avoidance can be improved by using the maximum of the clock value and next_idle_time as a measure of the LP's progress.[9]

This algorithm is shown in Figure 4.4. Note that it is may still be desirable to

[9]Using `next_idle_time` in this way is an example of exploiting model semantics to improve an LP's ability to predict its future behavior. Such techniques are the topic of the next chapter.

```
next_idle_time = 0;
while not finished do
    m = receive();
    case m.type of
        ARRIVAL:
            enqueue(m);
            if (clock > next_idle_time)
                /* server is idle */
                next_idle_time = clock+servicetime();
            else
                next_idle_time += servicetime();
            endif
            send(destination(), m, next_idle_time);
            schedule_departure(next_idle_time);
        DEPARTURE:
            /* keep statistics as necessary */
    endcase
endwhile
```

Figure 4.4: A more aggressive FCFS server algorithm.

schedule departure events, in order to simplify the calculation of statistical measures such as server utilizations and queue lengths.

It turns out that, when simulated sequentially, this algorithm is slower than the previous one because there are more events in the event list at any given time. Therefore, to keep things honest we have used the first version of the algorithm in all sequential simulations, and the second version in all parallel simulations.

4.6.4 Results

Unless otherwise noted, the results presented here are for networks that are fully interconnected, i.e., that have a connectivity that is one less than the number of LPs. We favored this topology because it is most likely to be a worst-case scenario for blocking avoidance protocols. The number of messages in the network is expressed as a *message density*, defined as the total number of messages divided by the number of communication channels (the number of LPs multiplied by the connectivity of each LP). Unless otherwise noted, message density for all experiments was one.

Each simulation was run until every LP had processed 5000 messages. Because

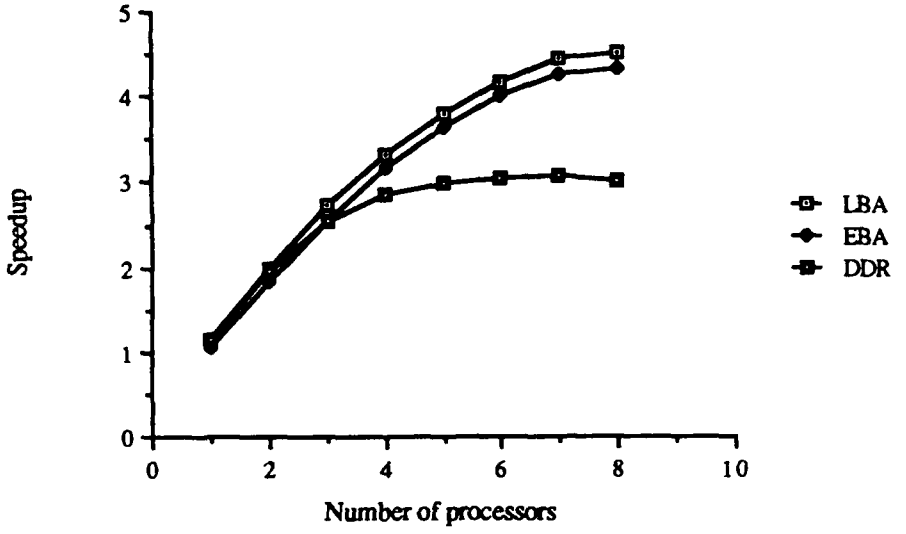


Figure 4.5: Speedup vs. number of processors, 8 node fully-interconnected network, deterministic service times.

of the stochastic nature of the benchmarks, the elapsed time of the simulation varied depending on the random number seed that was used. The data in our graphs represent the average of measurements from four parallel simulation runs, each using a different random number seed. (For speedup calculations, eight simulation runs were involved: four sequential and four parallel.) In all legends, "LBA" signifies lazy blocking avoidance, "EBA" signifies eager blocking avoidance, and "DDR" signifies deadlock detection and recovery.

Figure 4.5 shows speedup as a function of the number of processors used for a fully-interconnected network with 8 servers. The message density of this benchmark is one. Service times distributions at all servers are deterministic with identical means. Note the poor performance of deadlock detection and recovery relative to both blocking avoidance strategies.

(Although the fact that speedup with one processor is slightly greater than one may appear suspicious, it is a consequence of the use of different algorithms for the parallel and sequential simulators. This apparent anomaly is due to at least two factors:

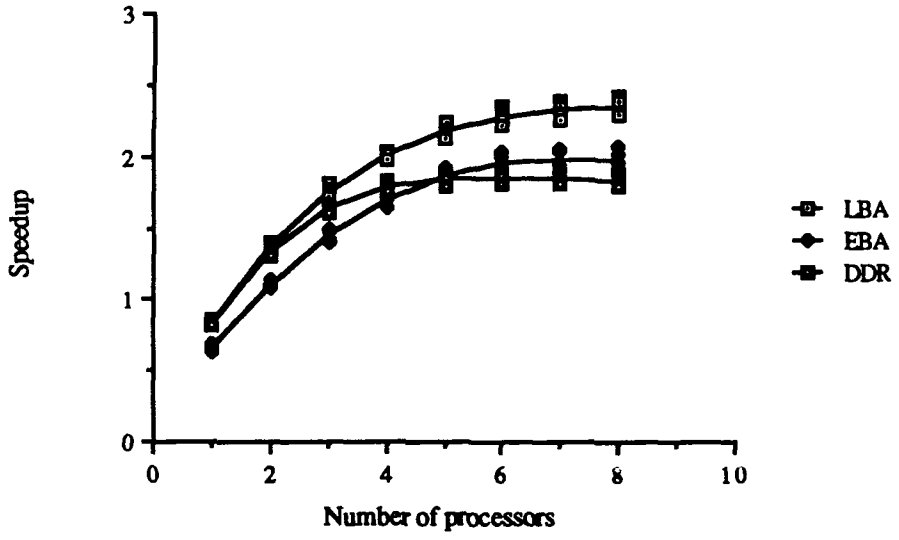


Figure 4.6: Speedup vs. number of processors, 8 node fully-interconnected network, exponential service times.

first, the overhead of sending a message is smaller in the parallel simulator than in the sequential simulator (Section 4.1). Second, since the parallel simulator does not have to execute events in strict timestamp order, it context switches less frequently than does the sequential simulator. As we shall see, this situation is the exception to the rule: in most cases the parallel simulator running on a single processor does not perform as well as the sequential simulator.)

Figure 4.6 is for a benchmark that is identical to the previous one in all respects except that the service time distribution at the servers is exponential. This figure also depicts the typical variability of our measured speedups by plotting individually the four speedup values that comprise each average value. (For the sake of clarity, in all other graphs we plot only the average of the four values.)

Performance in this case is much worse than before (note that the y-axis scales differ). What is the reason for such a dramatic difference? We propose that when service times are constant, there are many events scheduled for execution within any small interval of simulation time, so the LPs tend to progress almost synchronously

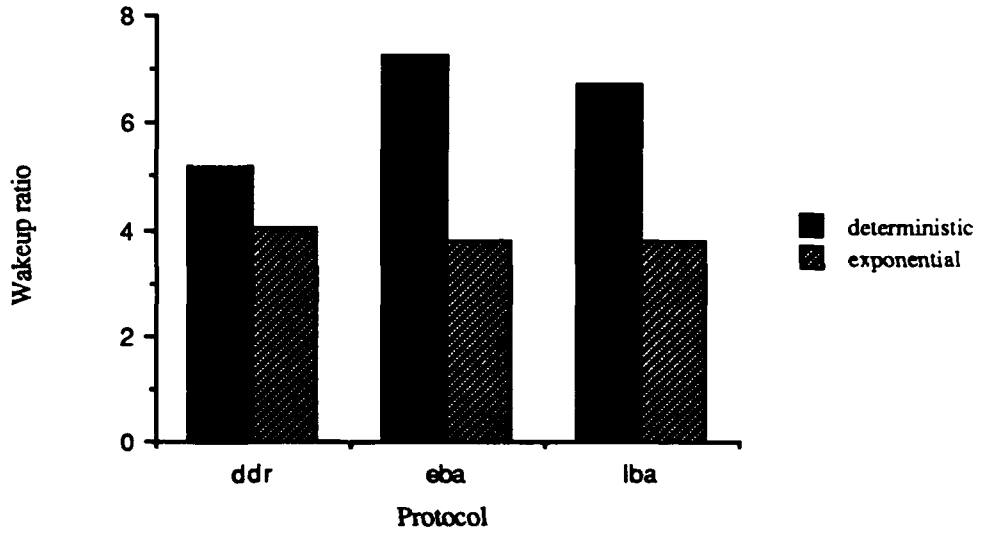


Figure 4.7: Wakeup ratio vs. protocol, 8 node fully-interconnected network.

through the simulation.

This hypothesis is supported by Figure 4.7, which shows the difference in *wakeup ratio* (the average number of LPs awakened per deadlock recovery) between the two benchmarks when 8 processors are used. A large wakeup ratio means that many LPs are being awakened during each deadlock recovery (Section 4.4), which indicates that there are many events scheduled close together in simulated time. Note that when blocking avoidance is used and service times are deterministic, nearly every LP is awakened each time a deadlock is broken, indicating that the LPs are staying fairly close together in simulated time. Because of this phenomenon, LPs are less likely to block, and speedup is improved. With a highly variable service time distribution such as the exponential, the LPs are more "scattered" in simulated time and hence they block more often.

Despite the large difference in absolute performance between the two benchmarks, the performance of the three protocols relative to each other is qualitatively consistent across the benchmarks. Lazy blocking avoidance performs *at least as well* as either of the other alternatives across the entire range of physical parallelism. At high levels of physical parallelism, both blocking avoidance techniques outperform deadlock detection

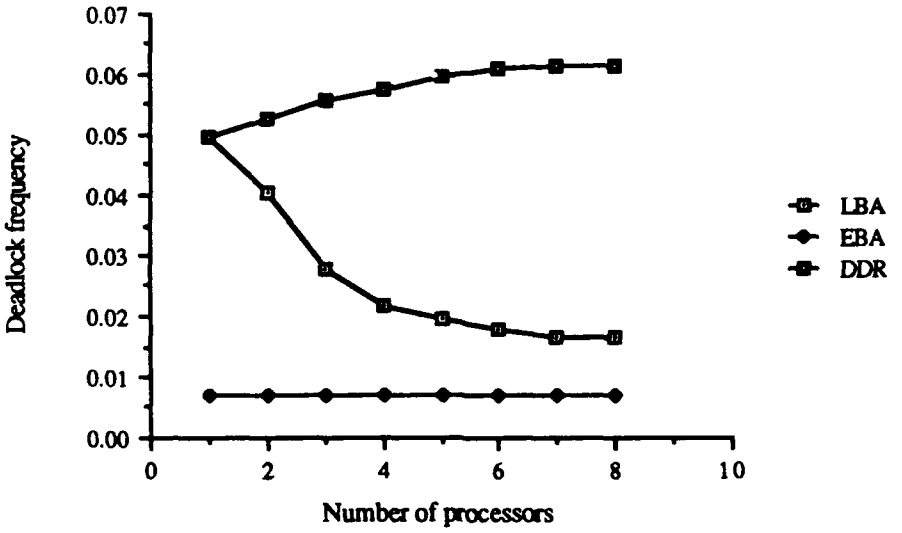


Figure 4.8: Deadlock frequency vs. number of processors, 8 node fully-interconnected network, deterministic service times.

and recovery, because there is a surplus of processors over logical parallelism. However, the virtue of lazy blocking avoidance is that it incurs no performance penalty when the number of processors in use is smaller than the average level of logical parallelism in the simulation. (As a special case, when only a single processor is used lazy blocking avoidance is equivalent to deadlock detection and recovery because there is never an idle processor available to run the protocol.) On the other hand, eager blocking avoidance is actually a liability rather than an asset at low levels of physical parallelism — in the case of exponential service times, it performs worse than deadlock detection and recovery until at least *five processors* are available.

To graphically illustrate the increasing effectiveness of lazy blocking avoidance as the number of processors increases, Figure 4.8 plots *deadlock frequency* (the number of deadlocks divided by the total number of messages processed in the simulation) as a function of the number of processors, for deterministic service times. Note the dramatic decrease in deadlock frequency when using lazy blocking avoidance as processors are added. This is due to the fact that increasing the number of processors increases the

percentage of time that each processor is idle, and hence more processing power is devoted to the lazy blocking avoidance protocol.

A rather counter-intuitive phenomenon also appears in Figure 4.8: deadlock frequency for deadlock detection and recovery is an *increasing* function of the number of processors used. This is because "missed" clock advances are causing false deadlocks. In other words, if LP_a is unable to receive a message from LP_b because LP_c is too far behind, and subsequently LP_c advances its clock, LP_a will not find this out unless one of the other LPs sends it a message. (This observation makes a compelling argument for the use of deadlock avoidance.) With fewer processors, this is less likely to happen because the execution of events is serialized.

In contrast, the deadlock frequency for eager blocking avoidance is steady across the entire range of physical parallelism. The reason for this is that the strategy of eager blocking avoidance is to avoid deadlocks whenever possible, so the deadlock detection mechanism is disabled as long as any LP is still propagating clock information. Therefore, no matter what the order of LP execution in real time, the set of simulation states in which deadlock occurs is completely pre-determined by the initial conditions of the simulation.

Figures 4.9 and 4.10 show the effect of message density on the speedup of an 8 node fully-interconnected network for service times that are deterministic and exponential, respectively. These graphs corroborate our intuition that increasing the number of messages in the network ought to decrease the amount of blocking, since it becomes less likely that an LP has any empty input channels. This effect is so important that it diminishes the effect of the service time distribution on performance.

(The most likely reason for the downturn in the graphs at very high message densities is that the event list data structure used in the sequential simulator (a heap) has a smaller asymptotic complexity than the structure that is used in the parallel simulator (a collection of linear lists).)

A corollary of the previous result is that if the number of channels in the simulation is

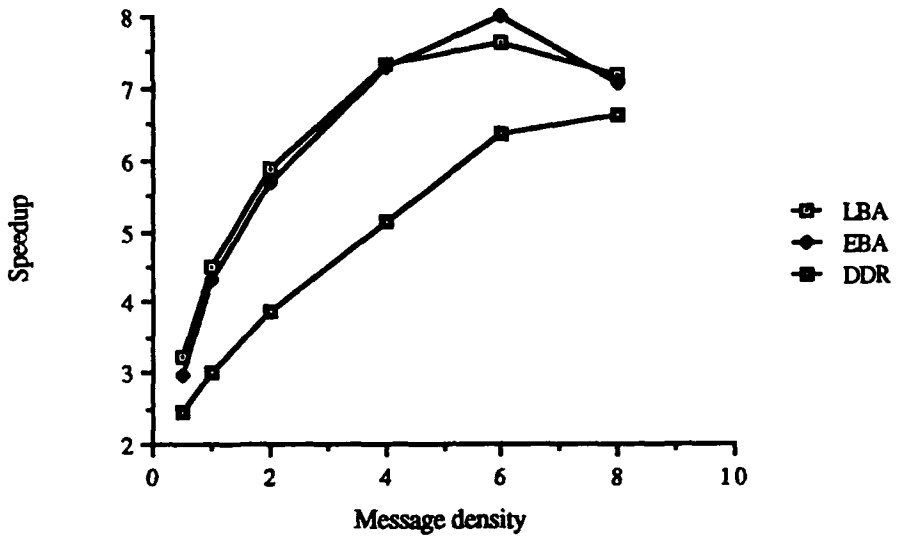


Figure 4.9: Speedup vs. message density, 8 node fully-interconnected network, deterministic service times.

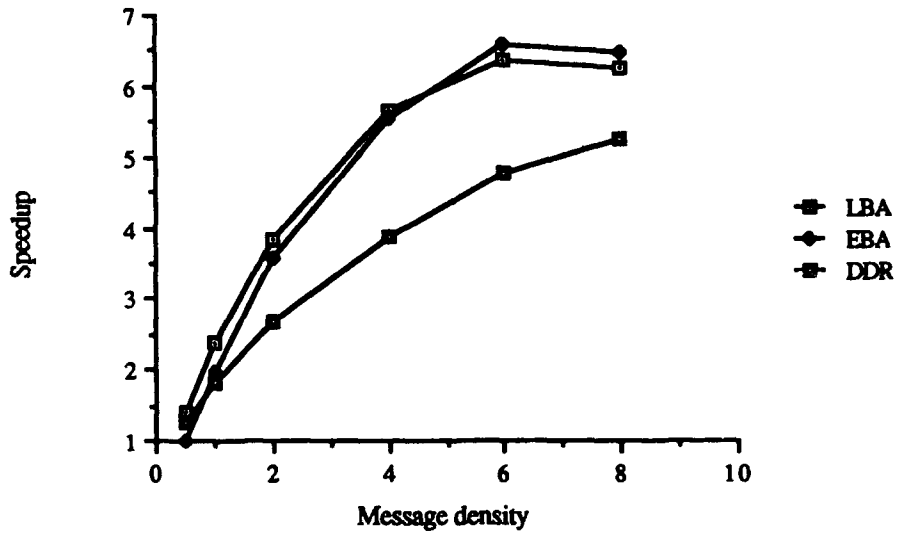


Figure 4.10: Speedup vs. message density, 8 node fully-interconnected network, exponential service times.

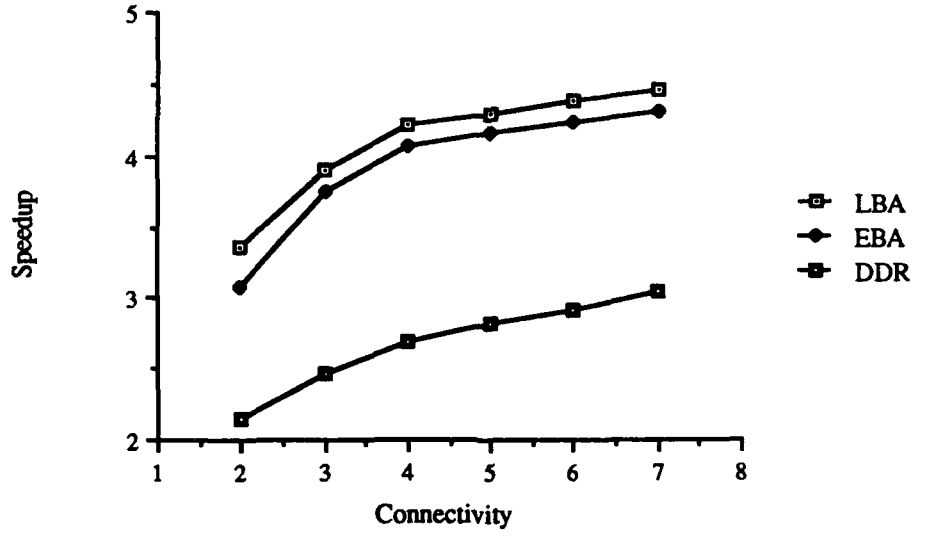


Figure 4.11: Speedup vs. connectivity, 8 node network, message density one, deterministic service times.

increased without also increasing the number of messages, performance will suffer. This brings up an obvious question: what will be the effect on performance of an increase in connectivity if the average number of messages per channel (the message density) is held constant? Figures 4.11 and 4.12 show the effect of connectivity on the speedup of an 8 node network with a message density of one, for service times that are deterministic and exponential, respectively. Speedup seems to be an increasing function of connectivity. This is somewhat counter-intuitive, for the following reason: even though the average number of message per channel is the same in all cases, the probability that n out of n input channels at the same LP all have at least one message is significantly smaller than the probability that $n-1$ out of $n-1$ input channels all have a message. Thus, we would expect there to be an increase in blocking as connectivity is increased.

We can explain our results by noting that increasing the number of messages in the network increases the average number of buffered messages at each LP. Because of this, more LPs are awakened each time a deadlock is broken, thus reducing the frequency of deadlock. Figures 4.13 and 4.14 support this hypothesis. They show the wakeup ratio

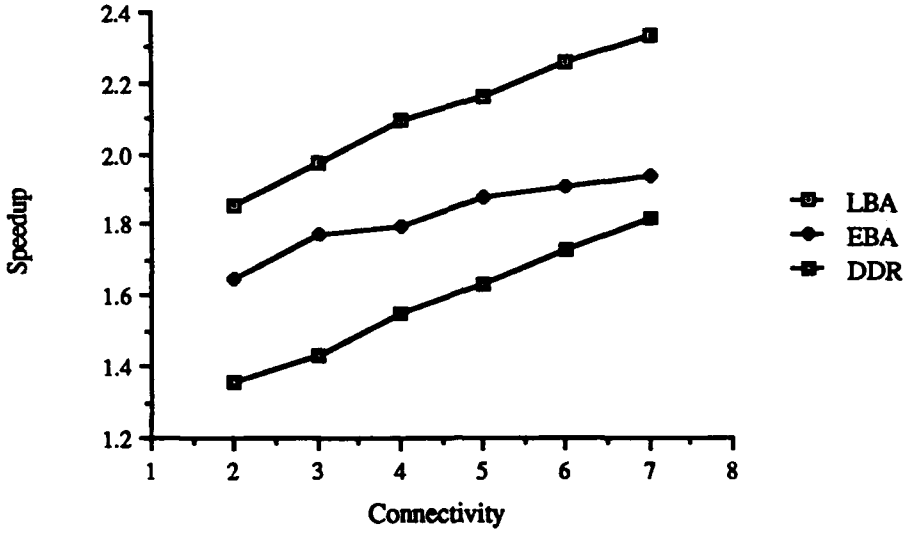


Figure 4.12: Speedup vs. connectivity, 8 node network, message density one, exponential service times.

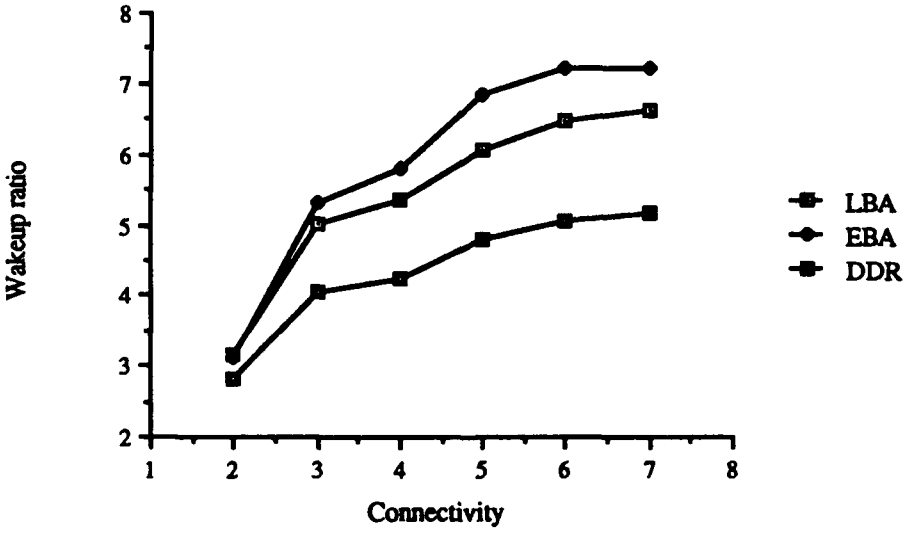


Figure 4.13: Wakeup ratio vs. connectivity, 8 node network, deterministic service times.

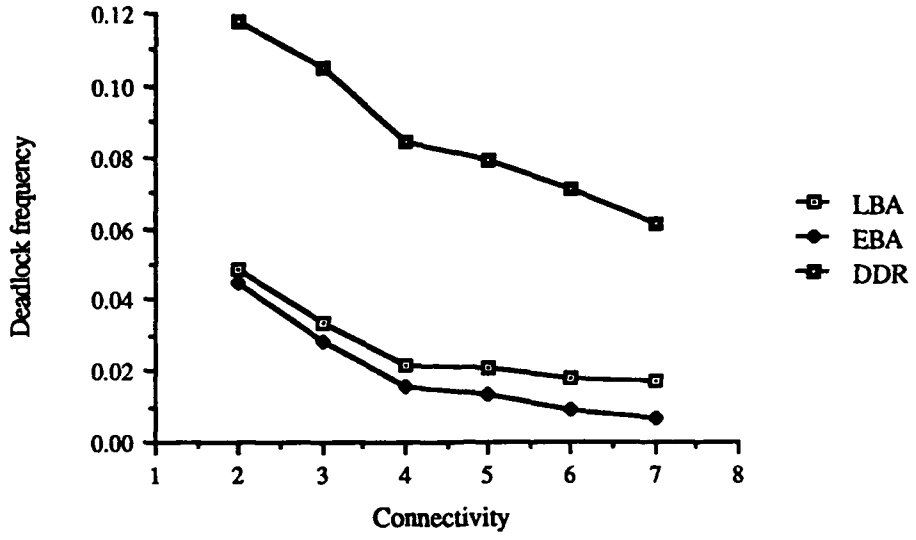


Figure 4.14: Deadlock frequency vs. connectivity, 8 node network, deterministic service times.

and deadlock frequencies for the benchmark with deterministic service times. As connectivity increases, wakeup ratio increases and consequently deadlock frequency decreases. (The results for exponential service times are similar.) In the next chapter we will see that when deadlocks are completely eliminated, speedup does not increase as connectivity increases. However, the overall gain in performance resulting from the elimination of deadlocks more than compensates for this.

Returning to Figures 4.11 and 4.12, we find another interesting phenomenon. The performance of eager blocking avoidance seems to be much more sensitive to connectivity in the exponential case than in the deterministic case. In the deterministic case, eager blocking avoidance tracks the performance of lazy blocking avoidance quite closely, but in the exponential case the difference between the two protocols increases with connectivity. This is because the overhead of deadlock avoidance is increased as connectivity is increased; this overhead is felt much more keenly by eager blocking avoidance than by lazy blocking avoidance.

4.6.5 Conclusions

The measurements presented in this section have demonstrated that the performance of conservative parallel simulation is dependent on a number of interacting factors such as message timestamp increment distributions, deadlock frequency, wakeup ratio, and the overhead of deadlock avoidance.

The variability of message timestamp increments was shown to have a profound effect on performance. Highly variable timestamp increments caused the LPs to be "scattered" in simulated time, increasing blocking and thus deadlock frequency. A secondary effect of this scattering is that when deadlocks did occur, fewer LPs could be awakened. Evidence for this was provided by our measurements of deadlock frequency and wakeup ratio. (In the case of deterministic service times, nearly every LP was awakened each time a deadlock was broken, indicating that the LPs were fairly close together in simulated time.)

In general, our measurements have also shown that it is necessary to use some form of blocking avoidance in order to achieve reasonable performance. The effect of artificial blocking on the performance of deadlock detection and recovery was illustrated by the fact that deadlock frequency was an *increasing* function of the number of processors used. As the level of physical parallelism is increased, the likelihood that clock advances are "missed" increases. This causes the LPs that miss the advances to remain blocked, possibly leading to false deadlocks.

At high levels of physical parallelism, both lazy and eager blocking avoidance outperform deadlock detection and recovery, because there is a surplus of physical parallelism (processors) over logical parallelism. However, at low levels of physical parallelism the overhead of the blocking avoidance protocol becomes an important factor. Our technique, lazy blocking avoidance, successfully balances these considerations. The virtue of lazy blocking avoidance is that it incurs no performance penalty when the number of processors in use is smaller than the average level of logical parallelism in the simulation. (The increasing effectiveness of lazy blocking avoidance as more processors are added

was reflected in our measurements of deadlock frequency.) On the other hand, eager blocking avoidance is actually a liability rather than an asset at low levels of physical parallelism, especially when service times are highly variable.

We also noted that deadlock frequency for eager blocking avoidance was unaffected by the number of processors used, because eager blocking avoidance disables the deadlock detection mechanism until all clock information has been propagated. In this sense, eager blocking avoidance has a deterministic flavor.

As expected, speedup was an increasing function of message density. Unexpectedly, speedup was an also an increasing function of connectivity when message density was held constant. The increase in speedup seemed to be the result of an increase in the wakeup ratio.

4.7 Chapter Summary

A central tenet of our thesis is that the availability of shared memory mandates a re-examination of traditional approaches to conservative parallel simulation. Towards this end, we began this chapter by examining the ramifications of distributed memory and shared memory on the implementation of conservative parallel simulation.

We identified four techniques for efficiently implementing a conservative parallel simulation in a shared memory environment:

1. A centralized scheduler eliminates the partitioning problem, and also makes deadlock detection trivial and deadlock breaking inexpensive.

2. Exact measures of progress of all LPs in the system can be obtained by inspecting shared variables, instead of relying on (possibly stale) information supplied by received messages.

3. Direct modification of channel clock values eliminates the need for explicit null messages.

4. Idle processors can be put to work checking for artificially blocked LPs.

This last observation motivated the development of a new protocol for artificial blocking and deadlock avoidance, which we call lazy blocking avoidance.

All of the above techniques have been implemented in our prototype parallel simulator, Synapse. Measurements of Synapse on a queueing network benchmark were presented to validate the efficacy of these techniques.

Our measurements have shown that the performance of conservative parallel simulation is dependent on a number of interacting factors such as message timestamp increment distributions, deadlock frequency, wakeup ratio, and the overhead of deadlock avoidance. Two of the more notable effects are:

- The variability of message timestamp increments has a profound effect on performance. Highly variable timestamp increments cause the LPs to be "scattered" in simulated time, increasing blocking and thus deadlock frequency. They also impede the effectiveness of deadlock recovery, by decreasing the wakeup ratio.

- The use of deadlock avoidance has the potential to be a big win, but can also be detrimental to performance if it excludes the early detection of deadlocks. Our technique, lazy blocking avoidance, successfully balances these two considerations.

Chapter 5

Exploiting Model Semantics

In this chapter we explore methods to improve speedup by exploiting the semantics of the problem being simulated. Probably the single most important technique in this domain is the calculation of *lookahead* [Misra 86, Fujimoto 88a, Nicol 88].

We begin with a review of the lookahead concept and some intuitive motivation for why it is important. We corroborate our intuition with measurement data from Synapse.

Next, we explore methods for calculating lookahead for LPs simulating queueing network servers with various types of service disciplines. (We focus on queueing network servers because many systems can be modeled as a network of queues (a queueing network model, or QNM [Kleinrock 75, Lazowska et al. 84]). Queueing network models are widely used in computer performance evaluation, operations research, industrial engineering, and traffic analysis.) We describe the seminal work in this area, Nicol's *future list* technique [Nicol 88], and we extend the future list technique to classes of service disciplines not considered in that study.

5.1 Lookahead

Lookahead characterizes the ability of an LP to predict future actions that it will take [Misra 86, Fujimoto 88a]. More precisely, if at real time τ an LP's simulation clock time

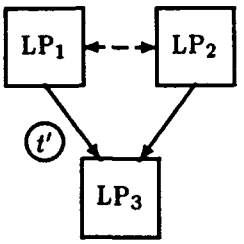


Figure 5.1: A lookahead scenario.

is t, then the lookahead, denoted by $L(\tau, t)$, is the largest value such that at real time τ the LP can predict all actions that the physical process it is simulating will take up to simulation time $t + L(\tau, t)$. Lookahead is a function of real as well as simulated time because it may be affected by events that do not change the LP's clock time, such as the arrival of a message that cannot be consumed immediately.[1]

We have already seen one simple example of lookahead, namely, the prediction of customer departure times used by the FCFS server LP that was introduced in the previous chapter (Section 4.6.3).

Although "bigger is better" where lookahead is concerned, large lookahead values do not in and of themselves guarantee good performance. We claim that in order to be useful, lookahead must be comparable to the expected amount by which LPs increment the timestamps of messages that they process. For example, consider the scenario shown in Figure 5.1. Let the clock values of LP_1 and LP_2 at real time τ equal $C_1(\tau)$ and $C_2(\tau)$, respectively. Assuming that LP_1 and LP_2 progress through simulation time at the same rate with respect to real time, we would expect that $C_1(\tau) \approx C_2(\tau)$.[2] Now suppose that LP_1 has just processed a message with timestamp $t = C_1(\tau)$ and forwarded it to LP_3 with a new timestamp $t' = C_1(\tau) + \delta$. LP_3 will be able to consume this message if t_{23}, the channel clock value of the channel from LP_2 to LP_3, is greater than or equal to t'. Assuming that t_{23} is up to date, $t_{23} = C_2(\tau) + L_2(\tau, C_2(\tau))$, that is, the channel clock

[1] For convenience of notation, in the remainder of this dissertation the dependence of L on τ and t will be implicit rather than explicit.

[2] This would seem to be especially true if LP_1 and LP_2 are in the same strongly connected component of the simulation graph, as suggested in the figure.

value is equal to the channel source's local clock plus the channel source's lookahead. Therefore, LP_3 will be able to consume the message from LP_1 if

$$C_2(\tau) + L_2(\tau, C_2(\tau)) \geq C_1(\tau) + \delta$$

Since we expect $C_1(\tau) \approx C_2(\tau)$, LP_2's lookahead will be of no use unless it is of the same order of magnitude as δ, the amount of simulated time by which the message's timestamp was incremented when it was processed by LP_1. As expected lookahead increases relative to the expected message timestamp increment, blocking should decrease and performance should improve. Thus, we expect *lookahead ratio (LAR)*, which is defined as the ratio of the mean message timestamp increment to the mean lookahead [Fujimoto 88a], to be inversely related to speedup.[3]

To see the importance of lookahead ratio, we enhanced the FCFS server LP shown in Figure 4.4 in the following way: after the server being simulated by an LP has forwarded all of its customers, the LP sets the channel clock values on all of its output channels to the sum of `next_idle_time` (*the time at which the server is known to become idle*) plus the minimum possible service time of the server. This serves to inform the other LPs that no further messages will be forthcoming from the given LP until at least that time.

Figure 5.2 shows the effect of lookahead ratio on the speedup of a benchmark composed of modified FCFS LPs. The benchmark used here is identical to the benchmark introduced in the previous chapter (a fully-interconnected network of 8 FCFS servers and a message density of one) and was run on 8 processors in all cases. Service times at each server are obtained from a biased exponential distribution: that is, an exponential random variable plus a constant (the bias). Lookahead ratio can be varied over any finite range by holding the bias fixed and varying the mean of the exponential. A lookahead ratio of ∞ is obtained by setting the bias to zero. (Note that in this case, the speedups in the graph are identical to the speedups shown in the last chapter for the exponential

[3]Although it might seem more intuitive to define LAR so that it is *directly* related to speedup, i.e., as the reciprocal of the definition given here, we use this definition for consistency with Fujimoto's work.

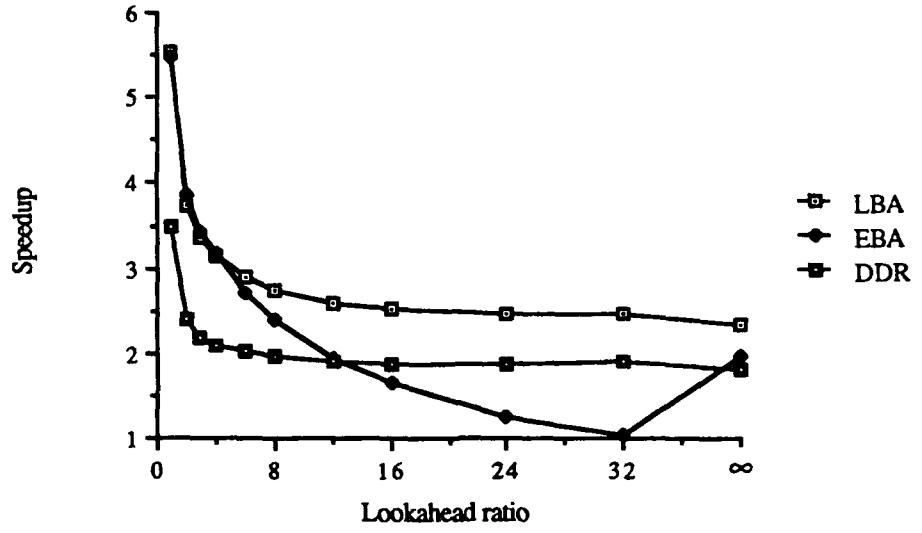


Figure 5.2: Speedup vs. lookahead ratio, 8 node fully-interconnected network, minimum service time lookahead.

distribution (Figure 4.6).)

As expected, speedup is a monotonically decreasing function of lookahead ratio,[4] with a dramatic drop going from LAR = 1 to 2 and then a more gradual decline all the way out to LAR = ∞. As was demonstrated on the model of the previous chapter, when LAR = 1 (i.e., service times are deterministic) the LPs tend to progress synchronously in simulated time, which reduces blocking and increases the wakeup ratio (the average number of LPs awakened by a deadlock recovery). Thus, deadlocks are less likely to happen in the first place, and when they finally do, a large number of LPs are enabled to receive a message.

As LAR continues to increase, the performance of lazy blocking avoidance and deadlock detection and recovery quickly level off, but eager blocking avoidance displays unexpected behavior: performance continues to decline all the way out to LAR=32, but then increases at LAR = ∞. Denote the speedup of the eager blocking avoidance protocol when LAR = x by $S_{\text{eba}}(x)$. We contend that $\lim_{x\to\infty} S_{\text{eba}}(x) = 0$, although clearly

[4]The single exception, eager blocking avoidance with LAR=∞, will be discussed shortly.

$S_{\text{cba}}(\infty) > 0$. Why should this be the case? It is because of eager blocking avoidance's insistence on exhausting all efforts at deadlock avoidance before allowing the deadlock breaking mechanism to operate. So long as LAR is finite, which is just another way of saying that there is a positive lower bound on service times, eager blocking avoidance will eventually succeed in enabling the receipt of a message at some LP. "Eventually" is the operative term here; as lookahead ratio increases, it takes longer and longer for eager blocking avoidance to do its job. However, when LAR = ∞ there is no lookahead (service times have no positive lower bound). Thus, clock values quickly reach a fixed point, eager blocking avoidance gives up, and the deadlock recovery mechanism saves the day.

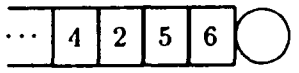
This points out a key philosophical difference between lazy and eager blocking avoidance: lazy blocking avoidance considers its job to be to unblock an LP whenever the opportunity presents itself, but to defer to the deadlock detection mechanism whenever possible. This casual attitude seems to be worthwhile, as the performance of lazy blocking avoidance has the robustness of deadlock detection and recovery while managing to maintain a comfortable superiority over it across the entire range of lookahead ratios.

The preceding discussion should help to motivate the exploration of lookahead calculation. In this next few sections we will analyze the lookahead characteristics of various types of queueing network servers and develop techniques for exploiting them.

5.2 First-Come-First-Served Servers

Consider an LP simulating a first-come-first-served (FCFS) server. Under FCFS scheduling, customers are served in the order in which they arrive. Therefore, at any given point in real time τ, it is possible for an LP simulating a busy FCFS server to predict the departure times of all customers that have arrived up to and including time τ, yielding excellent lookahead.

An example of FCFS lookahead is depicted in Figure 5.3. In this example, it is assumed that the server has three possible destinations, A, B, and C. Each customer



Destination	Lookahead
A	6
B	11
C	17

Figure 5.3: Example of FCFS lookahead.

in the queue is represented by its service time, and the destination of each customer is shown directly beneath its position in the queue.

As we saw in the last chapter, an LP simulating a FCFS server can actually forward customers before its local clock has reached their time of departure, effectively sending messages "into the future". In the example just presented, once all customers in the queue have been forwarded, the lookahead for all destinations is equal to the maximum lookahead, which is 17.

Unfortunately, if the server is idle then the LP can make only very weak predictions about the future behavior of the server. In the worst case, a customer could arrive immediately with an extremely small service time. Therefore, the lookahead prediction for all destinations is equal to the smallest possible service time for any customer at this server. Figure 5.2 showed the performance of Synapse when this simple capability was added to the FCFS LP implementation. Performance decreases rapidly as lookahead ratio increases, although it does eventually level off for deadlock detection and recovery and lazy blocking avoidance.

Typically, a logical process calculates lookahead based on service times and destinations for messages that have already been consumed by the LP. However, Nicol made the crucial observation [Nicol 88] that for queueing network models, it is typically the case that service times (and routing choices) at a server are determined by some probability distribution that is associated with the server. If this is indeed the case, then the

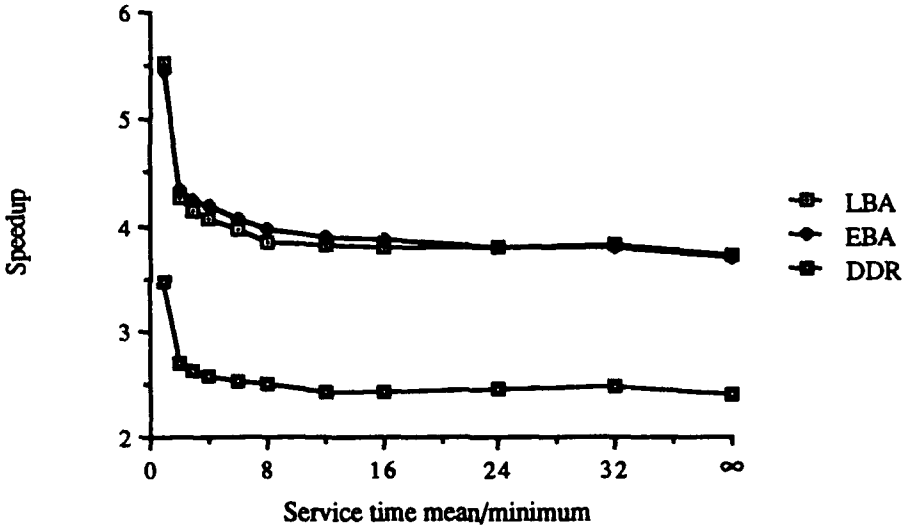


Figure 5.4: Speedup vs. mean/minimum service time, 8 node fully-interconnected network, single pre-sample lookahead.

service time and destination of a customer (message) can be determined by sampling these distributions *before* the customer actually does arrive. (Note that knowledge of the characteristics of yet-to-arrive messages is not limited to stochastic simulations. For example, if the simulation is trace driven, this information can be extracted from the trace data [Lin et al. 89].)

An important benefit of this technique is that there is no need for service times to be bounded away from zero in order to avoid deadlock. Moreover, even if such a bound existed, pre-sampling would be expected to improve performance, since the pre-sampled values would sometimes be significantly larger than the lower bound on service times.

The efficacy of this simple technique is demonstrated by the speedup curves in Figure 5.4, which were obtained by enhancing the Synapse FCFS LP to pre-sample service times exactly once. Speedup is now nearly insensitive to the ratio of mean to minimum service times (the exception once again being when this ratio is one) for each of the protocols, since the *expected* lookahead value used by an LP is the same (the mean of the service time distribution) no matter what the minimum service time. In other words,

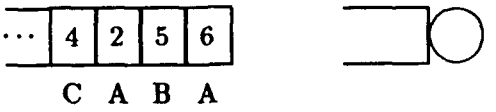


Figure 5.5: Calculation of lookahead using the future list.

lookahead ratio is unity in all cases.

In view of this fact, it is puzzling that the speedup curves should not be completely flat. The reason for the slow decline in performance (for LAR ≥ 2) is that the *variance* of the service time distribution, and hence of the lookahead values, is increasing as LAR increases. This increases the likelihood that at any given time, some LP has lookahead that is much poorer than average, which will impede the progress of all LPs that can receive messages from the given LP. In fact, the benchmark shown here ought to represent a worst case for this particular effect, since the LPs are fully interconnected. Thus, a single LP with poor lookahead will slow down the entire simulation.

In order to compute lookahead on a per-destination basis, it is necessary to pre-sample {*service time, destination*} pairs until the sampled destination matches the one for which a lookahead value is being computed. The statistical integrity of the simulation is preserved by keeping the pre-samples in a queue called the *future list* [Nicol 88]. Then, when a message is received by the LP, the arriving customer's service time and destination are taken from the head of the future list. (This implies that once the future list contains at least one message for every possible destination, further pre-sampling is unnecessary until another arrival occurs.)

Figure 5.5 depicts the calculation of lookahead using this method. In this example, the lookahead obtained is just as good as it was in the example in Figure 5.3, even though the server is idle.

If only a single pre-sample is used, the expected lookahead is obviously equal to μ, the mean of the service time distribution, and hence the lookahead ratio is unity. On the other hand, if lookahead is computed on a per-destination basis, then one destination

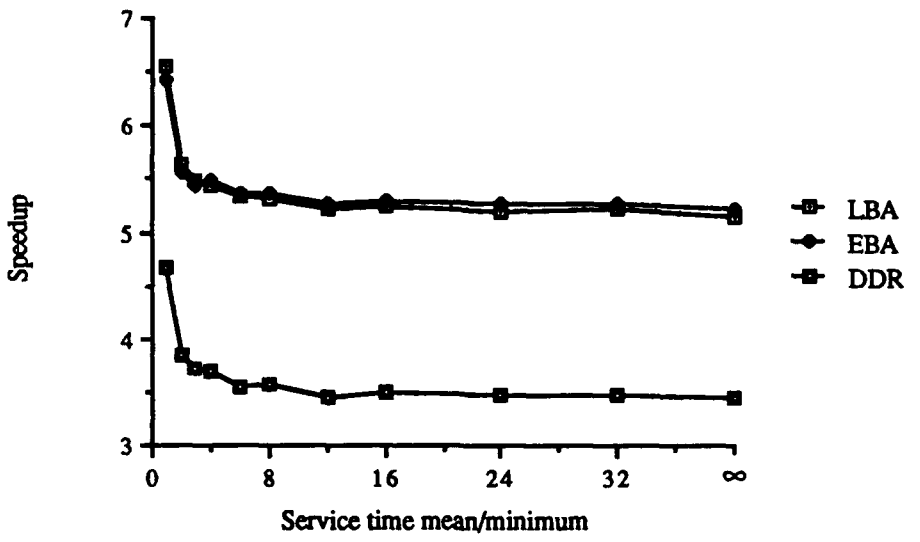


Figure 5.6: Speedup vs. mean/minimum service time, 8 node fully-interconnected network, future list lookahead.

will have a lookahead as large as in the single-sample case, and lookahead for every other destination will be even larger. To get a feeling for how much larger, assume that the probability that a customer is routed to destination d is equal to p_d (independent of previous choices). Then a simple conditional expectation calculation shows that the expected lookahead for destination d is μ/p_d. Therefore, the lookahead ratio for destination d is p_d, which may be significantly smaller than one.

Figure 5.6 shows that, indeed, the full future list improves performance a great deal compared to the single pre-sample technique. Speedup is even less sensitive to changes in the service time distribution than before.

To facilitate an overall comparison of the techniques presented so far, Figure 5.7 shows the performance of lazy blocking avoidance on our benchmark for each of the four lookahead alternatives: zero lookahead, minimum service time, single pre-sample, and complete future list. Figure 5.8, which is the same comparison for a benchmark that is twice as large (16 nodes and a connectivity of 15), demonstrates the scalability of these techniques.

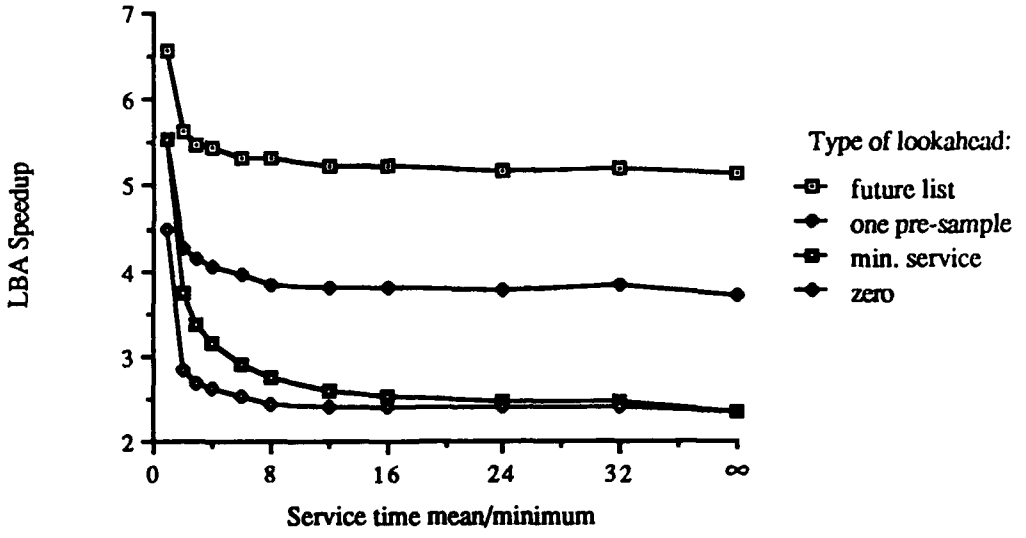


Figure 5.7: Comparison of the four lookahead alternatives, 8 node fully-interconnected network, lazy blocking avoidance.

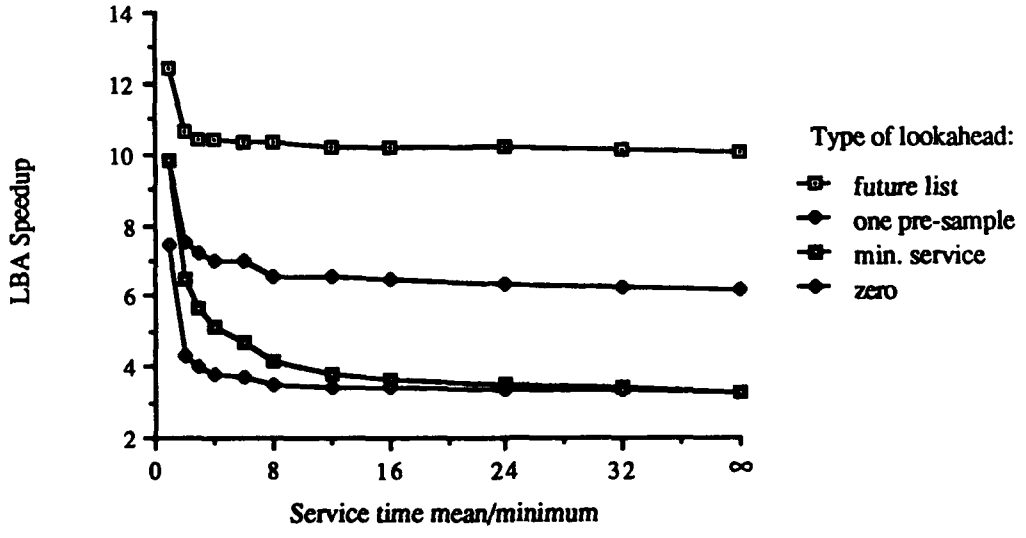


Figure 5.8: Comparison of the four lookahead alternatives, 16 node fully-interconnected network, lazy blocking avoidance.

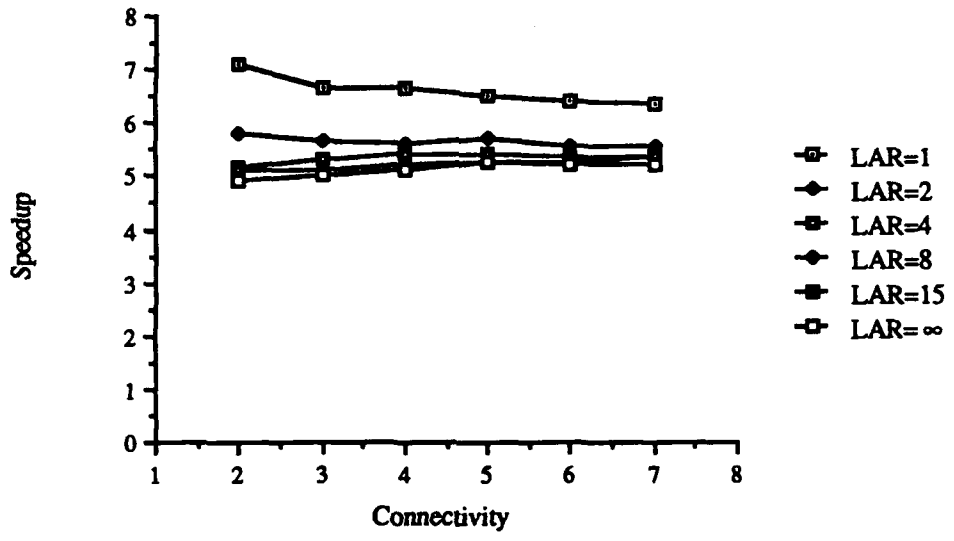


Figure 5.9: Speedup vs. connectivity, 8 node network, eager blocking avoidance, message density 1, future list lookahead.

Figure 5.9 shows speedup as a function of connectivity for an 8 node network using eager blocking avoidance and the future list. Each line in the graph represents the results of using a different service time distribution; a line labeled "LAR x" means that the ratio of the mean of the service time distribution to the minimum service time was x.

In the previous chapter, we observed that when lookahead was poor, speedup was an increasing function of connectivity (for a constant message density) because of the presence of deadlocks. That is, as the connectivity increased, and the total number of messages in the simulation with it, there were more LPs awakened after each deadlock and so the simulation had a higher level of average parallelism. Using the future list and eager blocking avoidance, however, deadlocks are completely eliminated, which improves speedup substantially. Moreover, it makes speedup essentially insensitive to connectivity. Thus, we conclude that when the future list technique is used, performance scales not only with the number of LPs but with LP connectivity as well.

Refer to Nicol's paper for the results of applying the FCFS future list technique to a wide variety of other FCFS queueing networks and service time distributions [Nicol 88].

5.3 Extensions of the Future List Technique

Whereas previous research concentrated on methods for *propagating* lookahead values, Nicol's work was the first to treat the *computation* of lookahead as a separate concern. However, that study was limited to the FCFS queuing discipline. The non-preemptive semantics of FCFS servers make them well suited to the calculation of lookahead; it is not clear that servers with other queueing disciplines can benefit as well.

In this section we extend the future list technique to other common queuing disciplines, and we calculate the resulting expected lookahead for some special cases. We also present measurements of an implementation of some of these techniques using Synapse.

5.3.1 Multiple Class Preemptive Priority Servers

Suppose there are C customer classes in the QNM, numbered $1 \ldots C$. A multiple class priority-based server schedules customers FCFS within each class, subject to the condition that all queued customers belonging to class i are served before any customers belonging to class j, if $i < j$. In addition, if scheduling is preemptive, a customer belonging to class j will be preempted from service if a customer belonging to class i, $i < j$, arrives while the class j customer is in service. We refer to such servers as multiple class preemptive priority (MCPP) servers.[5]

Suppose that a customer belonging to class j is in service at an MCPP server, and its residual service time is r_j. If the server is non-preemptive, then clearly the next customer to depart from the server will be the one in service, and hence the lookahead is r_j. On the other hand, if the server is preemptive, then the next customer to depart from the server might not be the one that is currently being served, since a higher priority customer could arrive in less than r_j time units. For each class i, let the lower bound on service times for that class be a_i. The worst case scenario, in terms of lookahead, would be if a higher priority customer, say, a class i customer, were to arrive immediately.

[5]There is also a choice of whether to restart or resume the preempted customer when the server becomes available again; for our purposes, it does not matter.

Since the service time of such a customer is unknown, a departure could take place in as little as a_i time units. Since a customer of any class $i < j$ can preempt the customer in service, then the lookahead, which must be a lower bound on the time until the next departure, is given by

$$L_j = \min(\{r_j\} \cup \{a_i : i < j\}) \tag{5.1}$$

If there is no customer currently in service, then Equation (5.1) still holds if we define j and r_j to be $C + 1$ and ∞, respectively.

Lookahead for an MCPP server can be improved by implementing a *future array* that is indexed by customer class. We consider first the calculation of a single lookahead value for all destinations, which we then extend to the case of per-destination lookahead.

The future array always contains one sample from the service time distribution of each customer class. Each time a job arrives, its service time is taken from the proper location in the future array and another sample is generated to take its place. If we denote the contents of the future array as $s_1 \ldots s_C$, then lookahead is given by

$$L_j = \min(\{r_j\} \cup \{s_i : i < j\})$$

where j, r_j are defined as before.

In general, the expected lookahead for a MCPP server depends on the service time distributions of the various customer classes. As an example, we calculate $E[L_j]$ for the case in which service times for every class are exponentially distributed. We shall make use of the following result from the theory of statistics [Allen 78, Thm. 3.2.1]: If $X_1, X_2, \ldots, X_n$ are independent random variables that are exponentially distributed with respective parameters $\lambda_1, \lambda_2, \ldots, \lambda_n$, then the statistic $V = \min\{X_1, X_2, \ldots, X_n\}$ is exponentially distributed with parameter $(\lambda_1 + \lambda_2 + \ldots + \lambda_n)$. Thus,

$$E[\min\{X_1, X_2, \ldots, X_n\}] = \frac{1}{\lambda_1 + \lambda_2 + \ldots + \lambda_n} \tag{5.2}$$

Let the service time for class i be exponentially distributed with parameter λ_i. Note that, because of the memoryless property of the exponential distribution, the residual

service time of the customer in service (if there is one) has the same distribution as the service time for that customer. Since the exponential distribution has a lower bound of zero, then without using the future array technique $E[L_j]$ is obviously zero. On the other hand, using the future array we obtain:

$$E[L_j] = \begin{cases} \frac{1}{\lambda_1+\ldots+\lambda_j} & 1 \le j \le C \\ \frac{1}{\lambda_1+\ldots+\lambda_C} & \text{otherwise (i.e., no customer in service)} \end{cases}$$

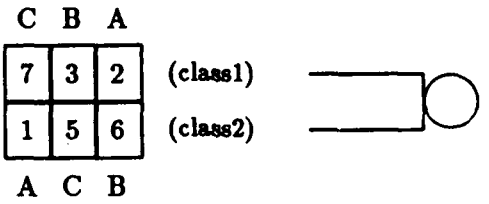
Note that the quantity we have calculated is the expected lookahead given that a class j customer is in service. In order to evaluate $E[L]$, the *unconditional* expected lookahead, we would need to know the probability that a customer of any given class (or no customer at all) is in service at a given time. (This might be accomplished by introducing assumptions about the arrival distributions for each customer class.) If we denote

$$p_j = \begin{cases} \Pr(\text{a customer of class } j \text{ is in service}) & \text{if } 1 \le j \le C \\ \Pr(\text{no customer is in service}) & \text{if } j = C+1 \end{cases}$$

and define $\lambda_{C+1} \equiv 0$, then we can write

$$E[L] = \sum_{j=1}^{C+1} \left(\frac{p_j}{\lambda_1 + \ldots + \lambda_j} \right)$$

In order to extend the future array technique to compute per-destination lookahead values, each entry of the future array must be a future list in the sense of the previous section (Figure 5.5). We make the following observation: the earliest time that any given class i arrival can depart from the server is the time computed by assuming that no arrivals of any higher-priority class take place. (Arrivals of lower-priority customers cannot affect the departure time of the customer.) In this case, the customer's time of departure will be the same as if the server were FCFS serving only class i customers. Therefore, we compute per-destination lookahead for each of the per-class future lists as for a FCFS server; then, the lookahead for a given destination is equal to the minimum



Destination	Lookahead
A	min(2,12) = 2
B	min(5,6) = 5
C	min(12,11) = 11

Figure 5.10: Per-destination lookahead computation for a MCPP server.

lookahead for that destination taken over all the per-class future lists.[6] In general, the lookahead for different destinations may be based on different components of the future array; this is necessary since lookahead always represents a worst-case scenario.

An example of the general technique is shown in Figure 5.10. Note that the lookahead values for destinations A and B are based on the future list for class 1, while for destination C the lookahead is based on the future list for class 2 (to take into account the possibility that no customers of class 1 arrive before the next two class 2 customers complete service).

5.3.2 Processor Sharing Servers

Processor sharing (PS) is an idealization of round robin (RR) scheduling. Under RR, each job receives a *quantum* of service before it must relinquish control to the next job in the queue, rejoining the queue at its tail. PS is defined as the limiting case of the RR algorithm as the quantum goes to zero, so that control of the processor circulates infinitely rapidly among all jobs. The effect is that jobs are served simultaneously, but each of the n jobs in service receives only $1/n$-th of the full power of the processor.

[6]This implicitly assumes that the routing as well as the service times of customers of different classes are independent random variables.

PS often is appropriate to model cpu scheduling in systems where some form of RR scheduling actually is employed [Lazowska et al. 84].

Suppose that, at local simulation time t, a single customer is in service at a PS server, and the customer's residual service time is r. Then it cannot be concluded that this customer will depart from the server at time $t + r$. The reason for this is that another customer may arrive before time $t + r$, effectively "stealing service" from the original customer, and thus delaying its departure. Even worse, the newly arrived customer may have a service time so small that it will depart even *earlier* than time $t + r$. As an example, suppose that $r = 6$ and a second customer with service time 2 arrives immediately. Then in the absence of further arrivals, the second customer will depart from the server at time $t + 4$ and the original customer will depart at time $t + 8$.

If the service time distribution of a PS server is bounded from below by some constant $a > 0$, then the simplest technique for calculating lookahead is to assume that a customer with the minimal service time arrives immediately. If there are n customers in service with residual service times $r_1 \ldots r_n$, the lookahead given by this technique is

$$L_n = \min(\{nr_i : 1 \leq i \leq n\} \cup \{(n+1)a\}) \tag{5.3}$$

To prove this, there are two cases to consider:

Case 1. No new customers arrive before one of the current customers completes service. Then the next customer to depart will be the one with residual service time $\min\{r_i : 1 \leq i \leq n\}$. Since it must share the processor with $n - 1$ other customers, it will require $n \cdot \min\{r_i\} = \min\{nr_i\}$ time units to complete service.

Case 2. A new customer arrives before any departure takes place. In the worst case, the new customer will arrive immediately. If the new customer's service time is less than the residual service time of any of the customers already in service, then the new customer will be the next to depart. Since there are now a total of $(n + 1)$ customers receiving service, the new customer will require at least $(n + 1)a$ time units to complete service.

Therefore, the minimum time that must elapse until any customer could finish service, which is the lookahead of the LP simulating this server, is given by Equation (5.3).

We next consider the application of the future list technique to the calculation of lookahead for a PS server with a positive lower bound on service times. Let there be m pre-samples in the future list, denoted by $s_1 \ldots s_m$. As before, if no customer arrives before any current customer completes service, the next departure will take place in $\min\{nr_i : 1 \leq i \leq n\}$ time units. Suppose now that one or more arrivals occur before any such departure. The worst case scenario would be if the arrivals occurred immediately. If a single customer were to arrive, then it could depart in no less than $(n+1)s_1$ time units, since it would have to share the processor with n other customers. Similarly, if two customers were to arrive, the earliest that the second arrival could depart is $(n+2)s_2$ time units in the future. The first arrival would be slowed down by the second, and thus it could not depart as early as if the second arrival had not occurred. Therefore, the earliest departure time of a future arrival given that no more than two such arrivals occur is $\min\{(n+1)s_1, (n+2)s_2\}$. Proceeding in this fashion, we deduce that if no more than m additional customers were to arrive, the soonest that any of the new arrivals could depart would be in $\min\{(n+1)s_1, (n+2)s_2, \ldots, (n+m)s_m\}$ time units.

Finally, we must take into account the possibility that more than m customers might arrive by presuming that an $m+1$-st customer with the minimal service time also arrives. This leads to the following expression for $L_{n,m}$, the lookahead given that there are n customers in service and m entries in the future list:

$$L_{n,m} = \min \left\{ \begin{array}{c} \{nr_i : 1 \leq i \leq n\} \cup \\ \{(n+i)s_i : 1 \leq i \leq m\} \cup \\ \{(n+m+1)a\} \end{array} \right\} \tag{5.4}$$

The first term in the minimum represents the n customers already in service; the second term represents the next m customers that might arrive, whose service times have been pre-sampled; and the last term represents the arrival of an $m+1$-st customer with an arbitrarily small service time.

	service time	number customers	minimum departure
in service	6	2	12
	5	2	10
future	2	3	6
⋮	4	4	16
⋮	3	5	15
(stop)	≥ 1	6	≥ 6

Figure 5.11: Calculation of PS lookahead.

The number of pre-samples m used in the calculation of $L_{n,m}$ is a free variable. In practice, the calculation of the second term in Equation 5.4 would continue until either the lookahead were acceptably large or until the cumulative minimum was less than or equal to $(n+m+1)a$, at which point further pre-sampling would not do any good. An example of this situation is shown in Figure 5.11, in which we have assumed a minimum service time of one time unit.

If there is no positive lower bound on the service time distribution, the future list technique can still be used as long as there is some limit N on the number of jobs that can be simultaneously in service at the server. As discussed earlier in this chapter, there are a number of scenarios common to QNMs in which the population at a server is externally constrained. Even if there is no such constraint, it would be reasonable for the server to limit the number of jobs that can be served simultaneously, forcing additional jobs to queue until some departures occur.[7]

The lookahead in this case can be obtained by setting $m = N - n$ in Equation 5.4 and dropping the term representing the $m+1$-st arrival. This yields:

$$L_n^N = \min \left\{ \begin{array}{c} \{nr_i : 1 \leq i \leq n\} \cup \\ \{(n+i)s_i : 1 \leq i \leq N-n\} \end{array} \right\} \tag{5.5}$$

[7]In fact, it would be unrealistic to allow a PS server to serve an unlimited number of jobs simultaneously, since then the approximation that the overhead of switching between jobs is negligible can no longer be justified.

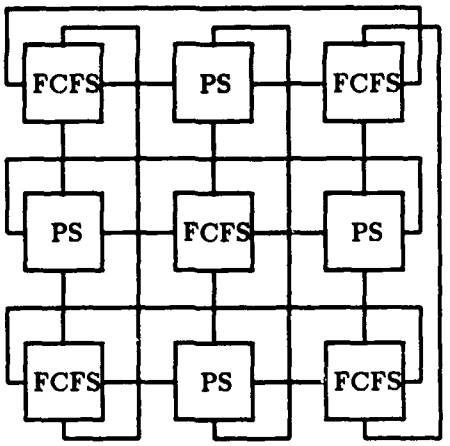


Figure 5.12: A mixed network of PS and FCFS servers.

We demonstrate the effectiveness of these techniques by applying them to the network shown in Figure 5.12, which consists of four PS and five FCFS servers arranged as a 3x3 edge-connected mesh. The message density for this benchmark was two.

Figure 5.13 shows the results. Each of the curves in the figure was obtained using the lazy blocking avoidance technique. In all three curves, the FCFS server LPs are using the simple one-pre-sample technique introduced in Section 5.2. The "LBA 0" curve represents the performance of lazy blocking avoidance in the absence of any PS server lookahead; the "LBA 1" middle curve represents the performance obtained by calculating PS server lookahead under the pessimistic assumption of an immediate arrival of a customer with the minimal service time; and the "LBA 2" curve represents the performance obtained by using the technique described in the preceding paragraph.

The most important and, perhaps, most surprising result is that even though the majority of the LPs in this benchmark (the five FCFS servers) have excellent lookahead, there is no speedup unless some lookahead calculation is also applied to the PS server LPs. The reason for this is that it is not possible for a PS server LP to predict with certainty the next customer departure time. (Recall that an arriving customer at a PS server will delay the departure of all customers that are already in service.) Therefore, no customer can depart from a PS server until the local simulation time at that server has

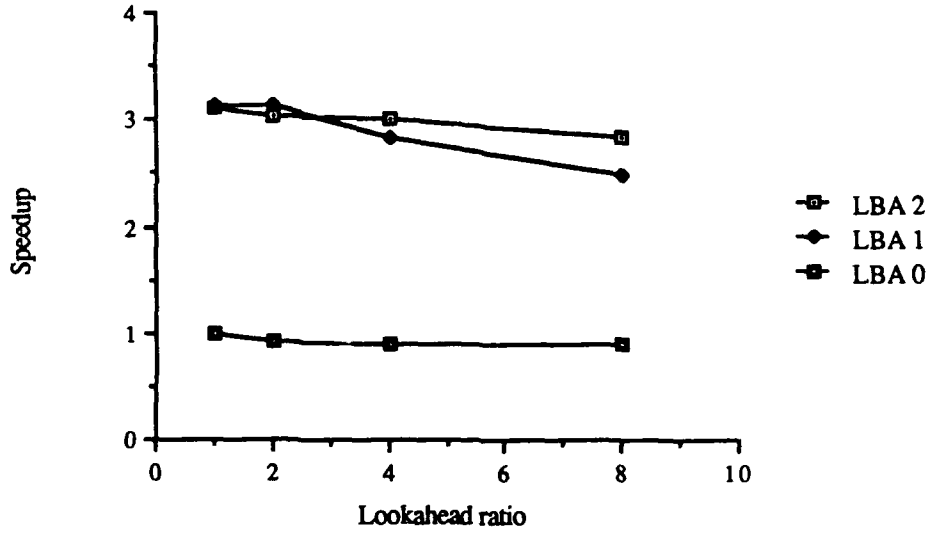


Figure 5.13: Speedup vs. lookahead ratio for a mixed network of PS and FCFS servers.

advanced to the departure time of the customer, implying that a PS server LP cannot send messages "into the future" in the way that an FCFS server LP can. Thus it is vitally important for such LPs to have at least some lookahead in order not to impede the progress of the rest of the simulation too much.

Another interesting result is that the addition of the full-olown future list calculation to the PS server LPs does not seem to make a very large difference in performance over the trivial scheme, although the difference does increase with increasing lookahead ratios.

Calculating the expected lookahead for a PS server is very difficult. One exception to this general rule is if the service times at the server are exponentially distributed. Let the parameter of the service time distribution be λ. Define new random variables $u_i = nr_i$ and $v_i = (n+i)s_i$, where r_i and s_i as defined as before. Then each random variable u_i is exponentially distributed (because of the memoryless property) with mean n/λ, and each random variable v_i is exponentially distributed with mean $(n+i)/\lambda$. Applying Equation (5.2), we obtain

$$E[L_{PS}^{N}(n)] \quad = \quad E[\min(\{u_i : 1 \leq i \leq n\} \cup \{v_i : 1 \leq i \leq N-n\})]$$

$$= \frac{1}{\sum_{i=1}^{n}(\lambda/n) + \sum_{i=n+1}^{N}(\lambda/i)}$$

$$= \begin{cases} \frac{1/\lambda}{\sum_{i=1}^{N}(1/i)} & \text{if } n = 0 \\ \frac{1/\lambda}{1+\sum_{i=n+1}^{N}(1/i)} & \text{if } 1 \leq n < N \\ 1/\lambda & \text{if } n = N \end{cases}$$

where N is the population constraint at the server.

In order to calculate $E[L^N]$, the expected lookahead independent of n, we would need to know the distribution of the number of customers in service at any given time, which would require us to make additional assumptions. The important thing to note is that, for a given n and N, $E[L_n^N]$ is a constant times $1/\lambda$, the mean service time. Since $E[L^N]$ is a linear combination of $E[L_n^N]$, $0 \leq n \leq N$, it too will be linear in $1/\lambda$. Therefore, under the given assumptions (exponentially distributed service times and a limit on the number of customers simultaneously receiving service), the expected lookahead ratio, equal to $(1/\lambda)/E[L^N]$, depends only on N and not on the mean service time. For this reason, the behavior of $E[L_n^N]$ as the population constraint N gets large is of interest to us.

Intuitively, one would expect lookahead to decrease as N grows, since the more samples that must be included in the calculation, the more likely it is that one of the samples will be extremely small. (Put another way, the minimum of N i.i.d. random variables is a decreasing function of N [Hogg & Craig 70, sec. 4.6].) What saves us, however, is that the multiplicative coefficient representing the processor sharing effect increases for each sample taken. It is not clear whether or not this is sufficient to keep the expected lookahead bounded away from zero as N increases. As we shall show next, either case is possible, depending on the service time distribution.

Consider once again the case in which service times are exponentially distributed. The summation in the denominator of Equation 5.5 is a subsequence of the *harmonic*

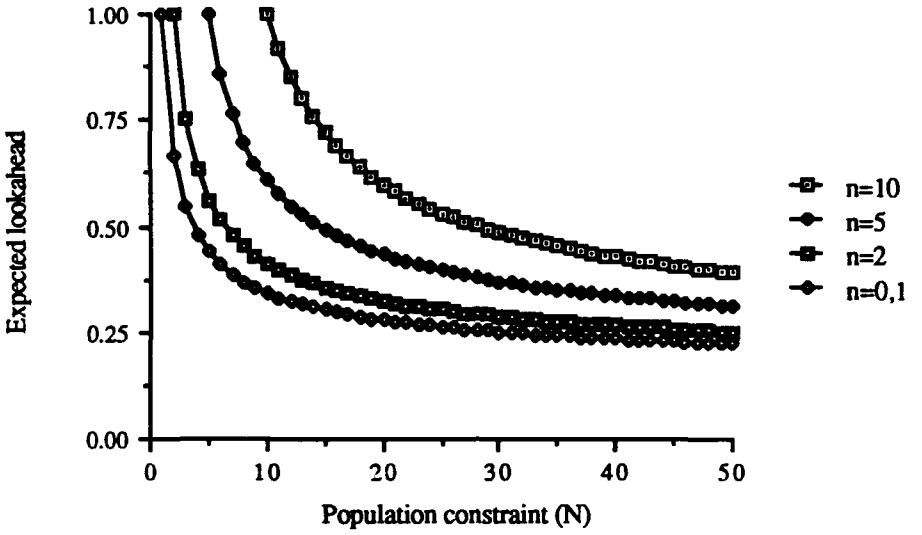


Figure 5.14: Expected lookahead as a function of population constraint for a processor sharing server with exponential service times.

series $\sum_{i=1}^{\infty}(1/i)$, which diverges. Thus, for $0 \leq n < N$,

$$\lim_{N \to \infty} E[L_n^N] = 0.$$

As lookahead decreases, lookahead ratio increases, and performance is expected to suffer. However, it is a well known fact that the partial sums of the harmonic series grow logarithmically; hence, $E[L_n^N]$ decreases to zero very slowly. Thus, even for "large" customer populations L_n^N is expected to be substantially better than having no lookahead at all. Figure 5.14, which shows L_n^N as a function of N for various values of n, clearly illustrates this behavior.

Given that expected lookahead has no positive lower bound as N increases and n remains fixed, it seems counter-intuitive that it should be a constant equal to the mean of the service time distribution when $n = N$. The explanation of this fact is as follows. As $n = N$ increases without bound, the minimum of the residual service times will tend towards zero. However, because the number of customers in service is also becoming infinite, each customer gets only an infinitesimal share of the server's attention. Thus, the expected time until the next departure, which is the ratio of these two quantities, is

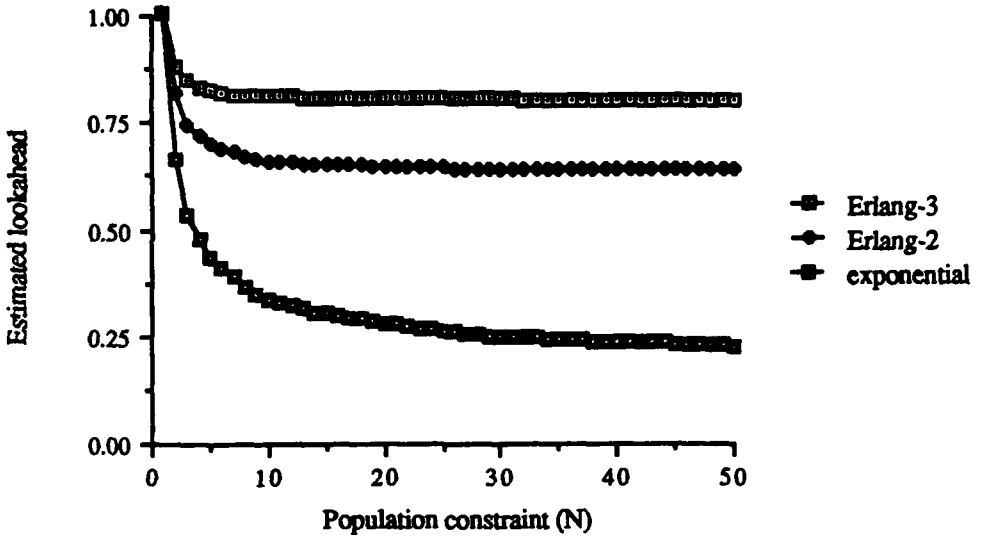
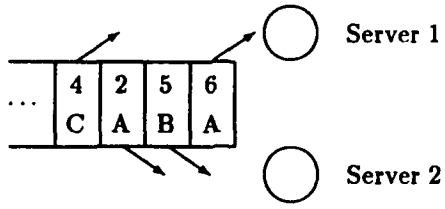


Figure 5.15: Estimated lookahead as a function of population constraint for an idle ($n = 0$) processor sharing server with various service time distributions.

finite and non-zero.

Next, we show the existence of service time distributions for which $\lim_{N\to\infty} E[L_n^N] > 0$ for all values of n. We claim that the Erlang-k distributions [Allen 78, sec. 3.2.5] for $k > 1$ have this property. (The Erlang-1 distribution is the same as the exponential distribution, and so obviously does not have this property.) The proof of this claim is contained in Appendix B.

Although we were unable to obtain a closed-form solution for $E[L_n^N]$ when service times are Erlang-k distributed for $k > 1$, we have used Monte Carlo simulation to estimate this quantity. Figure 5.15 shows $\overline{L_0^N}$, the *estimated* lookahead when $n = 0$ (idle server) as a function of population constraint, for service times having exponential, Erlang-2, and Erlang-3 distributions. The data suggest that $E[L_n^N]$ for a PS server with Erlang-2 service times approaches $1 - 1/e$ as N increases (although we have been unable to prove this). Furthermore, the expected lookahead is even larger for larger values of k.



Destination	Lookahead
A	6
B	5
C	10

Figure 5.16: Example of FCFS multiple-server center lookahead.

5.3.3 FCFS Multiple-Server Centers

This type of queueing network server has multiple servers serving a single FCFS queue of customers. There are many real-life examples of this service discipline (e.g., bank teller lines, airport ticket counter lines).

As for a simple FCFS server, it is possible to predict the departure times of all customers that have already arrived as well as customers that have yet to arrive. This is accomplished by keeping track of the next time that each server will become idle; an arriving customer will be served by the server that is anticipated to become idle the soonest. An example of this calculation is shown in Figure 5.16. In this example, the second and third customers are both served by server 2 because the second customer will complete service at server 2 before the first customer completes service at server 1. (Subsequently, the fourth customer will be served by server 1.)

The general lookahead computation for future arrivals is shown in Figure 5.17. In the algorithm, the next time that each server will become idle is kept in an array called `next_idle_time`. As usual, lookahead is calculated by assuming that all future arrivals up to and including the one of interest will happen immediately. The next idle times are copied to a working array and sorted, and as each new service time is added to the smallest value in this array, that value is bubbled to the right in order to maintain the ordering.

```
for i = 1 to num_destinations do
    lookahead[i] = 0;
endfor
for i = 1 to num_servers do
    copy[i] = next_idle_time[i];
endfor
sort(copy); /* in increasing order */
for i = 1 to future_list_size do
    copy[1] += future_service[i];
    destination = future_dest[i];
    if (lookahead[destination] == 0) then
        lookahead[destination] = copy[1];
    endif
    bubble_right(copy[1]);
endfor
```

Figure 5.17: The algorithm for FCFS multiple-server lookahead.

We will have more to say about this type of service center in Chapter 7, in which we apply the technique presented in this section to a simulation problem taken from the literature.

As an aside, we would like to point out that although calculating the lookahead of a multiple-server center is straightforward, it is extremely difficult to find a compact expression for lookahead, let alone calculate the expected lookahead! The reason for this difficulty is illustrated in Figure 5.18. This figure depicts the four possible cases that might occur in determining the departure time of the fourth future customer at the center. If we denote the service time of future customer i by s_i, then the quantity of interest could be $s_1 + s_4$, $s_2 + s_4$, $s_1 + s_3 + s_4$, or $s_2 + s_3 + s_4$, depending on the relative magnitudes of s_1, s_2, and s_3. Thus, the expected departure time for the fourth customer is given by

$$\begin{aligned} \mathrm{E}[t_2 - t_1] \;=\; & \mathrm{E}[s_1 + s_4 \mid s_2 < s_1 < s_2 + s_3] \cdot \Pr(s_2 < s_1 < s_2 + s_3) \\ & + \mathrm{E}[s_2 + s_4 \mid s_1 < s_2 < s_1 + s_3] \cdot \Pr(s_1 < s_2 < s_1 + s_3) \end{aligned}$$

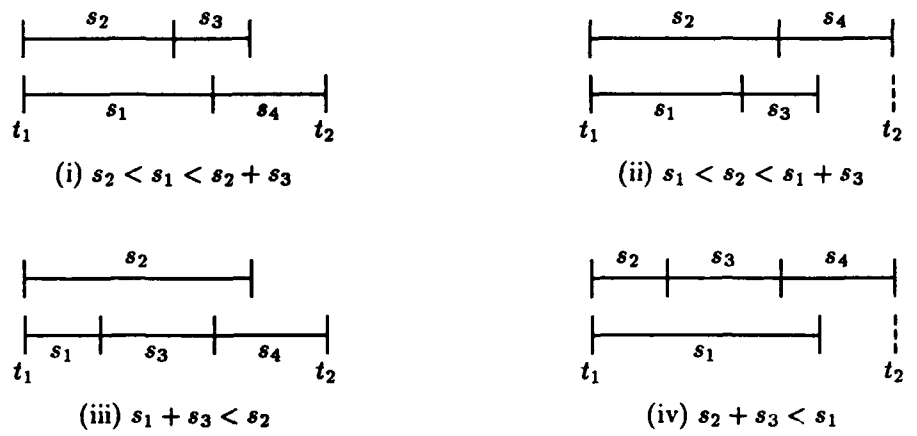


Figure 5.18: An enumeration of all possible cases for a departure time from a multiple-server center.

$$
\begin{aligned}
& + \mathrm{E}[s_1 + s_3 + s_4 \mid s_1 + s_3 < s_2] \cdot \Pr(s_1 + s_3 < s_2) \\
& + \mathrm{E}[s_2 + s_3 + s_4 \mid s_2 + s_3 < s_1] \cdot \Pr(s_2 + s_3 < s_1) \\
= \; & \mathrm{E}[s_1 \mid s_2 < s_1 < s_2 + s_3] \cdot \Pr(s_2 < s_1 < s_2 + s_3) \\
& + \mathrm{E}[s_2 \mid s_1 < s_2 < s_1 + s_3] \cdot \Pr(s_1 < s_2 < s_1 + s_3) \\
& + \mathrm{E}[s_1 + s_3 \mid s_1 + s_3 < s_2] \cdot \Pr(s_1 + s_3 < s_2) \\
& + \mathrm{E}[s_2 + s_3 \mid s_2 + s_3 < s_1] \cdot \Pr(s_2 + s_3 < s_1) \\
& + \mathrm{E}[s_4]
\end{aligned}
$$

Not only does the n ..ber of terms in this expression grow combinatorially with the index of the customer for which the expected departure time is being computed; but also each of the conditional expectations is extremely difficult to calculate, since the random variables involved are *not* independent.

5.4 Chapter Summary

In this chapter we have presented evidence for one of the central arguments of our thesis, which is that the semantics of the problem being simulated can and should be exploited to improve performance.

We began by reviewing the concept of lookahead and motivated the importance of its study. We asserted that lookahead is inversely correlated with the amount of blocking encountered by the LPs in the simulation. Empirical evidence in the form of measurement data from the Synapse parallel simulator corroborated this assertion.

The bulk of this chapter was then devoted to an investigation of methods for improving the lookahead of various types of queueing network servers, since queueing network models are a widely applicable performance analysis tool in many diverse fields. These techniques are based largely on the ability of the simulation LPs to predict characteristics such as service times and routing choices of messages that have not yet arrived, a technique that was pioneered by Nicol [Nicol 88]. We extended Nicol's technique to classes of service disciplines that were not considered in his original paper, such as multiple class preemptive priority servers, processor sharing servers, and multiple-server queues.

Chapter 6

System and Language Support

One of the major goals of this research was to design and implement a prototype parallel simulator, both as a tool and as a laboratory for the comparison of the synchronization mechanisms presented in the Chapters 4 and 5.

Our goals for the simulator included:

- That it relieve the programmer of the burden of LP synchronization. The programmer should be able to write a correct parallel simulation without any knowledge of how the underlying algorithms work.
- That it allow transparent (source-code compatible) comparisons of various synchronization mechanisms: deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance.
- That it provide a mechanism to give the programmer the *option* of tuning the program for performance. (A large body of empirical evidence indicates that application specific optimization can be crucially important for good performance of the conservative approach.) The simulation should still run correctly, albeit perhaps slowly, if the programmer chooses not to do any optimization.
- That it accomplish the foregoing while being as efficient as possible.

6.1 Presto

Many of the algorithms presented in the previous chapter require explicit control over low-level parallel programming mechanisms such as process synchronization and scheduling in order to be implemented efficiently. Fortunately, we were able to leverage our implementation off of the Presto parallel programming environment [Bershad et al. 88a, Bershad et al. 88b].

Presto is a set of tools for building parallel programming systems on shared memory multiprocessors. Presto's goal is to provide a framework within which one can easily build efficient support for any of a wide variety of "models" of parallel programming. Presto allows for easy modification and extension by employing an *object-oriented* programming style. Such extension is possible not only at the level of the primitives and structures made available for the application programmer's use, but also at the level of the run-time kernel that supports parallel applications (e.g., scheduling, preemption, processor control).

Furthermore, Presto's primitives are lightweight enough that the structure of the parallel application is not compromised due to the inefficiency of underlying primitives (e.g., thread context switch). The program can be coded in the way that is most natural, rather than being artificially structured due to lack of efficient primitives.

We exploited this flexibility to provide a run-time environment for conservative parallel simulation. Synapse[1] is a customized Presto enviroment that is designed explicitly to support conservative parallel simulation. Synapse was in fact the first major Presto application, and the development of Synapse contributed in various ways to the development of Presto [Bershad et al. 88b].

Presto and Synapse are implemented in the object-oriented language C++ and currently run on the Sequent Symmetry multiprocessor architecture.

[1]Synapse is an acronym for *So, You Need A Parallel Simulation Environment?*

6.2 Synapse

Synapse gives programmers several *classes* (abstract data types) to facilitate writing parallel simulations. The ones that are most visible to the programmer are the LogicalProcess and the Message. Equally important, but unseen by the programmer, is the SimScheduler class. These classes will be described later in this section.

For convenience, Synapse also provides utility classes to represent sundry abstractions such as random number streams and queues of messages.

Synapse has the following features:

- It provides virtual time semantics [Jefferson 85]: messages are received by LPs in non-decreasing timestamp order, regardless of their real-time order of arrival.
- It supports three deadlock handling mechanisms: deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance.
- It facilitates the comparison of the supported deadlock handling mechanisms by making the selection of a particular mechanism external to the program. In other words, the same application code runs no matter which mechanism is being used.
- It provides more sophisticated users with the *option* of tuning for performance. The programming interface is structured in a way that cleanly separates model specification from performance tuning, thus lessening the likelihood of introducing behavioral bugs during the tuning phase.

6.2.1 The LogicalProcess and Message Classes

The LogicalProcess (LP) and Message classes correspond directly to the concept of LP and message that were presented in the discussion of parallel simulation in Chapter 2. That is, an instance of class LP is an independent thread of control with a local simulation clock that can send and receive timestamped messages to other LP instances; an instance of class Message is simply an abstraction of a timestamped message. Neither of these

classes embodies any information that is specific to any particular simulation problem. That is, an instance of class Message contains no (programmer-visible) data other than a timestamp; an instance of class LP contains no (programmer-visible) data other than its clock, and no operations other than send and receive.

Instead, the intent is for the programmer to use the inheritance mechanism of the underlying object-oriented language to derive problem-specific classes from these base classes. In this way, the programmer-defined LPs transparently inherit the synchronization mechanisms that ensure a correct parallel execution of the simulation. (In particular, the LP and classes derived from it enjoy the property that all messages are received in non-decreasing timestamp order.) This approach has the side benefit of encouraging the programmer to use an object-oriented programming style.

(For a full description of the Synapse interface, refer to [Wagner 89].)

In addition to message ordering, the LP class automatically does eager blocking avoidance. Deadlock detection and recovery and lazy blocking avoidance are handled cooperatively by the LP and the SimScheduler class. Their interaction is illustrated in more detail in Section 6.2.4.

6.2.2 The SimScheduler Class

The SimScheduler assigns ready LPs to processors, detects and recovers from deadlock, and coordinates the operation of the lazy blocking avoidance protocol. It is invisible to the programmer.

Before continuing with our description of the SimScheduler, we would like to clear up a common misconception about Synapse: namely, that Synapse is not a true distributed simulator because it contains a centralized scheduler. This misconception probably arises because the term "scheduler" is somewhat misleading. The determination of whether or not an LP is runnable is not made by the SimScheduler; rather, it is made by the LPs themselves, using a decentralized algorithm.[2] An LP is placed in the ready list

[2]The exception to this is that the SimScheduler runs a centralized deadlock recovery algorithm after a deadlock has been detected.

whenever it (or some other LP) determines that it possesses a buffered message that can be received. We maintain that it is the decentralized determination of when events can be processed that is the salient feature of a distributed simulator; we argue, therefore, that Synapse is truly a distributed simulator. The centralized scheduler is simply a convenience that is possible because of the small-scale shared memory multiprocessor on which Synapse is implemented. If the implementation were done on a larger scale multiprocessor, it would probably be necessary to distribute the SimScheduler to prevent its ready list from becoming a bottleneck, but this would not have any effect on the basic operation of the LPs.

The SimScheduler is a straightforward derivation of the default Presto Scheduler class. We will describe its design in some detail because it is an excellent illustration of the power and flexibility of the Presto approach to building parallel programming environments.

The structure of the Presto Scheduler is particularly simple. Presto associates with each processor a distinguished thread of control called the *scheduler thread*. A processor runs its scheduler thread whenever the processor would otherwise be idle. The scheduler thread tries to find work for its processor by repeatedly invoking a routine called Scheduler::getreadythread(), which returns runnable user threads on a first-in, first-out basis.[3] If successful at obtaining a user thread, the scheduler thread performs a context-switch to that user thread. The user thread runs until it blocks on some synchronization primitive or until it receives a preemption interrupt, at which point it context switches back to the scheduler thread.

The SimScheduler class is identical to the Presto Scheduler class except for the behavior of the getreadythread() routine and the lack of preemption. More precisely, SimScheduler differs from Scheduler in the following ways:

[3]A thread that is blocked by some instance of a Presto SynchroObject (which may be a monitor, condition variable, barrier, etc.) is automatically added to the Scheduler's ready list by the SynchoObject when the reason for blocking no longer exists (i.e., a monitor becoming available, a condition variable being signaled, or all threads reaching a barrier.)

- Both types of scheduler maintain enough state to determine when all processors are idle. Scheduler::getreadythread() assumes that this condition indicates that the computation has terminated; on the other hand, SimScheduler::getreadythread() assumes that it indicates the presence of a deadlock, and initiates deadlock recovery.

- If no runnable thread is available but there is activity elsewhere in the system, Scheduler::getreadythread() simply returns a null value to the calling scheduler thread. On the other hand, SimScheduler::getreadythread() uses the scheduler thread to run the lazy blocking avoidance protocol (if enabled).

- The SimScheduler disables preemption interrupts. This is because fairness is not an issue in a conservative parallel simulation: as long as an LP has useful work to do, it seems unwise to block it.[4]

Although neither the Scheduler nor the SimScheduler prioritizes the execution of threads, the getreadythread routine would be the logical place to add such intelligence if it were deemed useful.

During Presto's initialization phase [Bershad 88], Synapse substitutes an instance of SimScheduler for the default Scheduler instance. This run-time integration is possible because the SimScheduler class is derived from the Scheduler class. During Presto's multithreaded phase of operation, the inheritance mechanism of the underlying language ensures that invocations of getreadythread() are handled by SimScheduler::getready/-thread(), whereas all other scheduler functions continue to be handled by the methods defined in the base Scheduler class.

From this example, we hope the reader will conclude that the openness of Presto

[4]Note that in an optimistic parallel simulation, preemption is necessary because allowing one LP to get arbitrarily far ahead of others increases the likelihood that it will have to roll back. In fact, the argument that the optimistic approach wastes no more work due to rollback than a conservative approach wastes due to blocking is valid only if processes are preemptively scheduled using their clock values as priorities (earliest virtual time first).

makes it a powerful tool for constructing special-purpose parallel programming environments.

6.2.3 Performance Tuning

Experience has shown that conservative parallel simulators that are general-purpose do not perform as well as those that are tailored to a particular application [Fujimoto 88a, Fujimoto 88b, Nicol 88, Reed & Malony 88, Reed et al. 88, Wagner et al. 89]. As an extreme example, Lin, Baer, and Lazowska [Lin et al. 89] showed that even a parallel simulator designed specifically for trace-driven simulation of multiprocessor cache coherence protocols could not perform as well as a simulator tailored to a *particular* cache coherence protocol. Thus, it seems crucial to provide a mechanism that allows the programmer to exploit the semantics of the simulation model to improve performance. The mechanism we chose for Synapse is the specification of lookahead values by the LP. (We have shown both analytically (Chapter 3) and experimentally (Chapter 5) that dramatic improvements in performance are possible through the use of lookahead.

Given that we wish for the programmer to be able to specify lookahead values, the question then becomes *how?*

Clearly, there are many possibilities. Perhaps one of the first schemes that comes to mind would be for the programmer to code derived LPs to explicitly send null messages with timestamps that are based on the calculated lookahead values. This is a bad idea for a number of reasons. First of all, as pointed out in the Chapter 4, explicit null messages are unnecessary in a shared memory implementation; instead, the timestamp of the "last null message sent" on each channel can be maintained in the channel clock value for that channel. Second, we also presented in that chapter extensive empirical evidence that lazy blocking avoidance is superior to eager blocking avoidance.

Finally, requiring the derived LP to send null messages is incompatible with our goal of allowing the same application code to run transparently on top of any one of the three supported synchronization mechanisms. Therefore, a more desirable mechanism would

be for the derived LP simply to provide lookahead values to the underlying Synapse LP, which would then "do the right thing" with the information.

That having been decided, we must now answer the question of *when* the derived LP should calculate lookahead. It may be the case that the amount of computation involved is significant, which argues for the calculation of lookahead only when the derived LP has nothing better to do (i.e., is about to block). The problem is, the derived LP has no way of determining when it is about to block because the buffering of messages is transparent to it. To make this buffering visible at the user level would be hard to reconcile with our goal of making parallel simulation as transparent as possible to the programmer.

Furthermore, it can be the case that an LP's clock advances without the LP being awakened (as a result of the arrival of a "null message" from a neighboring LP in the case of eager blocking avoidance, or as a result of running the lazy blocking avoidance protocol on the LP). Under such circumstances, we would like to be able to avoid having to awaken an LP just so it can calculate lookahead. Ideally, the lookahead calculation would be done directly by an idle processor's scheduler thread, without having to context switch to the LP's thread.

The solution that we have adopted is suggested by the object oriented programming philosophy: we consider the computation of lookahead to be an instrinsic behavior of the LogicalProcess class, and allow this behavior to be redefined using the standard class derivation mechanism provided by the language. More concretely, the LP class defines a member function called lookahead() that is invoked via an upcall from the run-time system at times suggested by the heuristics outlined in the preceding paragraphs. The LP then automatically takes the appropriate course of action, depending on the synchronization mechanism being used. By redefining LP::lookahead() when deriving a sub-class from the base LP class, the programmer can make the simulation use application-specific lookahead values.

Structuring the user-system interaction in this way has another advantage: it separates the code that specifies simulation behavior from the code that is used for per-

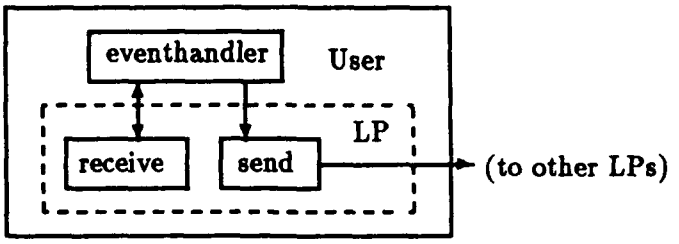


Figure 6.1: Normal LP operation.

formance tuning. This means that a naive programmer can ignore performance tuning altogether (because the system's version of LP::lookahead() will be used automatically if there is no programmer-supplied version), but that a sophisticated programmer who wishes to tune for performance can easily do so *after* having written and debugged the simulation. Furthermore, if the lookahead routine's access to simulation state is read-only (as it should be), then changes to the lookahead routine cannot affect the correctness of the simulation, only its speed.[5]

6.2.4 Overall Synapse Structure

In this section we will illustrate the interaction of Synapse's (and Presto's) component classes.

Figure 6.1 show the normal operation of an LP. The LP's thread of control animates a routine defined by the user-derived LP, which we will call User::eventhandler(). Eventhandler() interfaces with the rest of the simulation through the LP::receive() and LP::send() primitives.

[5]Of course, if the lookahead routine provided lookahead values that were larger than what could be guaranteed by the semantics of the application, LPs might process messages too optimistically; the likely result of a bug of this type is the arrival of a "stale" message at an LP (i.e., a message with timestamp less than the LP's incoming message horizon). Note that this may or may not happen, depending on the magnitude of the error and sheer luck. If no such arrival ever occurs, then the simulation is still correct, and the programmer "got away with something". However, if such an arrival does occur, the run-time system will balk and notify the programmer that the lookahead calculation is incorrect.

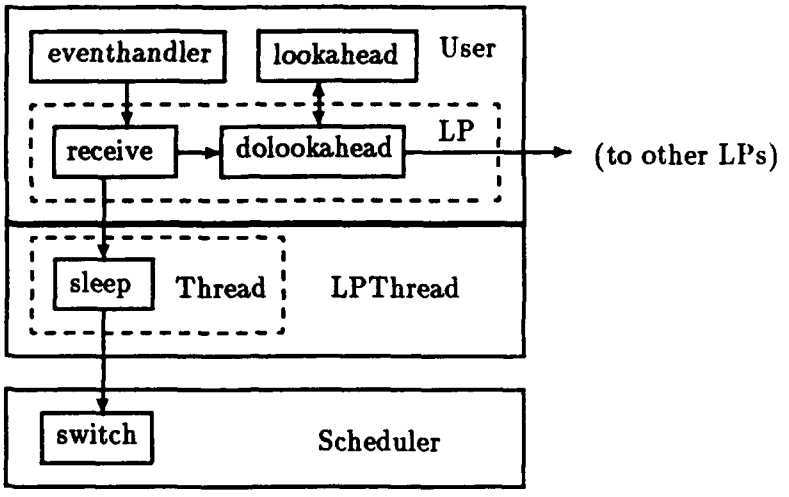


Figure 6.2: Blocking an LP.

Figure 6.2 shows what happens when an LP is unable to receive an event. First, LP::receive() checks to see if the LP's clock has advanced since the last time lookahead was computed; if it has, it calls LP::dolookahead() (1). LP::dolookahead() invokes the lookahead() routine (2), which may or may not have been provided by the User class (as explained in the previous section). The lookahead values returned by lookahead() are posted in shared memory and, if eager blocking avoidance is being used, each destination for which lookahead has changed is notified (3).

Finally, the LP is blocked by invoking Thread::sleep() (4), which modifies the thread's state appropriately and then context switches back to the scheduler thread for the processor on which it is running (5).

Note that although Presto provides monitors and condition variables (with Mesa-like semantics [Lampson & Redell 80]), the LP does not use them because they are "too big a hammer". A brief digression is in order at this point to explain why Synapse does not need the full power of the synchronization objects provided by Presto. The binding of threads to condition variables in Presto is dynamic and changes over time; furthermore, many threads can be waiting on the same condition variable. On the other hand, Synapse blocking semantics are much simpler in that there is only one reason for

an LPThread to block: namely, that its associated LP is unable to receive a message. Consequently, there is an *implied condition variable* to which the LPThread is statically bound (and vice-versa). Since there is only one LPThread that can be waiting for this implied condition, there is no need to maintain a queue of blocked threads as in the case of an ordinary, explicit condition variable.

Furthermore, the semantics of a Presto monitor are that a thread waiting to acquire the monitor is blocked (i.e., loses its processor). Early experiments showed this not to be a good idea: since the critical section inside an LP is used only to prevent race conditions that could cause wakeups to be missed, the amount of time typically spent waiting to enter the critical section is small compared to the time required for a context switch. Therefore, the LP uses a simple Presto spinlock to protect the data that determines whether or not the LP should block (or be awakened).

Finally, when a thread waiting on an ordinary Presto condition variable is signaled, it resumes execution inside the associated monitor object. However, when an LP is awakened it no longer needs to be in its critical section; therefore, such a policy would not only slow down this LP, but could also delay other LPs that might be trying to enter the critical section to deliver messages.

Fortunately, Presto's open architecture makes it possible to sidestep the standard monitor and condition variable classes and directly use the primitives (Thread::sleep and Thread::wakeup) from which they are built.

Continuing now with our walk-through of Synapse, Figure 6.3 shows the sequence of actions that takes place when an LP is notified of some change by another LP (as a result of a message arrival or eager blocking avoidance). First of all, the other LP invokes this LP's notify() primitive (1). LP::notify() checks to see if the LP is blocked and, if so, whether or not the new information carried by the notification would likely allow it to proceed. If both these criteria are satisfied, it calls LPThread::wakeup() (2). LPThread::wakeup() marks the thread as being awake and calls Scheduler::reschedule() to insert the thread in the scheduler's ready list (3).

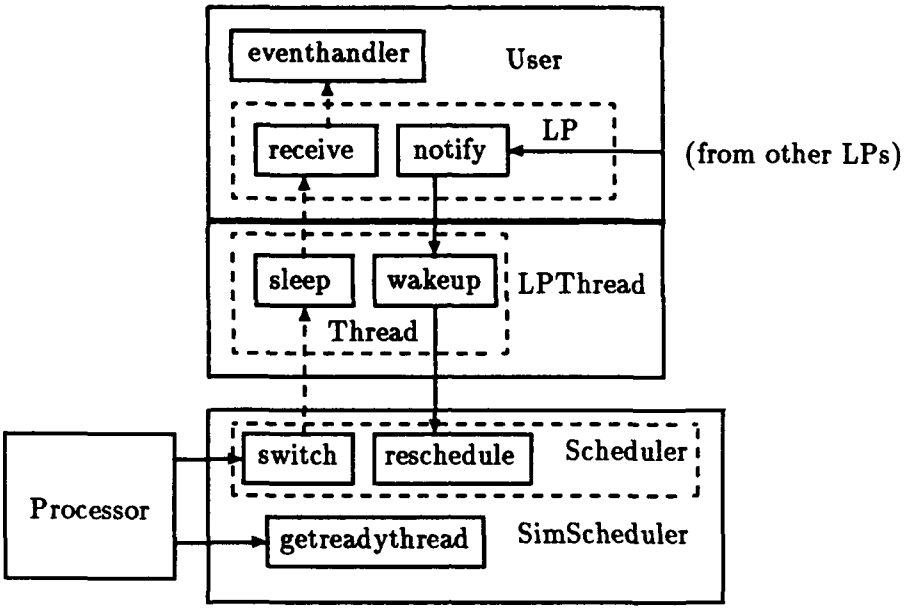


Figure 6.3: Notification of one LP by another.

At some later time, an idle processor invoking SimScheduler::getreadythread() (4) will be given this LPThread to run. The processor's scheduler thread will then perform a context switch to the LPThread (5), causing the associated LP to resume execution at the point in LP::receive just after the call to LPThread::sleep (6).

In addition to being awakened by another LP, an LP can also be awakened by the run-time system as a result of a deadlock being broken or as a result of the lazy blocking avoidance protocol (Figures 6.4, 6.5). Either scenario begins when a scheduler thread invokes SimScheduler::getreadythread on behalf of its processor and discovers that there are no ready threads (1).

At this point, the scheduler checks to see if *any* processor are busy. If not (Figure 6.4), it assumes that a deadlock has occurred and calls SimScheduler::deadlock_breaker() (2). The deadlock_breaker() routine determines the earliest simulation time at which any LP can be causally affected by a message (using the algorithm described in Chapter 4). It then invokes LP::awaken() with this value for each LP that can be awakened (3). LP::awaken() simply advances the LP's incoming message horizon to the specified time

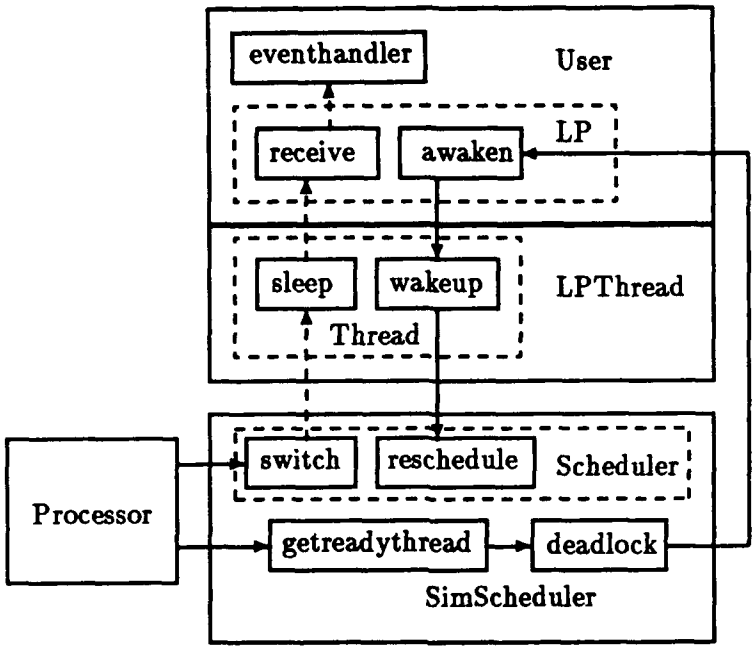


Figure 6.4: Deadlock recovery.

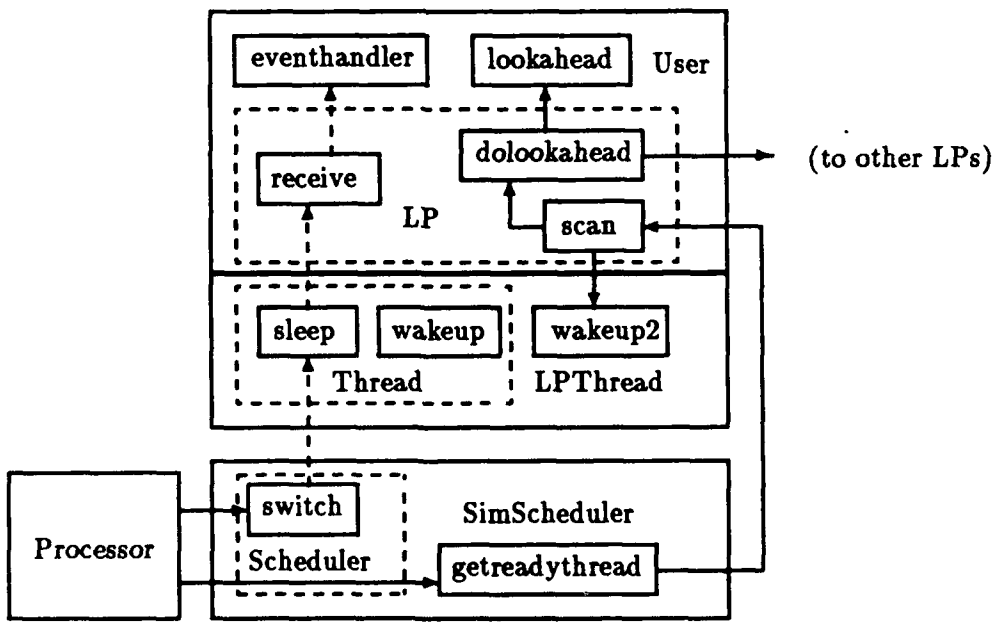


Figure 6.5: Lazy blocking avoidance.

and wakes up the associated LPThread exactly as described previously (4–7).

If, on the other hand, there is activity elsewhere in the simulation at the time getreadythread() comes up empty-handed, the lazy blocking avoidance protocol is run (Figure 6.5). Getreadhthread() selects a blocked LP at random and invokes LP::scan() (2). LP::scan() checks all of the LP's input dependencies, and can have three possible outcomes:

1. Nothing changes. In this case scan() returns a failure status.

2. The LP cannot receive a message, but its clock can be advanced. Then scan() calls dolookahead(), whose execution proceeds as described previously (3a). Scan() returns failure in this case as well.

3. The LP can receive a message. Then scan() calls the special primitive LPThread/-::wakeup2(), which adjusts the thread's state appropriately but does *not* insert the thread back into the scheduler's ready list (3b). Scan() returns success and the scheduler thread immediately performs a context switch to the LPThread (4–5).

Case (3) gives an example wherein we have delved even more deeply into the Presto run-time system: not only do we bypass the use of a pre-defined Presto synchronization object, we even bypass the most basic synchronization primitive (Thread::wakeup()) that the system provides! Why do we go to this much trouble? It is because doing so saves us from having to insert a thread into the scheduler's ready list and then immediately removing it. Each of these operations is costly because each requires the acquisition of the most heavily-contested spinlock in the system, namely, the spinlock protecting the scheduler's ready list.

6.3 Chapter Summary

This chapter describes the design and some implementation details of our prototype parallel simulator, Synapse.

Synapse is a custom-tailored programming environment designed specifically for the efficient implementation of the concepts discussed in Chapters 4 and 5. Its most notable design features are:

- It relieves the programmer of the burden of LP synchronization. The programmer can write a correct parallel simulation without any knowledge of how the underlying algorithms work.
- It allows transparent (source-code compatible) comparisons of three conservative synchronization mechanisms: deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance.
- Internal synchronization primitives are exactly matched to the needs of the application; a Synapse program is not forced to sacrifice performance for the sake of generality.
- The run-time scheduler, SimScheduler, is aware of the high-level structure of the application (conservative parallel simulation). In some cases (e.g., Figures 6.4 and 6.5) the SimScheduler bypasses the low-level Thread class completely and interacts directly with the high-level LP class.
- The custom-tailoring philosophy of Synapse allows programmers to exploit detailed semantics of the model being simulated in order to improve performance. The object-oriented programming model cleanly separates code that is necessary for correctness from that which is necessary for performance.

As demonstrated in Chapters 4 and 5, Synapse is able to achieve good speedups on a wide variety of simulation problems.

Chapter 7

Case Study

In this chapter we illustrate the techniques of the previous three chapters on a real-world simulation problem. The problem we chose was the Colombian health care delivery system presented by Lomow, Cleary, Unger, and West [Lomow et al. 88]. They simulated this problem using the optimistic approach, and conjectured that it would be a difficult problem for the conservative approach. Subsequently, Leung, Cleary, Lomow, Baezner, and Unger compared the results of a conservative parallel simulation of this problem to their earlier results, and concluded that while the presence of cycles in the model reduced the speedup of the optimistic technique, it removed almost all of the speedup from the conservative technique [Leung et al. 89].

We feel that their experiences make this problem "fair game" for our case study.

7.1 Description of the Problem

We quote liberally from [Lomow et al. 88] in the following description of this simulation problem.

The government of Colombia provides health care for the majority of the population through a multi-tiered system of services and referrals. At each level in the system certain services are provided while other problems are referred to the next higher level.

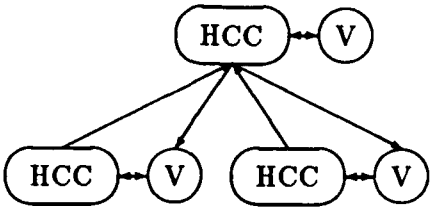


Figure 7.1: Abstract view of the Colombian health care system (V = village, HCC = health care center).

A simplified view of this system is shown in Figure 7.1. Each generic health care center represents a facility at one tier in the health system and serves both its local village and those health care centers below it in the hierarchy.

Each village generates a stream of patient arrivals for its local health care center. Each health care center has one or more health care providers (HCPs) who can serve patients independently. Each patient arriving at a health care center is either served immediately or, if all of the HCPs at this health care center are busy, placed in a queue of waiting patients. Each patient's condition is assessed and either the patient is treated at this health care center and then returned to his village, or the patient is referred to this health care center's parent health care center.

The simulation assumes that the health care system is organized as a full, four-way branching tree of height three. The number of HCPs at each health care center is based on its level in the tree: health care centers nearer to the root represent larger hospitals and have correspondingly more providers. A health care center at level i (where level zero nodes are leaves in the tree) is assumed to have 2^i HCPs.

The following parameters are used in the simulation: villages generate patients according to a Poisson process with arrival rate 0.3 patients per time unit; the assessment time for a patient at a health care center is a constant 0.3 time units; the probability that a patient can be treated at a given health care center is assumed to be 0.9 (except that it is assumed that all patients can be treated by the health care center that is at the

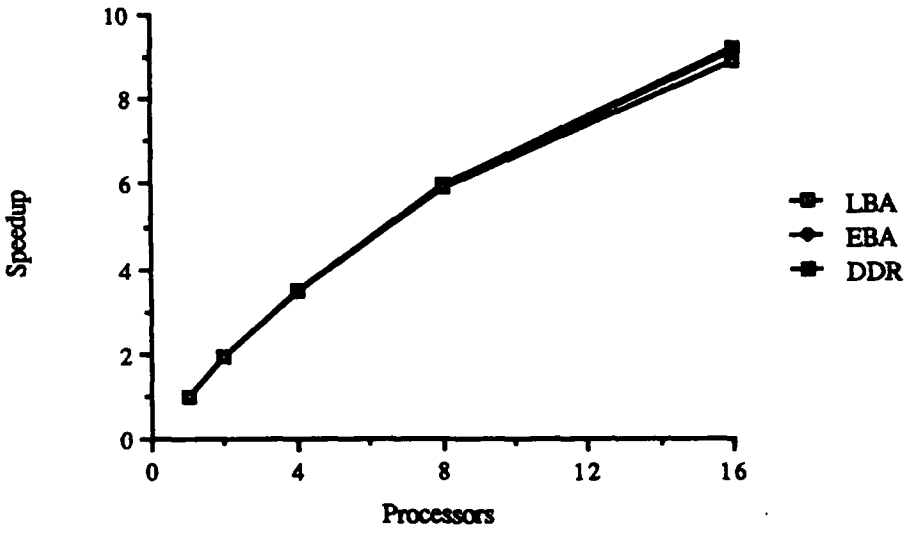


Figure 7.2: Speedup vs. number of processor for the no-feedback model.

root of the tree); and the treatment time (for patients that do not need to be referred) is an additional constant 1.0 time units.

Two different models of this system were simulated by Lomow et al.: one in which a patient leaving a health care center departs from the system, and one in which it returns to its home village. It is expected to be much harder to achieve good speedup for the latter model because of the presence of feedback.

7.2 The No-Feedback Model

In this model, after a patient has been treated the patient exits the system rather than returning to its home village. The system is thus an acyclic feed-forward network of LPs, and should yield good speedup.

Figure 7.2 shows the speedup obtained when this model is simulated using Synapse. The speedups in this case are quite good, which is to be expected: the absence of feedback means that there are never any deadlocks in this simulation. What is even more interesting is that a parallel simulation running on a single processor was faster than our

event list-based process-oriented sequential simulator! For this reason, the speedups shown in the figure were calculated relative to a 1-processor parallel simulation.

One reason for the superiority of the parallel algorithm for this model is that the sequential simulation algorithm causes more context switches between LPs than does the parallel algorithm. It is easy to understand why this is the case. Since the sequential simulator processes all events in timestamp order, there is usually only one or at most a few events simulated between successive context switches. On the other hand, the parallel simulation algorithm running on a single processor could theoretically execute the entire simulation with only as many context switches as there are LPs: each level-0 node could simulate all the way to the stopping time of the simulation without ever blocking; subsequently, each level-1 node could do the same, and so on. In fact, we chose to *force* each LP to relinquish the processor periodically in order to allow patient records to be recycled, thus reducing the memory requirements of the simulation. The net result was that the parallel simulation performed approximately 1/20-th as many context switches as the sequential simulation.

However, measurements of context-switch overhead confirmed that the number of extra context switches accounts for only about one-fourth of the difference in elapsed times between the sequential run and the one-processor "parallel" runs. In an attempt to isolate the source of the remaining difference, we modified Synapse so that we could control the frequency with which LPs were pre-empted, and thus control indirectly the number of context switches. This allowed us to compare various runs of the exact same algorithm, differing only in the number of context switches that were performed. Even in this case, we found differences in execution times that were much larger than context switch overhead alone could account for. The most likely explanation is that increasing the number of context switches adversely affects the data locality of the program. This hypothesis is bolstered by the fact that the relative differences in execution times grew very quickly as the number of LPs in the benchmark was increased, which increased the memory requirements of the simulation.

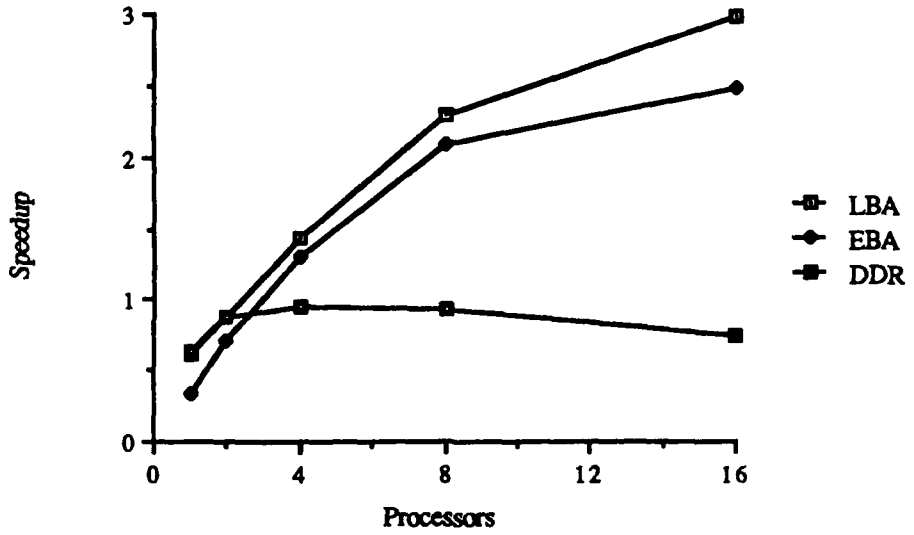


Figure 7.3: Speedup for the feedback model, first attempt.

7.3 The Feedback Model

In this model, after a patient has been treated the patient returns to its home village. The presence of feedback is expected to slow down the parallel simulation, and thus should not be included unless absolutely necessary in order to represent the system realistically. For example, if the patient arrival rate from a particular village were dependent on the number of villagers that were already seeking treatment, it would be necessary to include feedback in the simulation. Although the current model has no such dependencies, we make no attempt to exploit this fact, because the whole purpose of introducing feedback is to see how it affects the performance of the benchmark.

For our first attempt, lookahead for was simply the minimum possible service time at a health care center (the assessment time). Although this gives a reasonable lookahead ratio (1.3/0.3 = 4.33), Figure 7.3 shows that performance is surprisingly poor. Note the qualitative similarities between this figure and Figure 4.6, in which there was no lookahead at all. In particular, the extremely poor performance of eager blocking avoidance for small numbers of processors (where it performs even worse than deadlock detection

and recovery) indicates that something is wrong somewhere. (As is usually the case in situations like this, lazy blocking avoidance is able to perform at least as well as either of the other two synchronization mechanisms.)

Performance is poor because there is so little message feedback along some of the cycles in the communication graph. For example, only one out of every thousand messages generated by a level-0 village will eventually return to that village *via the path through the root of the tree.* The infrequency of feedback makes deadlock avoidance an absolute necessity: without it, it turns out that the deadlock frequency is 0.5 and the wakeup ratio is approximately 2. This implies that each time a deadlock is broken, approximately two LPs process a single message each before the next deadlock occurs!

Clearly, the lookahead needs to be improved on the channels pointing towards the root of the tree. The future list is likely to help a great deal, since the small probability that messages are sent to the parent health center (0.1) ought to lead to substantial lookahead values.

Since health care centers may contain multiple health care providers, each one is an instance of the FCFS multiple-server center discussed in Chapter 5. However, the particular semantics of this application allow for a significant simplification of the lookahead calculation.

First of all, note that this application is an example of one in which per-destination lookahead is not possible for certain channels. This is because a patient that is returned to a village must be returned to its original home village, and this information is carried with the patient itself. Therefore, it is not possible to predict *to which village* a patient will be sent, but only whether or not the patient is sent to *some* village or sent to the parent health care center. (Fortunately, this should not be a significant problem, since we are mainly trying to improve the lookahead on the channels between health care centers and their parents.)

The major simplification of the lookahead calculation is a consequence of the following two facts: first, there are only two possible service times, and second, a patient's service

time is implied by its destination. Therefore, the future list only needs to keep track of whether or not the patient will be treated locally. The random variable that determines whether or not a patient is locally treatable (with value 1 or 0, respectively) is pre-sampled until its value is 0. Observe that since the future list in this case is simply a string of ones followed by a zero, it is not necessary to maintain an explicit list at all, but only a count of the number of patients that must arrive before one shows up that is not locally treatable. Call this value n.

Now consider a level-0 health care center, in which there is only a single health care provider. Since the service times for all of the next n patients to arrive will be equal to the assessment time a plus the treatment time t, and the service time for the $n+1$-st patient to arrive will be the assessment time only, then the soonest that any patient could be sent to the parent health care center is in $n(a+t)+a$ time units. Note that if $n = 0$, meaning that the very next patient to arrive will have to be referred to the parent health care center, this expression reduces to the assessment time.

Finally, we generalize the preceding analysis to the case of a health care center with multiple health care providers. Since the service times of all of the locally treatable patients are equal, then we can assume that the future arrivals will served by the providers in a cyclic fashion. In other words, if the p providers are sorted in order of increasing next idle time and then renumbered $0 \ldots p-1$, then the $k+1$-st future patient will be served by provider $k \bmod p$. Furthermore, its service will commence no sooner than $\lfloor k/p \rfloor \cdot (a+t)$ time units after provider $k \bmod p$ becomes idle. Combining these two observations leads to the following simple expression for the lookahead of the channel leading to a health care center's parent, where a denotes the assessment time and t denotes the treatment time of a patient:

$$L_{\text{parent}} = \text{idle}[n \bmod p] + \left\lfloor \frac{n}{p} \right\rfloor \cdot (a+t) + a \tag{7.1}$$

As a quick sanity check, note that when $n = 0$ this expression reduces to $\text{idle}[0] + a$, i.e., a patient could be referred by the next available provider in as little as a time units after that provider becomes idle.

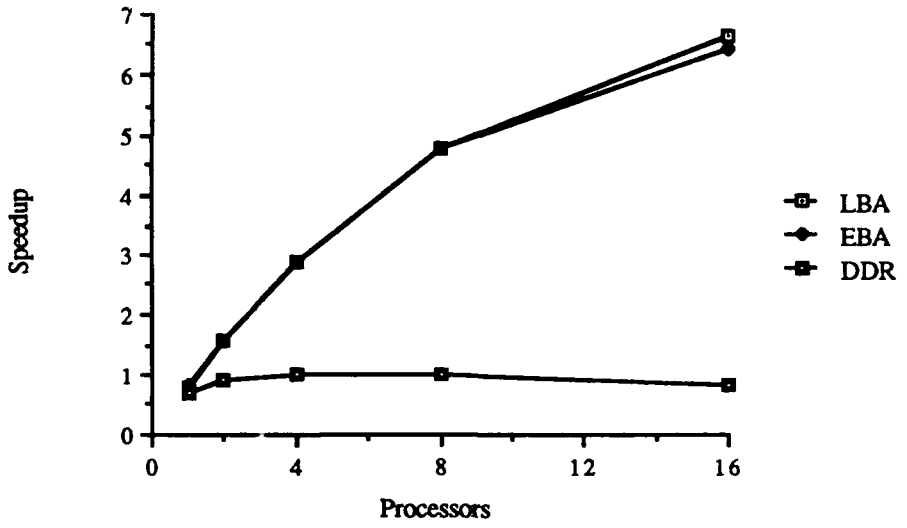


Figure 7.4: Speedup for the feedback model, using pre-sampling.

The lookahead for all channels leading to villages will be idle[0] + a + t, except in the special case when $n = 0$ (the next patient to arrive will be referred), in which case the lookahead is min(idle[0] + a, idle[1]) + a + t.

The speedup curves using this lookahead calculation are shown in Figure 7.4. Note the dramatic gains in speedup compared to the first attempt at simulating this problem.

7.4 Chapter Summary

There are two important results of this chapter. First, we showed that for simulations with an acyclic feed-forward communication structure, the conservative approach yields excellent speedup. In fact, because of its ability to execute events out of global timestamp order, the conservative parallel simulation algorithm has so many fewer context switches and so much better locality than the sequential process-oriented simulation algorithm that it outperforms that algorithm when run on a single processor.

The second important result is that, contrary to popular belief, conservative parallel simulation can achieve good performance in spite of the presence of a large number of

feedback paths with low message traffic. This is especially significant since this problem was originally posed by proponents of the optimistic approach as one that would be particularly taxing for the conservative approach [Lomow et al. 88]. The excellent speedups that we obtained were made possible by our technique for exploiting the semantics of multiple-server queueing centers (Section 5.3.3) to improve lookahead on the paths with low message traffic, and by the overall efficiency of Synapse's shared memory implementation (Chapters 4 and 6).

The bottom line, of course, is how our speedups compare to those achieved by Lomow et al. and Leung et al. for the optimistic simulation algorithm. In fact, the speedups reported in the former paper are slightly higher than ours for the case with feedback, but they calculated speedup relative to a one-processor parallel simulation rather than to a sequential simulation. If we also calculate speedup in this way, then our speedups are comparable to the figures they achieved using a random distribution of LPs on processors.

Leung et al. also calculated speedup relative to a one-processor parallel simulation. Even so, our speedups for the model with feedback are substantially better than theirs.

This problem was also used by Baezner, Cleary, Lomow, and Unger to study *algorithmic optimization* — in other words, modification of the simulation algorithm to exploit the semantics of the problem — for optimistic parallel simulation [Baezner et al. 89]. What is especially interesting about their work is that it demonstrated that algorithmic changes that affected lookahead had a dramatic effect on the performance of their optimistic simulator (as much as a factor of 2.6 improvement in speedup). They also noted that performance decreased sharply when the probability of successful treatment was reduced. Both of these observations cast some doubt on one of the frequently-claimed advantages of the optimistic approach, namely, that no special consideration is required on the part of the programmer to achieve good speedups.

Chapter 8

Conclusions and Future Directions

The main tenets of our thesis are the following:

1. *Conservative loose event-driven* parallel simulation is an effective way to reduce the time-to-completion of simulations of a wide variety of models.
2. The aggressive use of shared memory eliminates or alleviates most of the major obstacles to good performance.
3. Exploiting model semantics on a case-by-case basis can have a dramatic effect on speedup.
4. Point (3) seems to be contrary to the idea that "transparent is better" where parallel programming is concerned. Consequently, a parallel simulator should provide orthogonal mechanisms for the specification of model behavior and the performance tuning of model execution.

The original contributions of this dissertation include:

1. We have proposed using simple analytic techniques to bound from above the achievable speedup of parallel simulations. We have applied these techniques to show that

it is often the case that simulations of systems having a great deal of logical parallelism often can not achieve the same level of physical parallelism, regardless of the parallel simulation method that is used. Failure to recognize this in the past has led to overly pessimistic conclusions about the practicality of conservative parallel simulation.

2. We have proposed a novel technique for deadlock and artificial blocking avoidance called lazy blocking avoidance. Our measurements have shown that some form of blocking avoidance is necessary for good performance; however, when the available number of processors is smaller than the average level of parallelism, the overhead of blocking avoidance can actually be detrimental to performance.

   Lazy blocking avoidance successfully balances these considerations. Its principal virtue is that it incurs no overhead when the number of processors in use is smaller than the average level of parallelism of the simulation. However, it devotes more resources to blocking avoidance as the number of processors is increased. Our measurement data show that the net effect is that lazy blocking avoidance always performs at least as well as, and often substantially better than, two other synchronization methods that have been widely discussed in the literature (deadlock detection and recovery, and eager blocking avoidance).

3. We have shown how to exploit the semantics of various types of queueing network servers that had not been heretofore considered. We showed analytically that these modifications improve expected lookahead, and we validated the benefits of these modifications with measurement data from our parallel simulator. We also applied some of our techniques to a case study taken from the literature.

   The wide range of experiments we have conducted provides a much greater insight into the behavior of conservative parallel simulation than was available previously. These experiments are a significant contribution to the body of literature on parallel simulation.

4. We have implemented a custom-tailored programming environment for conservative loose event-driven parallel simulation, called Synapse. Synapse provides its own lightweight thread scheduler and synchronization primitives that are designed from the ground up with conservative parallel simulation in mind. Consequently, Synapse takes full advantage of the techniques for exploiting shared memory that we have identified.

   Synapse allows a programmer to write correct parallel simulations without any detailed knowledge of the underlying algorithms. At the same time, Synapse is flexible enough to allow the sophisticated user to exploit the lookahead characteristics of his or her simulation application. Its object-oriented philosophy cleanly separates code that is necessary for correctness from that which is necessary for performance. In addition, Synapse allows transparent (source-code compatible) comparisons of three conservative synchronization mechanisms: deadlock detection and recovery, eager blocking avoidance, and lazy blocking avoidance.

   Synapse was used for all measurement data reported in this dissertation.

5. Synapse was the first significant application of the Presto parallel programming environment. The development of Synapse contributed to the development of Presto and was a driving force behind Presto's "environment as a toolkit" paradigm.

This work could be extended in a number of ways.

First of all, the analytic modeling of parallel simulation in general, and conservative parallel simulation in particular, is still in its infancy. Although the model presented in Chapter 3 successfully predicted a number of behaviors of our parallel simulator, it does not model deadlock. This can lead to some anomalous predictions, as deadlock is frequently unavoidable (or it may be inadvisable, from a performance standpoint, to completely eliminate it). An obvious goal, therefore, is the development of a model of conservative loose event-driven parallel simulation that takes deadlock into account.

Our model allows the specification of lookahead characteristics only indirectly, via the probability that an empty input channel causes an LP to block. Any work that provided a *quantitative* relationship between these two concepts would be an important contribution.

Our measurements indicated that the implementation techniques presented here scale up to at least 16 processors, which approaches the limits of our current hardware base. It would be interesting to implement these techniques on a larger-scale parallel architecture to see how well certain aspects of our implementation scale up. Also of interest would be to investigate whether it would be possible to obtain good performance on a NUMA (non-uniform memory access) architecture, upon which it would no longer be possible to treat interprocessor communication as "free".

Another implementation consideration that is worth investigating is the impact of the scheduling policy on performance. In an optimistic parallel simulation, strong arguments can be made for particular scheduling policies (e.g., earliest virtual time first), but it is not obvious if different policies would produce any significant effect in a conservative parallel simulation.

In order to produce a widely usable conservative parallel simulator it will no doubt be necessary to provide tailored environments, written by sophisticated programmers, that are oriented to particular modeling domains such as queueing networks or Petri nets. The object-oriented design of Synapse allows this to be done fairly easily, and we have already implemented a number of LP classes such as FCFS and PS queueing network servers that are optimized for conservative parallel simulation; this work could be continued.

One topic that we have done some preliminary research on is the handling of globally readable state in a conservative parallel simulation. We believe it is possible to borrow the concept of *state histories* from optimistic parallel simulations in order to reduce the amount of synchronization that global state would require.

Finally, a much-needed contribution to the field would be both analytic and experimental comparisons of the optimistic and conservative approaches. This work represents a daunting challenge, because of the tremendous implementation effort required, and because the methodology of such research is subject to intense scrutiny by the proponents of whichever approach comes out on the "losing end". Furthermore, the introduction of new techniques frequently invalidates the results of earlier comparisons (as our experience with our case study has shown). Nevertheless, a number of open questions need to be resolved:

- We and others have demonstrated that in order to achieve good performance, conservative parallel simulations often need to be tuned to take advantage of the semantics of the simulation problem. An open question is whether or not optimistic parallel simulation is in general more "robust" than conservative parallel simulation, in the sense that its performance might be less dependent on the talents of the programmer. (This has been cited, without demonstration, as one of the advantages of the optimistic approach.)

- For parallel simulation, as for any parallel application domain, there is some minimum computational grain size that is needed in order to make the benefits gained from parallelization outweigh its overhead. Yet to the best of our knowledge, it is an open question whether or not there is any fundamental difference in the necessary grain sizes of the two approaches.

- This point is actually a generalization of the previous one. There is a general consensus that there are some problems for which the conservative approach is better-suited than the optimistic approach, and other problems for which the reverse is true, but no consensus as to which are which! Unfortunately, the collection of simulation benchmarks used in studies to date has tended to be rather ad-hoc. A set of benchmarks needs to be devised that can better characterize these two classes of problems. Until this is done, simulation practitioners will find themselves choos-

ing between the two paradigms on the basis of little more than anecdotal evidence and folklore, and parallel simulation will never become widely accepted outside of the computer science research community.

- A wide-open question is whether or not there is any advantage to be gained by combining the two approaches into a hybrid optimistic-conservative paradigm. The idea here is to introduce some amount of "conservativism" into an optimistic parallel simulator in order to reduce the frequency of rollbacks. One possible approach would be to use lookahead as a hint rather than as a hard constraint on LP behavior. Another possibility would be to dynamically analyze the temporal characteristics of message traffic between pairs of LPs in the hope of gaining some advantage in a statistical sense.

In summary, there are at least as many unanswered as there are answered questions about parallel simulation. The importance of simulation to the scientific community, combined with its seemingly insatiable appetite for computational resources, ensures that parallel simulation will be an important research topic for many years to come.

Bibliography

[Abrams 88] Abrams, M. The Object Library for Parallel Simulation (OLPS). In *Proc. 1988 Winter Simulation Conference*. Society for Computer Simulation International, December 1988.

[Agre 89] Agre, J. Simulation of Time Warp Distributed Simulations. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Aho et al. 74] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Allen 78] Allen, A. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, New York, NY, 1978.

[Amdahl 67] Amdahl, G. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. of the AFIPS*, 1967.

[Ayani 89] Ayani, R. A Parallel Simulation Scheme Based on Distances Between Objects. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Baezner et al. 89] Baezner, D., Cleary, J., Lomow, G., and Unger, B. Algorithmic Optimizations of Simulations on Time Warp. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Bailey & Snyder 89] Bailey, M. and Snyder, L. The Effect of Timing on the Speedup of Parallel Chip Simulation. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Berry & Jefferson 85] Berry, O. and Jefferson, D. Critical Path Analysis of Distributed Simulation. In *Distributed Simulation 1985*, pages 57–60. Society for Computer Simulation International, San Diego, CA, January 1985.

[Berry 86] Berry, O. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism*. PhD dissertation, University of Southern California, Los Angeles, CA, May 1986.

[Bershad 88] Bershad, B. The PRESTO User's Manual. Technical Report 88–01-04, Department of Computer Science, University of Washington, January 1988.

[Bershad et al. 88a] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8):713–732, August 1988.

[Bershad et al. 88b] Bershad, B., Lazowska, E., Levy, H., and Wagner, D. An Open Environment for Building Parallel Programming Systems. In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*. ACM SIGPLAN, July 1988.

[Birtwistle et al. 79] Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. *SIMULA BEGIN*. Chartwell-Bratt Ltd, England, 1979.

[Braun 75] Braun, M. *Differential Equations and Their Applications*, volume 15 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1975.

[Bryant 77] Bryant, R. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT,LCS,TR-188, Massachusetts Institute of Technology, Cambridge, MA, 1977.

[Chandy & Misra 79] Chandy, K. and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[Chandy & Misra 81] Chandy, K. and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[Chandy & Misra 88] Chandy, K. and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[Chandy & Sherman 89a] Chandy, K. and Sherman, R. Space-Time and Simulation. In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Chandy & Sherman 89b] Chandy, K. and Sherman, R. The Conditional Event Approach to Distributed Simulation. In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Chandy et al. 79] Chandy, K., Holmes, V., and Misra, J. Distributed Simulation of Networks. *Computer Networks*, 3:105–113, 1979.

[Chandy et al. 83] Chandy, K., Haas, L., and Misra, J. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[Comfort 82] Comfort, J. The Design of a Multi-Microprocessor Based Simulation Computer – I. In *Proc. 15th Annual Simulation Symposium*, pages 45–53, 1982.

[Comfort 83] Comfort, J. The Design of a Multi-Microprocessor Based Simulation Computer – II. In *Proc. 16th Annual Simulation Symposium*, pages 197–209, 1983.

[Crane & Lemoine 77] Crane, M. and Lemoine, A. *An Introduction to the Regenerative Method for Simulation Analysis*, volume 4 of *Lecture Notes in Control and Information Science.* Springer-Verlag, New York, 1977.

[Ebling et al. 89] Ebling, M., Loreto, M. D., Presley, M., Wieland, F., and Jefferson, D. An Ant Foraging Model Implemented on the Time Warp Operating System. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Eggers et al. 89] Eggers, S., Lazowska, E., and Lin, Y.-B. Techniques for the Trace-Driven Simulation of Cache Performance. Invited paper, *1989 Winter Simulation Conference*, December 1989.

[Even 79] Even, S. *Graph Algorithms*. Computer Science Press, Potomac, MD, 1979.

[Ferrari 78] Ferrari, D. *Computer Systems Performance Evaluation*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Fishman 78] Fishman, G. *Principles of Discrete Event Simulation*. John Wiley and Sons, New York, 1978.

[Fujimot 38a] Fujimoto, R. Lookahead in Parallel Discrete Event Simulation. In *Proc. Int. Conf. on Parallel Processing*, August 1988.

[Fujimoto 88b] Fujimoto, R. Performance Measurements of Distributed Simulation Strategies. In *Distributed Simulation 1988*, pages 14–20. Society for Computer Simulation International, San Diego, CA, July 1988.

[Fujimoto 89] Fujimoto, R. Time Warp on a Shared Memory Multiprocessor. In *Proc. Int. Conf. on Parallel Processing*, August 1989.

[Fujimoto et al. 88] Fujimoto, R., Tsai, J., and Gopalakrishnan, G. Design and Performance of Special Purpose Hardware for Time Warp. In *Proc. 15th Annual Int. Symposium on Computer Architecture*. IEEE Computer Society, June 1988.

[Gafni 85] Gafni, A. *Space Management and Cancellation Mechanisms for Time Warp*. PhD dissertation, University of Southern California, December 1985. TR-85-341.

[Gates & Marti 88] Gates, B. and Marti, J. An Empirical Study of Time Warp Request Mechanisms. In *Distributed Simulation 1988*, pages 73–80. Society for Computer Simulation International, July 1988.

[Gilmer, Jr. 88] Gilmer, Jr., J. An Assessment of "Time Warp" Parallel Discrete Event Simulation Algorithm Performance. In *Distributed Simulation 1988*, pages 45–49. Society for Computer Simulation International, July 1988.

[Gonnet 84] Gonnet, G. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA, 1984.

[Heidelberger & Welch 83] Heidelberger, P. and Welch, P. Simulation Run Control Length in the Presence of an Initial Transient. *Operations Research*, 31(6):1109–1144, June 1983.

[Heidelberger 86] Heidelberger, P. Statistical Analysis of Parallel Simulations. In *Proc. 1986 Winter Simulation Conference*, pages 290–295. Society for Computer Simulation International, 1986.

[Hillier & Lieberman 86] Hillier, F. and Lieberman, G. *Introduction to Operations Research.* Holden-Day, Oakland, CA, 4th edition, 1986.

[Hogg & Craig 70] Hogg, R. and Craig, A. *Introduction to Mathematical Statistics.* Macmillan, London, 3rd edition, 1970.

[Hontalas et al. 89] Hontalas, P., Beckman, B., Loreto, M. D., Blume, L., Reiher, P., Sturdevant, K., Warren, L. V., Wedel, J., Wieland, F., and Jefferson, D. Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior and Sectoring). In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Hwang et al. 89] Hwang, V., Sokol, L., and Stucky, B. MTW: A Control Mechanism for Parallel Discrete Simulation. Technical report, The MITRE Corp., McLean, VA, January 1989. Working paper.

[Iglehart & Shedler 80] Iglehart, D. and Shedler, G. *Regenerative Simulation of Response Times in Networks of Queues*, volume 26 of *Lecture Notes in Control and Information Science*. Springer-Verlag, New York, 1980.

[Jefferson & Sowizral 83] Jefferson, D. and Sowizral, H. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. Rand Note N-1906-AF, The RAND Corporation, Santa Monica, CA, June 1983.

[Jefferson & Witkowski 84] Jefferson, D. and Witkowski, A. An Approach to Performance Analysis of Timestamp-Driven Synchronization Mechanisms. In *Proc. ACM Conf. on Principles of Distributed Computing*, pages 243–253. ACM, 1984.

[Jefferson 85] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Jefferson 89] Jefferson, D., April 1989. Private communication.

[Jefferson et al. 87] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Time Warp Operating System. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93, Austin, TX, November 1987. ACM.

[Jones 86] Jones, D. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[Kaudel 87] Kaudel, F. A Literature Survey on Distributed Discrete Event Simulation. *ACM Simuletter*, 18(2):11–21, June 1987.

[Kelton & Law 84] Kelton, W. and Law, A. An Analytical Evaluation of Alternative Strategies in Steady-State Simulation. *Operations Research*, 32(1):169–184, January 1984.

[Kleinrock 75] Kleinrock, L. *Queueing Systems Volume I: Theory.* Wiley, New York, 1975.

[Kreutzer 86] Kreutzer, W. *System Simulation: Programming Styles and Languages.* Addison-Wesley, Reading, MA, 1986.

[Lamport 78] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson & Redell 80] Lampson, B and Redell, D. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Lavenberg & Reiser 80] Lavenberg, S. and Reiser, M. Stationary State Probabilities of Arrival Instants for Closed Queueing Networks with Multiple Types of Customers. *J. Applied Probability*, December 1980.

[Law & Kelton 82] Law, A. and Kelton, W. *Simulation Modeling and Analysis.* McGraw-Hill, New York, 1982.

[Law 83] Law, A. Statistical Analysis of Simulation Output Data. *Operations Research*, 31(6):983–1029, June 1983.

[Lazowska et al. 84] Lazowska, E., Zahorjan, J., Graham, G., and Sevcik, K. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Leung et al. 89] Leung, E., Cleary, J., Lomow, G., Baezner, D., and Unger, B. The Effects of Feedback on the Performance of Conservative Algorithms. In *Dis-*

tributed Simulation 1989. Society for Computer Simulation International, San Diego, CA, March 1989.

[Lin & Lazowska 89] Lin, Y.-B. and Lazowska, E. Optimality Considerations for "Time Warp" Parallel Simulation. Technical Report 89–07-05, Univeristy of Washington, Seattle, WA, 1989.

[Lin et al. 89] Lin, Y.-B., Baer, J.-L., and Lazowska, E. Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols. In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Little 61] Little, J. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[Livny 85] Livny, M. A Study of Parallelism in Distributed Simulation. In *Distributed Simulation 1985.* Society for Computer Simulation International, San Diego, CA, January 1985.

[Lomow et al. 88] Lomow, G., Cleary, J., Unger, B., and West, D. A Performance Study of Time Warp. In *Distributed Simulation 1988.* Society for Computer Simulation International, San Diego, CA, July 1988.

[Lubachevsky 88] Lubachevsky, B. Efficient Distributed Event Driven Simulation of Multiple-Loop Networks. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 12–21. ACM, May 1988.

[Lubachevsky 89] Lubachevsky, B. Scalability of the Bounded Lag Distributed Discrete Event Simulation. In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Madisetti et al. 89a] Madisetti, V., Walrand, J., and Messerschmitt, D. Efficient Distributed Simulation. In *Proc. of the 22nd Annual Simulation Symposium*, pages

5–21. Society for Computer Simulation International, San Diego, CA, March 1989.

[Madisetti et al. 89b] Madisetti, V., Walrand, J., and Messerschmitt, D. On Estimating the Progress of Optimistic Distributed Computation. Preprint, March 1989.

[Madisetti et al. 89c] Madisetti, V., Walrand, J., and Messerschmitt, D. On Estimating the Progress of Optimistic Distributed Computation — Multiple Processor Case. Preprint, May 1989.

[Misra 86] Misra, J. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–60, March 1986.

[Mitra & Mitrani 87] Mitra, D. and Mitrani, I. Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized by Rollback. *Performance Evaluation*, 7:111–124, 1987.

[Muntz & Wong 74] Muntz, R. and Wong, J. Asymptotic Properties of Closed Queueing Network Models. In *Proc. 8th Princeton Conference on Information Sciences and Systems*, 1974.

[Najjar et al. 87] Najjar, W., Jezouin, J.-L., and Gaudiot, J.-L. Parallel Execution of Discrete-Event Simulation. In *Proc. International Conference on Computer Design*, October 1987. (Port Chester, NY).

[Naylor 71] Naylor, T. *Computer Simulation Experiments with Models of Economic Systems.* Wiley, New York, 1971.

[Nicol & Reynolds 85a] Nicol, D. and Reynolds, P. A Statistical Approach to Dynamic Partitioning. In *Distributed Simulation 1985*, pages 53–56. Society for Computer Simulation International, San Diego, CA, 1985.

[Nicol & Reynolds 85b] Nicol, D. and Reynolds, P. An Operational Repartitioning Policy. In *Proc. 1985 Winter Simulation Conference*, pages 493–497. Society for Computer Simulation International, December 1985.

[Nicol 88] Nicol, D. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[Nicol 89] Nicol, D. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. Preprint, 1989.

[Peacock et al. 79] Peacock, J., Wong, J., and Manning, E. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, March 1979.

[Perry 89] Perry, T. Million Transistor Microchip: Intel's Three-Year Secret. *IEEE Spectrum*, 26(4):22–28, April 1989.

[Poole & Szymankiewicz 77] Poole, T. and Szymankiewicz, J. *Using Simulation to Solve Problems.* McGraw-Hill Ltd (UK), London, 1977.

[Presley et al. 89] Presley, M., Ebling, M., Wieland, F., and Jefferson, D. Benchmarking the Time Warp Operating System with a Computer Network Simulation. In *Distributed Simulation 1989.* Society for Computer Simulation International, San Diego, CA, March 1989.

[Reed & Malony 88] Reed, D. and Malony, A. Parallel Discrete Event Simulation: the Chandy-Misra Approach. In *Distributed Simulation 1988*, pages 8–13. Society for Computer Simulation International, San Diego, CA, July 1988.

[Reed et al. 88] Reed, D., Malony, A., and McCredie, B. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[Reiser & Lavenberg 80] Reiser, M. and Lavenberg, S. Mean Value Analysis of Closed Multichain Queueing Networks. *JACM*, 27(2):313–322, April 1980.

[Reitman 81] Reitman, J. *Computer Simulation Applications: Discrete-Event Simulation for Synthesis and Analysis of Complex Systems.* Robert E. Krieger, Malabar, FL, 1981.

[Reynolds 82] Reynolds, P. A Shared Resource Algorithm for Distributed Simulation. In *Proc. 9th International Symposium on Computer Architecture*, pages 259–266, Austin, TX, 1982. IEEE.

[Reynolds 85] Reynolds, P., editor. *Distributed Simulation, 1985*, volume 15, no. 2 of *Simulation Series.* Society for Computer Simulation International, San Diego, CA, January 1985.

[Roberts et al. 83] Roberts, N., Andersen, D., Deal, R., Garet, M., and Shaffer, W. *Introduction to Computer Simulation: A Systems Dynamics Modeling Approach.* Addison-Wesley, Reading, MA, 1983.

[Sauer & Chandy 81] Sauer, C. and Chandy, K. *Computer Systems Performance Modeling.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Sauer et al. 80] Sauer, C., MacNair, E., and Salza, S. A Language for Extended Queueing Networks. *IBM Journal of Research and Development*, 24(6):747–755, November 1980.

[Schruben 83] Schruben, L. Confidence Interval Estimation Using Standardized Time Series. *Operations Research*, 31(6):1090–1108, June 1983.

[Seethalakshmi 79] Seethalakshmi, M. A Study and Analysis of Performance of Distributed Simulation. Master's thesis, University of Texas at Austin, Austin, TX, 1979.

[Sevcik & Mitrani 81] Sevcik, K. and Mitrani, I. The Distribution of Queueing Network States at Input and Output Instants. *JACM*, 28(2):358–371, April 1981.

[Sokol et al. 88] Sokol, L., Briscoe, D., and Wieland, A. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In *Distributed Simulation 1988*. Society for Computer Simulation International, San Diego, CA, July 1988.

[Su & Seitz 89] Su, W.-K. and Seitz, C. Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Thesen 78] Thesen, A. *Computer Methods in Operations Research*. Academic Press, New York, 1978.

[Unger & Fujimoto 89] Unger, B. and Fujimoto, R., editors. *Distributed Simulation, 1989*, volume 21, no. 2 of *Simulation Series*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Unger & Jefferson 88] Unger, B. and Jefferson, D., editors. *Distributed Simulation, 1988*, volume 19, no. 3 of *Simulation Series*. Society for Computer Simulation International, San Diego, CA, July 1988.

[Wagner 89] Wagner, D. SYNAPSE User's Manual. University of Washington, Seattle, WA, 1989.

[Wagner et al. 89] Wagner, D., Lazowska, E., and Bershad, B. Techniques for Efficient Shared-Memory Parallel Simulation. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Wieland et al. 89] Wieland, F., Hawley, L., Feinberg, A., Loreto, M. D., Blume, L., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson, D. Distributed

Combat Simulation and Time Warp: the Model and its Performance. In *Distributed Simulation 1989*. Society for Computer Simulation International, San Diego, CA, March 1989.

[Wilson 86] Wilson, A. Parallelization of an Event Driven Simulator on the Encore Multimax. Technical Report ETR 86–005, Encore Computer Corp., November 1986.

[Wilson 87] Wilson, A. Parallelization of an Event Driven Simulator for Computer System Simulation. *Simulation*, 49(2):72–78, 1987.

[Yu et al. 89] Yu, Q., Towsley, D., and Heidelberger, P. Time-Driven Parallel Simulation of Multistage Interconnection Networks. In *Distributed Simulation 1989*. Society for Computer Simulation International, March 1989.

Appendix A

The Tradeoff Between Inter- and Intra-Replication Parallelism for Stochastic Simulations

Heidelberger examined the tradeoff between the number of replications and the degree of parallelism used in each replication of a stochastic simulation when the total number of processors is fixed [Heidelberger 86]. More concretely, given P processors and a time limit t, what is the combination of number of replications K and intra-replication parallelism $M = P/K$ that minimizes the mean square error of an estimator for the mean of an arbitrary model output? Increasing K (consequently, decreasing M) has the effect of reducing the variance of the estimator by providing more sample points. On the other hand, increasing M allows each replication to run longer in terms of *simulation time*, and hence diminishes the effect of the initial transient. However, since the parallel efficiency of the simulation algorithm is almost certainly less than unity (and likely decreases as M increases), the total number of observations (summed across all replications) decreases as M is increased.

Heidelberger's metric for comparison was *statistical efficiency*: that is, given a fixed amount of computing time, which choice of K and M yields the smallest mean square

error? A general statement of his results is that the degree of intra-replication parallelism M should be favored at the expense of the number of replications K under the following circumstances: for short runs (i.e., small t); for systems with a strong initial transient; or if a large number of processors are available. Of course, the optimal values depend on statistical characteristics of the output values of the simulation (which are generally unknown) as well as the total number of processors P and the computation time limit t.

Unfortunately, Heidelberger's results, in the form they are given, are not directly applicable. Consider: what does it matter if the mean square error of one alternative is 10, 100, or 1000 times smaller than the mean square error of a second alternative, if both are within acceptable limits? From a practical standpoint, we would rather be able to answer the following question: *How much faster can the statistically more efficient of two approaches achieve a given mean square error than the statistically less efficient one?* In other words, rather than fixing t for both approaches and comparing their mean square errors, it would be more illuminating to fix the mean square error and compare the amount of computation using each approach.

Furthermore, Heidelberger's results were based on a particular assumption regarding the parallel efficiency function of the simulation algorithm. It would be useful to be able to present a comparison that does not require this type of assumption.

Figure A.1 shows such a comparison. Using Heidelberger's analytic model, we have calculated *break-even speedup* as a function of M, for various values of P. We define break-even speedup as the speedup that each parallel replication would need to achieve in order for the parallel strategy to achieve the same mean square error as a purely sequential strategy running for the same amount of wall-clock time. This can be interpreted in one of two ways: if the parallel simulation is capable of exceeding the break-even speedup, then either (a) the parallel strategy can produce the same level of statistical confidence as the purely sequential strategy in less elapsed time, or (b) the parallel strategy can produce a higher level of statistical confidence than the purely sequential strategy in the same amount of elapsed time.

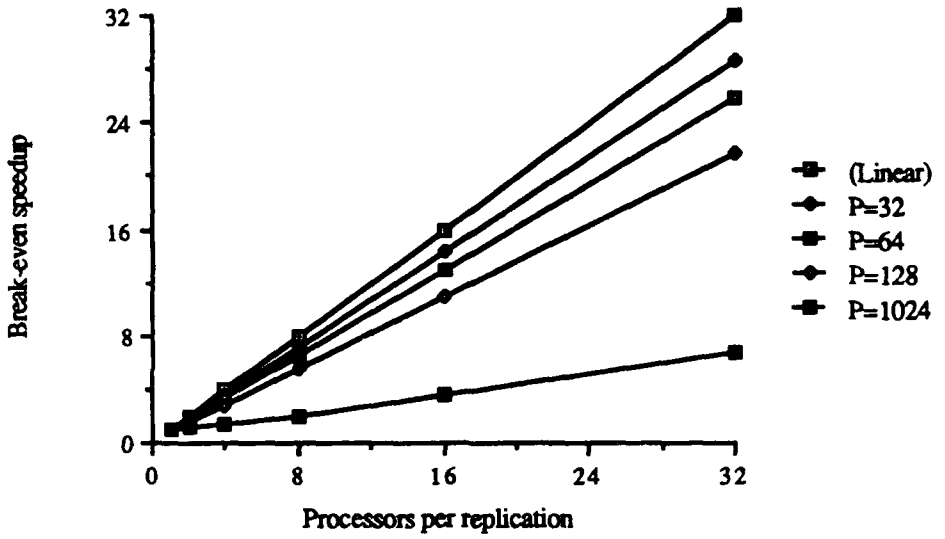


Figure A.1: Break-even speedup as a function of intra-replication parallelism.

Because the graph in Figure A.1 is for the particular model presented in [Heidelberger 86] (a single-server queue with exponential service times and Poisson job arrivals at a rate of 90% of the server's service rate), the reader is cautioned not to read too much into the *absolute* break-even speedup values. The general trend that is illustrated is much more important: as the total number of processors P increases, the break-even speedup decreases dramatically, even for large values of intra-replication parallelism M. Thus, given a large enough number of processors, the case for parallelizing individual replications can always be made.

Note that the analysis presented here does not take into account the underutilization of the processors due to stopping rule effects (Section 2.7.1); hence, the analysis presented here understates the case for intra-replication parallelism.

Appendix B

Expected Lookahead for PS Servers with Erlang-k Service Times

The Erlang-k probability distribution is given by [Allen 78]

$$F_k(x) = 1 - e^{-k\lambda x}\left[1 + \frac{k\lambda x}{1!} + \frac{(k\lambda x)^2}{2!} + \cdots + \frac{(k\lambda x)^{k-1}}{(k-1)!}\right] \tag{B.1}$$

Our goal in this section is to prove the following theorem:

Theorem B.1 *Suppose that service times at a PS server are distributed according to the Erlang-k probability distribution for some $k \geq 2$. Then for any $n \geq 0$,*

$$\lim_{N\to\infty} E[L_{PS}^N(n)] > 0$$

To prove this, we need the following two lemmas.

Lemma B.1 *If $X_1, X_2, \ldots, X_m$ are i.i.d. Erlang-2 random variables with parameter λ, n is a non-negative integer, and c is a real number such that $0 < c \leq (n+1)/2\lambda$, then*

$$\lim_{m\to\infty} \Pr(\min\{jX_{j-n} : n+1 \leq j \leq n+m\} > c) > 0 \tag{B.2}$$

Proof.

$$\begin{aligned} P &= \Pr(\min\{jX_{j-n} : n+1 \le j \le n+m\} > c) \\ &= \Pr\left[\bigwedge_{j=n+1}^{n+m} (jX_{j-n} > c)\right] \\ &= \prod_{j=n+1}^{n+m} \Pr(jX_{j-n} > c) \\ &= \prod_{j=n+1}^{n+m} \Pr(X_{j-n} > c/j) \\ &= \prod_{j=n+1}^{n+m} (1 - F_2(c/j)) \\ &= \prod_{j=n+1}^{n+m} e^{-2\lambda c/j}(1 + 2\lambda c/j) \end{aligned}$$

Taking the natural logarithm of both sides of this expression,

$$\begin{aligned} \log P &= \sum_{j=n+1}^{n+m} \log\left[e^{-2\lambda c/j}(1 + 2\lambda c/j)\right] \\ &= \sum_{j=n+1}^{n+m} [\log(1 + 2\lambda c/j) - 2\lambda c/j] \end{aligned}$$

Note that since $j \ge n+1$ and $c \le (n+1)/2\lambda$, $2\lambda c/j \le 1$. Making use of the fact that $\log(1+x) \ge x - x^2/2$ for $0 \le x \le 1$, we have

$$\begin{aligned} \log P &\ge \sum_{j=n+1}^{n+m} \left[\left(2\lambda c/j - \frac{(2\lambda c/j)^2}{2}\right) - 2\lambda c/j\right] \\ &= -\frac{1}{2}\sum_{j=n+1}^{n+m} \left(\frac{2\lambda c}{j}\right)^2 \end{aligned}$$

Therefore,

$$\lim_{m\to\infty} \log P \ge -\frac{1}{2}\sum_{j=n+1}^{\infty} \left(\frac{2\lambda c}{j}\right)^2$$

Since $2\lambda c/j < 1$, this infinite series converges, and thus

$$\lim_{m\to\infty} \log P > -\infty$$

$$\lim_{m\to\infty} P > 0 \blacksquare$$

Lemma B.2 *For any $k \geq 2$, if $X_1, X_2, \ldots, X_m$ are i.i.d. Erlang-k random variables, Equation (B.2) holds.*

Proof. Let $Y_1, Y_2, \ldots, Y_m$ be i.i.d. Erlang-2 random variables. Let $F_X(X_i)$ and $F_Y(Y_i)$ be the distribution functions of the X_i and Y_i, respectively. From (B.1), $1 - F_X(X_i) \geq 1 - F_Y(Y_i)$, i.e., for any $c \geq 0$,

$$\Pr(X_i > c) \geq Pr(Y_i > c)$$

The result follows directly from the proof of Lemma B.1. ∎

We are now ready to pro.e Theorem B.1.

Proof. Let $R_1, \ldots, R_n$ be the random variables representing the residual service times of the n customers in service and let $S_1, \ldots, S_m$ be the random variables representing the $m = N - n$ entries in the future list. Then for $c > 0$,

$$\begin{aligned}
&\Pr\left[L_{PS}^N(n) > c\right] \\
&\quad = \Pr[\min(\{nR_i : 1 \leq i \leq n\} \cup \{(n+i)S_i : 1 \leq i \leq N-n\}) > c] \\
&\quad = \Pr[(\min\{nR_i : 1 \leq i \leq n\} > c) \wedge (\min\{(n+i)S_i : 1 \leq i \leq N-n\} > c)] \\
&\quad = \Pr[\min\{nR_i : 1 \leq i \leq n\} > c] \cdot \Pr[\min\{(n+i)S_i : 1 \leq i \leq N-n\} > c]
\end{aligned}$$

Therefore, since n is fixed,

$$\begin{aligned}
&\lim_{N\to\infty} \Pr\left[L_{PS}^N(n) > c\right] = \\
&\qquad \Pr[\min\{nR_i : 1 \leq i \leq n\} > c] \\
&\qquad \cdot \lim_{N\to\infty} \Pr[\min\{(n+i)S_i : 1 \leq i \leq N-n\} > c] \qquad \text{(B.3)}
\end{aligned}$$

Since residual service times cannot be identically zero, $\Pr[\min\{nR_i : 1 \leq i \leq n\} > c]$ is the product of a finite number of non-zero terms, and hence is non-zero. Furthermore, if $c \leq (n+1)/2\lambda$, the limit on the right side of Equation (B.3) is non-zero by Lemma B.1. Therefore,

$$\lim_{N\to\infty} \Pr\left[L_{PS}^N(n) > c\right] > 0$$

which implies that

$$E[L_{PS}^{N}(n)] > 0 \blacksquare$$

Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols*

Yi-Bing Lin, Jean-Loup Baer, and Edward D. Lazowska

Department of Computer Science
University of Washington
Seattle, WA 98195

Technical Report 88-03-02

March 1988

*This work was supported in part by the National Science Foundation (Grants CCR-8619663, CCR-8702915, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, Boeing Computer Services, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

Abstract

This paper concerns the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols.

Simulations are often used in computer system design. Since simulations are time-consuming, it is natural to attempt to use parallel computing to accelerate them.

In a previous paper we devised and analyzed a general technique for the parallel trace-driven simulation of multiprocessor cache coherence protocols. In this paper we optimize our general technique to simulate various specific protocols. The surprising result is that using our technique, the processes simulating the caches often need to do little or no communication even when simulating shared references. Thus, linear or near-linear speedup is possible.

1 Introduction

This paper concerns the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols.

Trace-driven simulation is the usual approach for evaluating memory hierarchies [6][9]. Traces of memory references are gathered by interpretively executing programs and recording the memory locations that have been referenced. The simulation consists of processing this trace information while varying certain parameters. In the case of cache simulations, these parameters might include cache size and organization, and the goal would be to obtain estimates of, for example, hit ratios and processor and bus utilizations.

Since simulations are time-consuming, it is natural to attempt to use parallel computing to accelerate them. But obtaining reasonable speedups from parallel discrete-event simulations has proven to be challenging, as has analyzing the performance of such algorithms to determine, prior to implementation, whether or not they are worthwhile.

In a previous paper, we devised and analyzed a general technique for the parallel trace-driven simulation of multiprocessor cache coherence protocols. Roughly, this technique works as follows:

We begin with a separate reference trace for each processor/cache. Each trace includes two types of memory references: *private* and *shared*. A private reference to a cache j does not have any effect on caches other than j, whereas a shared reference may update the status of other caches. All caches can be simulated asynchronously in parallel as long as there are no shared references, but concurrency control of some sort is required when shared references are encountered. To illustrate this, consider the following example: Suppose two consecutive references e_1 and e_2 are private to cache j with time-stamps t_1 and t_2, $t_1 < t_2$. Once cache j has been simulated up to event e_1, event e_2 cannot be simulated unless it is known that no shared reference originating at some other cache occurs during the time period (t_1, t_2) which affects the state of cache j.

The basic idea of our simulation technique is to pre-process the input traces so that all shared references are inserted into each input trace. That is, for all j, $j \neq i$, the shared references to cache j are inserted in trace i (we call these references *inserted references*). Thus, all potential interactions among caches are identified before the simulation commences; we transform *conditional events* into *unconditional events* as proposed by Chandy and Misra [5]. The result is that deadlocks (if the Chandy-Misra simulation algorithm [4] [18] is used) or rollbacks (if the Jefferson algorithm [13][14] is used) can be avoided during the simulation. The overhead of the pre-processing is low (its time is proportional to the length of all traces, i.e., $O(kN)$ where k is the number of traces and N the number of references in a single trace), and it may be amortized over many simulations since one typically performs simulations with different parameters using the same input traces.

Of course, the fact that all traces include all shared references does not eliminate the need for

communication between the "simulation processes" representing the various caches. When a shared reference occurs, the execution of the cache coherence protocol must be simulated. Our earlier paper analyzed a general technique in which the communication was not tailored to any specific cache coherence protocol. The present paper shows that, beginning from our merged traces, huge savings in communication (and thus huge speedups in simulation time) can be achieved by tailoring the simulation technique to the specific cache coherence protocol under investigation.

Section 2 of this paper describes the input traces that are used by our technique. Section 3 summarizes the results from [16] about our basic algorithm. Section 4 describes the modified algorithm, tailored to specific cache coherence protocols. Section 5 analyzes the time requirements of the algorithm. Section 6 analyzes the buffer storage requirements of this algorithm. Finally, Section 7 presents our conclusions.

2 The Input Traces

There is one input trace for each cache; trace j corresponds to cache j. A trace initially consists of a string of $(operation, address)$ references where an operation is either *read* or *write* (an instruction fetch is considered a read). We assume that for each trace j, the shared references can be distinguished from the private references. We assume that the relative position of these shared references in each other trace i is known, so that we can insert all shared references in each trace j into the other traces i. The inserted references together with the shared references are called *interaction points*. The relationship among private references, shared references, inserted references, and interaction points is shown in Figure 1.

Construction of the merged trace, either on-line or off-line, is not expensive. In multiprocessor systems that require the programmer to explicity identify shared variables [19], we can write on-line the shared references simultaneously on all traces. If this is not possible, tracing techniques can either provide relative order of references [1] or a time stamp for each event [10][12] such that the order of two events can be decided in spite of the fact that they are from different traces. Such information can be used off-line to post-process the traces [9][6] so that the types of references are distinguished. This post-processing (really a preprocessing for either the sequential or the parallel simulations) has to be done only once. Its time is proportional to the length of all traces, i.e., $O(kN)$ where k is the number of traces and N the number of references in a single trace.

A basic assumption that we make in the analysis of the simulation process, and that we will indeed have to make when we perform the simulations themselves, is that the relative locations of the *interaction points* in the traces are independent of the system's configuration. Thus they can remain in the position in which they were originally recorded or placed in the pre-processing stage. This assumption is certainly valid for the shared references. In

the case of inserted references, the positions in which they can appear in a given trace depend on the time spent processing private references between shared references. Thus, modifying the system's organization, for example by increasing the cache sizes and hence their hit ratios, could influence the placement of inserted references. However, if all elements of the multiprocessor system are modified in a homogeneous way (e.g., if all cache sizes are increased by the same amount), then we can assume that the relative processing times of the private references remain in the same ratio and that keeping the inserted references in the same positions will not distort the simulation results.

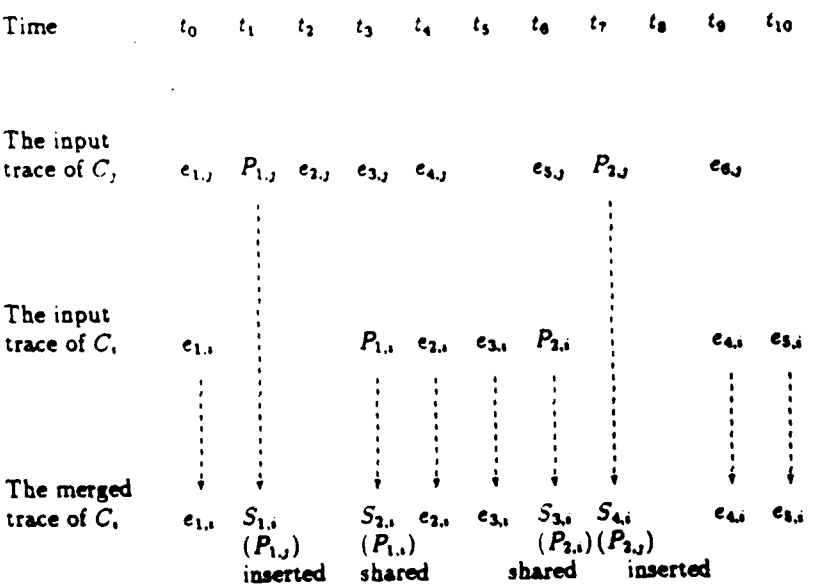


Fig. 1. The shared references, inserted references, and interaction points.
$e_{k,i}$: the k^{th} private reference to cache i.
$P_{k,i}$: the k^{th} shared reference to cache i.
$S_{k,i}$: the k^{th} interaction point to cache k.

3 The Basic Simulation Algorithm

In this section we describe our basic parallel simulation algorithm and summarize the results presented in [16]. We consider only the cache processes, which simulate the operations of caches, and do not dwell on the mundane input processes, which read the input traces and send the reference events to the cache processes.

In the basic algorithm, there is a barrier that synchronizes the execution of the cache processes at each interaction point. For each reference contained in merged trace i, there is a corresponding event at cache process C_i in the simulation. There are three possibilities for each event:

I. The event is a private reference. C_i continues its execution.

II. The event is a shared reference. C_i waits on the barrier until all other cache processes reach[1] this interaction point. Then C_i checks other caches' status, and decides its own status using the cache coherence protocol under investigation. All cache processes then leave the barrier, continuing execution.

III. The event is an inserted reference. C_i waits on the barrier until some other cache process (the one with the corresponding "shared" reference) consults C_i's status; it then continues execution.

It is not difficult to see that this algorithm is correct and deadlock-free. Before summarizing the performance analysis of this algorithm, we note that it is not necessary for all cache processes to "rendezvous" at each interaction point. Modifications that relax this restriction, sometimes with dramatic gains in efficiency, are the subject of Sections 4, 5, and 6 of this paper.

We begin our summary of the timing analysis of the basic algorithm by introducing some notation that is used throughout the paper:

- N: the total number of references to a cache (not including the inserted references).
- p_s: the proportion of references to a cache that are shared references.
- k: the number of caches to be simulated. (Throughout this paper, we assume $k \geq 2$.)
- Δt_s: the average time for processing a private reference event[2] in the uniprocessor environment.
- Δt_p: the average time to process an event (a private reference, or an interaction point) in the multiprocessor environment. Note that for an interaction point, Δt_p does not include the waiting time.
- $\rho = \frac{\Delta t_p}{\Delta t_s} \geq 1$: the synchronous factor that represents the extra overhead to support synchronization in the multiprocessor environment. This factor is determined by the overhead of the primitives available in the system (semaphore, monitor, or message passing primitives) and the programming ability of the user who implements the simulation.
- t_s: the sequential simulation time.
- t_p: the parallel simulation time when there are an unlimited number of available processors.

[1] We say that C_j "reaches" an event R or R "arrives at" C_j, iff the event to be processed by C_j is R.

[2] The time to process a shared reference is longer than that to process a private reference in the uniprocessor environment. (See the discussion in Section 5.)

The main results in [16] are:

Theorem 3.1: If the arrivals of interaction points to each cache form a Poisson process, with the same mean for each cache, then the expected value of the parallel simulation time with an unlimited number of processors, $E[t_p]$, is bounded by

$$E[t_p] \leq [1 + (k-1)p_s](1 + \frac{k-1}{\sqrt{2k-1}})N\Delta t_p$$

Theorem 3.2: If the arrivals of interaction points to each cache form a Poisson process, with the same mean for each cache, then the speedup with an unlimited number of processors, defined as $S_\infty = \frac{E[t_s]}{E[t_p]}$, is bounded by

$$S_\infty \geq \frac{k}{\rho(1 + \frac{k-1}{\sqrt{2k-1}})}$$

that is, the speedup is $O(\sqrt{k})$.

(We expect a degradation when p_s increases. Surprisingly, p_s does not have any effect on S_∞, because the degradation in parallel simulation time is matched by a degradation in sequential simulation time.)

Theorem 3.3: If there are n processors available, $1 \leq n \leq k$, and k is large, then a lower bound of the speedup with n processors is

$$S(n) \geq \sqrt{k} - \frac{k-n}{\sqrt{k}+n}$$

4 The Modified Algorithm

The basic algorithm requires that all cache processes synchronize at each interaction point. In this section we describe a modification that reduces this restriction. We prove that the modified algorithm is correct and deadlock free.

The idea of the modified algorithm is that at some interaction points, a cache process can determine its cache status using only local information (including that provided by its (merged) trace), so there is no need to consult the status of other caches. Thus we can process such interaction points in the same way as private references.

Whether an interaction point can be processed locally is dependent on the type of interaction (a shared reference, which is a *processor-induced transaction*, or an inserted reference, which is a "possible" *bus-induced transaction*) and the cache coherence protocol under investigation. In Table I below, the first column lists various pieces of information that are required by various cache coherence protocols, while the second and third columns indicate the sources of this information for processor-induced and bus-induced transactions, respectively:

information required	Source of information	
	processor-induced transaction	bus-induced transaction
current state of the block	status of the cache process	status of the cache process
read/write	arrival event	arrival event
hit/miss	status of the cache process	status of the cache process or other cache processes
actual sharing	status of other cache processes	status of the cache process

Table I: The sources of information required in a transaction.

From Table I, we see that synchronization is required during the simulation if

- the cache coherence protocol requires information about actual sharing for a processor-induced transaction, or
- the protocol cannot provide information about hit/miss for a bus-induced transaction.

We say a transaction (or an interaction point) is *synchronous* iff it requires synchronization in the simulation. We call an interaction point that requires synchronization a *synchronous point*. Otherwise, it is a *non-synchronous point*. As just noted, the specific cache coherence protocol under investigation has a bearing on whether an interaction point requires synchronization or not. Table II gives "synchronization information" about five protocols: Berkeley [15], Illinois [20], FBWO (Future bus write-once) [11], Firefly [23], and Dragon [17]. The first three of these protocols are based on "invalidation" while the last two are of the "distributed-write" type [2].

Protocols	Shared				Inserted	
	Read		Write		Read	Write
	hit	miss	hit	miss		
Berkeley	non-sync	non-sync	non-sync	non-sync	non-sync	non-sync
Illinois	non-sync	sync	non-sync	non-sync	non-sync	non-sync
FBWO	non-sync	sync	non-sync	non-sync	non-sync	non-sync
Firefly	non-sync	sync	non-sync	sync	non-sync	non-sync
Dragon	non-sync	sync	non-sync	sync	non-sync	non-sync

Table II: "Synchronization information" for different protocols.

What Table II shows is that a surprisingly large number of event types can be simulated without the need for synchronization. For example, an ownership-based invalidation protocol such as the Berkeley protocol does not require *any* synchronization. The other two

invalidation-based protocols require synchronization only at read-misses. For the distributed-write protocols, synchronization on write-misses is also required.

To illustrate this important observation, we shall prove the first row (Berkeley protocol) of Table II. The remaining rows can be proved by similar arguments. The following states [3][15] are used in the Berkeley protocol:

- **INVALID**: Copy is not up to date.
- **UNMOD-SHD**: Unmodified-shared; copy is not modified with respect to main memory. Other caches *may* have a copy.
- **MOD-SHD**: Modified-shared; other caches may have copies. Block needs to be written back when replaced.
- **MOD-EXC**: Modified-exclusive; no other copies exist. Block must be written back on replacement.

In the Berkeley protocol, a cache with a **MOD-SHD** or **MOD-EXC** copy is called the *owner*. There can be at most one such cache. Although a **MOD-SHD** copy indicates that other caches may have copies, the protocol ensures that they will be **UNMOD-SHD** copies. Thus there is at most one owner cache, and the owner is responsible for writing the block back to memory. If a block is not owned by any cache, memory is the owner.

The protocol works as follows: Let i be the requesting cache, and let j be any other cache that contains the requested block. [3]

A. The action of cache i:

I. Read hit: The state of cache i does not change; the requested block is sent to the processor.

II. Read miss: Cache i gets the copy either from cache j (if such j exists) or from memory (if no caches have the copy). The state of the block in cache i is set to **UNMOD-SHD**.

III. Write hit: If the block in cache i is **MOD-EXC**, the write proceeds with no delay. If it is **MOD-SHD** or **UNMOD-SHD**, an invalidation signal is sent and the state of the block in cache i is set to **MOD-EXC**.

IV. Write miss: As with a read miss, cache i gets the copy either from cache j (if such j exists) or from memory (if no caches have the copy). The state of the block in cache i is set to **MOD-EXC**.

[3] For those "other" caches that do not have the requested block, no action is taken

B. The response of cache j:

I. No signal is sent from cache i to cache j if cache i has the copy (read hit): No action is taken in cache j.

II. Read miss signal: If it is a read miss in cache i, then cache i sends a "read miss" signal to cache j. Cache j sends its copy to cache i, if it is the owner of the block. If the block is in state **UNMOD-SHD** then no action is taken. If the block is in state **MOD-SHD** then it remains in the same state. If the block is in state **MOD-EXC** then its state is set to **MOD-SHD**.

III. Invalidation signal: If it is a write hit in cache i, then cache i sends an "invalidation" signal to cache j. The block in cache j is set to **INVALID**.

IV. Write miss signal: If it is a write miss in cache i, then cache i sends a "write miss" signal to cache j. Cache j sends its copy to cache i, if it is the owner of the block. The block in cache j is set to **INVALID**.

Figure 2 shows the Berkeley protocol state transition diagram.

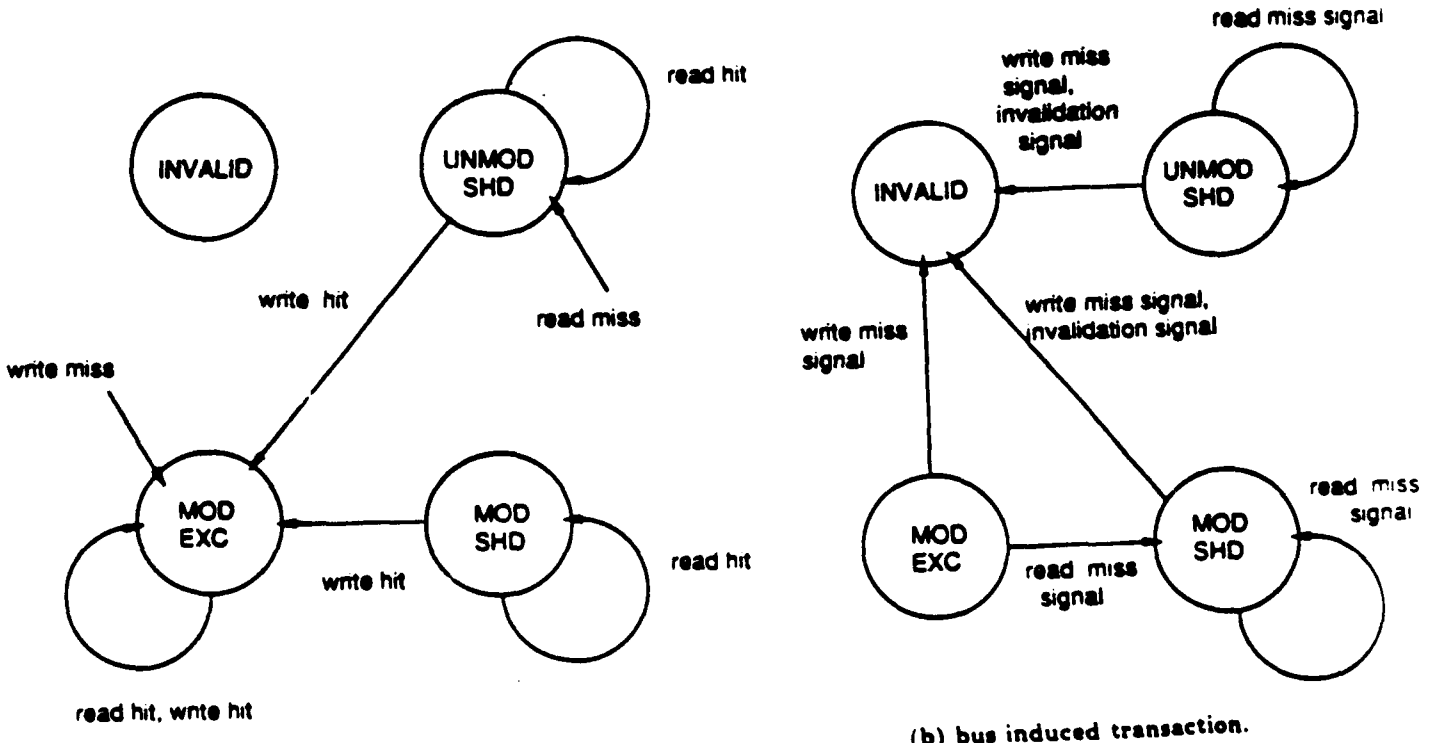


Fig. 2. The Berkeley protocol state diagram.

Theorem 4.1: All interaction points are *non-synchronous* it the Berkeley cache coherence protocol is used.

Proof : All shared references are non-synchronous since "sharing" information is not used in this protocol (**A** of the protocol description).

All inserted "write" references are non-synchronous since the next state of the block is set to **INVALID** independently of whether the operation is a hit or miss (**B.III** and **B.IV**).

All inserted "read" references are non-synchronous. If the block is in state **UNMOD-SHD** or **MOD-SHD**, the next state remains the same (**B.I** and **B.II**). If the block is in state **MOD-EXC**, then it corresponds to a miss in cache i and the next state is **MOD-SHD**.

QED

We reiterate the surprising result that if one begins from our easily-constructed merged traces, then the Berkeley protocol can be simulated in parallel with absolutely no communication between simulation processes. As shown in Table II, the Berkeley protocol is unique in this regard. However, for all protocols the required synchronization is considerably less than one would expect based on the number of interaction points, even for complex protocols such as the EIP protocol [3] and the CMU RW protocol [22] that have features of both the invalidation and the distributed-write protocols. We now present an algorithm that is general enough to handle any of these protocols. We refer to it as the "modified algorithm", in contrast to the "basic" algorithm appearing in our earlier paper, which required synchronization at each interaction point.

As in the basic algorithm, there are three possibilities for an event in the execution of a cache process C_i:

I. The event is a private reference. C_i updates its status, and advances to the next event.

II. The event is an inserted reference. Suppose that this inserted reference was generated by a shared reference, say r, in cache j.

II (a). The reference r is of the "non-sync" type for cache j: C_i updates its status, and advances to the next event.

II (b). The reference r is of the "sync" type for cache j: C_i first updates its status, and then generates a message, R_i to cache process C_j, which indicates whether there is a copy of the referenced block in cache i. C_i puts R_i in a queue called B_i[4] so C_j may check C_i's status via B_i. Then C_i advances to the next event.

[4] If B_i is full, C_i will wait till there is room in the buffer; cf. Section 6 for the analysis of space requirements

III. The event is a shared reference.

III (a). If the event is "non-sync", then C_i updates its status and advances to the next event.

III (b). If the event is "sync", then C_i waits until it has received messages R_j from each other process C_j, $(1 \leq j \leq k, j \neq i)$. C_i updates its status and continues.

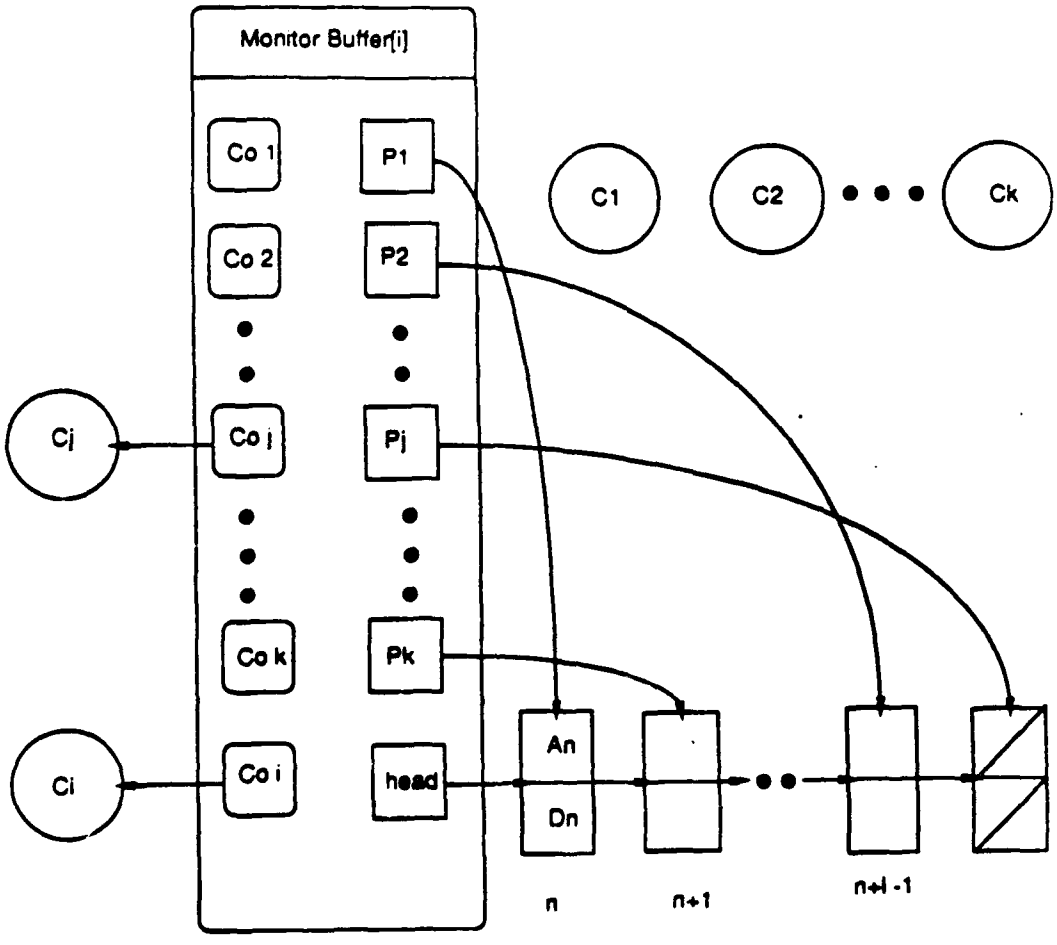


Fig. 3. An implementation of cache processes synchronization: C_j is blocked since it tries to insert a message in the queue which is full. When C_i needs to access the head of the queue, it is blocked since $D_n < k-1$ (C_i has not yet inserted a message at the head of the queue).

The communication of cache processes in the modified algorithm is analogous to the bounded buffer producer/consumer problem. An example of implementing a solution to this problem is shown in Figure 3; there is a monitor $Buffer[i]$ for each cache process C_i. The monitor consists of

- a queue, B_i, of length $l+1$,
- $k-1$ pointers ($P_j, 1 \leq j \leq k, j \neq i$) which point to elements of B_i, and
- k conditions $Co_j, 1 \leq j \leq k$.

An element in B_i consists of two parts: A_n denotes the number of caches that "share" the requested block of cache i at I_n [5]; D_n denotes the number of cache processes that have sent messages to C_i corresponding to I_n. C_i removes the first element of B_i only when $D_n = k-1$. The pointer P_j for C_j points to an element of B_i in which C_j must record its next message to C_i. This message, which corresponds to the insertion of I_n, always increments D_n by 1 and increments A_n by 1 if C_j was sharing the requested block. When P_j points to the $l+1^{th}$ element of B_i, the buffer is full. Therefore, if C_j tries to send a message to C_i, then it is blocked on the condition Co_j (see Figure 3). If C_i wants to access the first element of B_i, and some cache process has not yet sent the corresponding message, then C_i is blocked on condition Co_i.

Theorem 4.2: The modified algorithm is correct.

Proof: Correctness consists of two parts:

(a) Let R_1 and R_2 be events occurring at cache i, and let R_1' and R_2' be their simulation counterparts. We must show that R_1 arrives at cache i earlier than R_2 does, iff C_i processes R_1' earlier than R_2'. It is obvious that (a) is satisfied by the algorithm.

(b) We must show that the status of C_i before and after the arrival of R' reflects the status of cache i before and after the arrival of R. To show this, we must consider the three cases described earlier. It is easy to see that **I** satisfies (b). In **II**, the status of cache i only depends on the operation (read/write) of the shared reference to cache j. Such information is provided by R, and the change of status of C_i reflects that of cache i. Thus, the execution of C_i continues without being blocked. In **III (a)**, the status of cache i is determined by the arrival event. In **III (b)**, the status of cache i depends on the status of all other caches. Such information is provided by $R_j, 1 \leq j \leq k,\ j \neq i$ by other cache processes (**II (b)**). Thus C_i must wait until other caches reach the same synchronous point, and the change of C_i's status reflects that of cache i.

QED

Two important issues that arise in parallel and distributed algorithms are *termination* and *deadlock*. In the modified algorithm, the termination problem is easily solved by adding an end-mark event at the end of each input trace. Freedom from deadlock is shown in Theorem 4.3:

[5] The information provided by A_n is required to determine "shared high" or "shared low" in protocols such as FBWO, Firefly, and Dragon. In fact, a single bit would be sufficient to indicate the presence/absence of sharing.

Theorem 4.3: The modified algorithm is deadlock free.

Proof: We show that there is no *circular wait* [21] among cache processes as follows:

Let I and J be shared references to cache i and j respectively. Then there are only two cases in which C_i waits for C_j (denoted as $C_i \rightarrow C_j$). The first case is that C_i reaches I earlier than C_j does. The second case is that C_i reaches J earlier than C_j does and B_i is full. (It is obvious that we can ignore the case in which C_i reaches L ($L \neq I, J$) earlier than C_j does.)

In both cases, C_i reaches some synchronous point earlier than C_j does. If a circular wait is formed during the execution, then there exist n cache processes C_{i_1}, $C_{i_2}, ..., C_{i_n}$, $2 \leq n \leq k$, $1 \leq i_1, i_2, ..i_n \leq k$, such that $C_{i_1} \rightarrow C_{i_2}$, $C_{i_2} \rightarrow C_{i_3}$, ..., $C_{i_n} \rightarrow C_{i_1}$. This circular wait exhibits a contradiction involving C_{i_1}.

QED

Our previous paper [16] showed a generally-applicable cache simulation technique. (It also included a careful performance analysis, which was the real point of the paper.) Our current paper shows that dramatic improvements can be achieved by considering the details of the specific cache coherence protocol being simulated. This was shown qualitatively in this section and will be analyzed quantitatively in the next one. Our results support the claim that it may not be highly worthwhile to try to build a general parallel simulation system, without considering the structure of the problem under consideration. Of course, it is worthwhile to build tools for building special-purpose simulation systems.

5 Timing Analysis

5.1 Sequential Simulation

Problems such as the performance analysis of shared-memory multiprocessor systems reveal the substantial costs of sequential simulation [2]. In addition to the overhead of maintaining a large event queue, the costs of processing shared references are large: searches in all caches are required, so the time to process a shared reference is k times the cost of processing a private reference. One may argue that under most coherence protocols, searches in all caches are not necessary when there is a read hit. Since we can modify the modified algorithm such that a cache process never waits at "read hit" shared references, it is likely that the modified algorithm would benefit more from this optimization than any sequential algorithm does. To keep the time complexity analyses simple and clean, however, we assume that:

- the time to process a "read hit" shared reference is k times the cost to process a private reference in sequential simulation,

- a cache process always waits at a "read hit" shared reference in parallel simulation, and
- a cache process always waits at a "write" shared reference.

We add the following notation to that introduced in Section 3:

- $m = Np_s$: the total number of shared references to a cache.

The sequential execution time is

$$E[t_s] = k \times [\textit{time to process private references} + \textit{time to process shared references}]$$

$$= k[N - m + km]\Delta t_s = k[1 + (k-1)p_s]N\Delta t_s \quad (1)$$

If $N\Delta t_s$ is approximated as being constant, i.e., independent of k[6], then the relationship between $E[t_s]$, k, and p_s is shown in Figure 4. Note that k, the number of caches being simulated, has a dominant effect on $E[t_s]$.

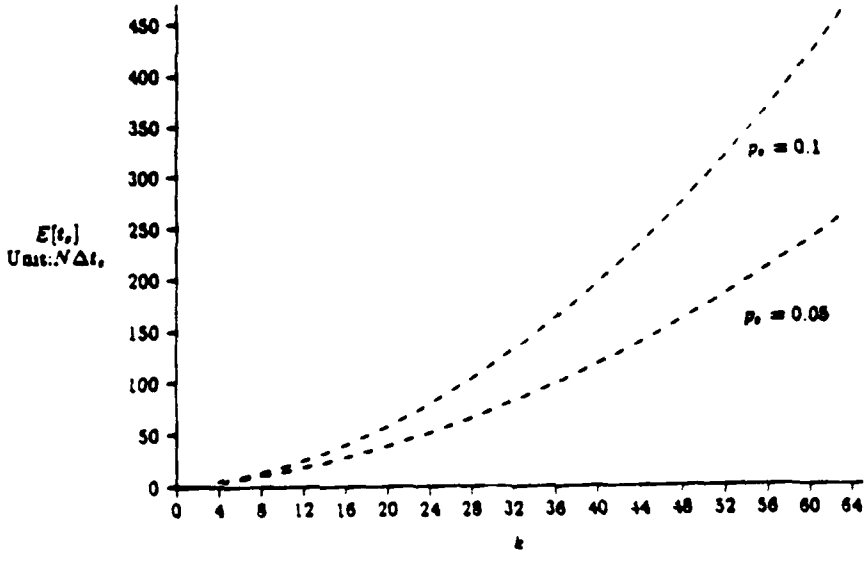


Fig. 4. The relationship between the sequential time $E[t_s]$, the number of cache k and the probability p_s that a reference is shared.

[6] In fact, Δt_s should increase when k increases, since the overhead of operations on some data structures (e.g., the event queue) increases.

5.2 Parallel Simulation

5.2.1 The Parallel Simulation Time with an Unlimited Number of Processors

The parallel simulation for the Berkeley cache coherence protocol does not require any synchronization (Theorem 4.1). With an unlimited number of available processors, the parallel simulation time is just the time to process a merged trace. We expect the speedup with an unlimited number of processors to be close to k, the number of caches (of course, the speedup is less than k, partly because the merged traces are larger, and partly because of our pessimistic assumptions on the processing of "read hit" shared references).

The time analyses for simulations of the protocols listed in Table II other than the Berkeley protocol are not trivial because synchronizations are required in the simulation. Thus processes will have to wait for each other at the various synchronous points. One main result is that the expected time spent waiting for synchronization is at worst equal to the time processing events, i.e., at least half of the simulation time is spent in useful work. In order to derive analytically this upper bound on the expected parallel computation time $E[t_p]$, we make the following assumptions:

P.1 We have an unlimited number of processors. The case of a limited number of processors will be treated in the next subsection.

P.2 A cache process never waits at its inserted references. Or. in other words, the probability of buffer overflow is nil. (We shall see in Section 6 that this is a very reasonable assumption since the probability of overflow for an $O(k)$ amount of buffering is low.)

P.3 The behaviors of the processes being traced are statistically identical; in particular they have the same length (N references) prior to the insertion of shared references , the same number of shared references m for a total length of $N+(k-1)m$ references, and the times to process events such as private and "non-sync" references are of the same order of magnitude.

P.4 The arrivals of synchronous points to each cache form a Poisson process with the average arrival rate λ being the ratio of the number of synchronous points over the time to process the whole merged trace, where

$$\lambda = \frac{km}{(N-m+km)\Delta t_p}$$

This follows from assuming that the time for a cache process to handle all private and "non-sync" references between two synchronous points is exponentially distributed.

P.5 If we denote by $W^{(j)}$ the time that C_i waits for C_j, then the waiting times $W^{(1)}, W^{(2)}, \ldots, W^{(k)}$ (excluding $W^{(i)}$ which is null) are independent, identically distributed random variables. This assumption follows **P.3**.

P.6 Let S_n be a synchronous point where two cache processes, say C_i and C_j, handle their next references, most likely private references, at the same time and let S_{n+s}, $s \geq 1$, be the next synchronous point where one of these two processes has to wait for the other. Then $[S_n, S_{n+s}]$ is called a *synchronization interval*. We assume that during a synchronization interval between cache processes C_i and C_j, C_i will wait for all cache processes aside from C_j the same amount of time that C_j will wait for all cache processes aside from C_i.

If a synchronous point S is a shared reference to cache i, then the cache process C_i waits until all cache processes reach that synchronous point. The expected time for the parallel simulation (i.e., to process one trace, since they all take approximately the same amount of time) is therefore the time to process all private references plus the expected waiting time at each shared reference. An upper bound of the parallel simulation time $E[t_p]$ is given by the following expression:

$$E[t_p] = N[(1 + (k-1)p_s)\Delta t_p + p_s E[W]] \tag{2}$$

where W is the maximum of the waiting times at a given synchronous point:

$$W = \max_{1 \leq j \leq k, i \neq j} W^{(j)} \tag{3}$$

David showed [7] that if $W^{(1)}, W^{(2)}, ..., W^{(k)}$ are independent, identically distributed random variables with mean u and standard deviation s then

$$E[W] = E[\max_{1 \leq j \leq k} W^{(i)}] \leq u + \frac{k-1}{\sqrt{2k-1}} s \tag{4}$$

The mean u, or $E[W^{(j)}]$, is derived as follows: As shown in Figure 5, C_i and C_j are synchronized at S_n (i.e., both C_i and C_j handle the next references, most likely private references, at the same time). After s synchronous points, a shared reference to cache i arrives. There are two possible cases:

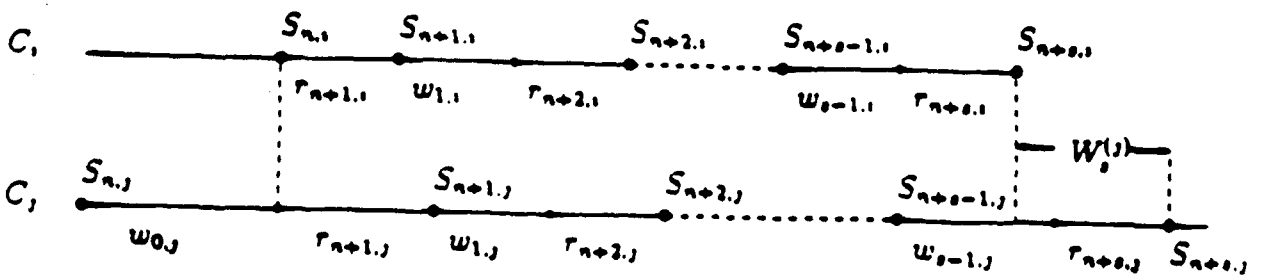


Fig. 5 The waiting time diagram.

- C_j reaches S_{n+s} earlier than C_i does. In this case, C_i does not wait for C_j and $W_s^{(j)} = 0$.
- C_i reaches S_{n+s} earlier than C_j does. Then

$$W_s^{(j)} = \sum_{l=1}^{s} r_{n+l,j} + \sum_{l=1}^{s-1} w_{l,j} - \sum_{l=1}^{s} r_{n+l,i} - \sum_{l=1}^{s-1} w_{l,i} > 0 \tag{5}$$

where $r_{n+l,j}$ is the time for C_j to handle private and non-sync references between S_{n+l-1} and S_{n+l}, and $w_{n+l,j}$ is the waiting time of C_j at S_{n+l}. (Note that $w_{n+l,j} = 0$ if S_{n+l} is not a J.)

According to assumption **P.6** the waiting times for all caches other than C_i and C_j cancel out and

$$\sum_{l=1}^{s} w_{l,j} \simeq \sum_{l=1}^{s} w_{l,i} \tag{6}$$

With this simplification, we have

$$W_s^{(j)} = \begin{cases} 0 & \text{if } C_j \text{ reaches } S_{n+s} \text{ earlier than } C_i \text{ does} \\ t_1 - t_2 & \text{otherwise} \end{cases}$$

where

$$t_1 = \sum_{l=1}^{s} r_{n+l,j}, \quad t_2 = \sum_{l=1}^{s} r_{n+l,i}$$

According to the assumption of Poisson arrivals for synchronous points (**P.4**), the time to handle private and non-sync references between S_n and S_{n+s} has an ***Erlang distribution*** with the probability density function

$$\frac{\lambda}{(s-1)!}(\lambda t)^{s-1} e^{-\lambda t}$$

This will allow us to derive as an upper bound of $E[W^{(j)}]$ (cf. Appendix B):

$$E[W^{(j)}] \leq \frac{\sqrt{k}}{\lambda}$$

Similarly, we can derive an upper bound of the standard deviation s of $W^{(j)}$, $\sqrt{V[W^{(j)}]}$, as:

$$\sqrt{V[W^{(j)}]} \leq \frac{\sqrt{k}}{\lambda}$$

This leads us to our main result:

Theorem 5.1: For all $k \geq 2$, $E[t_p] \leq 2[1 + (k-1)p_s]N\Delta t_p$

Proof: See Appendix B.

$E[t_p]$ consists of three parts:

- $E[t_{private}]$: the expected elapsed time for handling all private references

$$E[t_{private}] = (N - m)\Delta t_p = (1 - p_s)N\Delta t_p \tag{7}$$

- $E[t_{sync}]$: the expected elapsed time for handling all synchronous points

$$E[t_{sync}] = km\Delta t_p = kp_sN\Delta t_p \tag{8}$$

- $E[t_{wait}]$: the expected waiting time in the simulation.

Therefore, it follows from Theorem 5.1 that

$$E[t_{wait}] \leq [1 + (k - 1)p_s]Nt_p = E[t_{private}] + E[t_{sync}]$$

The importance of this result is that a processor spends more time doing useful work than waiting. This is illustrated in Figure 6.

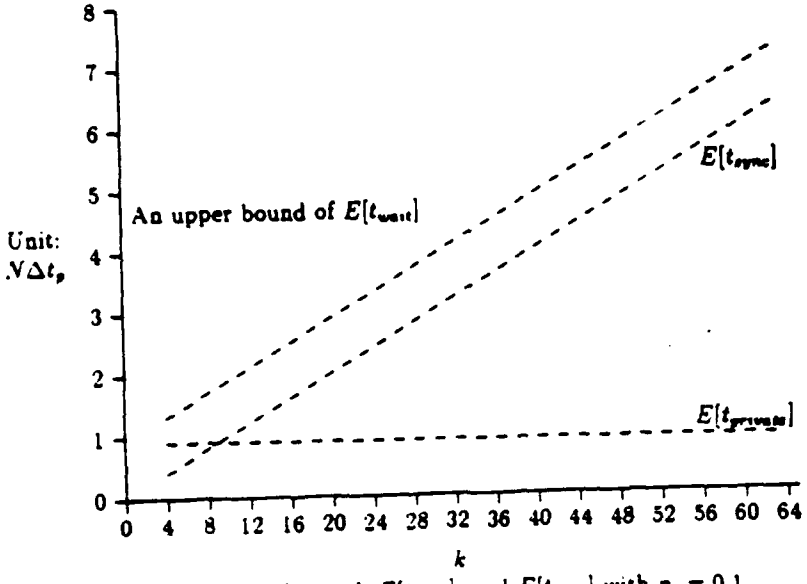


Fig. 6. $E[t_{private}]$, $E[t_{sync}]$, and $E[t_{wait}]$ with $p_s = 0.1$.

We observe that for small k, ($k \leq 8$), the parallel simulation times with different p_s values are roughly the same. On the other hand, when k increases, $E[t_p]$ with larger p_s grows faster. This phenomenon is explained as follows. When $(k-1)p_s << 1$, i.e., with very little sharing, we have

$$E[t_p] \approx N\Delta t_p$$

which is independent of k and p_s and represents an ideal case. When $(k-1)p_s >> 1$, i.e., with a fair amount of sharing and a large number of caches,

$$E[t_p] \approx (k-1)p_s N \Delta t_p$$

which is linearly proportional to k and p_s.

5.2.2 Speedup with an Unlimited Number of Processors

With an upper bound on the parallel simulation time, we can now derive lower bounds on achievable speedup.

Corollary 5.1: An upper bound on the speedup with an unlimited number of processors is

$$S_\infty \geq \frac{k[1+(k-1)p_s]N\Delta t_s}{2[1+(k-1)p_s]N\Delta t_p} = \frac{k}{2\rho}$$

Proof: Directly from Section 5.1 and Theorem 5.1.

QED

We expect a degradation when p_s increases. As with the basic algorithm, though, p_s does not have any effect on S_∞, because the degradation in $E[t_p]$ is matched by a degradation in $E[t_s]$. The relationship between S_∞ and k is linear.

5.2.3 Speedup with a Finite Number of Processors

If we consider the (realistic) case in which the number of runnable subtasks (the cache processes) exceeds the number of available processors, the speedup is not equal to S_∞. We assume that the processor scheduling discipline is *processor sharing*. Under this discipline, if k subtasks are eligible for execution and there are n available processors, $n < k$, each subtask receives service at a rate that is n/k times the rate at which it would receive service if a processor were dedicated to it. According to Eager, Zahorjan, and Lazowska [8] we have the following theorem:

Theorem 5.2 [Eager-Zahorjan-Lazowska]: Let S_∞ denote the speedup with an unlimited number of available processors, m_{max} the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number is available), and $S(n)$ the speedup with n processors. If $n < m_{max}$, with processor sharing scheduling:

$$S(n) \geq \frac{nS_\infty}{n + S_\infty - 1 - \frac{(n-1)(S_\infty-1)}{m_{max}-1}}$$

Thus, we have

Corollary 5.2: If there are n processors available in the parallel simulation, $1 \leq n \leq k$, and $k \geq 2$, then

$$S(n) \geq \frac{n(k-1)}{(2\rho-1)n+k-2\rho}$$

Proof: According to Theorem 5.2

$$S(n) \geq \frac{nS_\infty}{n+S_\infty-1-\frac{(n-1)(S_\infty-1)}{k-1}} = \frac{nS_\infty(k-1)}{(n+S_\infty-1)k-nS_\infty}, \quad 1 \leq n \leq k$$

Applying Corollary 5.1

$$S(n) \geq \frac{n(k-1)}{(2\rho-1)n+k-2\rho}$$

QED

Figure 7 shows the relationship among S, n, and k for $\rho = \sqrt{2}$. There are two simple upper bounds on speedup [8]. The *hardware bound* reflects the limitation imposed by the hardware, and is given by the number n of available processors (line *hb* in Figure 7). This bound can be achieved only if all n processors can be kept busy all of the time. The ***software bound***, S_∞ reflects the limitation imposed by the software (lines $S_{\infty,36}$ and $S_{\infty,64}$. Corollary 5.2 gives the lower bound of speedup with $k = 36$ and $k = 64$ (lines S_{36}, and S_{64})).

We observe that:

- For fixed n, the speedup S increases when k increases. This phenomenon is explained as follows: When an active cache process blocks itself, the associated processor, P, becomes a free processor and can execute the next cache process on the ready queue. If the ready queue is empty, then P is idle. The probability that the ready queue is not empty increases when k increases. In other words, when k increases, processors are unlikely to be idle, and are devoted to useful work, and thus increase the speedup.
- For fixed k, the speedup S is about 80% of S_∞ when $n = 0.6k$. After that point, an increase in n does not provide much gain. This phenomenon is explained as follows: When n increases, it is likely that the number of free processors is more than the number of processes on the ready queue. Thus, there are more opportunities for some free processors to be idle, and the speedup cannot be improved significantly.

An important conclusion to draw is that we can choose an appropriate number, n', of processors (say, $n' = 0.6k$ and $n' \leq n$) to perform the simulation. The remaining $n - n'$ processors can be used to support functions such as statistics collection [24]. Or, in the simulation of set-associative caches, we may partition processors into groups to simulate different sets of caches.

Similar observations can be drawn for the basic algorithm. Figure 8 shows the speedup for the basic algorithm with the same parameters as in Figure 7. Note that when n is large, increasing n in the modified algorithm gains more than that in the basic algorithm. This is because the opportunity for a cache process to wait in the modified algorithm is less than that in the basic algorithm. Thus the added processors in the modified algorithm always contribute more than that in the basic algorithm (as long as $n \leq k$).

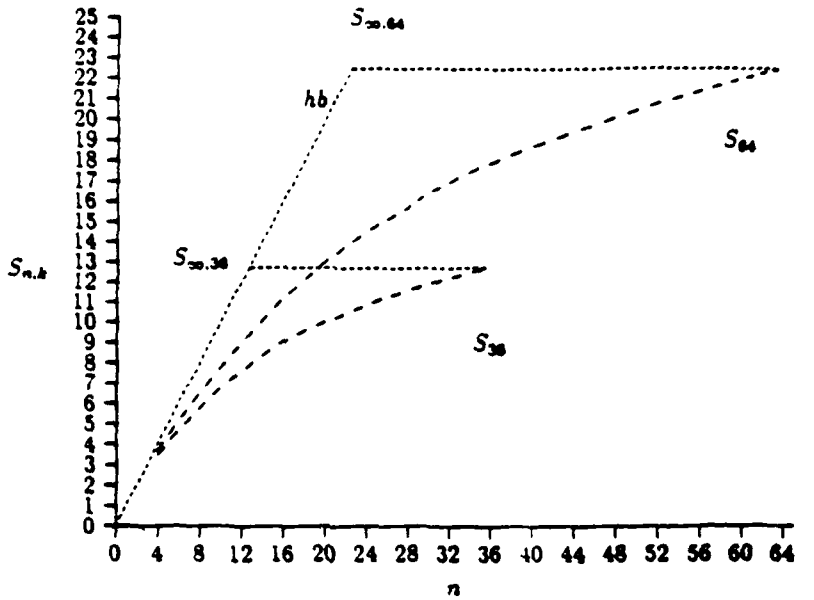


Fig. 7. The relationship between S, n, and k (the modified algorithm).

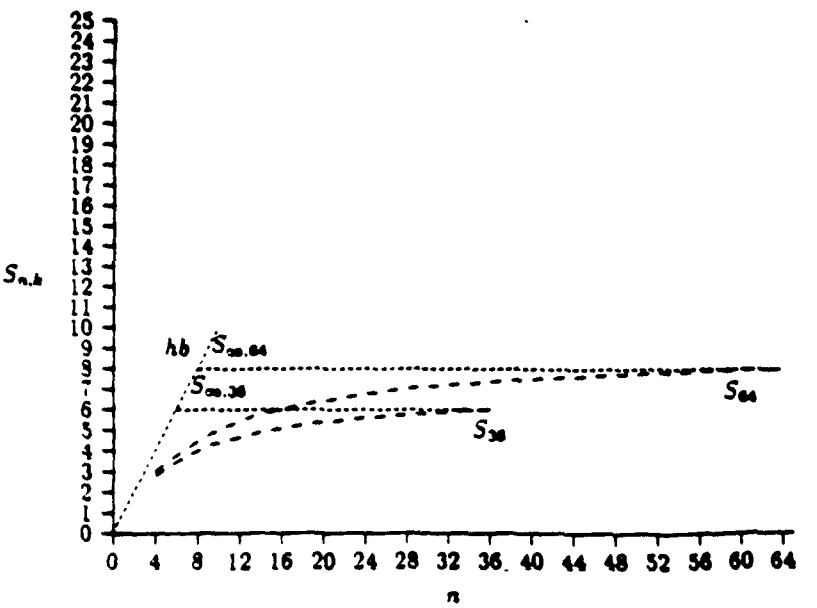


Fig. 8. The relationship between S, n, and k (the basic algorithm).

6 Space Analysis

In the modified algorithm, we use an array of buffers, or queues, $B = \{B_i | \ 1 \leq i \leq k\}$ for communication between cache processes (see Figure 3). If B had infinite capacity then the cache processes would never have to wait at the synchronous points induced by inserted references (case II (b) of the modified algorithm in Section 4). But since B is necessarily finite, we need to balance space requirements (amount of buffering) and time requirements (waiting time due to buffer congestion). In this section we show that with a low probability of buffer overflow, the space complexity of the total buffer size, $|B|$, is $O(k)$ with a small constant. We first introduce the additional notation:

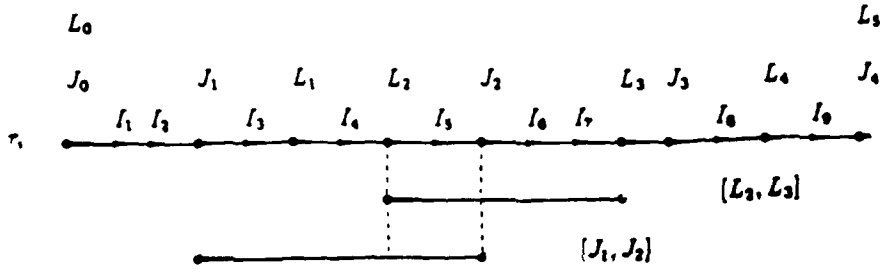


Fig. 9. The input trace, the interval, the boundary set, and the latest interval.

- m_i (m_j): the number of shared references that require synchronization (the "sync" references in Table II) to cache i (j).
- I_n (J_n): the n^{th} such shared reference to cache i (j), $1 \leq n \leq m_i$ (m_j).
- τ_i: the merged input trace for cache i.
- $[J_n, J_{n+1}] \subseteq \tau_i$, $1 \leq j \leq k, 0 \leq n \leq m_j$: an interval of the input trace i which starts at J_n and ends at J_{n+1} with no other J in between. By convention, we have two *virtual* events J_0 and J_{m_j+1} at the beginning and the end of τ_i.

  Note that each shared reference I_n in τ_i is included in exactly $k-1$ intervals, with each of the intervals being contributed by a cache process C_j, $1 \leq j \leq k$, $j \neq i$.
- Ω_{I_n}, or *the boundary set* of I_n: the set of all intervals that contain I_n. Note that there are exactly $k-1$ intervals in a boundary set.
- ψ_{I_n}, or *the latest interval* in Ω_{I_n}: Let $\psi_{I_n} = [J_{n'}, J_{n'+1}] \in \Omega_{I_n}$. Then $J_{n'+1}$ has the largest "time-stamp" among all events in Ω_{I_n}. i.e., all events in Ω_{I_n} are processed earlier than $J_{n'+1}$ is.
- $\theta_{J_n,I}$: the number of I's in $[J_n, J_{n+1}]$.

- b_i: the number of buffers in B_i that have not yet been consumed by C_i.
- $b_{i,j}$: the number of messages sent from C_j to C_i. Those messages have not been processed by C_i yet. It is clear that $b_{i,j} \leq b_i$.

Figure 9 shows an input trace τ_i for cache i for $k = 3$ (i.e., there are three cache processes i, j, and l): Interval $[L_2, L_3] = \{L_2, I_5, J_2, I_6, I_7, L_3\}$, and interval $[J_1, J_2] = \{J_1, I_3, L_1, I_4, L_2, I_5, J_2\}$. The boundary set for I_5 is $\Omega_{I_5} = \{[L_2, L_3], [J_1, J_2]\}$ since $I_5 \in [L_2, L_3] \cap [J_1, J_2]$. The latest interval in Ω_{I_5} is $\psi_{I_5} = [L_2, L_3]$ and $e_l = L_3$. $\theta_{J_1,i} = 3$ is the number of I's in $[J_1, J_2]$.

Lemma 6.1. If C_i reaches J_n, $1 \leq n \leq m_j$, then $b_{i,j} = 0$.

Proof: It is clear that if C_i reaches S_l then C_i has processed all $S_{l'}$, $1 \leq l' \leq l$. When C_i reaches J_n, C_j is in one of the following two states:

1. C_j has not yet reached J_n: it is obvious that $b_{i,j} = 0$.
2. C_j reaches J_n earlier than C_i does: C_j waits until all other cache processes reach J_n. Thus, C_j does not send any new message until C_i reaches J_n. When C_i reaches J_n, it has already consumed all events in the queue, and thus $b_{i,j} = 0$.

QED

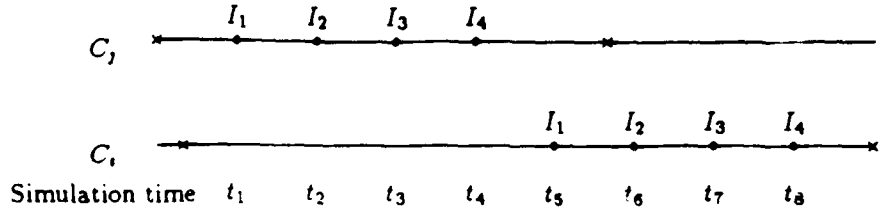
Let S be an arbitrary synchronous point, and let I and J be arbitrary shared references to cache i and j respectively. Lemma 6.1 tells us that $b_{i,j}$ is periodically reduced to 0; that is, if there are s I's between J_n and J_{n+1}, then the value of $b_{i,j}$ is bounded by s in the interval $[J_n, J_{n+1}]$. Figures 10 (a) and (c) show the worst case of the maximum number of required buffers in $[J_n, J_{n+1}]$ when $s = 4$: C_j produces 4 messages before C_i consumes any. Thus $b_{i,j}$ grows from 0 to 4, and then reduces to 0. Figures 10 (b) and (d) show the best case when $b_{i,j}$ alternates between 0 and 1 during the interval. For the buffer analysis we shall consider the worst case.

We first give an expression for an upper bound, b_i, of the number of messages that C_i might have to process when reaching a shared reference in its own trace.

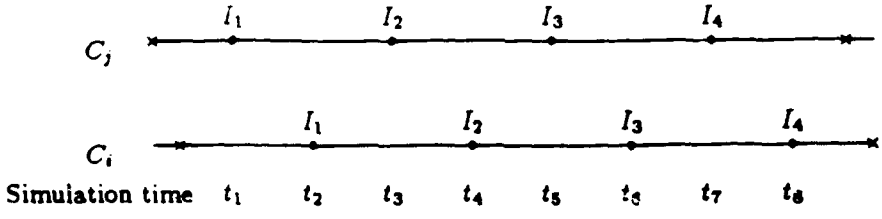
Theorem 6.1: Let Ω_{I_n} be the boundary set of I_n, and $\psi_{I_n} = [J_{n'}, J_{n'+1}]$ be the latest interval. If C_i reaches I_n, then an upper bound on b_i is $\theta_{J_{n'},i}$.

Proof: Let $[L_{n_l}, L_{n_l+1}] \in \Omega_{I_n}$, for all $1 \leq l \leq k$, $l \neq i$. When C_i reaches I_n, the maximum value for b_i is equal to the number of I's in $[I_n, J_{n'+1}]$, because those processes C_l that reach L_{n_l+1} earlier than C_i does, have to wait for C_i, and cannot send any more messages to C_i (by Lemma 6.1). Since $I_n \in [J_{n'}, J_{n'+1}]$, we have $[I_n, J_{n'+1}] \subseteq [J_{n'}, J_{n'+1}]$. Thus, $b_i \leq \theta_{J_{n'},i}$.

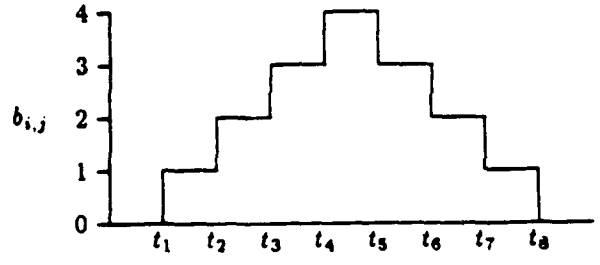
QED



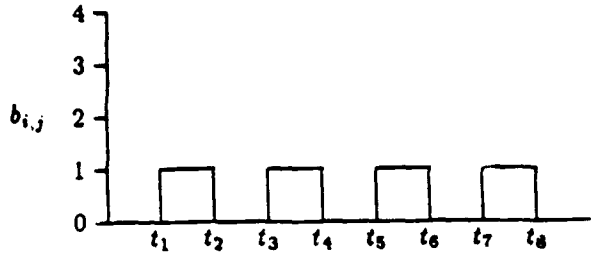
(a) The order of arrival events in C_i and C_j in the worst case
A "×" denotes a J, and a "·" denotes an I.



(b) The order of arrival events in C_i and C_j in the best case
A "×" denotes a J, and a "·" denotes an I.



(c) The behavior of $b_{i,j}$ in the worst case



(d) The behavior of $b_{i,j}$ in the best case

Fig. 10 The behavior of $b_{i,j}$.

Consider the example in Figure 9: When C_i reaches I_5, C_j can at most reach J_2 and must wait at J_2 since C_i has not reached J_2 yet. Similarly, C_l can at most reach L_3. Thus, the possible messages queued in B_i are messages for I_6, and I_7. And $b_i \leq 2 < 3 = \theta_{L_2,i}$.

The next step is to find the value $\theta_{J_{n'},i}$ for the latest interval. Let S be a synchronous point. From our assumptions of identical statistical cache behaviors, we have

$$p_{i,j} = Prob\{\ an\ S\ is\ a\ J \mid an\ S\ is\ either\ an\ I\ or\ a\ J\ \} = \frac{m_j}{m_i + m_j}$$

and

$$q_{i,j} = 1 - p_{i,j} = Prob\{\ an\ S\ is\ an\ I \mid an\ S\ is\ either\ an\ I\ or\ a\ J\ \} = \frac{m_i}{m_i + m_j}$$

To estimate the buffer space complexity, we construct a two-state Markov chain $\{Q\}$ with transition probability matrix

$$P = \begin{bmatrix} q_{i,j} & p_{i,j} \\ 0 & 1 \end{bmatrix}$$

and the initial probabilities

$$P\{Q_0 = 0\} = q_{i,j}, \quad P\{Q_0 = 1\} = p_{i,j}$$

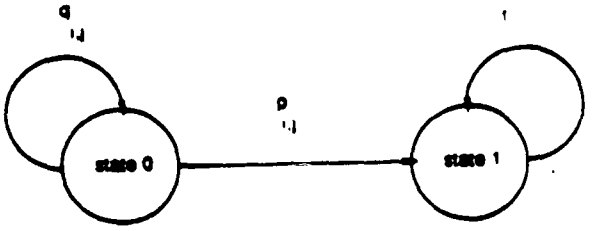


Fig. 11 The state diagram of the buffer model.

Figure 11 shows the state diagram. State 0 represents the fact that we had a sequence of I's after J_n before encountering J_{n+1}. State 1 is absorbing. The number of steps that the Markov chain stays in state 0 is equal to $\theta_{J_{n'},i}$, i.e., an upper bound on the amount of buffering needed by C_i. In the following analyses, we answer two questions:
(1) How long does it take to reach state 1, i.e., what is this upper bound, and
(2) What is the probability that the Markov chain remains in state 0 longer than some

predetermined number of steps $l_{i,j}$, i.e., given some buffer length what is the probability of overflow?

Let $\theta_{i,j} = E[\theta_{J_n,i}]$ where $\theta_{i,j}$ can be viewed as the expected number of steps to reach state 1, or the expected number of I's between two consecutive J's. Thus, we rephrase the above questions as: (1) What is the expected number of I's between two consecutive J's, and (2) What is the probability δ that $\theta_{i,j} > l_{i,j}$?

Lemma 6.2: Let T be the number of steps that it takes for the Markov chain to reach state 1, and let $\delta = Prob\{T > l_{i,j}\}$. Then

$$(a)\ \theta_{i,j} = \frac{m_i}{m_j} \quad (b)\ \delta = q_{i,j}^{l_{i,j}+1}$$

Proof:

(a) $\theta_{i,j} = E[T] = \sum_{n=0}^{\infty} n q_{i,j}^n p_{i,j} = \frac{p_{i,j}q_{i,j}}{(1-q_{i,j})^2} = \frac{q_{i,j}}{p_{i,j}} = \frac{m_i}{m_j}$

(b) Since

$$Prob\{T = k|Q_0 = 0\} = Prob\{Q_k = 1\}Prob\{Q_{k-1} = 0\} = p_{i,j}(q_{i,j})^{k-1}, \quad k = 1, 2, ...$$

$\forall l_{i,j} > 0$ we have

$$Prob\{T \leq l_{i,j}|Q_0 = 0\} = \sum_{k=1}^{l_{i,j}} p_{i,j}(q_{i,j})^{k-1} = 1 - (q_{i,j})^k \text{ and } Prob\{T > l_{i,j}|Q_0 = 0\} = (q_{i,j})^k$$

Clearly, $\forall l_{i,j} > 0, \quad Prob\{T > l_{i,j}|Q_0 = 1\} = 0$. According to the above relations,

$$\forall l_{i,j} > 0, \quad \delta = Prob\{T > l_{i,j}\} = (q_{i,j})^{l_{i,j}+1}$$

QED

Now, if we give ourselves some probability of overflow δ, we can derive an upper bound on the number of buffer entries needed before we reach the overflow with δ probability. We compute the amount of buffering needed by each cache process and sum up over all processes.

Let $q_{i_m,j_m} = \max_{1 \leq i,j \leq k} q_{i,j}, i \neq j$. Then we have the following theorem:

Theorem 6.2: The upper bound of the buffer size $|B|$ with overflow probability δ is

$$|B| \leq k(\lceil \log_{q_{i_m,j_m}} \delta \rceil - 1)$$

Proof: According to Lemma 6.2 (b), we have

$$l_{i,j} = \begin{cases} \lceil \log_{q_{i,j}} \delta \rceil - 1 & \text{if } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

According to Theorem 6.1 and the fact that, if $0 \leq q' \leq q \leq 1$ and $0 < \delta < 1$, $\lceil \log_q \delta \rceil \geq \lceil \log_{q'} \delta \rceil$, we have

$$b_i \leq \lceil \log_{q_{i_m,j_m}} \delta \rceil - 1 \quad \forall i,\ 1 \leq i \leq k$$

Thus,

$$|B| = \sum_{i=1}^{k} b_i \leq k(\lceil \log_{q_{i_m,j_m}} \delta \rceil - 1)$$

QED

The relationship between this upper bound of b_i, δ, and q_{i_m,j_m} is shown in Figure 12 with $\delta = 0.1$ and 0.05.

Since the caches exhibit similar behaviors, we can assert that $0.3 \leq q_{i,j} \leq 0.7$ hence $q_{i_m,j_m} = 0.7$. Then, according to Theorem 6.2, the upper bound of $|B|$ is $6k$; that is, under reasonable assumptions for the distribution of shared references, the space complexity of B is $O(k)$. This shows the practicality of the modified algorithm and also justifies our timing analysis with infinite buffers.

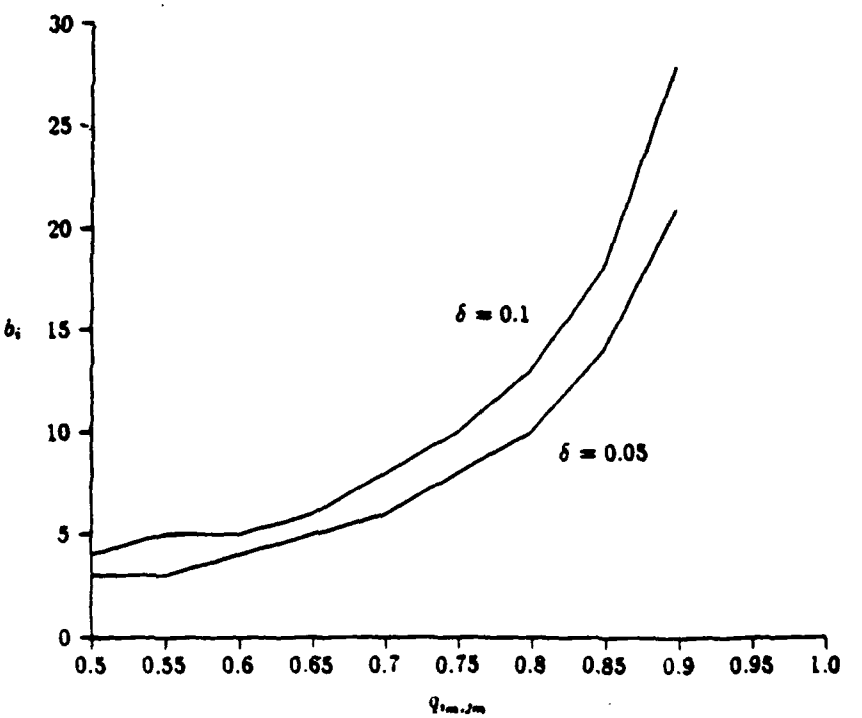


Fig. 12. The relationship between b_i and q_{i_m,j_m}

7 Conclusions

This paper concerns the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols.

We began from the observation that "merged traces", in which the trace information provided to the process simulating each cache includes the shared references generated by all other caches, ease the task of simulation, because "conditional events" are transformed into "unconditional events" by this approach.

Whereas one might think that synchronization between cache simulation processes would be required at each "interaction point" (each shared reference). A key result of this paper is that this is not true. The "synchronous points" are a subset of the interaction points – an empty subset in some cases. This lack of synchronization means that parallel trace-driven simulation of multiprocessor cache coherence protocols can yield much greater speedup than one would expect.

We showed how the synchronization requirements vary with the specific cache coherence protocol under investigation. We presented a simulation algorithm, and analyzed its time and space requirements.

Acknowledgments

We would like to thank John Zahorjan for extensive comments on an earlier, related paper.

References

[1] Agarwal, A., Sites, R.L., and Horowitz, M. ATUM: A New Technique for Capturing Address Traces Using Microcode. *Proc. 13th Annual International Symposium on Computer Architecture*, 119–127, 1986.

[2] Archibald, J., and Baer, J.-L. An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[3] Archibald, J.K. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA 98195, February 1987. Technical Report 87-02-06.

[4] Chandy, K.M., and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[5] Chandy, K.M., and Misra, J. Conditional Knowledge as a Basis for Distributed Simulation. Technical Report TR-87-5251, Computer Sciences Department, University of Texas at Austin, 1987.

[6] Cheriton, D.R., Gupta, A., Boyle, P.D., and Goosen, H.A. The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation. Technical report, Department of Computer Science, Stanford University, 1988.

[7] David, H.A. *Order Statistics.* Wiley and Sons, 2nd edition, 1962.

[8] Eager, D.L., Zahorjan, J., and Lazowska, E.D. Speedup Versus Efficiency in Parallel Systems. Technical Report 86-08-01, Department of Computer Science, University of Washington. To appear, IEEE Transactions on Computers, 1988.

[9] Eggers, S.J. and Katz, R.H. A Characterization of Sharing in Parallel Programs and Its Applicability to Coherency Protocol Evaluation. Technical Report UCB/CSD 87/387, Computer Science Division (EECS), University of California at Berkeley, December 1987.

[10] Fromm, H., et al. Experiments with Performance Measurement and Modeling of a Processor Array. *IEEE Transactions on Computers*, C-32(1):15–31, January 1983.

[11] Goodman, J.R. Cache Memory Optimization to Reduce Processor/Memory Traffic. *Journal of VLSI and Computer Systems*, 2(1):61–86, 1987.

[12] Hercksen, U., Kla, R., Kleinoder, W., and Kneissl, F. Measuring Simultaneous Events in a Multiprocessor System. *Proc. 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 77–82, 1982.

[13] Jefferson, J.D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[14] Jefferson, J.D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles*, 77–93, November 1987.

[15] Katz, R., Eggers, S., Wood, D.A., Perkins, C., and Sheldon, R.G. Implementing A Cache Consistency Protocol. *Proc. 12th Annual International Symposium on Computer Architecture*, 276–283, 1985.

[16] Lin, Y.B., Baer, J.-L., and Lazowska, E.D. Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis. Technical report, Department of Computer Science, University of Washington; submitted for publication, 1988.

[17] McCreight, E. The Dragon Computer System: An Early Overview. Technical report, Xerox Corp., 1984.

[18] Misra, J. Distributed Discrete Event Simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[19] Osterhaug, A. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 1986.

[20] Papamarcos, M., and Patel, J. A Low Overhead Coherence Solution for Multiprocessors With Private Cache Memories. *Proc. 11th Annual International Symposium on Computer Architecture*, 348–354, 1984.

[21] Peterson, J.L., and Silberschatz, A. *Operating System Concepts.* Addison-Wesley Publishing Company, Inc., 2nd edition, 1985.

[22] Rudolph, L. and Segall, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *Proc. 11th Annual International Symposium on Computer Architecture*, 340–347, 1984.

[23] Thacker, C.P., Stewart, L.C. Firefly: A Multiprocessor Workstation. *Proc. of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 164–172, October 1987.

[24] Wyatt, D. Simulation Programming On a Distributed System: A Preprocessor Approach. *Distributed Simulation 1985*, 32–36, January 1985.

Appendix A

Definition 1. The probability density function for one-parameter Gamma distribution (or Erlang distribution) is

$$f_s(t) = \frac{\lambda}{(s-1)!}(\lambda t)^{s-1}e^{-\lambda t}.$$

Fact A.1.

$$t f_s(t) = \frac{s}{\lambda} f_{s+1}(t) \quad \textit{and} \quad t^2 f_s(t) = \frac{s(s+1)}{\lambda^2} f_{s+2}(t).$$

Proof: Directly from definition 1.

QED

Fact A.2.

$$\int_{t=0}^{\infty} f_s(t) f_j(t) dt = \frac{\lambda(s+j-2)!}{2^{s+j-1}(s-1)!(j-1)!}$$

Proof: Directly from definition 1.

QED

Theorem A.1. for $t \geq 0$, the distribution function for $f_s(t)$ is

$$F_s(t) = 1 - \sum_{j=0}^{s-1} \frac{(\lambda t)^j}{j!} e^{-\lambda t} = 1 - \frac{1}{\lambda} \sum_{j=1}^{s} f_j(t)$$

Proof: cf. any Statistics book.

QED

Fact A.3.

$$\sum_{j=1}^{s} \int_{t=0}^{\infty} f_s(t) f_j(t) dt = \frac{\lambda}{2}$$

Proof:

$$\int_{t_1=0}^{\infty} \int_{t_2=0}^{\infty} f_s(t_1) f_s(t_2) dt_2 dt_1$$

$$= \int_{t_1=0}^{\infty} \int_{t_2=t_1}^{\infty} f_s(t_1) f_s(t_2) dt_2 dt_1 + \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_s(t_1) f_s(t_2) dt_1 dt_2 = 1$$

$$\Rightarrow X = \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_s(t_1) f_s(t_2) dt_1 dt_2 = \frac{1}{2}$$

If we integrate X directly, then

$$X = \int_{t_2=0}^{\infty} [1 - F_s(t_2)] f_s(t_2) dt_2 = \frac{1}{\lambda} \sum_{j=1}^{s} \int_{t=0}^{\infty} f_s(t) f_j(t) dt = \frac{1}{2}$$

QED

Theorem A.2

$$M[W_s^{(j)}] = \frac{s}{\lambda^2}$$

Proof:

$$M[W_s^{(j)}] = \int_{t_2=0}^{\infty} \int_{t_1=t2}^{\infty} (t_1 - t_2)^2 f_s(t_1) f_s(t_2) dt_1 dt_2 = X + Y + Z \quad \textit{where}$$

$$X = \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_1^2 f_s(t_1) f_s(t_2) dt_1 dt_2,$$

$$Y = \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_2^2 f_s(t_1) f_s(t_2) dt_1 dt_2, \quad \textit{and}$$

$$Z = -2 \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_1 t_2 f_s(t_1) f_s(t_2) dt_1 dt_2$$

X is rewritten as

$$\frac{s(s+1)}{\lambda^2} \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_{s+2}(t_1) f_s(t_2) dt_1 dt_2 \quad (\textit{Fact A.1})$$

$$= \frac{s(s+1)}{\lambda^3} \int_{t_2=0}^{\infty} [\sum_{j=1}^{s+2} f_j(t_2)] f_s(t_2) dt_2 \quad (Theorem\ A.1)$$

$$= \frac{s(s+1)}{\lambda^3} [\int_{t_2=0}^{\infty} \sum_{j=1}^{s+2} f_j(t_2) f_s(t_2) dt_2 + \int_{t_2=0}^{\infty} f_{s+1}(t_2) f_s(t_2) dt_2 + \int_{t_2=0}^{\infty} f_{s+2}(t_2) f_s(t_2) dt_2]$$

$$= \frac{s(s+1)}{\lambda^3} [\frac{\lambda}{2} + \frac{\lambda(2s-1)!}{2^{2s} s!(s-1)!} + \frac{\lambda(2s)!}{2^{2s+1}(s+1)!(s-1)!}] \quad (Fact\ A.3,\ Fact\ A.2).$$

Y is rewritten as

$$\frac{s(s+1)}{\lambda^2} \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_s(t_1) f_{s+2}(t_2) dt_1 dt_2 \quad (Fact\ A.1)$$

$$= \frac{s(s+1)}{\lambda^3} \int_{t_2=0}^{\infty} [\sum_{j=1}^{s} f_j(t_2)] f_{s+2}(t_2) dt_2 \quad (Theorem\ A.1)$$

$$= \frac{s(s+1)}{\lambda^3} \int_{t_2=0}^{\infty} \sum_{j=1}^{s+2} f_j(t_2) f_{s+2}(t_2) dt_2 - \int_{t_2=0}^{\infty} f_{s+1}(t_2) f_{s+2}(t_2) dt_2 - \int_{t_2=0}^{\infty} f_{s+2}(t_2) f_{s+2}(t_2) dt_2]$$

$$= \frac{s(s+1)}{\lambda^3} [\frac{\lambda}{2} - \frac{\lambda(2s+1)!}{2^{2s+2} s!(s+1)!} - \frac{\lambda(2s+2)!}{2^{2s+3}(s+1)!(s+1)!}] \quad (Fact\ A.3,\ Fact\ A.2).$$

Z is rewritten as

$$-\frac{2s^2}{\lambda^2} \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_{s+1}(t_1) f_{s+1}(t_2) dt_1 dt_2 \quad (Fact\ A.1)$$

$$= -\frac{s^2}{\lambda^2}.$$

Thus

$$X + Y + Z = \frac{s(s+1)}{\lambda^2} [1 + \frac{(2s-1)!}{2^{2s}(s!)(s-1)!} (1 + \frac{s}{s+1} - \frac{(2s+1) \cdot 2s}{2(s+1)s})] - \frac{s^2}{\lambda^2}$$

$$= \frac{s(s+1)}{\lambda^2} - \frac{s^2}{\lambda^2} = \frac{s}{\lambda^2}$$

QED

Appendix B

Theorem B.1: If there are an unlimited number of processors, then an upper bound of the expected parallel simulation time $E[t_p]$ is

$$E[t_p] \leq N[(1 + (k-1)p_s)\Delta t_p + p_s(E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}} \sqrt{V[W^{(j)}]})]$$

where $\sqrt{V[W^{(j)}]}$ is the standard deviation of $W^{(j)}$.

Proof: The waiting time of C_i at point S is W:

$$W = \max_{1 \le j \le k, i \ne j} W^{(j)} \tag{9}$$

Since there are m shared references to each cache, $E[t_p]$ is approached by:

$$E[(N - m + km)\Delta t_p + mW] = N[(1 + (k-1)p_s)\Delta t_p + p_s E[W]] \tag{10}$$

David showed [7] that if $W^{(1)}, W^{(2)}, ..., W^{(k)}$ are independent, identically distributed random variables (cf. **P.5**) with mean u and standard deviation s then

$$E[W] = E[\max_{1 \le j \le k} W^{(i)}] \le u + \frac{k-1}{\sqrt{2k-1}} s \tag{11}$$

Thus we have:

$$E[W] \le E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}} \cdot \sqrt{V[W^{(j)}]}, \quad k \ge 2 \tag{12}$$

and by substitution

$$E[t_p] \le N[(1 + (k-1)p_s)\Delta t_p + p_s(E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}}\sqrt{V[W^{(j)}]})]$$

QED

Lemma B.1: Let $P = \frac{1}{k}$ be the probability that a synchronous point is an inserted reference. Then:

$$M[W_s^{(j)}] = \frac{k}{\lambda^2}$$

Proof:

$$M[W_s^{(j)}] = \int_{t_2=0}^{\infty} \int_{t_1=0}^{\infty} (W_s^{(j)})^2 \frac{\lambda}{(s-1)!}(\lambda t_1)^{s-1} e^{-\lambda t_1} \frac{\lambda}{(s-1)!}(\lambda t_2)^{s-1} e^{-\lambda t_2} dt_1 dt_2$$

which is rewritten as

$$M[W_s^{(j)}] = \int_{t_2=0}^{\infty} \int_{t_1=t2}^{\infty} (t_1 - t_2)^2 \frac{\lambda}{(s-1)!}(\lambda t_1)^{s-1} e^{-\lambda t_1} \frac{\lambda}{(s-1)!}(\lambda t_2)^{s-1} e^{-\lambda t_2} dt_1 dt_2$$

According to Theorem A.2 in Appendix A,

$$M[W_s^{(j)}] = \frac{s}{\lambda^2}$$

and

$$M[W^{(j)}] = \sum_{s=1}^{\infty} p(1-p)^{s-1} M[W_s^{(j)}] = \sum_{s=1}^{\infty} p(1-p)^{s-1} \frac{s}{\lambda^2} = \frac{1}{p\lambda^2} = \frac{k}{\lambda^2}$$

QED

Lemma B.2: An upper bound of $E[W^{(j)}]$ is $\frac{\sqrt{k}}{\lambda}$

Proof: Directly from Lemma B.1, and the fact that $E[W^{(j)}] \leq \sqrt{M[W^{(j)}]}$.

QED

Lemma B.3: An upper bound of $V[W^{(j)}]$ is $\frac{k}{\lambda^2}$.

Proof: Directly from Lemma B.1 and the fact that $V[W^{(j)}] \leq M[W^{(j)}]$.

QED

Theorem 5.1: For all $k \geq 2$

$$E[t_p] \leq 2[1 + (k-1)p_s]N\Delta t_p$$

Proof: According to Theorem B.1 and Lemmas B.2 and B.3

$$E[t_p] \leq N[(1 + (k-1)p_s)\Delta t_p + p_s(\frac{\sqrt{k}}{\lambda} + \frac{k-2}{\sqrt{2k-3}} \cdot \frac{1}{\lambda}\sqrt{k})] \tag{13}$$

Applying **P.4**

$$E[t_p] \leq [1 + (k-1)p_s + p_s\frac{N-m+km}{km}(\sqrt{k} + \frac{k-2}{\sqrt{2k-3}} \cdot \sqrt{k})]N\Delta t_p$$

$$= [(N + (k-1)p_s)(1 + \frac{1}{\sqrt{k}} + \frac{k-2}{\sqrt{k(2k-3)}})]N\Delta t_p$$

Since

$$\forall k \geq 2, \quad \frac{1}{\sqrt{k}} + \frac{k-2}{\sqrt{k(2k-3)}} < 1,$$

We have $\forall k \geq 2$

$$E[t_p] \leq 2[1 + (k-1)p_s]N\Delta t_p$$

QED

Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis *

Yi-Bing Lin, Edward D. Lazowska and Jean-Loup Baer
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

This paper concerns the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols. Simulations are often used in computer system design. Since simulations are time-consuming, it is natural to attempt to use parallel computing to accelerate them. In this paper we devise and analyze a general technique for the parallel trace-driven simulation of multiprocessor cache coherence protocols. Then we optimize our general technique to simulate various specific protocols. The surprising result is that using our technique, the processes simulating the caches often need to do little or no communication even when simulating shared references. Thus, linear or near-linear speedup is possible.

*This work was supported in part by the National Science Foundation (Grants CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

1 Introduction

This paper concerns the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols.

Trace-driven simulation is a common approach for evaluating memory hierarchies [4][7]. Traces of memory references are gathered by interpretively executing programs and recording the memory locations that have been referenced. The simulation consists of processing this trace information while varying certain parameters. In the case of cache simulations, these parameters might include cache size and organization, and the goal would be to obtain estimates of, for example, hit ratios and processor and bus utilizations.

Since simulations are time-consuming, it is natural to attempt to use parallel computing to accelerate them. But obtaining reasonable speedups from parallel discrete-event simulations has proven to be challenging, as has analyzing the performance of such algorithms to determine, prior to implementation, whether or not they are worthwhile.

We first devise and analyze a general technique for the parallel trace-driven simulation of multiprocessor cache coherence protocols. Roughly, this technique works as follows:

We begin with a separate reference trace for each processor/cache. Each trace includes two types of memory references: *private* and *shared*. A private reference originating at processor/cache j does not have any effect on caches other than j, whereas a shared reference may update the status of other caches. All caches can be simulated asynchronously in parallel as long as there are no shared references, but concurrency control of some sort is required when shared references are encountered. To illustrate this, consider the following example: Suppose two consecutive references e_1 and e_2 are private to cache j with time-stamps t_1 and t_2, $t_1 < t_2$. Once cache j has been simulated up to event e_1, event e_2 cannot be simulated unless it is known that no shared reference originating at some other processor/cache occurs during the time period (t_1, t_2) which affects the state of cache j.

The basic idea of our simulation technique is to pre-process the input traces so that all shared references are inserted into each input trace. That is, for all j, where $j \neq i$, the shared references originating at processor/cache j are inserted in trace i (we call these references *inserted references*). Thus, all potential interactions among caches are identified before the simulation commences; we transform *conditional events* into *unconditional events* as proposed by Chandy and Misra [3]. The result is that deadlocks (if the Chandy-Misra simulation algorithm [15] is used) or rollbacks (if the Time Warp algorithm [11][12] is used) can be avoided during the simulation. The overhead of the pre-processing is low (its time is proportional to the length of all traces, i.e., $O(kN)$ where k is the number of traces and N the number of references in a single trace), and it may be amortized over many simulations since one typically performs simulations with different parameters using the same input traces.

Of course, the fact that all traces include all shared references does not eliminate the need for communication between the "simulation processes" representing the various caches. When a shared reference occurs, the execution of the cache coherence protocol must be simulated. We show that, beginning from our merged traces, huge savings in communication (and thus huge speedups in simulation time) can be achieved by tailoring the simulation technique to the specific cache

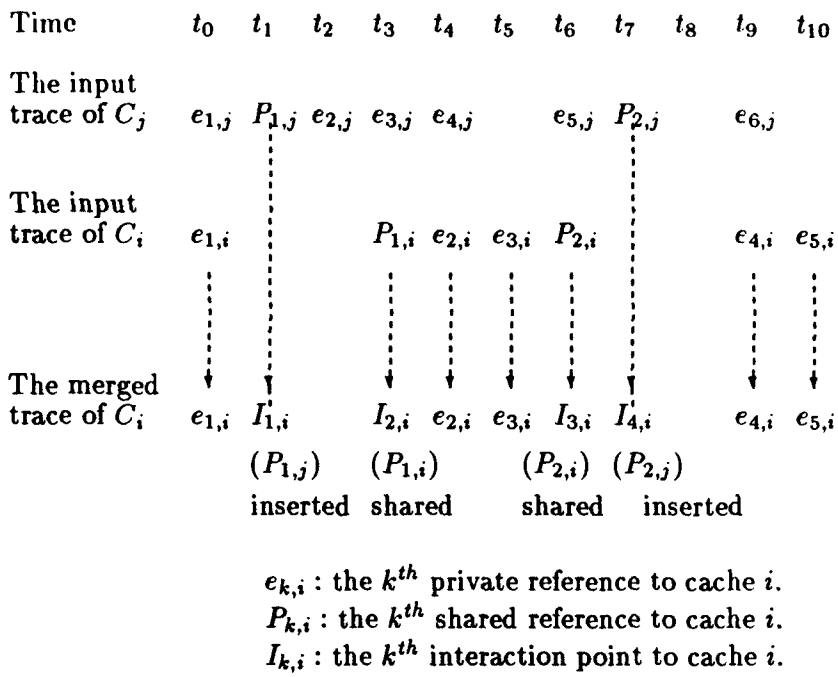


Figure 1: The shared references, inserted references, and interaction points.

coherence protocol under investigation.

Section 2 of this paper describes the input traces that are used by our technique. Section 3 describes and analyze our basic algorithm. Section 4 describes the modified algorithm, tailored to specific cache coherence protocols. Section 5 and Section 6 analyze the time requirements and the space requirements of the algorithm respectively. Finally, Section 7 presents our conclusions.

2 The Input Traces

There is one input trace for each cache; trace j corresponds to cache j. A trace initially consists of a string of $(operation, address)$ references where an operation is either *read* or *write* (an instruction fetch is considered a read). We assume that for each trace j, the shared references can be distinguished from the private references. We assume that the relative position of these shared references in each other trace i is known, so that we can insert all shared references in each trace j into the other traces i. The inserted references together with the shared references are called *interaction points*. The relationship among private references, shared references, inserted references, and interaction points is shown in Figure 1.

Construction of the merged trace, either on-line or off-line, is not expensive. In multiprocessor systems that require the programmer to explicitly identify shared variables [16], we can write on-line the shared references simultaneously on all traces. If this is not possible, tracing techniques can either provide the relative order of references [1] or a time stamp for each event [8][10] such that the order of two events can be decided in spite of the fact that they are from different traces. Such

information can be used off-line to post-process the traces [7][4] so that the types of references are distinguished. This post-processing (really a pre-processing for either the sequential or the parallel simulations) has to be done only once. Its time is proportional to the length of all traces, i.e., $O(kN)$ where k is the number of traces and N the number of references in a single trace.

A basic assumption that we make in the analysis of the simulation process, and that we will indeed have to make when we perform the simulations themselves, is that the relative locations of the *interaction points* in the traces are independent of the system's configuration. Thus they can remain in the position in which they were originally recorded or placed in the pre-processing stage. This assumption is certainly valid for the shared references. In the case of inserted references, the positions in which they can appear in a given trace depend on the time spent processing private references between shared references. Thus, modifying the system's organization, for example by increasing the cache sizes and hence their hit ratios, could influence the placement of inserted references. However, if all elements of the multiprocessor system are modified in a homogeneous way (e.g., if all cache sizes are increased by the same amount), then we can assume that the relative processing times of the private references remain in the same ratio and that keeping the inserted references in the same positions will not distort the simulation results.

Note that assumptions of this type are customary and often necessary in trace driven simulations. For example, many studies on parallel program behavior or cache coherence protocols, e.g., [1] and [7], assume "infinite caches" so they do not have to deal with the effects of hits and misses due to the various cache sizes. Assuming equal instruction times, and equal memory or cache reference times for read or writes, is also justified by an argument similar to ours, namely that the variation in timing would be identical across processors [7].

3 The Basic Simulation Algorithm

In this section we describe and analyze our basic parallel simulation algorithm. We consider only the cache processes, which simulate the operations of caches, and do not dwell on the mundane input processes, which read the input traces and send the reference events to the cache processes.

In the basic algorithm, there is a barrier that synchronizes the execution of the cache processes at each interaction point. For each reference contained in merged trace i, there is a corresponding event at cache process C_i in the simulation. There are three possibilities for each event:

I. The event is a private reference. In this case, C_i continues its execution.

II. The event is a shared reference. In this case, C_i waits on the barrier until all other cache processes reach[1] this interaction point. Then C_i checks other caches' status, and decides its own status using the cache coherence protocol under investigation. All cache processes then leave the barrier, continuing execution.

III. The event is an inserted reference. In this case, C_i waits on the barrier until some other cache process (the one with the corresponding "shared" reference) consults C_i's status; it then continues execution.

[1] We say that C_j "reaches" an event R or R "arrives at" C_j, iff the event to be processed by C_j is R.

It is not difficult to see that this algorithm is correct and deadlock-free. Before summarizing the performance analysis of this algorithm, we note that it is not necessary for all cache processes to "rendezvous" at each interaction point. Modifications that relax this restriction, sometimes with dramatic gains in efficiency, are the subject of forthcoming sections.

We begin our timing analysis of the basic algorithm by introducing some notation that is used throughout the paper:

- N: the total number of references to a cache (not including the inserted references).
- p_s: the proportion of references to a cache that are shared references.
- $k \geq 2$: the number of caches to be simulated.
- Δt_s: the average time for processing a private reference event[2] in the uniprocessor environment.
- Δt_p: the average time to process an event (a private reference, or an interaction point) in the multiprocessor environment. Note that for an interaction point, Δt_p does not include the waiting time.
- $\rho = \frac{\Delta t_p}{\Delta t_s} \geq 1$: the synchronous factor that represents the extra overhead to support synchronization in the multiprocessor environment. This factor is determined by the overhead of the primitives available in the system (semaphore, monitor, or message passing primitives) and the programming ability of the user who implements the simulation.
- t_s: the sequential simulation time.
- t_p: the parallel simulation time when there are an unlimited number of available processors.

The main results are (cf. Appendix A):

Theorem 3.1: If the arrivals of interaction points to each cache form a Poisson process, with the same mean for each cache, then the expected value of the parallel simulation time with an unlimited number of processors, $E[t_p]$, is bounded by

$$E[t_p] \leq [1 + (k-1)p_s](1 + \frac{k-1}{\sqrt{2k-1}})N\Delta t_p$$

Theorem 3.2: If the arrivals of interaction points to each cache form a Poisson process, with the same mean for each cache, then the speedup with an unlimited number of processors, defined as $S_\infty = \frac{E[t_s]}{E[t_p]}$, is bounded by

$$S_\infty \geq \frac{k}{\rho(1 + \frac{k-1}{\sqrt{2k-1}})}$$

[2]The time to process a shared reference is longer than that to process a private reference in the uniprocessor environment. (See the discussion in Section 5.)

Information Required	Source of information	
	Processor-induced transaction	Bus-induced transaction
Current state of the block	Status of the cache process	Status of the cache process
Read/write	Arrival event	Arrival event
Hit/miss	Status of the cache process	Status of the cache process or other cache processes
Actual sharing	Status of other cache processes	Status of the cache process

Figure 2: The sources of information required in a transaction.

that is, the speedup is $O(\sqrt{k})$.

(We expect a degradation when p_s increases. Surprisingly, p_s does not have any effect on S_∞, because the degradation in parallel simulation time is matched by a degradation in sequential simulation time.)

Theorem 3.3: If there are n processors available, $1 \leq n \leq k$, and k is large, then a lower bound of the speedup with n processors is

$$S(n) \geq \sqrt{k} - \frac{k-n}{\sqrt{k}+n}$$

4 The Modified Algorithm

The basic algorithm requires that all cache processes synchronize at each interaction point. In this section we describe a modification that reduces this restriction. We prove that the modified algorithm is correct and deadlock free.

The idea of the modified algorithm is that at some interaction points, a cache process can determine its cache status using only local information (including that provided by its (merged) trace), so there is no need to consult the status of other caches. Thus we can process such interaction points in the same way as private references.

Whether an interaction point can be processed locally is dependent on the type of interaction (a shared reference, which is a *processor-induced transaction*, or an inserted reference, which is a "possible" *bus-induced transaction*) and the cache coherence protocol under investigation. In Figure 2, the first column lists various pieces of information that are required by various cache coherence protocols, while the second and third columns indicate the sources of this information for processor-induced and bus-induced transactions, respectively.

From Figure 2, we see that synchronization is required during the simulation if:

- the cache coherence protocol requires information about actual sharing for a processor-induced

Protocol	Shared				Inserted	
	Read		Write		Read	Write
	hit	miss	hit	miss		
Berkeley						
Illinois		×				
FBWO		×				
Firefly		×		×		
Dragon		×		×		

Figure 3: "Synchronization information" for different protocols. An entry with "×" means that synchronization is required.

transaction, or

- the protocol cannot provide information about hit/miss for a bus-induced transaction.

We say a transaction (or an interaction point) is *synchronous* iff it requires synchronization in the simulation. We call an interaction point that requires synchronization a *synchronous point.* Otherwise, it is a *non-synchronous point.* As just noted, the specific cache coherence protocol under investigation has a bearing on whether an interaction point requires synchronization or not. Figure 3 gives "synchronization information" about five protocols: Berkeley [13], Illinois [17], FBWO (Future bus write-once) [9], Firefly [20], and Dragon [14]. The first three of these protocols are based on "invalidation" while the last two are of the "distributed-write" type.

What Figure 3 shows is that a surprisingly large number of event types can be simulated without the need for synchronization. For example, an ownership-based invalidation protocol such as the Berkeley protocol does not require *any* synchronization. The other two invalidation-based protocols require synchronization only at read-misses. For the distributed-write protocols, synchronization on write-misses is also required.

To illustrate this important observation, we shall prove the first row (Berkeley protocol) of Figure 3. The remaining rows can be proved by similar arguments. The following states [2][13] are used in the Berkeley protocol:

- **INVALID**: Copy is not up to date.
- **UNMOD-SHD**: Unmodified-shared; copy is not modified with respect to main memory. Other caches *may* have a copy.
- **MOD-SHD**: Modified-shared; other caches may have copies. Block needs to be written back when replaced.
- **MOD-EXC**: Modified-exclusive; no other copies exist. Block must be written back on replacement.

In the Berkeley protocol, a cache with a **MOD-SHD** or **MOD-EXC** copy is called the *owner.* There can be at most one such cache. Although a **MOD-SHD** copy indicates that other caches

may have copies, the protocol ensures that they will be **UNMOD-SHD** copies. Thus there is at most one owner cache, and the owner is responsible for writing the block back to memory. If a block is not owned by any cache, memory is the owner.

The protocol works as follows: Let i be the requesting cache, and let j be any other cache that contains the requested block. [3]

A. The action of cache i:

I. Read hit: The state of cache i does not change; the requested block is sent to the processor.

II. Read miss: Cache i gets the copy either from cache j (if such j exists) or from memory (if no caches have the copy). The state of the block in cache i is set to **UNMOD-SHD**.

III. Write hit: If the block in cache i is **MOD-EXC**, the write proceeds with no delay. If it is **MOD-SHD** or **UNMOD-SHD**, an invalidation signal is sent and the state of the block in cache i is set to **MOD-EXC**.

IV. Write miss: As with a read miss, cache i gets the copy either from cache j (if such j exists) or from memory (if no caches have the copy). The state of the block in cache i is set to **MOD-EXC**.

B. The response of cache j:

I. No signal is sent from cache i to cache j if cache i has the copy (read hit): No action is taken in cache j.

II. Read miss signal: If it is a read miss in cache i, then cache i sends a "read miss" signal to cache j. Cache j sends its copy to cache i, if it is the owner of the block. If the block is in state **UNMOD-SHD** then no action is taken. If the block is in state **MOD-SHD** then it remains in the same state. If the block is in state **MOD-EXC** then its state is set to **MOD-SHD**.

III. Invalidation signal: If it is a write hit in cache i, then cache i sends an "invalidation" signal to cache j. The block in cache j is set to **INVALID**.

IV. Write miss signal: If it is a write miss in cache i, then cache i sends a "write miss" signal to cache j. Cache j sends its copy to cache i, if it is the owner of the block. The block in cache j is set to **INVALID**.

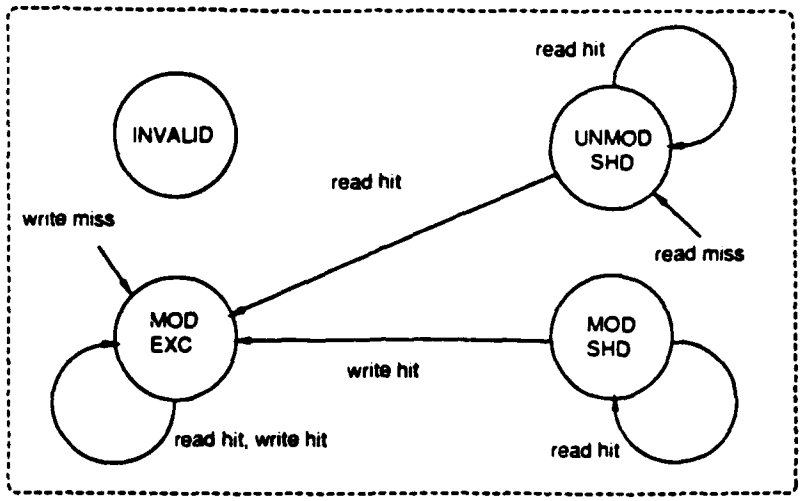
Figure 4 shows the Berkeley protocol state transition diagram.

Theorem 4.1: All interaction points are *non-synchronous* if the Berkeley cache coherence protocol is used.

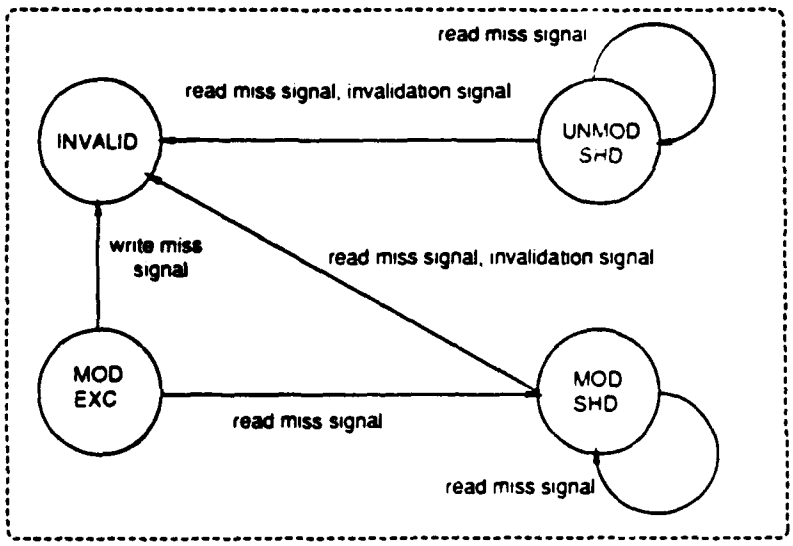
Proof: All shared references are non-synchronous since "sharing" information is not used in this protocol (**A** of the protocol description).

All inserted "write" references are non-synchronous since the next state of the block is set to **INVALID** independently of whether the operation is a hit or miss (**B.III** and **B.IV**).

[3] For those "other" caches that do not have the requested block, no action is taken.



(a) Processor induced transansaction.



(b) Bus induced transaction

Figure 4: The Berkeley protocol state transition diagram.

All inserted "read" references are non-synchronous. If the block is in state **UNMOD-SHD** or **MOD-SHD**, the next state remains the same (**B.I** and **B.II**). If the block is in state **MOD-EXC**, then it corresponds to a miss in cache i and the next state is **MOD-SHD**. ∎

We reiterate the surprising result that if one begins from our easily-constructed merged traces, then the Berkeley protocol can be simulated in parallel with absolutely no communication between simulation processes. As shown in Figure 3, the Berkeley protocol is unique in this regard. However, for all protocols the required synchronization is considerably less than one would expect based on the number of interaction points, even for complex protocols such as the EIP protocol [2] and the CMU RW protocol [19] that have features of both the invalidation and the distributed-write protocols. We now present an algorithm that is general enough to handle any of these protocols. We refer to it as the "modified algorithm", in contrast to the "basic" algorithm appearing in the previous section, which required synchronization at each interaction point.

As in the basic algorithm, there are three possibilities for an event in the execution of a cache process C_i:

I. The event is a private reference. C_i updates its status, and advances to the next event.

II. The event is an inserted reference. Suppose that this inserted reference was generated by a shared reference, say r, in cache j. There are two possibilities:

II (a). The reference r is of the "non-sync" type for cache j: C_i updates its status, and advances to the next event.

II (b). The reference r is of the "sync" type for cache j: C_i first updates its status, and then generates a message, R_i to cache process C_j which indicates whether there is a copy of the referenced block in cache i. C_i puts R_i in a queue called B_i [4] so C_j may check C_i's status via B_i. Then C_i advances to the next event.

III. The event is a shared reference.

III (a). If the event is "non-sync", then C_i updates its status and advances to the next event.

III (b). If the event is "sync", then C_i waits until it has received messages R_j from each other process C_j, where $1 \leq j \neq i \leq k$. C_i updates its status and continues.

The communication of cache processes in the modified algorithm is analogous to the bounded buffer producer/consumer problem. An example of implementing a solution to this problem is shown in Figure 5; there is a monitor $Buffer[i]$ for each cache process C_i. The monitor consists of

- a queue, B_i, of length $l + 1$,
- $k - 1$ pointers P_j, where $1 \leq j \neq i \leq k$, which point to elements of B_i, and
- k conditions Co_j, where $1 \leq j \leq k$.

[4] If B_i is full, C_i will wait till there is room in the buffer; cf. Section 6 for the analysis of space requirements.

An element in B_i consists of two parts: A_n denotes the number of caches that "share" the requested block of cache i at I_n (the n^{th} sync. point) [5]; D_n denotes the number of cache processes that have sent messages to C_i corresponding to I_n. C_i removes the first element of B_i only when $D_n = k - 1$. The pointer P_j for C_j points to an element of B_i in which C_j must record its next message to C_i. This message, which corresponds to the insertion of I_n, always increments D_n by 1 and increments A_n by 1 if C_j was sharing the requested block. When P_j points to the $l + 1^{th}$ element of B_i, the buffer is full. Therefore, if C_j tries to send a message to C_i, then it is blocked on the condition Co_j (see Figure 5). If C_i wants to access the first element of B_i, and some cache process has not yet sent the corresponding message, then C_i is blocked on condition Co_i.

Theorem 4.2: The modified algorithm is correct.

Proof: Correctness consists of two parts:

(a) Let R_1 and R_2 be events occurring at cache i, and let R_1' and R_2' be their simulation counterparts. We must show that R_1 arrives at cache i earlier than R_2 does, iff C_i processes R_1' earlier than R_2'. It is obvious that (a) is satisfied by the algorithm.

(b) We must show that the status of C_i before and after the arrival of R' reflects the status of cache i before and after the arrival of R. To show this, we must consider the three cases described earlier. It is easy to see that **I** satisfies (b). In **II**, the status of cache i only depends on the operation (read/write) of the shared reference to cache j. Such information is provided by R, and the change of status of C_i reflects that of cache i. Thus, the execution of C_i continues without being blocked. In **III (a)**, the status of cache i is determined by the arrival event. In **III (b)**, the status of cache i depends on the status of all other caches. Such information is provided by R_j, where $1 \leq j \neq i \leq k$, by other cache processes (**II (b)**). Thus C_i must wait until other caches reach the same synchronous point, and the change of C_i's status reflects that of cache i. ∎

Two important issues that arise in parallel and distributed algorithms are *termination* and *deadlock*. In the modified algorithm, the termination problem is easily solved by adding an end-mark event at the end of each input trace. Freedom from deadlock is shown in Theorem 4.3:

Theorem 4.3: The modified algorithm is deadlock free.

Proof: We show that there is no *circular wait* [18] among cache processes as follows:

Let I and J be shared references to cache i and j respectively. Then there are only two cases in which C_i waits for C_j (denoted as $C_i \rightarrow C_j$). The first case is that C_i reaches I earlier than C_j does. The second case is that C_i reaches J earlier than C_j does and B_i is full. (It is obvious that we can ignore the case in which C_i reaches L, (where $L \neq I, J$) earlier than C_j does.)

In both cases, C_i reaches some synchronous point earlier than C_j does. If a circular wait is formed during the execution, then there exist n cache processes $C_{i_1}, C_{i_2}, ..., C_{i_n}$, where $2 \leq n \leq k$, and $1 \leq i_1, i_2, ..i_n \leq k$, such that $C_{i_1} \rightarrow C_{i_2}$, $C_{i_2} \rightarrow C_{i_3}$, ..., $C_{i_n} \rightarrow C_{i_1}$. This circular wait exhibits a contradiction involving C_{i_1}. ∎

These results show that dramatic improvements can be achieved by considering the details of the specific cache coherence protocol being simulated. This was shown qualitatively in this section and

[5] The information provided by A_n is required to determine "shared high" or "shared low" in protocols such as FBWO, Firefly, and Dragon. In fact, a single bit would be sufficient to indicate the presence/absence of sharing.

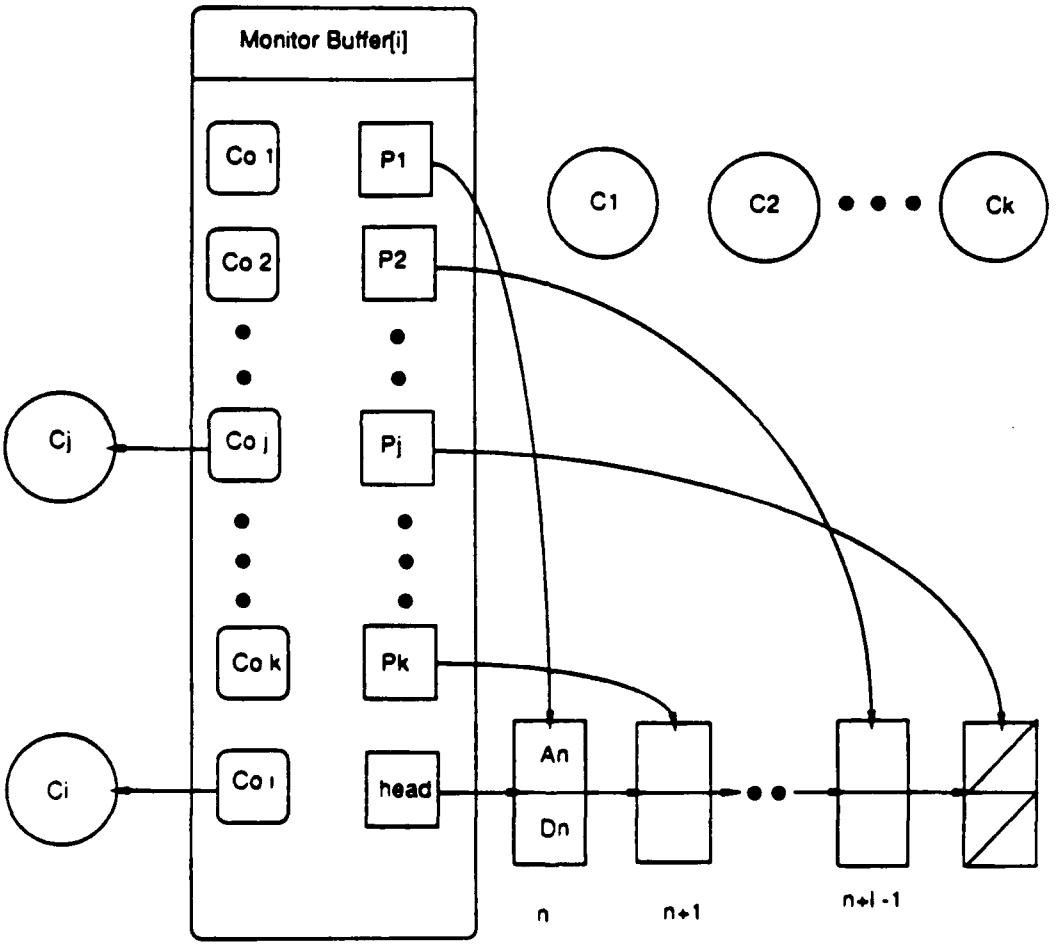


Figure 5: An implementation of cache process synchronization: C_j is blocked since it tries to insert a message in the queue which is full. When C_i needs to access the head of the queue, it is blocked because $D_n < k - 1$. (C_1 has not yet inserted a message at the head of the queue.)

will be analyzed quantitatively in the next one. Our results support the claim that it may not be highly worthwhile to try to build a general parallel simulation system, without considering the structure of the problem under consideration. Of course, it is worthwhile to build tools for building special-purpose simulation systems.

5 Timing Analysis

This section analyzes the time complexity of our approach. We first derive the execution time of the sequential simulation, then derive the execution time of the parallel simulation assuming that each process is assigned a dedicated processor. Finally, we derive the speedup of our approach when a finite number of processors are used.

5.1 Sequential Simulation

The performance analysis of shared-memory multiprocessor systems reveals the substantial costs of sequential simulation. In addition to the overhead of maintaining a large event queue, the costs of processing shared references are large: searches in all caches are required, so the time to process a shared reference is k times the cost of processing a private reference. One may argue that under most coherence protocols, searches in all caches are not necessary when there is a read hit. Since we can modify the algorithm such that a cache process never waits at "read hit" shared references, it is likely that the modified algorithm would benefit more from this optimization than any sequential algorithm does. To keep the time complexity analyses simple and clean, however, we assume that:

- the time to process a "read hit" shared reference is k times the cost to process a private reference in a sequential simulation,
- a cache process always waits at a "read hit" shared reference in parallel simulation, and
- a cache process always waits at a "write" shared reference.

Let $m = Np_s$ be the total number of shared references to a cache. Then the sequential execution time is

$$\begin{aligned} E[t_s] &= k \times (\text{time to process private references} \\ &\quad + \text{time to process shared references}) \\ &= k(N - m + km)\Delta t_s \\ &= k[1 + (k-1)p_s]N\Delta t_s \end{aligned}$$

If $N\Delta t_s$ is approximated as being constant, i.e., independent of k, [6] then the relationship between $E[t_s]$, k, and p_s is shown in Figure 6. Note that k, the number of caches being simulated, has a dominant effect on $E[t_s]$.

[6] In fact, Δt_s should increase when k increases, since the overhead of operations on some data structures (e.g., the event queue) increases.

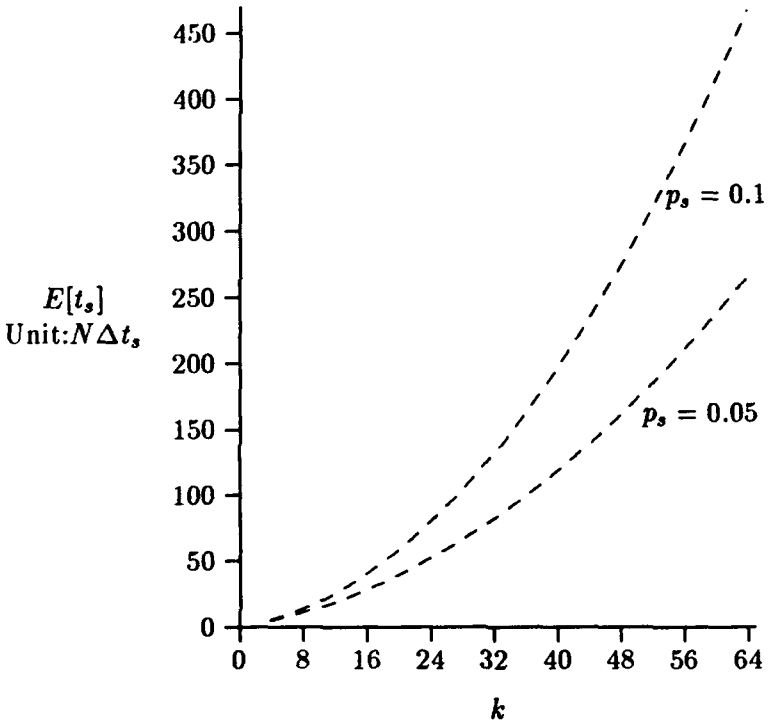


Figure 6: The relationship between the sequential simulation time, $E[t_s]$, the number of caches, k, and the probability of shared references, p_s.

5.2 Parallel Simulation

In this subsection we derive the execution time of our approach. We first consider the case that each cache process is executed by a dedicated process, then the case that the number of cache processes exceeds the number of available processors.

5.2.1 The Parallel Simulation Time with an Unlimited Number of Processors

The parallel simulation for the Berkeley cache coherence protocol does not require any synchronization (cf. Theorem 4.1). With an unlimited number of available processors, the parallel simulation time is just the time to process a merged trace. We expect the speedup with an unlimited number of processors to be close to k, the number of caches (of course, the speedup is less than k, partly because the merged traces are larger, and partly because of our pessimistic assumptions on the processing of "read hit" shared references).

The timing analyses for simulations of the protocols listed in Figure 3 other than the Berkeley protocol are not trivial because synchronizations are required in the simulations. Thus, processes will have to wait for each other at the various synchronous points. One main result is that the expected time spent waiting for synchronization is at worst equal to the time spent in processing events, i.e., at least half of the simulation time is spent in useful work. In order to derive analytically this upper bound on the expected parallel computation time $E[t_p]$, we make the following assumptions:

P.1 An unlimited number of processors is assumed. The case of a limited number of processors will be treated in the next subsection.

P.2 Cache processes never wait at their inserted references. In other words, the probability of buffer overflow is nil. (Section 6 shows that this is a very reasonable assumption since the probability of overflow for an $O(k)$ amount of buffering is low.)

P.3 The behaviors of the processes being traced are statistically identical; in particular they have the same length (N references) prior to the insertion of shared references , the same number of shared references m for a total length of $N+(k-1)m$ references, and the times to process events such as private and "non-sync" references are of the same order of magnitude.

P.4 The arrivals of synchronous points to each cache form a Poisson process with the average arrival rate λ being the ratio of the number of synchronous points over the time to process the whole merged trace, where

$$\lambda = \frac{km}{(N-m+km)\Delta t_p}$$

This follows from assuming that the time for a cache process to handle all private and "non-sync" references between two synchronous points is exponentially distributed.

P.5 Let $W^{(j)}$ be the time that C_i waits for C_j. The waiting times $W^{(1)}, W^{(2)}$, ..., $W^{(k)}$ (excluding $W^{(i)}$ which is null) are independent, identically distributed random variables. This assumption follows **P.3**.

P.6 Let S_n be a synchronous point where two cache processes, say C_i and C_j, handle their next references, most likely private references, at the same time and let S_{n+s}, where $s \geq 1$, be the next synchronous point where one of these two processes has to wait for the other. Then $[S_n, S_{n+s}]$ is called a *synchronization interval.* We assume that during a synchronization interval between cache processes C_i and C_j, C_i will wait for all cache processes aside from C_j the same amount of time that C_j will wait for all cache processes aside from C_i.

If a synchronous point S is a shared reference to cache i, then the cache process C_i waits until all cache processes reach that synchronous point. The expected time for the parallel simulation (i.e., to process one trace, since they all take approximately the same amount of time) is therefore the time to process all private/shared references plus the expected waiting time at each shared reference. An upper bound of the parallel simulation time $E[t_p]$ is given by the following expression:

$$E[t_p] = N\{[1+(k-1)p_s]\Delta t_p + p_s E[W]\}$$

where W is the maximum of the waiting times at a given synchronous point:

$$W = \max_{1 \leq j \leq k, i \neq j} W^{(j)}$$

David showed [5] that if $W^{(1)}, W^{(2)}, ..., W^{(k)}$ are independent, identically distributed random variables with mean u and standard deviation s then

$$E[W] = E[\max_{1 \leq j \leq k} W^{(i)}] \leq u + \frac{k-1}{\sqrt{2k-1}} s$$

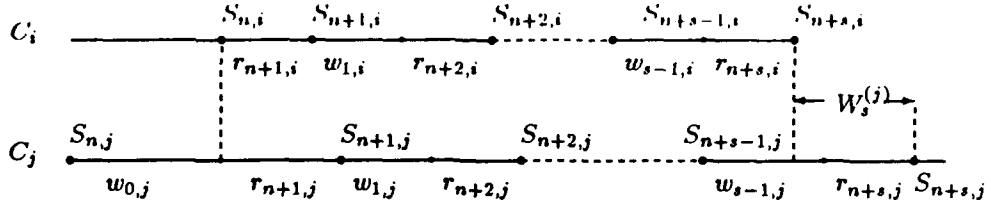


Figure 7: The waiting time graph.

The mean u, or $E[W^{(j)}]$, is derived as follows: As shown in Figure 7, C_i and C_j are synchronized at S_n (i.e., both C_i and C_j handle the next references, most likely private references, at the same time). After s synchronous points, a shared reference to cache i arrives. There are two possible cases:

1. C_j reaches S_{n+s} earlier than C_i does. In this case, C_i does not wait for C_j and $W_s^{(j)} = 0$.
2. C_i reaches S_{n+s} earlier than C_j does. Then

$$W_s^{(j)} = \sum_{l=1}^{s} r_{n+l,j} + \sum_{l=1}^{s-1} w_{l,j} - \sum_{l=1}^{s} r_{n+l,i} - \sum_{l=1}^{s-1} w_{l,i} > 0$$

where $r_{n+l,j}$ is the time for C_j to handle private and non-sync references between S_{n+l-1} and S_{n+l}, and $w_{n+l,j}$ is the waiting time of C_j at S_{n+l}. (Note that $w_{n+l,j} = 0$ if S_{n+l} is not a J.)

According to **P.6** the waiting times for all caches other than C_i and C_j cancel out and

$$\sum_{l=1}^{s} w_{l,j} \simeq \sum_{l=1}^{s} w_{l,i}$$

With this simplification, we have

$$W_s^{(j)} = \begin{cases} 0 & \text{if } C_j \text{ reaches } S_{n+s} \text{ earlier than } C_i \text{ does} \\ t_1 - t_2 & \text{otherwise} \end{cases}$$

where

$$t_1 = \sum_{l=1}^{s} r_{n+l,j} \quad \text{and} \quad t_2 = \sum_{l=1}^{s} r_{n+l,i}$$

According to the assumption of Poisson arrivals for synchronous points (**P.4**), the time to handle private and non-sync references between S_n and S_{n+s} has an *Erlang distribution* with the probability density function

$$f_s(t) = \frac{\lambda}{(s-1)!}(\lambda t)^{s-1} e^{-\lambda t}$$

This will allow us to derive as an upper bound of $E[W^{(j)}]$ (cf. Appendix C):

$$E[W^{(j)}] \leq \frac{\sqrt{k}}{\lambda}$$

Similarly, we can derive an upper bound of the standard deviation s of $W^{(j)}$, $\sqrt{V[W^{(j)}]}$, as:

$$\sqrt{V[W^{(j)}]} \leq \frac{\sqrt{k}}{\lambda}$$

This leads us to our main result (cf. Appendix C):

Theorem 5.1: For all $k \geq 2$, $E[t_p] \leq 2[1 + (k-1)p_s]N\Delta t_p$.

$E[t_p]$ consists of three parts, i.e., $E[t_p] = E[t_{private}] + E[t_{sync}] + E[t_{wait}]$:

- $E[t_{private}]$: the expected elapsed time for handling all private references

$$E[t_{private}] = (N - m)\Delta t_p = (1 - p_s)N\Delta t_p$$

- $E[t_{sync}]$: the expected elapsed time for handling all synchronous points

$$E[t_{sync}] = km\Delta t_p = kp_sN\Delta t_p$$

- $E[t_{wait}]$: the expected waiting time in the simulation.

Therefore, it follows from Theorem 5.1 that

$$E[t_{wait}] \leq [1 + (k-1)p_s]N\Delta t_p = E[t_{private}] + E[t_{sync}]$$

The importance of this result is that a processor spends more time doing useful work than waiting. This is illustrated in Figure 8. We observe that for small k, ($k \leq 8$), the parallel simulation times with different p_s values are roughly the same. On the other hand, when k increases, $E[t_p]$ with larger p_s grows faster. This phenomenon is explained as follows. When $(k-1)p_s << 1$, i.e., with very little sharing, we have

$$E[t_p] \approx N\Delta t_p$$

which is independent of k and p_s and represents an ideal case. When $(k-1)p_s >> 1$, i.e., with a fair amount of sharing and a large number of caches,

$$E[t_p] \approx (k-1)p_sN\Delta t_p$$

which is linearly proportional to k and p_s.

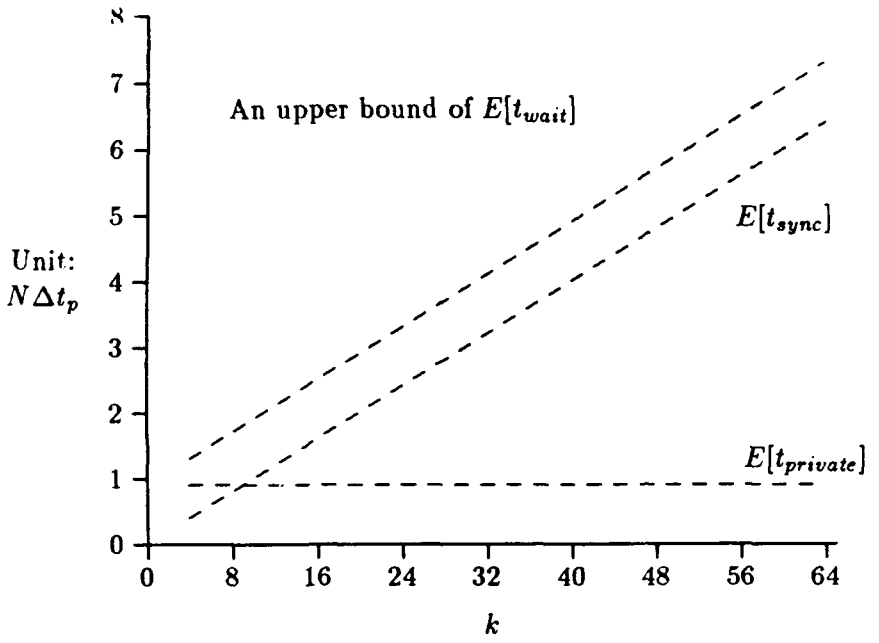


Figure 8: $E[t_{private}]$, $E[t_{sync}]$, and $E[t_{wait}]$ with $p_s = 0.1$.

5.2.2 Speedup with an Unlimited Number of Processors

With an upper bound on the parallel simulation time, we can now derive lower bounds on achievable speedup.

Corollary 5.1: An upper bound on the speedup with an unlimited number of processors is

$$S_\infty \geq \frac{k[1+(k-1)p_s]N\Delta t_s}{2[1+(k-1)p_s]N\Delta t_p} = \frac{k}{2\rho}$$

Proof: Directly from Section 5.1 and Theorem 5.1. ∎

We expect a degradation when p_s increases. As with the basic algorithm, though, p_s does not have any effect on S_∞, because the degradation in $E[t_p]$ is matched by a degradation in $E[t_s]$. The relationship between S_∞ and k is linear.

Figure 9 shows the incremental benefit of each approach: Curve (a) represents the speedup of the parallel simulation with original traces. Curve (b) represents the speedup of the parallel simulation with preprocessed traces. Curve (c) represents the speedup of the parallel simulation which takes advantage of knowledge of the cache coherence protocols as well as the preprocessed traces. Figure 9 show that, beginning from our merged traces, huge savings in communication (and thus huge speedups in simulation time) can be achieved by tailoring the simulation technique to the specific cache coherence protocol under investigation.

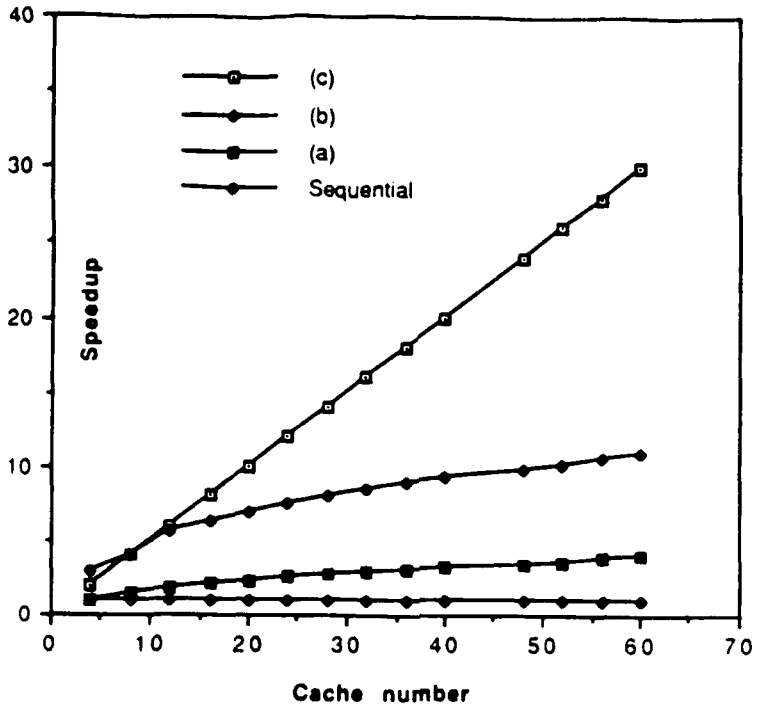


Figure 9: The speedups of the parallel simulation with different optimizations.

5.2.3 Speedup with a Finite Number of Processors

If we consider the (realistic) case in which the number of runnable subtasks (the cache processes) exceeds the number of available processors, the speedup is not equal to S_∞. We assume that the processor scheduling discipline is *processor sharing*. Under this discipline, if k subtasks are eligible for execution and there are n available processors, $n < k$, each subtask receives service at a rate that is n/k times the rate at which it would receive service if a processor were dedicated to it. According to Eager, Zahorjan, and Lazowska [6] we have the following theorem:

Theorem 5.2 [Eager-Zahorjan-Lazowska]: Let S_∞ denote the speedup with an unlimited number of available processors, m_{max} the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number is available), and $S(n)$ the speedup with n processors. If $n < m_{max}$, with processor sharing scheduling:

$$S(n) \geq \frac{nS_\infty}{n + S_\infty - 1 - \frac{(n-1)(S_\infty - 1)}{m_{max} - 1}}$$

Thus, we have

Corollary 5.2: If there are n processors available in the parallel simulation, where $1 \leq n \leq k$, and $k \geq 2$, then

$$S(n) \geq \frac{n(k-1)}{(2\rho - 1)n + k - 2\rho}$$

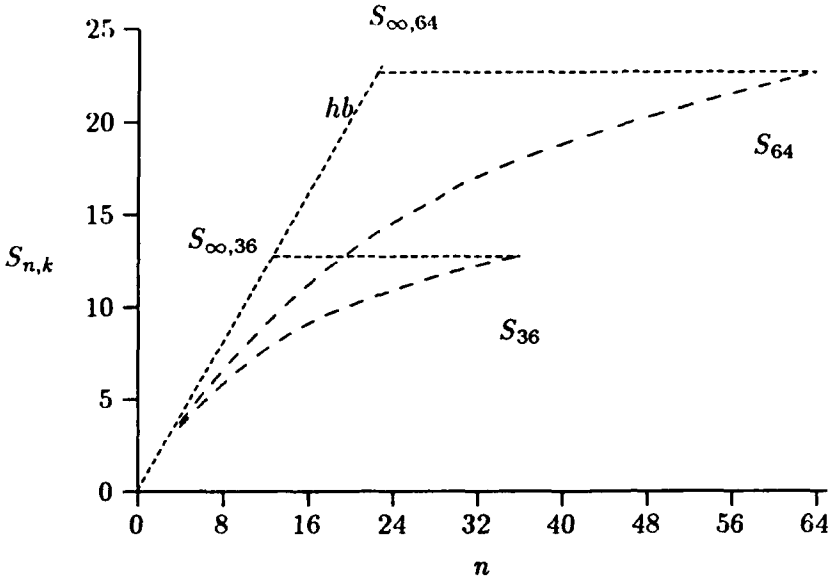


Figure 10: The relationship between S, n, and k (the modified algorithm).

Proof: According to Theorem 5.2, and let $1 \leq n \leq k$

$$S(n) \geq \frac{nS_\infty}{n + S_\infty - 1 - \frac{(n-1)(S_\infty - 1)}{k-1}} = \frac{nS_\infty(k-1)}{(n + S_\infty - 1)k - nS_\infty}$$

Applying Corollary 5.1

$$S(n) \geq \frac{n(k-1)}{(2\rho - 1)n + k - 2\rho} \quad \blacksquare$$

Figure 10 shows the relationship among S, n, and k for $\rho = \sqrt{2}$. There are two simple upper bounds on speedup [6]. The *hardware bound* reflects the limitation imposed by the hardware, and is given by the number n of available processors (line hb in Figure 10). This bound can be achieved only if all n processors can be kept busy all of the time. The *software bound* S_∞ reflects the limitation imposed by the software (lines $S_{\infty,36}$ and $S_{\infty,64}$. Corollary 5.2 gives the lower bound of speedup with $k = 36$ and $k = 64$ (lines S_{36}, and S_{64})). We observe that:

- For fixed n, the speedup S increases when k increases. This phenomenon is explained as follows: When an active cache process blocks itself, the associated processor, P, becomes a free processor and can execute the next cache process on the ready queue. If the ready queue is empty, then P is idle. The probability that the ready queue is not empty increases when k increases. In other words, when k increases, processors are unlikely to be idle, and are devoted to useful work, and thus increase the speedup.
- For fixed k, the speedup S is about 80% of S_∞ when $n = 0.6k$. After that point, an increase in n does not provide much gain. This phenomenon is explained as follows: When n increases, it is likely that the number of free processors is more than the number of processes on the ready queue. Thus, there are more opportunities for some free processors to be idle, and the speedup cannot be improved significantly.

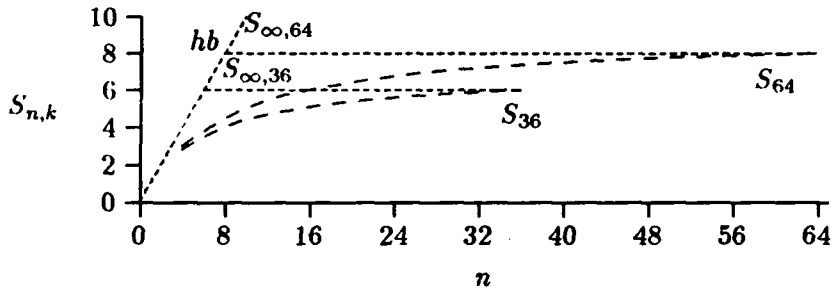


Figure 11: The relationship between S, n, and k (the basic algorithm).

An important conclusion to draw is that we can choose an appropriate number, n', of processors (say, $n' = 0.6k$ and $n' \leq n$) to perform the simulation. The remaining $n - n'$ processors can be used to support functions such as statistics collection [21]. Or, in the simulation of set-associative caches, we may partition processors into groups to simulate different sets of caches.

Similar observations can be drawn for the basic algorithm. Figure 11 shows the speedup for the basic algorithm with the same parameters as in Figure 10. Note that when n is large, increasing n in the modified algorithm gains more than that in the basic algorithm. This is because the opportunity for a cache process to wait in the modified algorithm is less than that in the basic algorithm. Thus the added processors in the modified algorithm always contribute more than that in the basic algorithm (as long as $n \leq k$).

6 Space Analysis

In the modified algorithm, we use an array of buffers, or queues,

$$B = \{B_i | \ 1 \leq i \leq k\}$$

for communication between cache processes (see Figure 5). If B had infinite capacity then the cache processes would never have to wait at the synchronous points induced by inserted references (case **II (b)** of the modified algorithm in Section 4). But since B is necessarily finite, we need to balance space requirements (amount of buffering) and time requirements (waiting time due to buffer congestion). In this section we show that with a low probability of buffer overflow, the space complexity of the total buffer size, $|B|$, is $O(k)$ with a small constant. We first introduce the additional notation:

- m_i (m_j): the number of shared references that require synchronization (the "sync" references in Figure 3) to cache i (j).
- I_n (J_n): the n^{th} such shared reference to cache i (j), where $1 \leq n \leq m_i$ (m_j).
- τ_i: the merged input trace for cache i.
- $[J_n, J_{n+1}] \subseteq \tau_i$, where $1 \leq j \leq k$, and $0 \leq n \leq m_j$: an interval of the input trace i which starts at J_n and ends at J_{n+1} with no other J in between. By convention, we have two *virtual* events J_0 and J_{m_j+1} at the beginning and the end of τ_i.

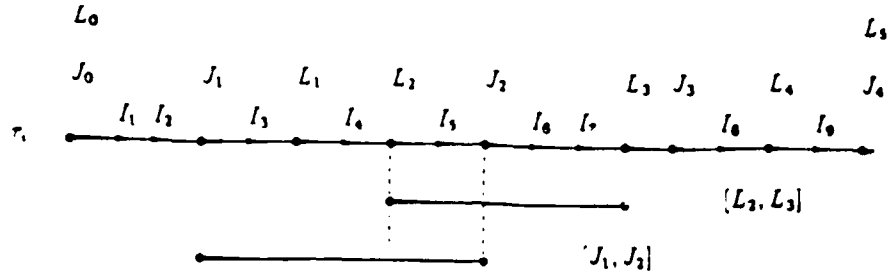


Figure 12: The input trace, the interval, the boundary set, and the least interval.

Note that each shared reference I_n in τ_i is included in exactly $k-1$ intervals, with each of the intervals being contributed by a cache process C_j, where $1 \leq j \neq i \leq k$.

- Ω_{I_n}, or *the boundary set* of I_n: the set of all intervals that contain I_n. Note that there are exactly $k-1$ intervals in a boundary set.
- ψ_{I_n}, or *the latest interval* in Ω_{I_n}: Let $\psi_{I_n} = [J_{n'}, J_{n'+1}] \in \Omega_{I_n}$. Then $J_{n'+1}$ has the largest "time-stamp" among all events in Ω_{I_n}. i.e., all events in Ω_{I_n} are processed earlier than $J_{n'+1}$ is.
- $\theta_{J_n,i}$: the number of I's in $[J_n, J_{n+1}]$.
- b_i: the number of buffers in B_i that have not yet been consumed by C_i.
- $b_{i,j}$. the number of messages sent from C_j to C_i. Those messages have not been processed by C_i yet. It is clear that $b_{i,j} \leq b_i$.

Figure 12 shows an input trace τ_i for cache i where $k = 3$ (i.e., there are three cache processes i, j, and l): Interval $[L_2, L_3] = \{L_2, I_5, J_2, I_6, I_7, L_3\}$, and interval $[J_1, J_2] = \{J_1, I_3, L_1, I_4, L_2, I_5, J_2\}$. The boundary set for I_5 is $\Omega_{I_5} = \{[L_2, L_3], [J_1, J_2]\}$ since $I_5 \in [L_2, L_3] \cap [J_1, J_2]$. The latest interval in Ω_{I_5} is $\psi_{I_5} = [L_2, L_3]$ and $e_l = L_3$. $\theta_{J_1,i} = 3$ is the number of I's in $[J_1, J_2]$.

Lemma 6.1. When C_i reaches J_n, where $1 \leq n \leq m_j$, we have $b_{i,j} = 0$.

Proof: It is clear that if C_i reaches S_l then C_i has processed all $S_{l'}$, where $1 \leq l' \leq l$. When C_i reaches J_n, C_j is in one of the following two states:

1. C_j has not yet reached J_n: it is obvious that $b_{i,j} = 0$.
2. C_j reaches J_n earlier than C_i does: C_j waits until all other cache processes reach J_n. Thus, C_j does not send any new message until C_i reaches J_n. When C_i reaches J_n, it has already consumed all events in the queue, and thus $b_{i,j} = 0$. ∎

Let S be an arbitrary synchronous point, and let I and J be arbitrary shared references to cache i and j respectively. Lemma 6.1 tells us that $b_{i,j}$ is periodically reduced to 0; that is, if there are s I's between J_n and J_{n+1}, then the value of $b_{i,j}$ is bounded by s in the interval $[J_n, J_{n+1}]$. Figure 13

(a) and (c) show the worst case of the maximum number of required buffers in $[J_n, J_{n+1}]$ when $s = 4$: C_j produces 4 messages before C_i consumes any. Thus $b_{i,j}$ grows from 0 to 4, and then reduces to 0. Figure 13 (b) and (d) show the best case when $b_{i,j}$ alternates between 0 and 1 during the interval. For the buffer analysis we shall consider the worst case.

We first give an expression for an upper bound, b_i, of the number of messages that C_i might have to process when reaching a shared reference in its own trace.

Theorem 6.1: Let Ω_{I_n} be the boundary set of I_n, and $\psi_{I_n} = [J_{n'}, J_{n'+1}]$ be the latest interval. If C_i reaches I_n, then an upper bound on b_i is $\theta_{J_{n'},i}$.

Proof: Let $[L_{n_l}, L_{n_l+1}] \in \Omega_{I_n}$, for all l, where $1 \leq l \neq i \leq k$. When C_i reaches I_n, the maximum value for b_i is equal to the number of I's in $[I_n, J_{n'+1}]$, because those processes C_l that reach L_{n_l+1} earlier than C_i does, have to wait for C_i, and cannot send any more messages to C_i (by Lemma 6.1). Since $I_n \in [J_{n'}, J_{n'+1}]$, we have $[I_n, J_{n'+1}] \subseteq [J_{n'}, J_{n'+1}]$. Thus, $b_i \leq \theta_{J_{n'},i}$. ∎

Consider the example in Figure 12: When C_i reaches I_5, C_j can at most reach J_2 and must wait at J_2 since C_i has not reached J_2 yet. Similarly, C_l can at most reach L_3. Thus, the possible messages queued in B_i are messages for I_6, and I_7. And $b_i \leq 2 < 3 = \theta_{L_2,i}$.

The next step is to find the value $\theta_{J_{n'},i}$ for the latest interval. Let S be a synchronous point. From our assumptions of identical statistical cache behaviors, we have

$$p_{i,j} = Prob\{\text{an } S \text{ is a } J \mid \text{an } S \text{ is either an } I \text{ or a } J\} = \frac{m_j}{m_i + m_j}$$

and

$$q_{i,j} = 1 - p_{i,j} = Prob\{\text{an } S \text{ is an } I \mid \text{an } S \text{ is either an } I \text{ or a } J\} = \frac{m_i}{m_i + m_j}$$

To estimate the buffer space complexity, we construct a two-state Markov chain $\{Q\}$ with transition probability matrix

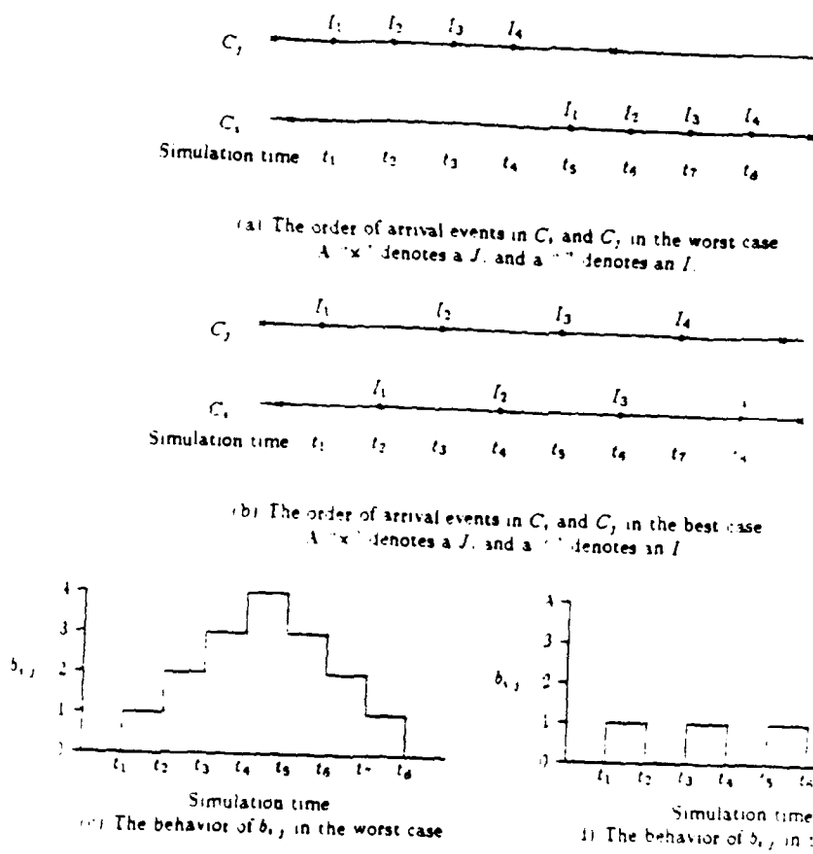
$$P = \begin{bmatrix} q_{i,j} & p_{i,j} \\ 0 & 1 \end{bmatrix}$$

and the initial probabilities

$$P\{Q_0 = 0\} = q_{i,j}, \quad P\{Q_0 = 1\} = p_{i,j}$$

Figure 14 shows the state transition diagram. State 0 represents the fact that we had a sequence of I's after J_n before encountering J_{n+1}. State 1 is absorbing. The number of steps that the Markov chain stays in state 0 is equal to $\theta_{J_{n'},i}$, i.e., an upper bound on the amount of buffering needed by C_i. In the following analyses, we answer two questions:

1. How long does it take to reach state 1, i.e., what is this upper bound?
2. What is the probability that the Markov chain remains in state 0 longer than some predetermined number of steps $l_{i,j}$, i.e., given some buffer length what is the probability of overflow?



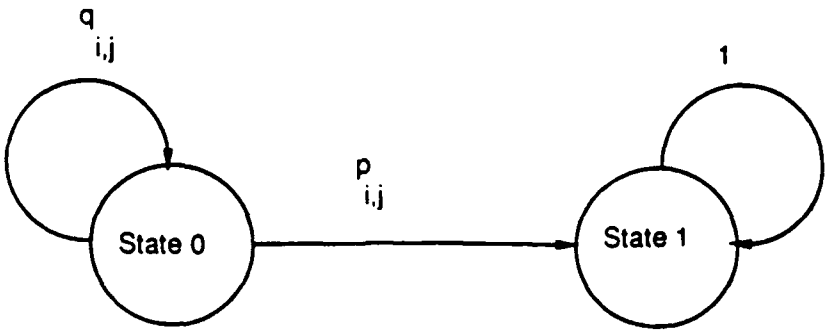
(a) The order of arrival events in C_i and C_j in the worst case
A "x" denotes a J_i and a "·" denotes an I_i.

(b) The order of arrival events in C_i and C_j in the best case
A "x" denotes a J_i and a "·" denotes an I_i

(c) The behavior of $b_{i,j}$ in the worst case

(d) The behavior of $b_{i,j}$ in the best case

Figure 13: The behavior of $b_{i,j}$.



State 0: the current arrival event is an I.
State 1: the current arrival event is a J.
Transition 0 to 0: the next arrival event is an I.
Transition 0 to 1: the next arrival event is a J.
Transition 1 to 1: the next arrival event is a J.

Figure 14: The state transition diagram of the buffer model.

Let $\theta_{i,j} = E[\theta_{J_{n'},i}]$ where $\theta_{i,j}$ can be viewed as the expected number of steps to reach state 1, or the expected number of I's between two consecutive J's. Thus, we rephrase the above questions as: (1) What is the expected number of I's between two consecutive J's? (2) What is the probability δ that $\theta_{i,j} > l_{i,j}$?

Lemma 6.2: Let T be the number of steps that it takes for the Markov chain to reach state 1, and let $\delta = Prob\{T > l_{i,j}\}$. Then (a) $\theta_{i,j} = \frac{m_i}{m_j}$ and (b) $\delta = q_{i,j}^{l_{i,j}+1}$.

Proof: (a) $\theta_{i,j} = E[T] = \sum_{n=0}^{\infty} n q_{i,j}^n p_{i,j} = \frac{p_{i,j} q_{i,j}}{(1-q_{i,j})^2} = \frac{q_{i,j}}{p_{i,j}} = \frac{m_i}{m_j}$

(b) For $k = 1, 2, ...,$

$$Prob\{T = k | Q_0 = 0\} = Prob\{Q_k = 1\} Prob\{Q_{k-1} = 0\} = p_{i,j}(q_{i,j})^{k-1}$$

we have

$$\forall l_{i,j} > 0 \quad Prob\{T \leq l_{i,j} | Q_0 = 0\} = \sum_{k=1}^{l_{i,j}} p_{i,j}(q_{i,j})^{k-1} = 1 - (q_{i,j})^k$$

$$\text{and} \quad Prob\{T > l_{i,j} | Q_0 = 0\} = (q_{i,j})^k$$

Clearly, for all $l_{i,j} > 0$, $\quad Prob\{T > l_{i,j} | Q_0 = 1\} = 0$. According to the above relations,

$$\forall l_{i,j} > 0, \quad \delta = Prob\{T > l_{i,j}\} = (q_{i,j})^{l_{i,j}+1} \quad \blacksquare$$

Now, if we give ourselves some probability of overflow δ, we can derive an upper bound on the number of buffer entries needed before we reach the overflow with δ probability. We compute the amount of buffering needed by each cache process and sum up over all processes.

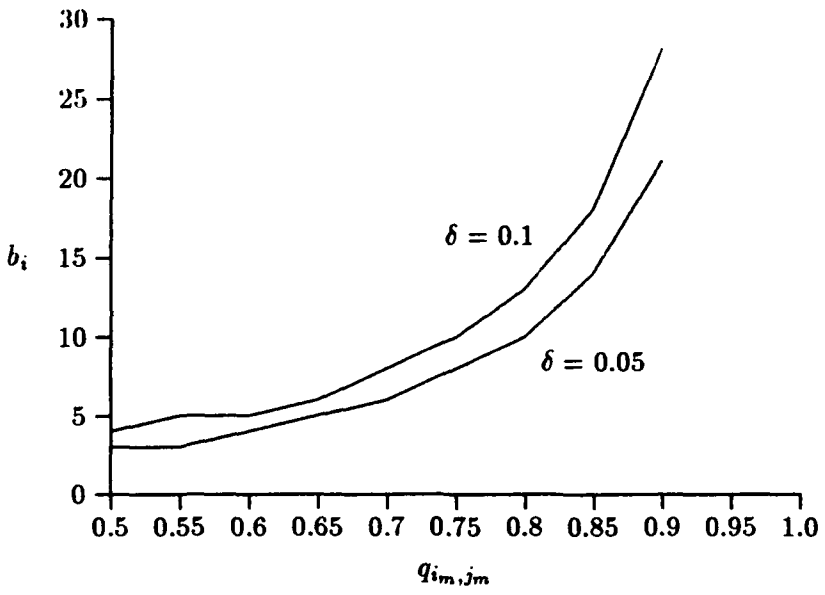


Figure 15: The relationship between b_i and q_{i_m,j_m}.

Let $q_{i_m,j_m} = \max_{1 \le i,j \le k} q_{i,j}$, where $i \neq j$. Then we have the following theorem:

Theorem 6.2: The upper bound of the buffer size $|B|$ with overflow probability δ is

$$|B| \le k(\lceil \log_{q_{i_m,j_m}} \delta \rceil - 1)$$

Proof: According to Lemma 6.2 (b), we have

$$l_{i,j} = \begin{cases} \lceil \log_{q_{i,j}} \delta \rceil - 1 & \text{if } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

According to Theorem 6.1 and the fact that, for $0 \le q' \le q \le 1$ and $0 < \delta < 1$, $\lceil \log_q \delta \rceil \ge \lceil \log_{q'} \delta \rceil$, we have

$$\forall i,\ 1 \le i \le k, \quad b_i \le \lceil \log_{q_{i_m,j_m}} \delta \rceil - 1$$

Thus,

$$|B| = \sum_{i=1}^{k} b_i \le k(\lceil \log_{q_{i_m,j_m}} \delta \rceil - 1) \quad \blacksquare$$

The relationship between this upper bound of b_i, δ, and q_{i_m,j_m} is shown in Figure 15 with $\delta = 0.1$ and 0.05.

Since the caches exhibit similar behaviors, we can assert that $0.3 \le q_{i,j} \le 0.7$ hence $q_{i_m,j_m} = 0.7$. Then, according to Theorem 6.2, the upper bound of $|B|$ is $6k$; that is, under reasonable assumptions for the distribution of shared references, the space complexity of B is $O(k)$. This shows the practicality of the modified algorithm and also justifies our timing analysis with infinite buffers.

7 Conclusions

We have discussed the design and analysis of parallel algorithms for the trace-driven simulation of multiprocessor cache coherence protocols.

We began from the observation that "merged traces", in which the trace information provided to the process simulating each cache includes the shared references generated by all other caches, ease the task of simulation, because "conditional events" are transformed into "unconditional events" by this approach.

One might think that synchronization between cache simulation processes would be required at each "interaction point" (each shared reference). A key result of this paper is that this is not true. The "synchronous points" are a subset of the interaction points – an empty subset in some cases. This lack of synchronization means that parallel trace-driven simulation of multiprocessor cache coherence protocols can yield much greater speedup than one would expect.

We showed how the synchronization requirements vary with the specific cache coherence protocol under investigation. We presented a simulation algorithm, and analyzed its time and space requirements.

Acknowledgments

We would like to thank John Zahorjan for extensive comments on an earlier, related paper.

References

[1] Agarwal, A., Sites, R.L., and Horowitz, M. ATUM: A New Technique for Capturing Address Traces Using Microcode. *Proc. 13th Annual International Symposium on Computer Architecture*, pages 119–127, 1986.

[2] Archibald, J.K. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA 98195, February 1987. Technical Report 87-02-06.

[3] Chandy, K.M., and Misra, J. Conditional Knowledge as a Basis for Distributed Simulation. Technical Report TR-87-5251, Computer Sciences Department, University of Texas at Austin, 1987.

[4] Cheriton, D.R., Gupta, A., Boyle, P.D., and Goosen, H.A. The VMP Multiprocessor: Initial Exeperience, Refinements, and Performance Evaluation. *Proc. 15th Annual International Symposium on Computer Architecture*, pages 410–421, 1988.

[5] David, H.A. *Order Statistics.* Wiley and Sons, 2nd edition, 1962.

[6] Eager, D.L., Zahorjan, J., and Lazowska, E.D. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[7] Eggers, S.J. and Katz, R.H. A Characterization of Sharing in Parallel Programs and Its Applications to Coherency Protocol Evaluation. *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–383, 1988.

[8] Fromm, H., et al. Experiments with Performance Measurement and Modeling of a Processor Array. *IEEE Transactions on Computers*, C-32(1):15–31, January 1983.

[9] Goodman, J.R. Cache Memory Optimization to Reduce Processor/Memory Traffic. *Journal of VLSI and Computer Systems*, 2(1):61–86, 1987.

[10] Hercksen, U., Kla, R., Kleinoder, W., and Kneissl, F. Measuring Simultaneous Events in a Multiprocessor System. *Proc. 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 77–82, 1982.

[11] Jefferson, J.D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[12] Jefferson, J.D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[13] Katz, R., Eggers, S., Wood, D.A., Perkins, C., and Sheldon, R.G. Implementing A Cache Consistency Protocol. *Proc. 12th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[14] McCreight, E. The Dragon Computer System: An Early Overview. Technical report, Xerox Corp., 1984.

[15] Misra, J. Distributed Discrete Event Simulation. *Computing Surveys*, 18(1):39-65, March 1986.

[16] Osterhaug, A. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 1986.

[17] Papamarcos, M., and Patel, J. A Low Overhead Coherence Solution for Multiprocessors With Private Cache Memories. *Proc. 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[18] Peterson, J.L., and Silberschatz, A. *Operating System Concepts.* Addison-Wesley Publishing Company, Inc., 2nd edition, 1985.

[19] Rudolph, L. and Segall, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *Proc. 11th Annual International Symposium on Computer Architecture*, pages 340–347, 1984.

[20] Thacker, C.P., Stewart, L.C. Firefly: A Multiprocessor Workstation. *Proc. of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[21] Wyatt, D. Simulation Programming On a Distributed System: A Preprocessor Approach. *Distributed Simulation 1985*, pages 32–36, January 1985.

A The Timing Analysis of the Basic Algorithm

This appendix gives the timing analysis of the basic algorithm. In addition to the notation described in Section 3, the following notation is introduced:

- $m' = Np_s$: the total number of shared references to a cache.
- $m = m'k$: the total number of synchronous points to a cache.
- $r_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq k$: the elapsed time for C_i to handle private references between the $i-1^{th}$ and i^{th} synchronous points. By convention, we have a 0^{th} synchronous point before the first event to each cache.
- n: the number of processors that perform the simulation in the multiprocessor environment. We assume that $n \leq k$.

Theorem A.1: If $\forall i$, $1 \leq i \leq m$, $r_{i,1}, r_{i,2}, ..., r_{i,k}$ are independent, identically distributed random variables with mean u_i and standard deviation s_i then

$$E[t_p] \leq \sum_{i=1}^{m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)$$

Proof:

$$t_p = \sum_{i=1}^{m}(\max_{1\leq j\leq k} r_{i,j})$$

and

$$E[t_p] = E[\sum_{i=1}^{m}(\max_{1\leq j\leq k} r_{i,j})]$$

$$= \sum_{i=1}^{m} E[(\max_{1\leq j\leq k} r_{i,j})]$$

David showed [5] that if $r_{i,1}, r_{i,2}, ..., r_{i,k}$ are independent, identically distributed random variables with mean u_i and standard deviation s_i then

$$E[\max_{1\leq j\leq k} r_{i,j}] \leq u_i + \frac{k-1}{\sqrt{2k-1}}s_i$$

Thus

$$E[t_p] \leq [\sum_{i=1}^{m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)] \quad \blacksquare$$

If we assume in addition that the mean and standard deviation of $r_{i,j}$ are the same for all caches, then we have following corollary:

Corollary A.1: If, in addition, $u_i = u_j = u$ and $s_i = s_j = s$ for $1 \leq i, j \leq m$, then

$$E[t_p] \leq m[u + \frac{k-1}{\sqrt{2k-1}}s] \tag{1}$$

Proof: Directly from Theorem A.1. ∎

If we further assume that the arrivals of synchronous points to each cache form a Poisson process, then we have

Theorem 3.1:

$$E[t_p] \leq [N + (k-1)m'](1 + \frac{k-1}{\sqrt{2k-1}})\Delta t_p$$

Proof: Since the arrivals of synchronous points to each cache form a Poisson process, Corollary A.1 holds, and $r_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq k$, has an exponential distribution with an average arrival rate of

$$\lambda = \frac{\textit{total number of synchronization points}}{\textit{time to handle all references in a trace}} = \frac{m}{(N - m' + km')\Delta t_p}$$

Thus, the mean and variance of $r_{i,1}, ..., r_{i,k}$ are

$$\mu = \frac{1}{\lambda}, \quad s^2 = \frac{1}{\lambda^2} \tag{2}$$

Applying Equation 2 to Equation 1, we have

$$E[t_p] \leq [N + (k-1)m'](1 + \frac{k-1}{\sqrt{2k-1}})\Delta t_p = [1 + (k-1)p_s](1 + \frac{k-1}{\sqrt{2k-1}})N\Delta t_p \quad ∎$$

Corollary A.2: If $\forall i$, $1 \leq i \leq m$, $r_{i,1}, r_{i,2}, ..., r_{i,k}$ are independent, identically distributed random variables with mean u_i and standard deviation s_i then

$$S_\infty \geq \frac{kN[1 + p_s(k-1)]}{[\sum_{1 \leq i \leq m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)]\rho}$$

Proof: According to Theorem A.1 and the definition of the sequential time in Section 5.1.

$$S_\infty = (\frac{E[t_s]}{E[t_p]}) \geq \frac{k[N + (k-1)m']\Delta t_s}{[\sum_{1 \leq i \leq m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)]\Delta t_p}$$

$$= \frac{k[N + (k-1)m']}{[\sum_{1 \leq i \leq m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)]\rho} = \frac{kN[1 + p_s(k-1)]}{[\sum_{1 \leq i \leq m}(u_i + \frac{k-1}{\sqrt{2k-1}}s_i)]\rho} \quad ∎$$

Theorem 3.2: If the arrivals of synchronous points are a Poisson process to each cache, then

$$S_\infty = \frac{E[t_s]}{E[t_p]} \geq \frac{k}{\rho(1 + \frac{k-1}{\sqrt{2k-1}})} \tag{3}$$

Proof: Directly from the definition of the sequential execution time and Theorem 3.1. ∎

Theorem 3.3: If there are n processors available in the parallel simulation, $1 \leq n \leq k$, and k is large, then

$$S(n) \geq \sqrt{k} - \frac{k-n}{\sqrt{k}+n}, \quad n \leq k$$

Proof: This proof uses Theorem 5.2 (cf. Section 5.2.3) which is re-written as

$$S(n) \geq \frac{nS_\infty}{(n-1)[1-\frac{S_\infty-1}{m_{max}-1}]+S_\infty}, \quad n \leq k$$

In the parallel cache simulation, $m_{max} = k$. Thus the lower bound of the speedup can be derived as follows: If k is large, then Equation 3 is expressed as

$$S_\infty \geq c'\sqrt{k}, \quad where\ c' = \frac{\sqrt{2}}{\rho} \tag{4}$$

If we assume that $\Delta t_p \leq \sqrt{2}\Delta t_s$ (this assumption is reasonable since the number of the synchronous points is much less than N in general), then $c' \leq 1$ in Equation 4, and the lower bound of S_∞ is $\sqrt{k}$.

Let $n = ck = c(S_\infty)^2$ where $0 < c < 1$, then

$$S(n) = S(ck) \geq \frac{cS_\infty^3}{(cS_\infty^2-1)[1-\frac{S_\infty-1}{S_\infty^2-1}]+S_\infty}$$

$$= S_\infty - \frac{(1-c)S_\infty}{cS_\infty+1}, \quad n \leq k$$

or

$$S(n) \geq \sqrt{k} - \frac{k-n}{\sqrt{k}+n}, \quad n \leq k \ \blacksquare$$

B The Second Moment of the Waiting Time

This appendix gives some facts about Erlang distribution, then uses these facts to derive the second moment of the time that cache process C_i waits for cache process C_j.

Definition 1: The probability density function for one-parameter Gamma distribution (or Erlang distribution) is $f_s(t) = \frac{\lambda}{(s-1)!}(\lambda t)^{s-1}e^{-\lambda t}$.

Facts B.1, B.2 and Theorem B.1 are directly from Definition 1:

Fact B.1: $tf_s(t) = \frac{s}{\lambda}f_{s+1}(t)$ and $t^2 f_s(t) = \frac{s(s+1)}{\lambda^2}f_{s+2}(t)$.

Fact B.2: $\int_{t=0}^{\infty} f_s(t)f_j(t)dt = \frac{\lambda(s+j-2)!}{2^{s+j-1}(s-1)!(j-1)!}$.

Theorem B.1: for $t \geq 0$, the distribution function for $f_s(t)$ is

$$F_s(t) = 1 - \sum_{j=0}^{s-1}\frac{(\lambda t)^j}{j!}e^{-\lambda t} = 1 - \frac{1}{\lambda}\sum_{j=1}^{s} f_j(t)$$

Fact B.3: $\sum_{j=1}^{s} \int_{t=0}^{\infty} f_s(t) f_j(t) dt = \frac{\lambda}{2}$

Proof: $\int_{t_1=0}^{\infty} \int_{t_2=0}^{\infty} f_s(t_1) f_s(t_2) dt_2 dt_1$

$$= \int_{t_1=0}^{\infty} \int_{t_2=t_1}^{\infty} f_s(t_1) f_s(t_2) dt_2 dt_1 + \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_s(t_1) f_s(t_2) dt_1 dt_2 = 1$$

$$\Rightarrow X = \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_s(t_1) f_s(t_2) dt_1 dt_2 = \frac{1}{2}$$

If we integrate X directly, then

$$X = \int_{t_2=0}^{\infty} [1 - F_s(t_2)] f_s(t_2) dt_2 = \frac{1}{\lambda} \sum_{j=1}^{s} \int_{t=0}^{\infty} f_s(t) f_j(t) dt = \frac{1}{2} \quad \blacksquare$$

Theorem B.2: Let $M[W_s^j]$ be the second moment of the time that C_i waits for C_j, then $M[W_s^{(j)}] = \frac{s}{\lambda^2}$

Proof: $M[W_s^{(j)}]$ is expressed as

$$\int_{t_2=0}^{\infty} \int_{t_1=t2}^{\infty} (t_1 - t_2)^2 f_s(t_1) f_s(t_2) dt_1 dt_2 = X + Y + Z \quad \text{where}$$

$$\begin{aligned} X &= \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_1^2 f_s(t_1) f_s(t_2) dt_1 dt_2, \\ Y &= \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_2^2 f_s(t_1) f_s(t_2) dt_1 dt_2, \quad \text{and} \\ Z &= -2 \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} t_1 t_2 f_s(t_1) f_s(t_2) dt_1 dt_2 \end{aligned}$$

From Fact B.1 X is rewritten as

$$\frac{s(s+1)}{\lambda^2} \int_{t_2=0}^{\infty} \int_{t_1=t_2}^{\infty} f_{s+2}(t_1) f_s(t_2) dt_1 dt_2$$

According to Theorem B.1

$$\begin{aligned} X &= \frac{s(s+1)}{\lambda^3} \int_{t_2=0}^{\infty} [\sum_{j=1}^{s+2} f_j(t_2)] f_s(t_2) dt_2 \\ &= \frac{s(s+1)}{\lambda^3} [\int_{t_2=0}^{\infty} (\sum_{j=1}^{s+2} f_j(t_2) f_s(t_2) + f_{s+1}(t_2) f_s(t_2) + f_{s+2}(t_2) f_s(t_2)) dt_2] \end{aligned}$$

Applying Fact B.3 and Fact B.2

$$X = \frac{s(s+1)}{\lambda^3} [\frac{\lambda}{2} + \frac{\lambda(2s-1)!}{2^{2s} s!(s-1)!} + \frac{\lambda(2s)!}{2^{2s+1}(s+1)!(s-1)!}]$$

From Fact B.1 Y is rewritten as

$$\frac{s(s+1)}{\lambda^2}\int_{t_2=0}^{\infty}\int_{t_1=t_2}^{\infty} f_s(t_1)f_{s+2}(t_2)dt_1dt_2$$

According to Theorem B.1

$$\begin{aligned} Y &= \frac{s(s+1)}{\lambda^3}\int_{t_2=0}^{\infty}[\sum_{j=1}^{s} f_j(t_2)]f_{s+2}(t_2)dt_2 \\ &= \frac{s(s+1)}{\lambda^3}\int_{t_2=0}^{\infty}(\sum_{j=1}^{s+2} f_j(t_2)f_{s+2}(t_2) - f_{s+1}(t_2)f_{s+2}(t_2) - f_{s+2}(t_2)f_{s+2}(t_2))dt_2 \end{aligned}$$

Finally, from Fact B.3 and Fact B.2

$$Y = \frac{s(s+1)}{\lambda^3}[\frac{\lambda}{2} - \frac{\lambda(2s+1)!}{2^{2s+2}s!(s+1)!} - \frac{\lambda(2s+2)!}{2^{2s+3}(s+1)!(s+1)!}]$$

From Fact B.1 Z is rewritten as

$$-\frac{2s^2}{\lambda^2}\int_{t_2=0}^{\infty}\int_{t_1=t_2}^{\infty} f_{s+1}(t_1)f_{s+1}(t_2)dt_1dt_2 = -\frac{s^2}{\lambda^2}$$

Thus

$$\begin{aligned} X+Y+Z &= \frac{s(s+1)}{\lambda^2}[1 + \frac{(2s-1)!}{2^{2s}(s!)(s-1)!}(1 + \frac{s}{s+1} - \frac{(2s+1)\cdot 2s}{2(s+1)s})] - \frac{s^2}{\lambda^2} \\ &= \frac{s(s+1)}{\lambda^2} - \frac{s^2}{\lambda^2} = \frac{s}{\lambda^2} \quad \blacksquare \end{aligned}$$

C The Expected Parallel Simulation Time

This appendix derives an upper bound of the expected parallel simulation time $E[t_p]$.

Theorem C.1: If there are an unlimited number of processors, then an upper bound of the expected parallel simulation time $E[t_p]$ is

$$E[t_p] \leq N\{[1 + (k-1)p_s]\Delta t_p + p_s(E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}}\sqrt{V[W^{(j)}]})\}$$

where $\sqrt{V[W^{(j)}]}$ is the standard deviation of $W^{(j)}$.

Proof: The waiting time of C_i at point S is $W = \max_{1\leq j\leq k, i\neq j} W^{(j)}$. Since there are m shared references to each cache, $E[t_p]$ is approached by:

$$E[(N - m + km)\Delta t_p + mW] = N\{[1 + (k-1)p_s]\Delta t_p + p_s E[W]\}$$

David showed [5] that if $W^{(1)}, W^{(2)}, ..., W^{(k)}$ are independent, identically distributed random variables (cf. **P.5**) with mean u and standard deviation s then

$$E[W] = E[\max_{1\leq j\leq k} W^{(i)}] \leq u + \frac{k-1}{\sqrt{2k-1}}s$$

Thus for $k \geq 2$, we have

$$E[W] \leq E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}} \cdot \sqrt{V[W^{(j)}]}, \quad k \geq 2$$

and by substitution

$$E[t_p] \leq N\{[1 + (k-1)p_s]\Delta t_p + p_s(E[W^{(j)}] + \frac{k-2}{\sqrt{2k-3}}\sqrt{V[W^{(j)}]})\} \quad \blacksquare$$

Lemma C.1: Let $P = \frac{1}{k}$ be the probability that a synchronous point is an inserted reference. Then $M[W_s^{(j)}] = \frac{k}{\lambda^2}$

Proof: $M[W_s^{(j)}]$ can be expressed as

$$\int_{t_2=0}^{\infty}\int_{t_1=0}^{\infty}(W_s^{(j)})^2 \frac{\lambda}{(s-1)!}(\lambda t_1)^{s-1}e^{-\lambda t_1}\frac{\lambda}{(s-1)!}(\lambda t_2)^{s-1}e^{-\lambda t_2}dt_1dt_2$$

which is rewritten as

$$\int_{t_2=0}^{\infty}\int_{t_1=t2}^{\infty}(t_1-t_2)^2 \frac{\lambda}{(s-1)!}(\lambda t_1)^{s-1}e^{-\lambda t_1}\frac{\lambda}{(s-1)!}(\lambda t_2)^{s-1}e^{-\lambda t_2}dt_1dt_2$$

According to Theorem B.2 in Appendix B, we have $M[W_s^{(j)}] = \frac{s}{\lambda^2}$ and

$$M[W^{(j)}] = \sum_{s=1}^{\infty} p(1-p)^{s-1}M[W_s^{(j)}] = \frac{k}{\lambda^2} \quad \blacksquare$$

Lemma C.2: An upper bound of $E[W^{(j)}]$ is $\frac{\sqrt{k}}{\lambda}$.

Proof: Directly from Lemma C.1 and the fact that $E[W^{(j)}] \leq \sqrt{M[W^{(j)}]}$. $\blacksquare$

Lemma C.3: An upper bound of $V[W^{(j)}]$ is $\frac{k}{\lambda^2}$.

Proof: Directly from Lemma C.1 and the fact that $V[W^{(j)}] \leq M[W^{(j)}]$. $\blacksquare$

Theorem 5.1: For all $k \geq 2$,

$$E[t_p] \leq 2[1 + (k-1)p_s]N\Delta t_p$$

Proof: According to Theorem C.1 and Lemmas C.2 and C.3

$$E[t_p] \leq N\{[1 + (k-1)p_s]\Delta t_p + p_s(\frac{\sqrt{k}}{\lambda} + \frac{k-2}{\sqrt{2k-3}} \cdot \frac{1}{\lambda}\sqrt{k})\}$$

Applying **P.4**

$$\begin{aligned} E[t_p] &\leq [1+(k-1)p_s + p_s\frac{N-m+km}{km}(\sqrt{k}+\frac{k-2}{\sqrt{2k-3}}\cdot\sqrt{k})]N\Delta t_p \\ &= \{[N+(k-1)p_s](1+\frac{1}{\sqrt{k}}+\frac{k-2}{\sqrt{k(2k-3)}})\}N\Delta t_p \end{aligned}$$

Since

$$\forall k \geq 2, \quad \frac{1}{\sqrt{k}}+\frac{k-2}{\sqrt{k(2k-3)}} < 1$$

We have for $k \geq 2$

$$E[t_p] \leq 2[1+(k-1)p_s]N\Delta t_p \quad \blacksquare$$

Optimality Considerations of "Time Warp" Parallel Simulation *

Yi-Bing Lin and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The critical path in the event-precedence graph of a simulation application is a lower bound of execution time for any conservative parallel simulation. We refer to any parallel simulation that achieves this time as a conservative optimal simulation. This paper derives a relationship between a conservative optimal simulation and the "Time Warp" or "optimistic" parallel simulation, and compares the performance of the Time Warp simulation with Chandy-Misra conservative simulation.

We show that Time Warp simulation with aggressive cancellation is not conservative optimal in general, even under the assumption that the operational overhead (such as state saving/restoration and global virtual time calculation) is 0. We do, however, derive a sufficient condition for Time Warp to be conservative optimal, and we show some simulation problems that meet this condition. (We refer to such simulations as Time Warp simulations that satisfy the sufficient conservative optimal condition or TWSO.) We show that by applying the lazy cancellation technique to TWSO, Time Warp always outperforms a conservative optimal simulation. Given what we feel are equivalently favorable assumptions for both the Time Warp approach and the Chandy-Misra approach, we show that the Time Warp approach outperforms the Chandy-Misra approach in every feedforward network simulation. For feedback networks without lookahead, we show that in most cases Time Warp outperforms Chandy-Misra, even assuming that deadlock detection/recovery overhead is 0 in the Chandy-Misra simulation.

1 Introduction

Experience [5] has shown that the following properties are critical determinants of the efficiency of a parallel simulation: (1) granularity (relatively long computation per event), (2) high potential concurrency (short critical paths [1], balanced decomposition, etc.), (3) distributed geometry of computation; that is, the simulation should exhibit spatial locality (small communication fan out/in per object message directed to nearby objects) and temporal locality (events are scheduled in the near future), (4) balanced assignment of processes to processors, and (5) the *parallel simulation protocol* which is used to provide synchronization.

This paper concerns the last of these five critical issues: the parallel simulation protocol. The most common protocols are the Time Warp approach [4] and the Chandy-Misra approach [12]. Time Warp takes an optimistic approach. A process executes every message as soon as it arrives. If a message with a smaller timestamp subsequently arrives, the process must roll back its state to the time of the earlier message and re-execute from that point. Chandy-Misra takes a conservative approach. A process does not execute a message until it is certain that no message with an earlier timestamp can ever arrive. Since processes may have to wait for other processes to produce messages before they can proceed, deadlock may occur even if the system being modeled is deadlock-free. Deadlock resolution is required in this approach.

Experiments have shown that both approaches are viable. (The viability of the Time Warp approach was perhaps surprising to some practitioners of the earlier Chandy-Misra approach.) While neither approach has yielded phenomenal speedups, each has yielded good speedups. Reducing the running time of a simulation from a scale of hours to a scale of minutes, which is clearly achievable, dramatically changes the nature of simulation studies, since a design space can be investigated interactively.

Time Warp and Chandy-Misra simulations lie at opposite ends of a spectrum of approaches [12]. It is interesting to consider whether one is preferable to the other, either for certain problems or in general. This question is difficult to answer experimentally, because new optimizations of each approach are continually being invented. In this paper, we consider this question analytically: we make what we feel are "equivalently favorable" assumptions for each approach, and prove a number of results about their relative performance under these assumptions.

The critical path in the event-precedence graph of a simulation application is a lower bound of execution time for any conservative parallel simulation. We refer to any parallel simulation that achieves this time as a conservative optimal simulation. We show that "Time Warp" or "optimistic" parallel simulation is not conservative optimal in general, even under the assumption that the operational overhead (such as state saving/restoration and global virtual time calculation) is 0. We do, however, derive a sufficient condition for Time Warp to be conservative optimal, and we show some simulation problems that meet this condition. (We refer to such simulations as Time Warp simulations that satisfy the sufficient conservative optimal condition or TWSO.) We show that by applying the lazy cancellation technique to TWSO, Time Warp always outperforms a conservative optimal simulation. Given what we feel are equivalently favorable assumptions for both the Time Warp approach and the Chandy-Misra approach, we show that the Time Warp approach outperforms the Chandy-Misra approach in every feedforward network simulation. For feedback networks without lookahead, we show that in most cases Time Warp outperforms Chandy-Misra even assuming that deadlock detection/recovery overhead is 0 in the Chandy-Misra simulation.

*This work was supported by the National Science Foundation (Grants CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

2 Anachronous Phenomena

The behavior of Time Warp simulation is difficult to analyze. Besides the normal messages (called *true messages*) executed in the simulation, the Time Warp mechanism generates two other types of messages: *false messages* and *antimessages*. False messages are created by *anachronism* (out-of-order executions of messages). Antimessages are created to annihilate false messages. The following example illustrates the concepts of false messages and antimessages.

Example 2.1: Consider Figure 1. Message m_1 is sent from process p_1 to process p_3 with simulation timestamp 10. Message m_2 is sent from process p_2 to process p_3 with simulation timestamp 5. Suppose that p_3 receives m_1 at physical time 2, and m_2 at physical time 7. Since Time Warp takes an optimistic approach, m_1 is executed before m_2 arrives. Suppose that m_1's execution at time 2 [1] results in the scheduling of a message m_3, and m_3 is received by p_4 at time 4. Since m_1 should not be executed before m_2, m_3 is a false message, and any effect caused by m_3 must be undone. At time 7, this is detected. Hence m_1 is rolled back, and an antimessage is sent to cancel m_3. The true and false messages are defined more carefully as follows:

Definition 2.1: Consider a Time Warp simulation. A message is said to be *true* if execution of the message has an effect on the simulation. A message is said to be *false* if execution of the message has no effect on the simulation. A true message is possibly rolled back several times, but the effect of its *final execution* is preserved. A false message is possibly rolled back several times, and is eventually *annihilated* by an antimessage.

Note that Definition 2.1 is appropriate for Time Warp simulations with aggressive cancellation [4], but it may be confusing when the *lazy cancellation technique* is used. We will give more details about lazy cancellation later. There are two anachronous phenomena in Time Warp simulations:

Phenomenon 2.1: It is possible for a true message to be sent for the wrong reasons. Example 2.1 mentions that m_3 is a false message, and must be annihilated. It is possible that the execution of m_1 and m_2 in the correct order will result in the scheduling of a message m_4 to p_4 which is identical to m_3. In such a case, m_3's execution is correct although m_3 is sent for the wrong reason. A technique called *lazy cancellation* was developed to take advantage of this phenomenon: At time 7, m_1 is rolled back, but no antimessage is sent to annihilate m_3. Instead, m_2 is executed immediately. When m_4 is scheduled, the contents of m_4 and m_3 are compared. (Note that a copy of m_3 must have been saved in p_3's output message queue.) If they are the same, nothing will happen (and in effect, m_4 is correctly executed ahead of its scheduled time). Otherwise, m_4 is sent to p_4, and m_3 is undone. According to Definition 2.1, m_3 is considered as a true message if it is identical to m_4 (assuming that m_4 is a true message), otherwise it is a false message.

Phenomenon 2.2: It is possible for a correct computation to be rolled back by a false message. Consider Example 2.1. Suppose p_4 executes a true message m_0 at time 4. Suppose m_0 has timestamp 18, and p_4's computation up to time 4 is correct (i.e., after time 4, p_4 never receives a true message with timestamp smaller than 18). The arrival of the false message m_3 results in the rollback of m_0 (i.e., the correct computation is rolled back by a false message).

[1] From now on, we abbreviate "simulation timestamp" as "timestamp", and "physical time" as "time".

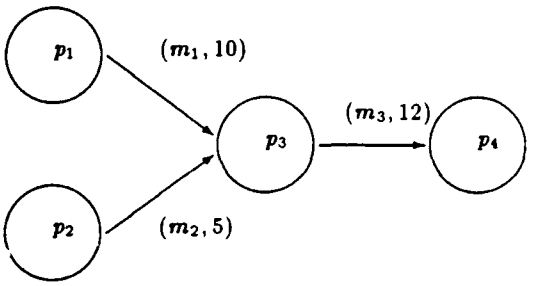


Figure 1: False message and antimessage.

And the computation caused by m_0 must be re-done some time after m_3 is annihilated. If lazy cancellation is used, the recomputation of m_0 is further delayed.

It should be obvious that while lazy cancellation may improve the performance of Time Warp by taking advantage of Phenomenon 2.1, it may degrade the performance of Time Warp by heightening the effect of Phenomenon 2.2. The effect of lazy cancellation on Time Warp simulation is discussed in [8]. This paper assumes that the aggressive cancellation [4] is used. The following assumptions are used throughout this paper.

Assumption 2.1: Each process in the simulation is assigned to a dedicated processor.

Assumption 2.2: Execution of a message m is rolled back immediately upon the arrival of a message m' if m's timestamp is larger than m''s. This is called *message preemption*. Message preemption is considered to be a requirement for Time Warp.

Assumption 2.3: The Time Warp operational overhead (such as state saving/restoration, global virtual time calculation) is 0.

The operational overhead identified in Assumption 2.3 merits further discussion. If we assume that the global virtual time calculation is done in the background then the main overhead of the Time Warp mechanism is due to the domino effect of antimessage transmissions instead of the operational overhead described above. Thus, Assumption 2.3 is reasonable when computation and message sending delay are long, or the size of the process state is small. Several techniques [2, 3] have been developed to efficiently reduce the overhead of the Time Warp operations.

In order to evaluate the performance of the Time Warp mechanism, we need to define what is *conservative optimal*. Our definition of the conservative optimal simulation is based on the *critical path analysis* of a simulation [1]: Let Ψ be the set of messages (events) executed by a group of processes in the simulation. Then there are two fundamental sequential constraints:

Constraint 2.1: If two messages m_1 and m_2 are scheduled for the same process p at timestamps ts_1 and ts_2 respectively, where $ts_1 < ts_2$, then m_1 must be executed before m_2.

Constraint 2.2: If a message m_1 at timestamp ts_1 causes message m_2 to be sent at timestamp ts_2 (and therefore $ts_1 < ts_2$) then m_1 must be executed before m_2.

Under Assumptions 2.1, 2.2, and 2.3, the Time Warp with aggressive cancellation is not conservative optimal because Phenomenon 2.2 may occur in the critical path [1] of the Time Warp simulation. The next section shows that the

Time Warp approach is always conservative optimal in absence of Phenomenon 2.2. We note that even when Phenomenon 2.2 exists, it is possible that Time Warp simulation outperforms the conservative optimal simulation if (1) lazy cancellation is assumed and (2) the effect of Phenomenon 2.1 is greater than the effect of Phenomenon 2.2. (In other words, the "conservative optimal" simulation under our definition is no longer optimal). No lower bound for the execution time of Time Warp simulation with lazy cancellation is known. A study about lazy cancellation mechanisms can be found in [8].

3 A Conservative Optimal Condition

This section describes a Time Warp simulation that satisfies a sufficient conservative optimal condition (TWSO), and proves the conservative optimality of TWSO. [2] Before we present TWSO, we shall give a formal definition of the execution time of a message in the conservative optimal simulation. We introduce the following notation:

- $ts(m)$: the timestamp of the message m.
- $O(m)$: the set of output messages scheduled as a result of the execution of the input message m.
- $I(m)$: $m' = I(m)$ iff $m \in O(m')$. In other words, the creation of m is a result of m''s execution.
- $Next(m)$: let m be an input message executed by the process p in the conservative optimal simulation, then $Next(m)$ is the next message executed by p.
- $Prev(m)$: $m' = Prev(m)$ iff $m = Next(m')$.
- $d(m)$: the message sending delay, or the elapsed time for sending m from a process p to another process q. $d(m)$ is assumed to be a constant.
- $\delta(m)$: the elapsed time to execute the input message m.
- $\tau_{op}(m)$: the time at which m's execution is started in the conservative optimal simulation.

Assumption 3.1: Let $m' \in O(m)$. We assume that m' is sent to the output channel at time $\tau_{op}(m) + \delta(m)$. (Our results generalize to the case that m' is sent at time t where $\tau_{op}(m) < t \leq \tau_{op}(m) + \delta(m)$.)

Definition 3.1: Consider any message m executed in a conservative optimal simulation. Let $m_1 = Prev(m)$ if $Prev(m)$ exists, and let $m_2 = I(m)$ if $I(m)$ exists. Let $\tau'_{op}(m) = \tau_{op}(m) + \delta(m)$. Then from Constraints 2.1 and 2.2

$$\tau_{op}(m) = \begin{cases} \max[\tau'_{op}(m_2), \tau'_{op}(m_1)], & \text{both } m_1 \text{ and } m_2 \text{ exist} \\ \tau'_{op}(m_2) + d(m), & m_1 \text{ does not exist} \\ \tau'_{op}(m_1), & m_2 \text{ does not exist} \\ 0, & \text{neither } m_1 \text{ nor } m_2 \text{ exist} \end{cases} \quad (1)$$

The execution time of the conservative optimal simulation is considered as a lower bound for conservative parallel simulation. It is clear that the set of messages executed in the conservative optimal simulation is the set of true messages in the Time Warp simulation. The functions defined for conservative optimal simulation (e.g., $d, \delta, ts, O, I, Next$, and $Prev$) are also defined for the true messages in Time Warp simulation. And the functions d, δ, ts, O, and I are defined for the false messages. We introduce two further notations:

- $P(m)$: the process that receives and executes the message m.
- $\tau_{tw}(m)$: the time at which m's final execution is started in the Time Warp simulation. Note that the final execution of a true message is never rolled back, and its execution is finished at time $\tau_{tw}(m) + \delta(m)$.

Now we define the TWSO as follows:

Definition 3.2: Consider a Time Warp simulation. Let m be any false message sent from a process p to another process q. Suppose that m is annihilated at time t. The simulation is said to satisfy the sufficient conservative optimal condition (the simulation is referred to as a TWSO) iff there exists a message m' sent from p to q after t such that $ts(m') < ts(m)$.

Lemma A.1 in [10] shows that, if a false message m is executed by a process in a TWSO, the message is eventually annihilated, and after the annihilation, p will receive a true message with timestamp less than $ts(m)$. This implies that a false message never rolls back correct computation. However, this is not enough to justify that TWSO is conservative optimal. We need to show that no other effect will make TWSO non-conservative optimal. In other words, we need to show that $\tau_{tw}(m) = \tau_{op}(m)$ for every true message m executed in the TWSO. The strategy is to number all true messages in the simulation (according to Constraint 3.1 and 3.2) such that if m_1 is executed before m_2 then the former is associated with a number less than the latter. Then we prove by induction on the associated numbers that $\tau_{tw}(m) = \tau_{op}(m)$ is true (cf. Theorem A.1 in [10]). always outperforms the conservative optimal simulation.

4 Comparing Time Warp and Chandy-Misra (Feedforward Systems)

This section shows that the Time Warp approach outperforms the Chandy-Misra approach in feedforward network [3] simulations. In other words, the execution time for the Time Warp simulation is never longer than the execution time for the Chandy-Misra simulation. This section uses Assumptions 2.1, 2.2, 2.3, and Assumptions 4.1-4.7 which we state now:

Assumption 4.1: The simulation never exhausts memory. This assumption implies that deadlocks do not occur in the Chandy-Misra simulation [7]. Thus, deadlock resolution (such as sending null messages or dynamic deadlock detection/recovery [12]) is not required, and thus there is no deadlock resolution overhead in the Chandy-Misra simulation.

Assumption 4.2: Lookahead exists in the simulated systems. We further assume that all processes in the simulated system have the same lookahead value ϵ. Our results generalize for the case that ϵ is not a constant. (This generalization is very important. In [9] we showed that lookahead may be a function of arrival messages.) This assumption implies that if a process p executes a message m at timestamp t, then any output message scheduled as a result of m's execution will have timestamp larger than $t + \epsilon$. (Our results generalize for the case that $ts(m) = t + \epsilon$.)

Assumption 4.3: All messages executed by a process have different timestamps. This assumption is introduced

[2] This section uses Assumptions 2.1, 2.2, 2.3, and Assumption 3.1 to be defined in this section.

[3] A feedforward graph (network) is a *dag* or a directed acyclic graph.

for analysis convenience. Our results generalize to the case that some messages have identical timestamps.

Assumption 4.4: Each source process p in the feedforward network behaves as follows: p computes for a while, then sends messages to its output channels. This procedure is repeated until p's local clock exceeds the end-of-simulation timestamp. Note that no message m executed in p is ever rolled back, and every message m is executed at the same time in both the Chandy-Misra and the Time Warp simulations.

Assumption 4.5: A process never schedules a message to itself. Our results generalize (cf. Corollary C.1 in [10]).

Assumption 4.6: No null messages are sent in the Chandy-Misra simulation. Our results generalize (cf. Corollary C.2 in [10]).

The messages executed in a Chandy-Misra simulation (without null messages) are the same as the messages executed in the conservative optimal simulation. These messages are the true messages executed in the Time Warp simulation.

In the remainder of this section, we prove that the Time Warp approach outperforms the Chandy-Misra approach in feedforward system simulation. The strategy is as follows: Since the behavior of a Chandy-Misra simulation must follow *the input waiting rule* and *the output waiting rule* (Definitions 4.2 and 4.3), it is not difficult to determine the time t at which a message m is executed in the Chandy-Misra simulation. Then from some properties of the final execution (Lemma 4.2), we prove that m's final execution in the Time Warp simulation always occurs before time t.

We first introduce some notation, then describe the input waiting rule and the output waiting rule of the Chandy-Misra simulation. (Some notation used in this section, such as $ts(m), \tau_{tw}(m)$, and $\delta(m)$, was defined in Section 3.)

- $\tau_{cm}(m)$: the time when m's (final) execution is started in the Chandy-Misra simulation. Note that all messages executed in the Chandy-Misra simulation are true messages, and they are only executed once.
- d : the message sending delay.
- $\Psi(p)$: the set of true input messages executed by process p in the simulation.

Definition 4.1: In a feedforward network, a process q is said to be a *parent* of a process p iff there is an output channel from q to p; in this case p is said to be a *child* of q.

Definition 4.2 [The input waiting rule]: Consider a Chandy-Misra simulation. Let m be a message executed by process p. Before p executes m, p must receive from each of its input channels a message with timestamp no less than $ts(m)$. Specifically, let $\Phi(q)$ be the set of output messages sent from q to p. Let $x_q(m) \in \Phi(q)$ such that $ts(x_q(m)) \geq ts(m)$ [4], and

$$\text{for all } m' \in \Phi(q), \quad ts(m') \geq ts(m) \Rightarrow ts(m') \geq ts(x_q(m))$$

then the message $x_q(m)$ must be received by p before m is executed.

Example 4.1: Consider Figure 2. The messages m_1 (with timestamp 50), m_2 (with timestamp 20) and m_3 (with timestamp 70) are input messages of p which are sent from s_1, s_2, and s_3 respectively. At time t, p selects m_2 as the next message to be executed. Then $x_{s_1}(m_2) = m_1, x_{s_2}(m_2) = m_2$, and $x_{s_3}(m_2) = m_3$.

[4] The equality holds when $m = x_q(m)$, and $ts(x_q(m)) > ts(m)$ if $x_q(m) \neq m$ (from Assumption 4.3).

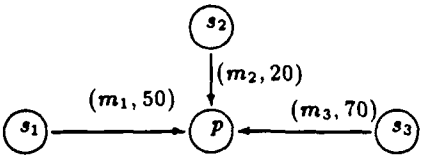


Figure 2: The input waiting rule.

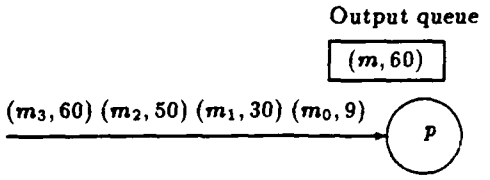


Figure 3: The output waiting rule.

Definition 4.3 [The output waiting rule]: Consider a Chandy-Misra simulation. Let ϵ be the lookahead, and let m be an output message sent from p. If m' is an input message executed by p such that $ts(m') + \epsilon \geq ts(m)$ then m cannot be sent before $\tau_{cm}(m')$. Specifically, let $\Psi(p)$ be the set of input messages executed by p. Let $y(m) \in \Psi(p)$ such that $ts(y(m)) + \epsilon \geq ts(m)$, and

$$\text{for all } m' \in \Psi(p), \quad ts(m') + \epsilon \geq ts(m) \Rightarrow ts(m') \geq ts(y(m))$$

then the message $y(m)$ must be available for execution before p sends m.

Example 4.2: Consider Figure 3. Let m_0, m_1, m_2 and m_3 be input messages of p with timestamps $9, 30, 50$ and 60 respectively. Let m be an output message of p. If the lookahead for p is $\epsilon = 10$, then m cannot be sent before p executes m_2. In other words, $y(m) = m_2$.

Let m be a true message executed by process p. According to Definition 4.2 and Definition 4.3, we proved that (cf. Lemma C.1 in [10]) m is available for execution at time t in the Chandy-Misra simulation, assuming that $s_1, s_2, \ldots, s_l$ are parents of p, where

$$t = \max_{1 \leq i \leq l} [\tau_{cm}(y(x_{s_i}(m))) + d]$$

After time t, m is executed when p finishes the execution of $Prev(m)$ [12]. Thus m is executed by p at the time

$$\begin{aligned} \tau_{cm}(m) \quad = \quad & \max\{\max_{1 \leq i \leq l} [\tau_{cm}(y(x_{s_i}(m))) + d], \\ & \tau_{cm}(Prev(m)) + \delta(Prev(m))\} \end{aligned} \tag{2}$$

in the Chandy-Misra simulation.

Then we prove (cf. Lemma C.2 and C.3 in [10]) that, in the Time Warp simulation, m is available for execution before time t', and is never rolled back after time t' where

$$t' = \max_{1 \leq i \leq l} [\tau_{tw}(y(x_{s_i}(m))) + d]$$

and m is executed at the time [4]

$$\begin{aligned} \tau_{tw}(m) \quad \leq \quad & \max\{\max_{1 \leq i \leq l} [\tau_{tw}(y(x_{s_i}(m))) + d], \\ & \tau_{tw}(Prev(m)) + \delta(Prev(m))\} \end{aligned} \tag{3}$$

Given the following condition

Condition 4.1: $\tau_{tw}(m_i) \leq \tau_{cm}(m_i), \forall m_i \in \Psi(s_i), 1 \leq i \leq l$.

And according to (2) and (3), we prove by induction that

$$\tau_{tw}(m) \leq \tau_{cm}(m), \text{ for all } m \in \Psi(p) \qquad (4)$$

(cf. Lemma C.4 in [10]). Finally the proof is completed by showing that (4) holds when Condition 4.1 is relaxed (cf. Theorem C.1 in [10]). Note that $\tau_{tw}(m) < \tau_{cm}(m)$ in general. The results of Theorem C.1 in [10] can be generalized in two respects: (1) The theorem holds when there are self-scheduled messages in the simulation (cf. Assumption 4.4). The proof is given in Corollary C.1 in [10]. (2) The theorem holds when null messages are sent in the Chandy-Misra simulation with the following assumption:

Assumption 4.7: The overhead of handling null messages is 0, and there is no message transmission contention (i.e., the message sending delay between two processes is not affected by the extra null message transmissions).

We shall elaborate more on (2). The primary purpose of sending null messages in the Chandy-Misra (deadlock avoidance) simulation is to avoid deadlocks. Sending null messages may also improve the performance of the Chandy-Misra simulation. Since deadlocks do not occur in feedforward networks (given Assumptions 3.1 and 3.2), the only reason for sending null messages here is for performance purposes. Assumption 4.7 implies that the existence of null messages never degrades the performance of the Chandy-Misra simulation, and the Chandy-Misra simulation can capture much of the performance gain possible by sending null messages. Under such an optimistic assumption, the Time Warp approach still outperforms the Chandy-Misra approach (cf. Corollary C.2 in [10]).

5 Comparing Time Warp and Chandy-Misra (Systems With No Lookahead)

This section shows that the Time Warp approach outperforms the Chandy-Misra approach in simulating feedback networks with no lookahead. [5] We first note that the Chandy-Misra deadlock avoidance scheme cannot simulate a feedback network with no lookahead. In other words, a deadlock recovery scheme must be used as deadlock resolution for this approach. We make an optimistic assumption for the deadlock recovery scheme:

Assumption 5.1: In the Chandy-Misra deadlock recovery scheme, the overhead of deadlock detection/recovery is 0. In other words, when a local deadlock (cf. Definition 5.2) is formed, it is detected and recovered from immediately.

To show that the Time Warp approach outperforms the Chandy-Misra approach, we observe that every feedback network can be partitioned into *strongly connected components* defined as follows:

Definition 5.1: Let $G = (P, E)$ be a feedback network where P is the set of processes, and E is the set of channels connecting processes. P can be partitioned into equivalence classes P_i, $1 \leq i \leq r$, such that processes p and q are equivalent iff there is a path from p to q, and there is a path from q to p. Let E_i, $1 \leq i \leq r$, be the set of channels connecting the pairs of processes in P_i. The subsystems $G_i = (P_i, E_i)$ are called the *strongly connected components* of G. Even though every process of G is in some P_i, G may have channels not in any E_i.

Definition 5.2: A *local deadlock* caused by a strongly connected component G' is a deadlock that includes G' and its descendants, and G''s ancestors are not involved in this deadlock.

Lemma 5.1 [11]: Consider a deadlock recovery simulation of a feedback network with no lookahead. Let $G = (P, E)$ be the network, and $G' = (P', E')$ be a strongly connected component of G. Let M' be the set of events/messages executed by processes in P' during the simulation. We assume that these events and messages have different timestamp values. Then the execution of every event/message in M' results in a local deadlock.

Lemma 5.1 implies that all messages in a strongly connected component are executed sequentially. When the overhead of deadlock detection/recovery is 0, the deadlock recovery simulation is equivalent to a Chandy-Misra simulation (without deadlock resolution) of a feedforward network G' where every process in G' is a strongly connected component in G.

We claim that the Time Warp approach can simulate G faster than G'. Since a process in G' is a feedback subnetwork G_s in G, it suffices to prove that Time Warp simulation outperforms sequential simulation in simulating G_s. In other words, assigning more processors to G_s improves the performance of Time Warp. The proof is similar to, but simpler than, the one for Theorem A.1. Finally, we have the following conclusion:

Since the Time Warp approach outperforms the Chandy-Misra approach in simulating G' (from Theorem 4.1), it is apparent that the Time Warp approach outperforms the deadlock recovery scheme in simulating G. (Lemma 4.1 and the aforementioned claim.)

Lemma 4.1 assumes that all messages executed in different processes of a strongly connected component have different timestamps. If some messages executed in different processes of a strongly connected component have the same timestamps, then they can be executed in parallel [11]. Since the above situation occurs infrequently, we expect that Time Warp simulation outperforms deadlock recovery in most cases.

6 Conclusions

The Time Warp approach and the Chandy-Misra approach have been studied in the past based on experiments. However, experimental studies cannot generally explain the behavior of the parallel simulation approaches because these studies are limited to specific simulation problems and simulator implementations. For example, many studies reported excellent performance of the Chandy-Misra approach in simulating FCFS tandem systems. However, to conclude the superiority of the Chandy-Misra approach based on these studies is misleading. It is easy to prove that under the assumptions given in this paper, [6] both the Time Warp approach and the Chandy-Misra approach are conservative optimal (Definition 3.1) in simulating FCFS tandem systems [8]. That is, good performance of the parallel simulations is due to the characteristics of FCFS tandem systems, and is independent of the parallel simulation approaches being used. Thus, it is important to study the parallel simulation approaches from a general viewpoint which is not limited to

[5] This section uses Assumptions 2.1-2.3, 4.1, 4.3, and Assumption 5.1 defined below.

[6] Assumptions 2.1-2.3, 4.1, 4.2, and 4.4.

specific simulation problems. This paper represents a step in this direction.

We have shown that Time Warp with aggressive cancellation is not in general conservative optimal (Definition 3.1). We have derived a sufficient condition for Time Warp to be conservative optimal. We have proved that Time Warp approach outperforms the Chandy-Misra approach in every feedforward network simulation. For feedback networks without lookahead, we have shown that the Time Warp outperforms the Chandy-Misra approach in most cases.

The above results, however, are derived under certain assumptions. To avoid misleading conclusions, we examine these assumptions as a final remark of this paper.

Assumption 2.1: Each process in the simulation is assigned to a dedicated processor. This assumption favors the Time Warp approach: Consider the case that the number of processes, k, is larger than the number of processors, n. In the Chandy-Misra simulation, the probability of an idle processor decreases as n decreases. In other words, when n is small, processors are unlikely to waste their power in waiting. This may not be true for the Time Warp simulation. Consider the case that $n < k$. It is possible that a processor executes incorrect computations when there are other correct computations waiting to be performed. Thus, the process scheduling for the Time Warp simulation is much difficult than the scheduling for the Chandy-Misra simulation. Although our results may not hold when Assumption 2.1 is relaxed, experiments [2] indicate that Time Warp outperforms Chandy-Misra simulation for $n < k$.

Assumptions 2.2 and 2.3: Execution of a message m is rolled back immediately upon the arrival of a message m' if m's timestamp is larger than m''s. This is called *message preemption*. Message preemption is considered as a requirement for Time Warp. Also, the Time Warp operational overhead (such as state saving/restoration, global virtual time calculation) is 0. These assumptions imply that Time Warp is supported by special hardware or special data structures. There is no doubt that the overhead of the Time Warp operations is higher than the overhead of the Chandy-Misra operations. This discrepancy is expected to diminish for long computation delays and message sending delays. Otherwise, special data structures [2, 6] and/or hardware [3, 2] are required to support fast Time Warp operations.

Assumption 4.1: The simulation never exhausts memory. Both the Time Warp and the Chandy-Misra approaches take advantage of this assumption. Without this assumption, both approaches may exhaust memory during the simulation even if the amount of memory used in the sequential simulation is bounded. Although Chandy and Misra claimed [12] that in their approach, the amount of memory required by all processors together is bounded and is no more than the amount required in sequential simulation, we have shown [11] that it is not true.

It is obvious that the Time Warp simulation uses more memory than the Chandy-Misra simulation does. In some simulation problems, the Time Warp approach may exhaust memory before the simulation terminates. (Before a message is safely committed, the state modified by the message must be saved in the Time Warp simulation.) Special hardware was proposed [3] to reduce the memory usage in the Time Warp simulation. **Assumption 5.1:** In the Chandy-Misra deadlock recovery scheme, the overhead of deadlock detection/recovery is 0. In other words, when a local deadlock is formed, it is detected and recovered from immediately. This assumption favors the Chandy-Misra approach in three respects: (1) In general, deadlock cannot be detected at the time it is formed. (2) Local deadlocks cannot be easily detected. Generally, only global deadlocks (i.e., the deadlock involves all processes) can be detected. (3) The procedures of deadlock detection/recovery involve many processes in the system, and the overhead is usually very high.

When Assumption 5.1 is relaxed, many studies [2, 11] show that deadlock recovery may not be a good approach for distributed simulation.

Other important issues comparing the Time Warp and the Chandy-Misra approaches (such as network topology, global mechanism, distributed termination, and reliability) can be found in [4] and [12].

7 Acknowledgements

We would like to thank Jean-Loup Baer, David Jefferson, Richard Fujimoto, and Ewan Tempero for extensive comments on this paper.

References

[1] Berry, O. and Jefferson, D. Critical Path Analysis of Distributed Simulation. *Proc. 1985 SCS Multiconference on Distributed Simulation*, pages 57–60, 1985.

[2] Fujimoto, R.M. Time Warp on a Shared Memory Multiprocessor. Technical Report UUCS-88-021a, Computer Science Department, University of Utah, January 1989.

[3] Fujimoto, R.M., Tsai, J.-J. and Gopalakrishnan, G. Design and Performance of Special Purpose Hardware for Time Warp. *Proc. 15th Symp. on Computer Architecture*, 1988.

[4] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[5] Jefferson, D. Parallel Simulation as a Fundamental Issue in Computer Science. Plenary lecture, 1989 SCS Multiconference on Distributed Simulation. 1989.

[6] Jefferson, D. Private communication. 1989.

[7] Kumar, D. Simulating Feedforward Systems Using a Network of Processors. *The 19th Annual Simulation Symposium*, pages 127–144, March 1986.

[8] Lin, Y.-B. and Lazowska, E.D. A Study of Time Warp Rollback Mechanisms. Technical Report 89-09-07, Department of Computer Science, University of Washington, 1989.

[9] Lin, Y.-B. and Lazowska, E.D. Exploiting Lookahead in Parallel Simulation. Technical Report , Department of Computer Science, University of Washington, 1989.

[10] Lin, Y.-B. and Lazowska, E.D. Optimality Considerations for "Time Warp" Parallel Simulation. Technical Report 89-07-05, Department of Computer Science, University of Washington, 1989.

[11] Lin, Y.-B., Lazowska, E.D., and Baer, J.-L. Conservative Parallel Simulation For Systems With No Lookahead. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[12] Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1):39–65, March 1986.

Conservative Parallel Simulation For Systems With No Lookahead Prediction*

Yi-Bing Lin, Edward D. Lazowska and Jean-Loup Baer
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The most popular conservative parallel simulation approach is the Chandy-Misra approach, referred to here as the Chandy-Misra basic scheme (CMB). When a CMB simulation includes a feedback loop (i.e., when a message may "circulate" in a loop of processes), there is the probability that deadlocks will occur. To deal with deadlocks in Chandy-Misra simulations, two modified algorithms, the Chandy-Misra deadlock avoidance (DA) and deadlock recovery (DR) algorithms, have been proposed. The DA algorithm is widely used for simulating systems with lookahead prediction. [1] The DR algorithm has been recognized, up to this point, as the only conservative approach for simulating systems with no lookahead prediction. This paper shows that a better approach for simulating systems with no lookahead prediction is to reconfigure the system such that there is no feedback loop, and use the CMB algorithm to perform the simulation. We identify the overheads of this approach, and devise both an analytical model and a number of simulation experiments to estimate its performance.

1 Introduction

The best approach to parallel simulation is dependent on the characteristics of the problem. One key characteristic is called *lookahead*. In Chandy-Misra parallel simulations, the simulated system can be classified into one of the two categories:

(I) Systems with explicit lookahead, referred to as E systems. In such parallel systems, each feedback loop contains at least one logical process p with some predefined lookahead value d, or minimal service time. [2] That is, if a message arrives at a logical process p at timestamp t, then no output message will be generated by p with timestamp less than $t + d$. In this kind of system, either the *Chandy-Misra deadlock avoidance (DA) algorithm* [Cha79] or the *Chandy-Misra deadlock recovery (DR) algorithm* [Cha81] can be used to address the deadlock problem. Experiments [Ree88] [Fuj88] [Wag89] have shown that the DA algorithm is better than the DR algorithm for E systems.

*This work was supported by the National Science Foundation (Grants CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

[1] In a system with lookahead prediction, each feedback loop of the system contains at least one logical process p with some lookahead value d such that if a message arrives at p at timestamp t, there is no output message scheduled by p with timestamp less than $t + d$.

[2] The service time here is *virtual time* or *timestamp*, not *real (execution) time*.

(II) Systems without explicit lookahead, referred to as $\bar{E}$ systems. In such systems, there is no minimal service time constraint. The DR algorithm has been proposed to solve the deadlock problem in these systems.

Nicol [Nic88] introduced the *future list technique*, which provides an orthogonal classification. The combination of the DA algorithm and the future list technique ($DAFL$ algorithm) is used to exploit lookahead values that are *implicit* in the simulated system. Experiments have shown that the larger the lookahead values, the better the performance. The lookahead values exploited by $DAFL$ algorithm are always larger than the explicitly predefined lookahead. Thus, using implicit lookahead values is better than using explicit lookahead values. Examples of systems with implicit lookahead values are FIFO systems [Nic88], systems with fixed message populations, and preemptive priority systems [Wag89]. Hence, using Nicol's concepts, systems can be classified into those with implicit lookahead (I systems) or those without implicit lookahead ($\bar{I}$ systems). Combining Chandy-Misra's partition and Nicol's partition, the domain of simulated systems can be partitioned into four subdomains:

- Systems with both explicit and implicit lookahead (EI systems).
- Systems with explicit lookahead, but without implicit lookahead ($E\bar{I}$ systems).
- Systems without explicit lookahead, but with implicit lookahead ($\bar{E}I$ systems).
- Systems without either explicit or implicit lookahead ($\bar{E}\bar{I}$ systems).

The E systems are studied in [Fuj88]. A subset of EI systems, the FIFO systems with explicit lookahead value, are studied in [Ree88], [Fuj88], and [Wag89]. The $\bar{E}I$ systems are studied in [Nic88]. Simulations of $\bar{E}I$ systems are seldom reported in the literature. The DR algorithm seems to be the only approach to $\bar{E}I$ system simulations. In this paper, we will show a better solution, to which we refer as the $\bar{E}IB$ algorithm (Definition 3.4 in Section 3), for simulating $\bar{E}I$ systems. This paper is organized as follows. Section 2 describes the Chandy-Misra simulation algorithm. Section 3 presents a general solution (the $\bar{E}IB$ approach) for simulating $\bar{E}I$ systems. Section 4 discusses the overhead of Chandy-Misra simulation. Section 5 analyzes the overhead of $\bar{E}IB$ simulation based on simulation experiments. Finally, Section 6 provides the conclusions.

2 The Chandy-Misra Algorithm

In Chandy-Misra simulations [Cha81], the topology of the logical system must be defined before simulation and cannot be changed during simulation. *Logical processes* are connected by *directed channels*. If a channel is directed from

process p_1 to process p_2 then it is called an *input (output)* channel of p_2 (p_1). There are three types of processes: a *source* process never receives messages from other processes; a *delay* process both receives and sends messages from/to other processes; a *sink* process never sends messages to other processes. In each process, there are three queues: an *input* queue for input messages; a *local* queue for events generated by the process itself; and an *output* queue for output messages. [3] The input (output) queue is always empty in a source (sink) process.

The *Chandy-Misra basic* (*CMB*) algorithm works as follows. Initially, some events are scheduled in the local event queues of processes (a non-source process may have an empty event queue initially, but each source process must have a non-empty event queue). Then each process p repeats the following sequence of steps until termination:

Step 1: p waits until at least one message from each input channel is in the input message queue. Let t be the minimum timestamp value of all messages in the input queue, and m be the corresponding message.

Step 2: p handles all events with timestamp smaller than t in the local event queue and the output message queue. Messages from the output message queue are sent to other processes. The execution of events from the local event queue may create new events. These events are inserted in either the local event queue or the output message queue.

Step 3: m is executed. If it is an *end-of-simulation* event, then another end-of-simulation message is sent to all output channels, and the process terminates. Otherwise, the local clock of p advances to t, and the process returns to Step 1.

For a source process, Step 1 is skipped; events from the local event queue and the output message queue are executed one by one in Step 2; and the event m in Step 3 is scheduled by p itself. For a sink process, no output message is sent in Step 2.

This algorithm is *pessimistic* or *conservative* because in Step 1, p must wait for one message from each input channel (this is called the *input waiting rule*), and in Step 2, an output message must wait (to be sent) until p's local clock advances to its timestamp value (this is called the *output waiting rule*). Deadlocks may occur in *CMB* simulations [Cha81]. To solve the deadlock problem, the *DA* algorithm and *DR* algorithm have been proposed. Here, we only give the description of the *DR* algorithm: The simulation (the basic scheme) is run until a system deadlock occurs. The deadlock is detected by a special process. Then the deadlock recovery method initiates a computation whereby the various logical processes can advance their local clock values. The basic idea for resuming the computation comes from sequential simulation: Each process posts into the event queue the time of its next output assuming it receives no inputs. The event with the smallest timestamp in the event queue is guaranteed to be the next event in the physical system.

3 The $\bar{E}\bar{I}B$ Approach

To our knowledge the *DR* algorithm is the only general conservative approach for simulating $\bar{E}I$ feedback systems. (Deadlocks would not occur in feedforward $\bar{E}I$ systems so other techniques would work.) In this section, we show that a better approach for simulating $\bar{E}I$ feedback systems is to restructure the network topology from a feedback network to a feedforward network, and use the *CMB* algorithm (i.e., the basic Chandy-Misra scheme without deadlock resolution) to perform the simulations. We only present the results. The proofs can be found in [Lin89a].

Definition 3.1: Let $G = (P, E)$ be a network, [4] where P is the set of processes, and E is the set of channels among processes. Let $p_1, p_2 \in P$. We say p_2 is *reachable* from p_1 (denoted as $p_1 \stackrel{*}{\rightarrow} p_2$) iff there is a path from p_1 to p_2. If p_1 and p_2 are the only two processes in the path then we denote the path as $p_1 \rightarrow p_2$.

Lemma 3.1: Let $G = (P, E)$ be a network. Consider two processes $p, q \in P$, and $p \stackrel{*}{\rightarrow} q$. If p is involved in a deadlock in the *DR* simulation, then q is also involved in the deadlock.

Definition 3.2: Let $G = (P, E)$ be a network. P can be partitioned into equivalence classes P_i, $1 \leq i \leq r$, such that processes p and q are equivalent iff $p \stackrel{*}{\rightarrow} q$ and $q \stackrel{*}{\rightarrow} p$. Let E_i, $1 \leq i \leq r$, be the set of channels connecting the pairs of processes in P_i. The subsystems $G_i = (P_i, E_i)$ are called the *strongly connected components* of G. Even though every process of G is in some P_i, G may have channels not in any E_i.

Corollary 3.1: Let $G_i = (P_i, E_i)$ be a strongly connected component of a network G, and $p_1, p_2 \in P_i$, then p_1 is involved in a deadlock in the *DR* simulation iff p_2 is involved in the deadlock.

Definition 3.3: Let $G_1, G_2, ..., G_r$ be the strongly connected components of a network G, and $p \stackrel{*}{\rightarrow} q$, where $p \in G_i, q \in G_j$, $1 \leq i, j \leq r$. Then G_i is called an *ancestor* of G_j, and G_j is called a *descendant* of G_i.

Lemma 3.2: Consider the *DR* simulation of an $\bar{E}I$ system. Let $G = (P, E)$ be the network, and $G' = (P', E')$ be a strongly connected component of G. Let M' be the set of events/messages executed by processes in P' during the simulation. We assume that these events and messages have different timestamp values. Then the execution of every event/message in M' results in a *partial deadlock*. [5]

Corollary 3.2: Consider the *DR* simulation of an $\bar{E}I$ system. Let G be the network, and $G_1, G_2, ..., G_r$ be the strongly connected components of G. If events/messages executed by $P_i, 1 \leq i \leq r$, have different timestamp values, then the speedup of parallel simulation can be no more than r, the number of strongly connected components in the network.

Corollary 3.2 shows that the performance of the deadlock recovery method is very poor when r is small. It's likely that sequential simulation greatly outperforms this method in many network structures (for example, $\bar{E}I$ systems with torus networks in [Fuj88] or central server queueing models in [Ree88]). In such a case, the Time Warp approach [Jef85] should be considered.

Definition 3.4 A *CMB* simulation of an $\bar{E}I$ system with network G is done as follows: Let $G_1, G_2, ..., G_r$ be the strongly connected components of G. Construct a new network G' such that $G_i, 1 \leq i \leq r$, are the processes of G', and the channels in G' are $\bar{E} - (\bigcup_{i=1}^{r} E_i)$. Then simulate G', a

[3] In practice, the local event queue and output message queue can be combined.

[4] The network of a simulation system specifies the input/output relationship among the processes in the system.

[5] A partial deadlock caused by a strongly connected component G' is a deadlock that includes G' and its descendants, and G''s ancestors are not involved in this deadlock.

feedforward network, via the *CMB* algorithm. The above simulation is referred to as the $\bar{E}IB$ simulation.

In addition to the network transformation, we need to implement a sequential simulation model for every strongly connected component G_i. The sequential model is constructed as follows: (i) The input channels of G_i are those from G_i's direct ancestors. (ii) The output channels of G_i are those from G_i to its direct descendants. (iii) The messages sent between processes in G_i become local events to G_i. (iv) All local event queue of processes in G_i are merged into one local event queue for G_i.

The implementation of the above sequential model is not difficult. It can be automatically done by a pre-processor to the simulator without user's knowledge.

Theorem 3.1: For an $\bar{E}I$ system, the $\bar{E}IB$ simulation is always faster than the *DR* simulation, if events/messages executed by every strongly connected component have different timestamp values.

We made a pessimistic assumption in Lemma 3.2 and Theorem 3.1, namely: The events/messages executed by every strongly connected component of a network have different timestamp values. In [Lin89a] we showed that this assumption has a negligible or secondary effect on the result of Lemma 3.2 and Theorem 3.1. The above analysis leads to the conclusion that we should use the $\bar{E}IB$ algorithm for $\bar{E}I$ system simulation. That is, we should pre-process the network by considering each strongly connected component in the old network as a "super" process in the new network. Thus, the new network is feedforward, and deadlocks never occur during simulation. The pre-processing of a network can be done very efficiently:

Theorem 3.2: The strongly connected components of G can be found in $O(\max(n,l))$ time on an $n-$processes, $l-$channels network G.

4 The Overhead of $\bar{E}IB$ Approach

The overhead of Chandy-Misra algorithms is primarily due to (1) the input waiting rule, (2) the output waiting rule, and (3) deadlock. We first define the input and output waiting overheads:

Definition 4.1: Suppose there are k input channels for a process p, and $m_i, 1 \leq i \leq k$ is the message from channel i that p waits to receive. Let t_i be the real time p waits before it receives m_i. Note that $t_i = 0$ if m_i is already in channel i when p requests an input message from channel i. Then the time that p waits before it can execute the next input message is $t = \max_{1 \leq i \leq k} t_i$, (Step 1 of *CMB* algorithm). Let $m_n, 1 \leq n \leq k$, be the next input message to be processed. If we have an optimal protocol that knows m_n is the next message to be executed in advance, then the time that p waits is t_n. In other words, $t - t_n \geq 0$ is the extra time that p waits due to the Chandy-Misra protocol. Thus, $t - t_n$ is defined to be the *input waiting overhead* of the *CMB* algorithm.

Definition 4.2: In the *CMB* algorithm, an output message m_o is sent when its timestamp, t_o, is equal to the value of p's local clock (Step 2 of *CMB* algorithm). Let $t_{r,o}$ be the real time when p advances its local clock to t_o. If we have an optimal protocol which knows that after real time $t \leq t_{r,o}$, p will not schedule any output message with timestamp small than t_o, then $t_{r,o} - t$ is the extra time that p waits to send m_o due to the Chandy-Misra protocol. Thus, $t_{r,o} - t$ is defined to be the *output waiting overhead* of the *CMB* algorithm.

Chandy-Misra algorithms are efficient in feedforward system simulations since the latter are deadlock-free. The performance of FIFO feedforward system simulation via Chandy-Misra algorithms is studied in [Kum86] and [Ree88]. In FIFO systems, there is no overhead due to the output waiting rule because every output message can be sent to an output channel at its generation time without violating the output waiting rule. This is not true for $\bar{E}I$ systems, where an output message is held in an output queue until it is safe to send that message.

If the input/output waiting overheads are reduced to 0, then the simulation is optimal, i.e., the concurrency of the simulation is fully exploited. Thus, it is important to reduce the input/output waiting overheads of the Chandy-Misra simulation. One way to do so is to take advantage of the *event knowledge* implied in the simulated system. An example is FIFO system simulations mentioned above; another example for trace-driven simulations is shown in [Lin89b]. Exploiting the event knowledge of simulations is beyond the scope of this paper. Instead, we are interested in finding the relationship between $\bar{E}IB$ simulations and optimal simulations. In the remaining part of this paper, $\bar{E}I$ feedforward system simulations are assumed. We first define *optimal simulations*, *input optimized simulations*, and *output optimized simulations*. Let P be an $\bar{E}IB$ simulation of a simulation application.

Definition 4.3: The *optimal simulation* of P is a modified $\bar{E}IB$ simulation in which some mechanism is used such that both input and output waiting overheads are reduced to 0.

Definition 4.4: The *input optimized simulation* of P is a modified $\bar{E}IB$ simulation in which some mechanism is used such that input waiting overhead is reduced to 0.

Definition 4.5: The *output optimized simulation* of P is a modified $\bar{E}IB$ simulation in which some mechanism is used such that output waiting overhead is reduced to 0.

In the above definitions, optimized simulations are not related to the simulation application, but are related to a specific $\bar{E}IB$ simulation of that application. It is clear that by network reconfiguration and event/message redecomposition of the application, the efficiency of $\bar{E}IB$ simulation may be different.

4.1 The Effect of the Input Waiting Rule

In this section, we consider the effect of the input waiting rule (Definition 4.1). The Chandy-Misra algorithm implies that after a process p handles the first input message, there is at most one empty input channel (i.e., if p has n input channels, then at least $n-1$ channels are non-empty) for p at any moment. Thus, the input waiting overhead of the Chandy-Misra algorithm is due to the unnecessary waiting for this empty channel if the next message to be handled is not from this channel (this waiting is called *artificial blocking* [Wag89]). Let θ_{op} be the real time that a process waits before it handles the next input message in the optimal simulation, and θ_{cm} be the waiting time in the Chandy-Misra simulation. The input waiting overhead can be expressed as $\theta = \theta_{cm}/\theta_{op}$. θ is determined by three factors:

(I) The average number of messages circulating in the system. As the number of messages in the system increases, it is less likely that an input channel is empty. In that case, we expect that both θ_{op} and θ_{cm} decrease. (cf. Section 5.2.)

(II) The number of input channels of a process. Since there are at most one empty input channel of a process at

Execution order	1	2	3	4	5	6	7	8	9
Input messages (arrived)	$(m_1, 10)$	$(m_2, 20)$	$(m_3, 30)$	$(m_4, 40)$	$(m_5, 50)$	$(m_6, 60)$	$(m_7, 70)$	$(m_8, 80)$	$(m_9, 90)$
Output messages (scheduled)	$(m'_1, 35)$	$(m'_2, 25)$	$(m'_3, 65)$	$(m'_4, 55)$	$(m'_5, 75)$	$(m'_6, 70)$	$(m'_7, 95)$	$(m'_8, 100)$	$(m'_9, 105)$
Output messages (sent) ($\bar{E}\bar{I}CMB$)			$(m'_2, 25)$	$(m'_1, 35)$		$(m'_4, 55)$	$(m'_3, 65)$ $(m'_6, 70)$	$(m'_5, 75)$	
Output messages (sent) (Optimal)		$(m'_2, 25)$ $(m'_1, 35)$		$(m'_4, 55)$ $(m'_3, 65)$		$(m'_6, 70)$ $(m'_5, 75)$			

Figure 1: An example of the output message sending in the optimal simulation and in the $\bar{E}\bar{I}B$ simulation.

any moment, one may suspect that θ should not be significantly affected by the number of input channels. In fact, the number of input channels N has a dominant effect on θ for the following two reasons:

- When N increases, it is likely that the next message to be handled is selected from a non-empty channel in the optimal simulation. Thus, θ_{op} decreases, and θ increases. Note that if $N = 1$ for all processes in the system, we have $\theta = 1$. That is, there is no input waiting overhead.
- Consider two systems with the same number of circulating messages, and assume that the numbers of processes in both systems are the same. If the number of input channels in System 1 is larger than that in System 2, then we expect that the number of messages queued on an input channel of System 1 is smaller than that of System 2. And the processes in System 1 are more likely to see an empty input channel than that in System 2. That is, we expect that $\theta_{sys.1} > \theta_{sys.2}$.

(III) The relationship between the timestamp of an input message and the real time when it arrives at the process p. If a system has the property that messages with smaller timestamps arrive at p earlier, then it is likely that a process p is artificially blocked before it handles a message in the Chandy-Misra simulation. On the other hand, p is likely to handle messages without any delay in the optimal simulation. In that case, large θ is expected.

4.2 The Effect of The Output Waiting Rule

The effect of the output waiting rule (Definition 4.2) can be demonstrated in the example shown in Figure 1 which shows the behavior of output messages in a given process. Figure 1 assumes that every arrival message m_i results in the scheduling of an output message m'_i. In the optimal simulation, the "future history" of the input message arrival and the output message scheduling is known in advance. For example, when m_2 is processed, m'_2 is generated, and is sent to an output channel immediately, because the process knows that no output messages scheduled later than m'_2 will have timestamps smaller than that of m'_2. In the case of m'_1, it cannot be sent at the time it is generated, because the process knows that m'_2 will be sent earlier than m'_1 is. In the $\bar{E}\bar{I}B$ simulation, an output message with timestamp t cannot be be sent unless the input message to be handled at that moment is with a timestamp larger than t. For example, m'_1 cannot be sent before m_4's arrival.

Let the time interval between the time m'_1 is generated and the time m'_1 is sent be $\delta_{opt,1}$ ($\delta_{cm,1}$) for the optimal simulation (the $\bar{E}\bar{I}B$ simulation) then, in the above example,

$\delta opt, 1$ = the execution time of m_2 - the execution time of m_1

$\delta cm, 1$ = the execution time of m_4 - the execution time of m_1

and the output waiting overhead can be expressed as $\delta = \delta_{cm}/\delta_{op}$. If the time between the executions of two consecutive messages is a constant, then $\delta_1 = \delta_{cm,1}/\delta_{opt,1} = 3$. That is, the time that m'_1 has to wait to be sent in the $\bar{E}\bar{I}B$ simulation is three times that in the optimal simulation. This extra waiting overhead is due to the output waiting rule. If we assume that the time intervals between the executions of two consecutive messages for the optimal simulation and the $\bar{E}\bar{I}B$ simulation have the same distribution, then the output waiting overhead is determined by the timestamps of the input messages, and the timestamps of the scheduled output messages.

Let the timestamp interval between two consecutive input messages be a random variable F, and the timestamp interval between an input message and its output message be a random variable S. In [Lin89a], we show that if both F and S are distributed exponentially with parameters λ_f and λ_s respectively, then the ratio of the waiting time in the $\bar{E}\bar{I}B$ simulation to that in the optimal simulation, δ, is

$$\delta = \frac{\delta_{cm}}{\delta_{op}} \simeq \frac{(2+\rho)(1+\rho)}{\rho}, \text{ where } \rho = \frac{\lambda_f}{\lambda_s} \qquad (1)$$

(cf. Theorem A.2, [Lin89a]). If S is uniformly distributed on the interval $[0, a]$, F is exponentially distributed with parameter λ_f, and $\rho = \lambda_f a$, then

$$\delta \simeq \rho \frac{\sum_{n=1}^{\infty} nF(n-1)}{\sum_{n=1}^{\infty} nF(n+1)}, \text{ where } F(n) = 1 - \sum_{j=1}^{n} \frac{(\rho)^j}{j!} e^{-\rho} \qquad (2)$$

(cf. Theorem A.4, [Lin89a]). Although (1) and (2) are quite different in their form, they have similar behavior. The parameter ρ is interpreted as the traffic load of the system. When $\rho < 1$, δ decreases as ρ increases, and when $\rho > 1$, the result is reversed. This phenomenon is explained as follows: Let T be the expected real time interval between the arrivals of two consecutive input messages. Let $m_1, m_2, \ldots$ be the sequence of input messages with timestamps $t_1, t_2, \ldots\ldots$, and $m'_1, m'_2, \ldots$ be the sequence of output messages with timestamps $t'_1, t'_2, \ldots\ldots$ We consider the δ for m'_1: If $\rho \to 0$, then we expect that $t'_1 - t_1 << t_2 - t_1$. In



Figure 2: The network configurations.

the optimal simulation, m_1' is sent at the time it is generated. In the $\hat{E}\hat{I}B$ simulation, m_1' is sent at the time m_2 arrives. Thus, $\delta = \frac{\delta_{cm}}{\delta_{op}} = \frac{1 \cdot T}{0 \cdot T} = \infty$. If $\rho \rightarrow \infty$, we expect that $t_1' - t_1 > t_n - t_1$, where $n \rightarrow \infty$. In the $\hat{E}\hat{I}B$ simulation, $\delta_{cm} \rightarrow \infty$. In the optimal simulation, $\delta_{op} = nT$, if $t_1' \leq t_i'$ then $i \geq n$, and n is expected to be finite. Thus, $\delta \rightarrow \infty$. The minimal value of δ occurs when $\rho \simeq 1$. In that case, both δ_{op} and δ_{cm} are finite. We conclude that when $\rho < 1$, output optimization is more important in lightly loaded systems [6] than in heavily loaded systems. When $\rho > 1$, we have the converse result.

5 Performance Result

In this section, the results of experiments concerning $\hat{E}\hat{I}B$ simulations are described. We consider four parallel simulation protocols: The optimal algorithm (protocol *opt*), the input optimized algorithm (protocol *in*), the output optimized algorithm (protocol *out*), and $\hat{E}\hat{I}B$ algorithm (protocol *cm*). (The optimized algorithms are defined in Definitions 4.3, 4.4, and 4.5, and are not achievable in practice.)

Critical path analysis [Ber85] allows us to implement the four protocols without difficulty. The idea is as follows: To measure an optimal simulation, we first run its sequential counterpart, and take the event trace of the execution. The trace is transformed into a directed graph whose edges are weighted with real times representing either computation or message sending delay. The weights are chosen to represent the timing characteristics of the hardware on which the parallel simulation is expected to run. Finally, the longest weighted path is found. The weight of that path then represents the execution time of the optimal simulation. The execution times of input optimized, output optimized, and $\hat{E}\hat{I}B$ simulations can be obtained form the same weighted, directed graph with slight modifications to the longest-weighted-path finding algorithm. Thus, in essence, we are doing a trace-driven analysis of the four parallel simulation algorithms.

5.1 Parameters and Measurements

The input parameters considered in our experiments follow those described in [Fuj88]:

Network topology: Two feedforward networks of high degree of branching are simulated (see Figure 2). Both networks have a bottleneck. Networks 2 is *balanced* in the sense that, when all output channels have the same routing probability, each process receives the same amount of workload from each of its parents, and produces the same amount of workload to each of its children.

Message population: In [Ree88], [Fuj88], and [Wag89], the simulated systems are *closed*. That is, messages are considered as jobs or costumers, and the message populations are fixed – messages are never created or destroyed. In our experiments, the simulated systems are *open* – messages are created at source processes, and destroyed at sink processes. No messages are created or destroyed in delay processes. Thus, the message population is not fixed during simulation.

Timestamp increment: This is the amount by which the timestamp of a message is increased as it travels through a process. In our experiments, timestamp increments with uniform distributions and exponential distributions are considered. Only the results with exponential distribution are presented. The results with uniform distribution are similar to those with exponential distributions. We consider two types of timestamp increment functions: In a *heavily loaded* system, each process has exponential timestamp increments with the same parameter λ_s. In a *lightly loaded* system with n processes, processes are numbered from 0 to $n-1$ such that a child is always assigned a number larger than that of its parents. For a process numbered i, its timestamp increment function is exponentially distributed with parameter $(n-i)\lambda_s$. Let $\lambda_{f,l}(\lambda_{f,h})$ be the message arrival rate of a process in lightly loaded (heavily loaded) systems, [7] then $\frac{\lambda_{f,h}}{\lambda_{s,h}} \geq \frac{\lambda_{f,l}}{\lambda_{s,l}}$.

Routing probability: After a message is received, it is forwarded to a randomly selected output channel. Our experiments follow Fujimoto's assumption [Fuj88] that each output channel is equally likely to be selected.

Computation delay and message sending delay: The computation delay represents the real time required to process a message. In our experiments, the computation delay for each process is exponentially distributed with the same parameter λ. The message sending delay represents the real elapsed time to forward a message from one process to another. The message sending delay is a constant. We vary the message sending delay from 0.1λ to 0.9λ to see the effect of tightly coupled systems and loosely coupled systems on the simulations.

Output measurements are:

- t_s : The sequential simulation time.
- t_x : The parallel simulation time of a protocol x.

[6] A system is lightly loaded (heavily loaded) if its ρ value is small (large).

[7] The arrival rate of a process p is determined by timestamp increment functions of p's ancestors.

- S_x : The speedup of a protocol x which is defined as $S_x = t_s/t_x$. $S_{x,h}(S_{x,l})$ is the speedup for heavily loaded (lightly loaded) systems.
- $I_x = \frac{S_x - S_{cm}}{S_{cm}} = \frac{t_x - t_{cm}}{t_{cm}}$: The improvement of a protocol x over protocol cm. $I_{x,h}(I_{x,l})$ is the improvement for heavily loaded (lightly loaded) systems.

5.2 Discussion

We discuss the experimental results in the following aspects:

The speedups of the optimal simulations ($S_{opt,h}$, $S_{opt,l}$) and the $\bar{E}\bar{I}B$ simulations ($S_{cm,h}$, $S_{cm,l}$) relative to the sequential simulations: (Recall that the l and h subscripts refer to light and heavy loads.) We observe that (i) Increasing the message delay time decreases both S_{op} and S_{cm}. (ii) When the workload of simulated systems increases, both S_{op} and S_{cm} increase. The increase of workload has the effect of increasing the number of messages circulating in the system, and thus increasing the potential for concurrency.

The improvements of the optimal simulations over the $\bar{E}\bar{I}B$ simulations ($I_{opt,l}$, $I_{opt,h}$): We observe that $I_{opt,h} < I_{opt,l}$. This is due to the effect of both input and output optimizations. We'll elaborate more on I_{in} and I_{out} later.

The improvements of the input optimizations ($I_{in,l}$, $I_{in,h}$) and the output optimizations ($I_{out,l}$, $I_{out,h}$) over the $\bar{E}\bar{I}B$ simulations: We observe that (i) $I_{in,l} > I_{in,h}$. In heavily loaded system, the probability that a process sees an empty input queue is low. According to the discussion in Section 4.1, we have $I_{in,l} > I_{in,h}$ (ii) $I_{out,l} > I_{out,h}$. Note that $\rho < 1$. This phenomenon is with the conclusion drawn in Section 4.2. It is interesting to point out that when $\rho > 1$, $I_{out,l} < I_{out,h}$ in our simulation experiments (the results are not shown in this paper). (iii) $I_{in} > I_{out}$. This is because the processes have many input channels.

6 Conclusion

In this paper, we concentrate on Chandy-Misra simulations of $\bar{E}\bar{I}$ systems: systems that have neither implicit nor explicit lookahead. We show that the deadlock recovery algorithm is not efficient for $\bar{E}\bar{I}$ system simulations, and a better approach is to reconfigure the network topology of an $\bar{E}\bar{I}$ system such that there is no feedback loop. In other words, we divide the network into a set of *feedback clusters*. Each feedback cluster is considered as a sequential model, and is simulated sequentially by a process. All feedback clusters are simulated in parallel, and are synchronized via the CMB algorithm. This is called the $\bar{E}\bar{I}B$ algorithm.

The overheads associated with $\bar{E}\bar{I}B$ algorithm are due to the input waiting rule and the output waiting rule. We show how the system load ρ (in term of virtual timestamps) affects the output waiting overhead. We conduct experiments to see the effects of the input and output optimizations on the $\bar{E}\bar{I}B$ simulations. The experiments indicate the following conclusions: (i) Optimizations are more important in lightly loaded system simulations than that in heavily loaded system simulations. (ii) For systems with a high branching degree network, the input optimization is more important than the output optimization.

The $\bar{E}\bar{I}B$ approach effectively eliminates the deadlock detect and recovery overhead in simulating an $\bar{E}\bar{I}$ system. However, it degenerates a parallel simulation to a sequential simulation if every process in the network is involved in a big feedback loop. In such a case, no Chandy-Misra-like approach can be used to exploit the parallelism in the system. Fortunately, a practical simulation problem is usually not a "pure" $\bar{E}\bar{I}$ system. Most of the processes in the system are FIFO, and only a small number of loops contain "pure" $\bar{E}\bar{I}$ processes. Thus, we can apply the network reconfiguration technique to remove these $\bar{E}\bar{I}$ loops, making it feasible for the deadlock avoidance algorithm to simulate the new network. Good performance can be expected for this combined approach.

References

[Ber85] Berry, O. and Jefferson, D. Critical Path Analysis of Distributed Simulation. *Proc. 1985 SCS Multiconference on Distributed Simulation*, pages 57–60, 1985.

[Cha79] Chandy, K.M. and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[Cha81] Chandy, K.M. and Misra J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[Fuj88] Fujimoto, R.M. Performance Measurements of Distributed Simulation Strategies. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 14–20, 1988.

[Jef85] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Kum86] Kumar, D. Simulating Feedforward Systems Using a Network of Processors. *The 19th Annual Simulation Symposium*, pages 127–144, March 1986.

[Lin89a] Lin, Y.-B., Lazowska, E.D., and Baer, J.-L. Conservative Parallel Simulation For Systems With No Lookahead. Technical Report 89-07-07, Department of Computer Science, University of Washington, July 1989.

[Lin89b] Lin, Y.-B., Lazowska, E.D., and Jean-Loup Baer. Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis. To appear in *Progress In Simulation*, edited by G. W. Zobrist, 1989.

[Nic88] Nicol, D.M. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137, 1988.

[Ree88] Reed, D.A., and Malony, A. Parallel Discrete Event Simulation: The Chandy-Misra Approach. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 8–13, 1988.

[Wag89] Wagner, D.B., Lazowska, E.D. and Bershad, B. Techniques for Efficient Shared-Memory Parallel Simulation. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 29–37, 1989.

Exploiting Lookahead in Parallel Simulation [1]

Yi-Bing Lin and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The most effective approach to parallel simulation depends on the characteristics of the system being simulated. One key characteristic is called *lookahead*. In Chandy-Misra deadlock avoidance simulation, lookahead must exist in the simulated system in order to avoid deadlock. Fujimoto recognized that, independent of its effect on deadlock, lookahead has a significant effect on the performance (speedup) of a simulation.

In Fujimoto's study, *explicit* lookahead is assumed, i.e., the lookahead is known before the simulation begins, and it does not change during the simulation. Another kind of lookahead, called *implicit* lookahead, was introduced by Nicol for simulating FCFS stochastic queueing systems; implicit lookahead can be exploited to yield performance benefits even when explicit lookahead does not exist.

In this paper we show the feasibility of implicit lookahead for non-FCFS systems. We propose several lookahead exploiting techniques for round-robin (RR) system simulations. We design an algorithm that generates lookahead in $O(1)$ time. Both analytical models and experiments are constructed to evaluate these techniques. We also evaluate a lookahead technique for preemptive priority (PP) systems using an analytical model.

The performance metric for these techniques is the *lookahead ratio*, which is correlated with other performance measures of more direct interest, such as speedup. Our analyses show that exploiting implicit lookahead can significantly improve the lookahead ratios of RR and PP system simulations.

[1]This work was supported in part by the National Science Foundation (Grants CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

1 Introduction

The most effective approach to parallel simulation depends on the characteristics of the system being simulated. One key characteristic is called *lookahead.* Lookahead is the ability of a process to predict its future behavior. For example, in a FCFS queueing network simulation, lookahead exists if each job requires a minimum service time δ.[2] Knowledge that each future job requires at least δ service allows a process to predict that a job that arrives immediately will not depart for at least δ time units. This knowledge can be used to reduce the synchronization requirements of other processes in simulation.

In Chandy-Misra deadlock avoidance (DA) simulation, lookahead must exist in the simulated system to ensure the absence of deadlock [7]. Fujimoto recognized that, independent of its effect on deadlock, lookahead has significant effect on the performance (speedup) of a DA simulation [2, 3]. In Fujimoto's study, *explicit* lookahead is assumed, i.e., the lookahead is known before the simulation, and it does not change during the simulation. Another kind of lookahead, called *implicit* lookahead, was introduced by Nicol [8]; exploiting implicit lookahead can yield performance benefits even when explicit lookahead does not exist.

The concept of exploiting implicit lookahead in a FCFS stochastic queueing network simulation involves pre-sampling a process's service time and routing distribution, thereby gaining information about messages that have yet to arrive at the process, which can be used in the lookahead calculation. The statistical integrity of the simulation is preserved by keeping these samples in a queue called the *future list.* Then, when a message is received by the process, its service time and destination are taken from the head of the future list [8, 9]. A modified DA algorithm based on the future list technique (we refer this modified algorithm as the $DAFL$ algorithm) works as follows: Even if no input message is available for processing at local clock t_c, it is possible to calculate lookahead as follows: the service time distribution can be pre-sampled to determine the service time t of the next arrival message. The next output message will, therefore, have a timestamp no less than $t + t_c$.

The $DAFL$ algorithm does not in general lend itself to non-FCFS systems. In a non-FCFS system, it is possible that the next output message will have a timestamp less than $t + t_c$. Thus, the future list technique fails. Nonetheless, by modifying the $DAFL$ algorithm, some non-FCFS systems can in fact be simulated:

- Systems with a fixed message population: there are N messages in the system, and after the initialization, no message is created or destroyed in the simulation.
- Round-robin (RR) systems with a minimum service time.
- Preemptive priority (PP) systems with no minimum service time [9].

Exploiting implicit lookahead for these systems is the subject of this paper. For systems with fixed message populations, this is straightforward: N service times are pre-sampled in a process p, one

[2]Throughout this paper, the term "time" means "simulation time" or "timestamp". The reader should not be confused with the execution time of the simulation.

for each message; the minimum of the N service times is the lookahead for p. We do not consider this case further.

We propose several techniques for exploiting implicit lookahead in RR systems and evaluate the *lookahead ratio* (cf. Section 2) of these techniques. If the minimum service time is much smaller than the expected service time, it is necessary to exploit implicit lookahead for performance purposes in RR system simulation. Then we evaluate the lookahead ratio of a previously proposed technique for PP systems. Chandy-Misra deadlock avoidance simulations for RR and PP systems are important because:

1. The RR and PP scheduling strategies are the most important scheduling strategies in computer system applications, and
2. Other distributed simulation approaches, such as Time Warp [4] and Chandy-Misra deadlock recovery approaches, may not be able to simulate these systems efficiently. The Time Warp approach may not be efficient because of the huge number of rollbacks involved. The inefficiency of the Chandy-Misra deadlock recovery approach for simulating non-FCFS systems has been shown in [6].

This paper is organized as follows: Section 2 defines lookahead ratio, the performance metric used for lookahead exploiting techniques. Section 3 presents techniques for RR systems and analyzes these techniques by both analytical models and experiments. Section 4 presents and analyzes, via a probabilistic model, a technique for PP systems. Finally, Section 5 states conclusions and proposes future research.

2 Lookahead Ratio

Speedup is the most interesting performance measure for parallel simulation. For several reasons, though, studies of performance improvement techniques for parallel simulation typically do not use speedup directly as a performance measure. For one thing, it is difficult to evaluate speedup analytically. Also, speedup is affected by many factors (such as network topology, timestamp increment function, message population, computation delay, message sending delay, etc. [3]). In using speedup to gain insight into the effect of one factor (e.g., lookahead), one may be misled by the interactions of other factors.

Since lookahead has significant effect on speedup, it is natural to study lookahead and relate it to speedup. It is not appropriate to directly relate lookahead to speedup, however. Consider a queueing network simulation using the DA algorithm where a logical process represents a "server", and messages moving among the processes represent "jobs". Suppose the scheduling strategy is FCFS, and the service time for a job is s [3] with a restriction that s is no less than some value δ. In other words, δ is the explicit lookahead or the minimum service time. Consider a server p where no job is in service at the local clock t_c. If no job arrives at p at timestamp $t > t_c$, then no job departs in the timestamp interval $[t, t + \delta]$. Thus, if a null message arrives at p with timestamp t,

[3] It can be a random variable with arbitrary distribution.

then we can send null messages with timestamps $t+\delta$ to p's output channels. It is obvious that the performance of the simulation becomes better as $\delta \to s$. Hence, it is meaningless to compare two systems by the absolute lookahead. (For example, consider a system with lookahead δ_1 and service time s_1, and a system with lookahead δ_2 and service time s_2. When $\delta_2 > \delta_1$, the former may still have better performance than the latter, given that $\delta_1 \approx s_1$ and $\delta_2 << s_2$.)

Thus, it is important to find another output measure which is correlated to speedup, and clearly shows the effect of lookahead. An appropriate measure, called *lookahead ratio*, was first introduced by Fujimoto [2, 3]: Consider a "cause" event with timestamp t_c that leads to an "effect" event with timestamp t_e. Then the lookahead ratio is defined as $\frac{\delta}{t_e - t_c}$, where δ is the lookahead value [4]. Thus, in this definition, the lookahead is *normalized* by $t_e - t_c$. The advantage of Fujimoto's definition is that it is easy to manipulate. (In the queueing network simulation, only the distribution of the service time is required to normalize lookahead.) It works well for FCFS systems with fixed distributions of message arrival interval and service time.

However, Fujimoto's definition has the following problem: Consider a FCFS server p with service time s and lookahead δ. Suppose that there is no job in service at timestamp t, and the next job will arrive at p at timestamp $t' > t$. Then no job will depart in the timestamp interval $[t, t'+s]$. In other words, the "true" next departure time is determined by s and $t'-t$ where $t'-t$ is a random variable referred to as the *message arrival interval*. If we simply normalize lookahead by s, then the effect of the message arrival interval is ignored. For two systems with the same service time distribution and lookahead but different message arrival intervals, it is clear that the system with a shorter message arrival interval is likely to have better performance. In that case, Fujimoto's definition does not appropriately reflect performance. A better definition for lookahead ratio is

$$\Theta \triangleq \frac{E[\Delta]}{E[T]} \tag{1}$$

where $E[\Delta]$ is the expected value of lookahead δ, and $E[T]$ is the expected value of $t'-t+s$. Expected values are used in (1) because both $t'-t$ and s are random variables, and δ is a function of $t'-t$ and s (if it is implicit; cf. Section 3 and 4).

In this paper, the performance of a lookahead exploiting technique is indicated by lookahead ratio defined by (1). For FCFS systems it is known that the larger the lookahead ratio, the better the performance [3, 2]. We expect the same for RR and PP systems. Thus, the technique yielding larger lookahead ratio is expected to have better performance (speedup).

3 Round-Robin Systems

In computer systems where a single processor provides service to a number of jobs running concurrently, some variation of the *round-robin* (RR) scheduling strategy is usually employed. The processor offers service in quanta of fixed size Q. The job at the head of the queue is given a quantum of service. If that job completes before the quantum elapses, it departs immediately;

[4] Fujimoto's original definition for lookahead ratio is $(t_e - t_c)/\delta$, and a low lookahead ratio corresponding to a high degree of lookahead.

otherwise it rejoins the queue at the tail. An RR system with $Q \rightarrow 0$ can be approximated by a processor-sharing system, and easily analyzed analytically. However, it is difficult to evaluate a RR system with large Q by an analytical model. In such a case, simulation may be a good approach.

In this section, we describe RR system simulation based on modified *DAFL* algorithms. We assume that the service demand of a job is no less than some minimum value which is equal to the quantum Q. Although our results generalize, we assume that each job starts/resumes its service at the beginning of a time quantum.

3.1 Lookahead When n Jobs Are in Service

Since an RR system does not preserve the FCFS property, Nicol's future list technique cannot be used to yield the lookahead value. New techniques need to be developed. In this subsection we describe a technique that yields lookahead, assuming that there are $n > 0$ jobs in service.

Lemma 3.1: Consider a process p in an RR system with time quantum Q. Let S be the set of jobs in service [5] at timestamp t, and let $|S| = n$. Then for every job J which departs in the timestamp interval $[t, t + nQ]$, we have $J \in S$.

Proof: Since each job at process p must consume at least one quantum before it departs, and because there are n jobs in service at timestamp t, no new job that arrives at p after timestamp t can receive its first service quantum in the timestamp interval $[t, t + nQ]$. Hence, if any job departs in that timestamp interval, it must be in the set S. ∎

Let J_D be the last job that departs in $[t, t + nQ]$, and let t_D be J_D's timestamp. (By convention, $t_D = t$ if no job departs in $[t, t + nQ]$.) If a message arrives at p at timestamp t, then we can simulate p's behavior up to t_D (i.e., we can advance p's local clock to t_D). Furthermore, we know that no job departs in $[t_D, t + nQ]$. Note that at timestamp t, J_D is already determined. The above discussion suggests a lookahead exploiting technique for RR systems:

Technique 1: If the history of a process in the timestamp interval $[0, t]$ is known, then we can simulate p up to timestamp t_D, and predict that no departure occurs in $[t_D, t + nQ]$.

Note that:

- Technique 1 is useful only when $n > 0$. We will elaborate more on the case when $n = 0$ in the next subsection.
- In the *DA* or *DAFL* algorithms, the value of a process p's local clock cannot exceed the timestamp of any input message p handles. On the other hand, in RR system simulations based on Technique 1, it is possible to advance p's local clock to $t_D \geq t$, when an input message with timestamp t is handled.
- The lookahead exploiting procedure based on Technique 1 does not take advantage of the future list technique. This shows that the future list technique is not the only tool we can use to exploit implicit lookahead.

[5] In the remainder of this paper, "a set of jobs in service" means the jobs on the ready queue as well as the job actually in service.

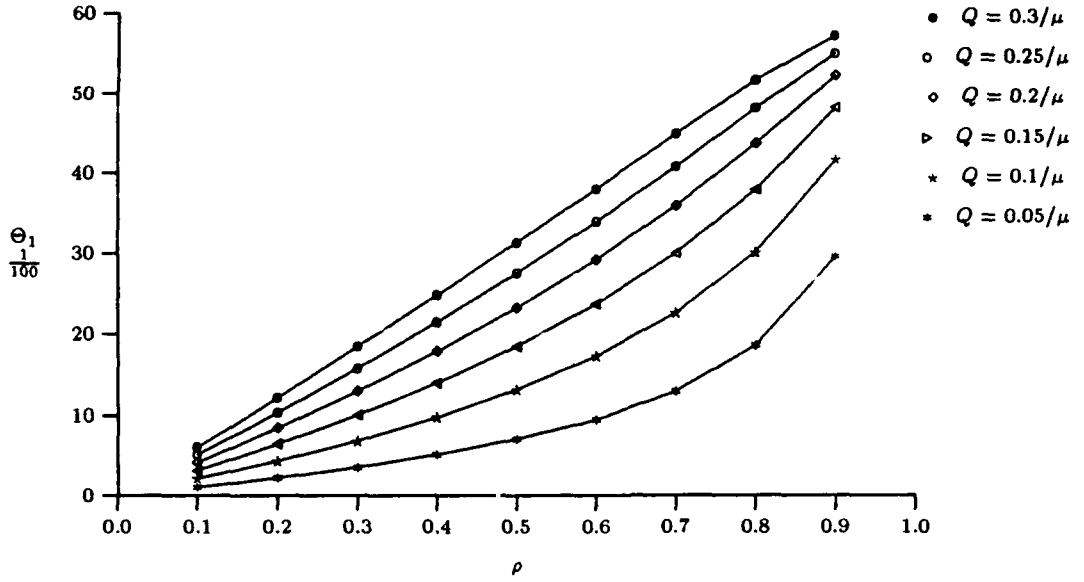


Figure 1: The lookahead ratio for Technique 1.

In the remainder of this subsection, we evaluate the performance (lookahead ratio) of Technique 1. To derive the lookahead ratio of Technique 1, we assume that the quantum size Q is small. Let $E[\Delta]$ be the average lookahead prediction at a particular time t, and $E[T]$ be the average time interval between t and the next departure. Then from (1), the lookahead ratio is defined as $\Theta = E[\Delta]/E[T]$. We assume that jobs arrive at a process p in a Poisson stream with rate λ, and that the jobs' required service times are always larger than Q and have some general distribution function with mean $1/\mu$. The load of the system, ρ, is defined as λ/μ. Theorem A.1 in Appendix A shows that the lookahead ratio Θ yielded by Technique 1 is

$$\Theta = \frac{1-(1-\lambda Q)^{n+2}}{2-\lambda Q-(1-\lambda Q)^{n+1}}, \quad \text{where } n = \frac{\rho}{1-\rho} \tag{2}$$

Based on (2), Figure 1 shows how the lookahead ratio Θ is affected by Q, λ, and μ:

- Assume that the load ρ is reasonably small, i.e., $\rho \not\to 1$. Figure 1 shows that if $\lambda Q \to 0$, then $\Theta \to 0$(cf. Figure 1: If $Q = 0.1/\mu$, and $\rho \leq 0.6$, then $\Theta < 0.1$). This phenomenon can be explained as follows: $E[\Delta]$ is bounded by the product of n (the number of jobs in service) and Q (the service quantum). Since $\rho \not\to 1$, n is a constant less than ∞. Thus, if $Q \to 0$, we have $E[\Delta] \to 0$, and thus $\Theta \to 0$ (note that $E[T]$ is not affected by Q). Since we assume that Q is small in an RR system, the lookahead ratio is not good for a lightly loaded system (i.e., a system with small ρ).

- If $\rho \to 1$, then $n \to \infty$. The system is nearly saturated and

$$\frac{1}{2} \leq \Theta = \frac{1}{2-\lambda Q} \leq 1$$

  Since Q is small, $\Theta \to 1/2$. This phenomenon is explained as follows: For a small Q, the departure process can be considered as a Markov process. Thus, both the *forward* and *backward*

recurrence times are exponentially distributed in the limit for $n \rightarrow \infty$ [1]. Corresponding to this symmetrical property, the expected lookahead value is half of the expected departure interval.

Figure 1 also shows that Θ increases when ρ increases. In other words, the performance of this technique relies on frequent arrival of jobs (Θ_1 when $\rho = 0.9$ is about 5 to 30 times larger than Θ_1 when $\rho = 0.1$). We conclude that for a heavily loaded system (i.e., a system with large ρ), our prediction is reasonably good even when Q is very small.

3.2 Lookahead When No Job Is In Service

When there is no job in service at a process p, Technique 1 only yields a small lookahead value for an arrival message (either a job or a null message). This situation may occur very often if ρ is small. Even if ρ is large so that this situation occurs infrequently, the influence may still be crucial. In this subsection, we describe two techniques that yield reasonably large lookahead values when no job is in service.

Lemma 3.2 [Technique 2]: Let Q be the service time quantum of a process p. Let there be no job in service at timestamp t, and let $s_i > Q, i = 0, 1, 2, ..,$ be the service time for the i^{th} arriving job in $[t, \infty]$. Then there is no departure in timestamp interval $[t, t + \delta]$ where

$$\delta = \min_{0 \le i \le j}(iQ + s_i), \quad \text{and } j = \left\lfloor \frac{s_0}{Q} \right\rfloor \tag{3}$$

Proof: Suppose J_0 arrives at timestamp t and no job is in service. If $j = 0$, then $s_0 < Q$, and J_0 is always the next one to depart. If $j > 0$, then the i^{th} arrival job $J_i, 1 \le i \le j$ cannot depart earlier than $t + iQ + s_i$ because the first i jobs $(J_0, ..., J_{i-1})$ receive their first service time quantum before J_i does. Note that we do not need to consider the effect of $J_k, k > j$, because it always departs later than $t + kQ + s_k > s_0$. Thus,

$$\delta = \min_{0 \le i \le j}(iQ + s_i) \quad \blacksquare$$

Another technique that yields larger lookahead values than Technique 2 is described in the following Lemma.

Lemma 3.3 [Technique 3]: Let Q be the service time quantum of a process p. Let there be no job in service at timestamp t, and let $s_i, i = 0, 1, 2, ..,$ be the service time for the i^{th} arrival job in $[t, \infty]$. Then there is no departure in timestamp interval $[t, t + \delta]$ where

$$\delta = \min_{0 \le i \le j}[iQ + (i + 1)s_i], \quad \text{and } j = \left\lceil \frac{s_0}{2Q} \right\rceil \tag{4}$$

Proof: A job $J_i, i \ge 0$, cannot start its service before $t + iQ$ (from the proof of Lemma 3.2).
(a) If $J_i, 0 \le i \le j$, is the first job to depart in $[t, \infty]$, then the timestamp interval T_i between when it starts and finishes service is longer than $(i + 1)s_i$ (because there are at least $i + 1$ jobs in service in the timestamp interval T_i).

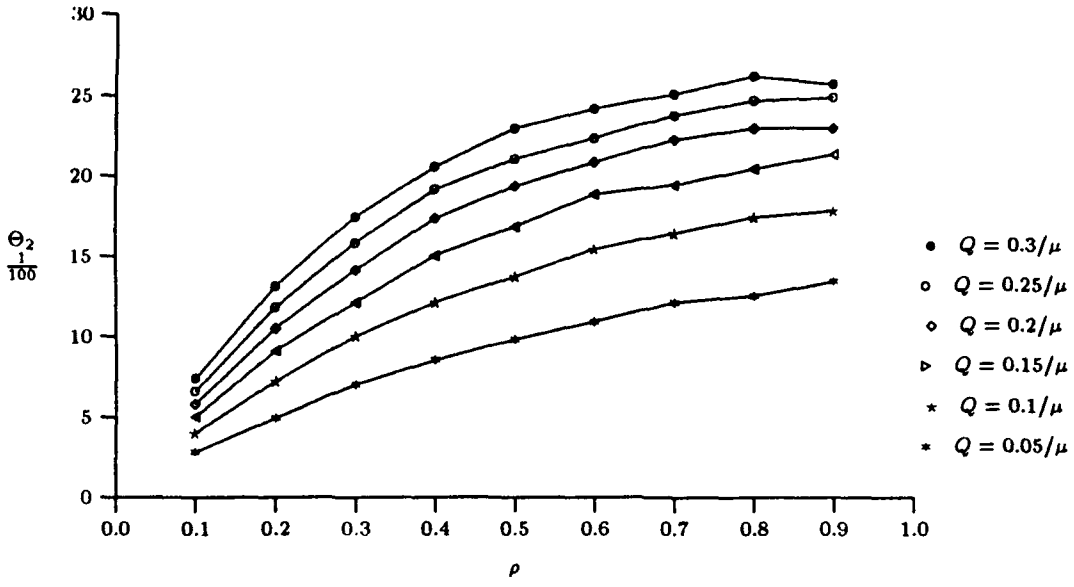


Figure 2: The lookahead ratio for Technique 2.

(b) If $J_i, i > j$, is the first job to depart in $[t, \infty]$, then let $r_i = s_i - Q > 0$. The timestamp interval T_i between when J_i starts and finishes service is

$$T_i \geq (i+1)s_i = (i+1)Q + (i+1)r_i \geq (i+1)Q$$

and J_i cannot depart earlier than the timestamp

$$t + iQ + T_i \geq t + iQ + (i+1)Q > t + 2iQ > t + 2jQ \geq t + s_0$$

From (a) and (b), we conclude that the first departure cannot occur earlier then $t + \delta$, where

$$\delta = \min_{0 \leq i \leq j} [iQ + (i+1)s_i] \quad \blacksquare$$

When Q approaches 0, we have

$$\lim_{Q \to 0} j = \infty, \quad \text{and} \quad \lim_{Q \to 0} \delta = \min_{0 \leq i \leq \infty} [(i+1)s_i] \tag{5}$$

Equation (5) is consistent with (4) in [10]. In that case, the scheduling strategy is processor-sharing.

Experiments were conducted to observe the lookahead ratio yielded by Technique 2 and Technique 3. In these experiments, we assume that the service times are i.i.d. random variables exponentially distributed with parameter $\mu = 1$, and the time interval between two arriving jobs also is an exponentially distributed random variable with parameter λ. Figure 2 and Figure 3 show the results for Technique 2 and Technique 3 respectively. Note that $E[\Delta]$ is determined by Q and μ, and $E[T]$ is determined by μ and λ. Let Θ_2 and δ_2 (Θ_3 and δ_3) be the lookahead ratio and lookahead value yielded by Technique 2 (Technique 3); then we have the following observations:

- Suppose μ and λ are fixed (i.e., Θ is linearly dependent on $E[\Delta]$). When Q increases, Θ_2 approaches Θ_3 (i.e., $E[\Delta_2]$ approaches $E[\Delta_3]$). This is explained as follows: δ_2 (cf. (3)) and

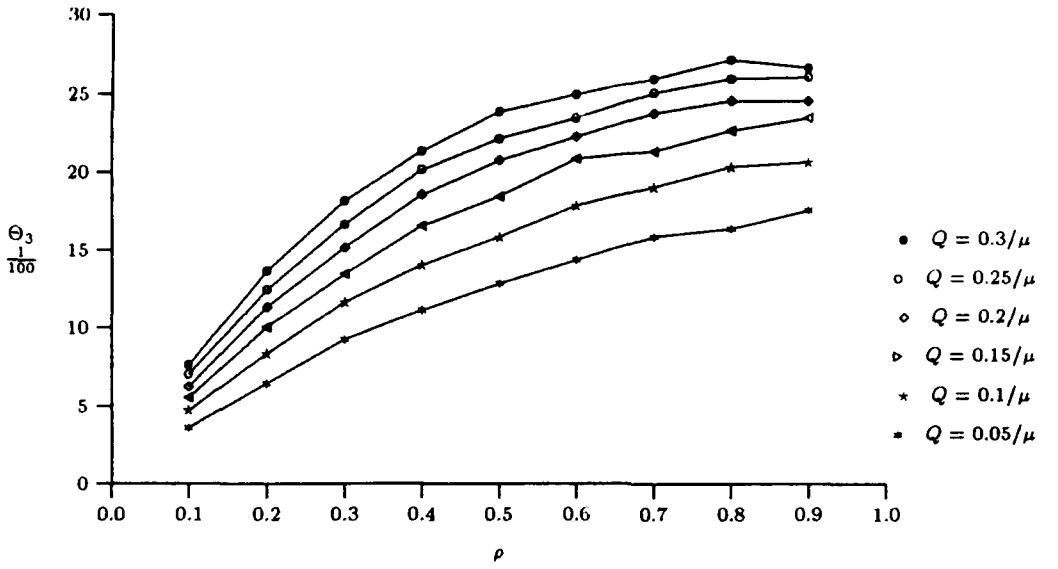


Figure 3: The lookahead ratio for Technique 3.

δ_3 (cf. (4)) are determined by two components in the above equations: the Q-term (which is iQ for both δ_2 and δ_3), and the s_i-term (which is s_i for δ_2, and $(i+1)s_i$ for δ_3). When Q is large, the Q-term dominates the s_i-term, and $\delta_2 \simeq \delta_3$. When Q is small, the s_i-term dominates the Q-term, and $\delta_2 < \delta_3$.

- Suppose Q and μ are fixed (i.e., $E[\Delta]$ is fixed, and $\Theta \approx 1/E[T]$). When there is no job in service, the expected residual arrival interval of the next arrival job J is $1/\lambda$ (the random observer property of a Poisson process). Let $E[S]$ be the elapsed time between the arrival of J and the first departure. Then $E[T]$ can be expressed as

$$E[T] = \frac{1}{\lambda} + E[S(\lambda)]$$

For fixed μ, $E[S]$ increases as λ increases. Simulation results show that $E[S]$ increases more slowly than λ does. This is because when J arrives at the process, it is the only job in service, and a job will depart before the queue length becomes too long (note that $\lambda < \mu$). Thus, $E[T]$ increases as λ decreases.

If jobs arrive in a Poisson stream with rate λ, and the service time has an exponential distribution with mean $\frac{1}{\mu} >> Q$, then Theorem A.2 in Appendix A shows that

$$\Theta_3 \simeq -\frac{\rho}{(1+\rho)\ln(2\mu Q)} \tag{6}$$

The curve in Figure 4 (a) shows Θ_3 derived by (6). The experiments (the curve in Figure 4 (b)) show that (6) is a good analytical model for Θ_3 when Q is small. Note that Θ_3 derived by (6) is larger than the true Θ_3 because (6) under estimates $E[T]$ (cf. the proof of Theorem A.2).

It is important to weigh the costs of generating lookahead against the performance benefits. In the previous section, we showed that Technique 3 yields larger lookahead values than does Technique 2.

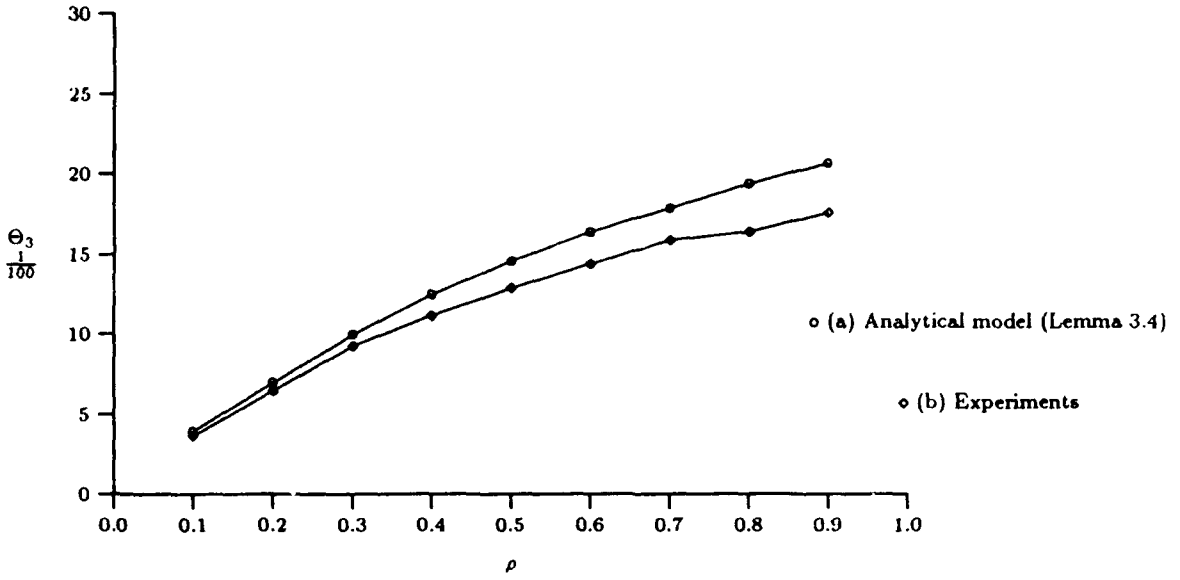


Figure 4: The lookahead ratio for Technique 3 ($Q = 0.05/\mu$). (a) Analytical model (Lemma 3.4). (b) Experiments.

However, the time complexity of generating a lookahead value by using Technique 3 is $O(E[S_0]/Q)$, where $E[S_0]$ is the expected service demand of a job. If $Q << E[S_0]$, then the generation overhead may not be acceptable. On the other hand, an $O(1)$ time complexity algorithm for Technique 2 exists (Theorem B.2 in Appendix B). Thus, Technique 2 may be a better choice although the lookahead ratio yielded by Technique 2 is a little worse than that yielded by Technique 3.

3.3 Combining Technique 1 and Technique 2

Figure 1 shows that the lookahead values yielded by Technique 1 are small for $\rho < 0.5$. Fortunately, it is possible to combine Technique 1 and Technique 2 (or Technique 3) to generate larger lookahead values for $0.2 < \rho < 0.8$. Corollary 3.1 and Theorem 3.1 show the feasibility of this combination:

Corollary 3.1: Suppose that S is the set of jobs in service at time t, and $r_{J'}$ is the residual service time for job $J' \in S$. Let $n = |S|$ be the size of S, and

$$r = \min_{J' \in S} r_{J'}, \quad \text{and } j = \left\lfloor \frac{nr}{Q} \right\rfloor$$

Then no job departs in the time interval $[t, t + t']$, where

$$t' = \min \left\{ nr, \min_{1 \le i \le j} [(n+i)s_i + iQ] \right\}$$

Proof: Directly from Lemma 3.2. ∎

From Lemma 3.1 and Corollary 3.1, we have

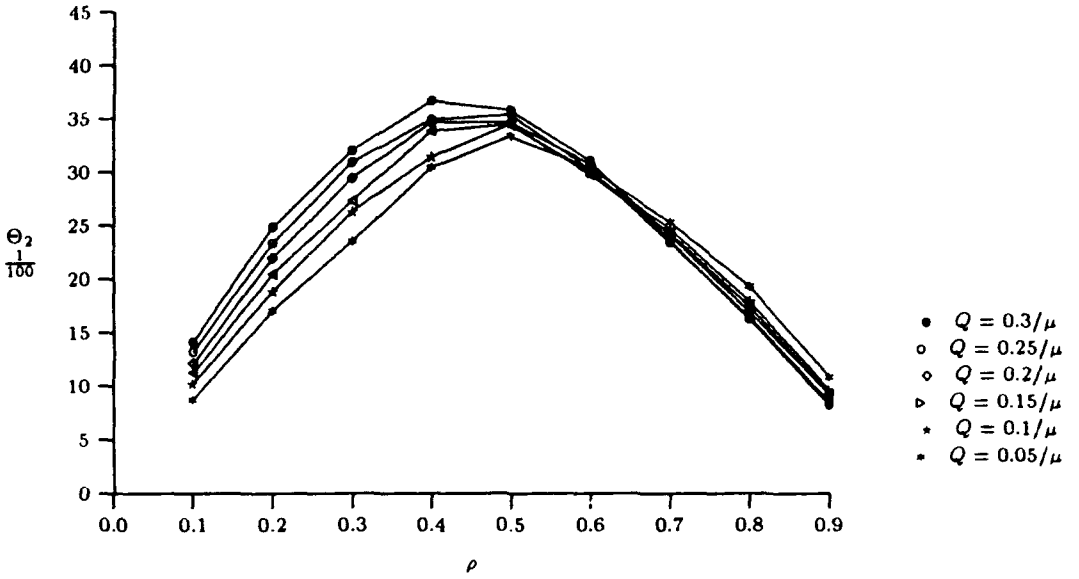


Figure 5: The lookahead ratio for Technique 2, after Technique 1 is applied.

Theorem 3.1 [Technique 4]: At any timestamp t, we can find a lookahead value δ_1 by using Technique 1, then apply Technique 2 or Technique 3 at timestamp $t + \delta_1$ to find another lookahead value δ_2 (as shown in Corollary 3.1). Thus, the total lookahead for timestamp t is $\delta_1 + \delta_2$.

Figure 5 shows the lookahead ratios yielded via Technique 2 after Technique 1 is applied. For small ρ, the lookahead ratios are similar to those shown in Figure 2 because it is likely that for small ρ no job is in service at most times. The lookahead ratio increases as ρ increases up to 0.5. For $\rho > 0.5$, the lookahead ratio decreases when ρ increases because a large portion of the lookahead value has been exploited by Technique 1, and there is not much room left for Technique 2.

Figure 6 shows the lookahead ratios yielded by Technique 4. Note that the poor performance of Technique 1 in the range $0.2 \leq \rho \leq 0.6$ is significantly improved by Technique 4. The lookahead ratios for $\rho < 0.1$ are still small. This inefficiency is generic to all lookahead exploiting algorithms based on the future list technique.

4 Preemptive Priority Systems

In a system with preemptive priority scheduling, there are N types of jobs. A job of type i has higher priority than a job of type j if $1 \leq i < j \leq N$. Let J_i be a job of type i. A server p schedules jobs FCFS within each type, satisfying the condition that all queued jobs with higher priorities are served before any jobs with lower priorities. In addition, a job J_i will be preempted from service if a job $J_j, j < i$, arrives while J_i is receiving service. When J_i eventually regains the server, its service can be continued from the point of interruption (this is called *preemptive-resume*), or restarted from the beginning (this is called *preemptive-restart*). For our purpose, it does not matter how the server deals with an interrupted job.

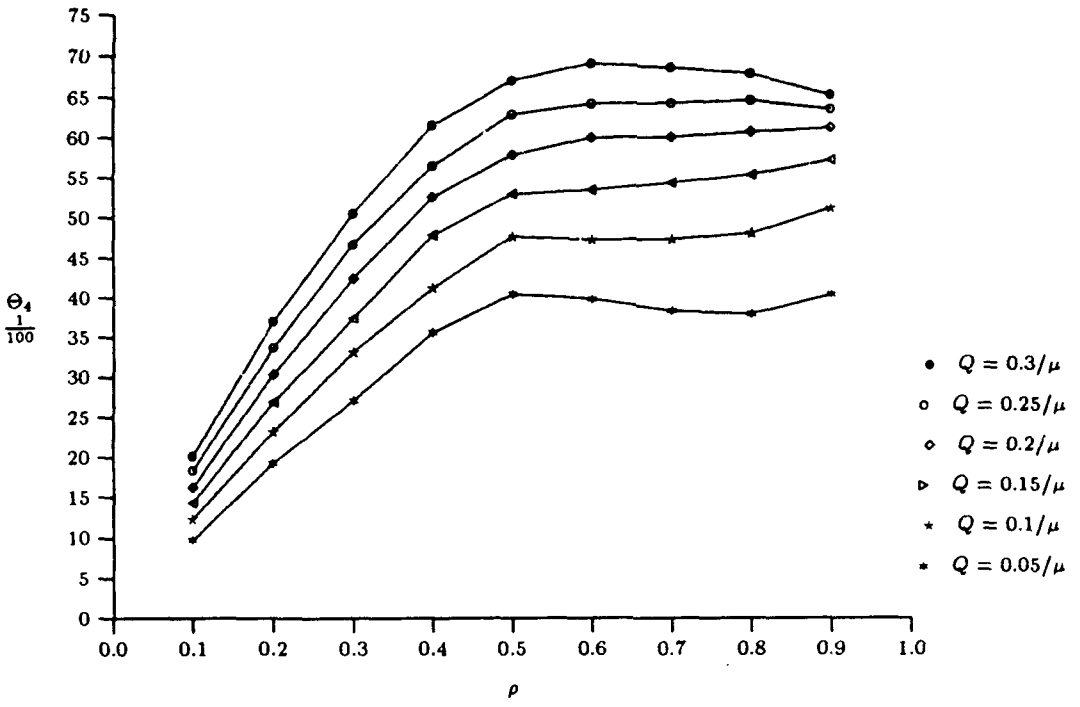


Figure 6: The lookahead ratio for Technique 4 (a combination of Technique 1 and Technique 2).

In this section, we describe an algorithm for preemptive priority system simulation[6], then discuss the lookahead ratio of this algorithm.

4.1 Algorithm

The preemptive priority simulation (PP) algorithm is a modified DA algorithm which uses the future list technique to estimate the lookahead values. The details of the algorithm are as follows.

Algorithm PP: In each server p, there are N future lists, one for each job type. Let t_c be the local clock of p. Let δ_i be the estimated lookahead value when an i type job J_i arrives. Suppose a job J_i arrives at p at time t. (A null message is considered as a type N job.) Note that $t \geq t_c$. Then the simulator simulates p in the time interval $[t_c, t]$ as follows:

Step 1. Send all jobs finished before t to output channels.

Step 2. Calculate the lookahead value as follows:

Case I. The server is idle or J_i has higher priority over the job receiving service. Then

$$\delta_i = \min_{1 \leq k \leq i} (s_k) \tag{7}$$

where s_k is the service time for job type k taken from the future list k.

[6]The idea of this algorithm was first presented in [9].

Case II. The job receiving service is J_j, and $j \leq i$. Suppose the remaining service time of J_j at time t is r_j, then

$$\delta_i = \begin{cases} \min\left[\min_{1 \leq k < j}(s_k), r_j\right] & , j > 1 \\ r_j & , j = 1 \end{cases} \tag{8}$$

In both cases, the null messages sent to output channels are with timestamp $t + \delta_i$.

Step 3. Associate J_i with service time s_i. For Case I, J_i is receiving service. For Case II, J_i is inserted into the priority queue.

Step 4. Wait for the next job arrival, and repeat the above procedure.

The correctness of the lookahead value generation in the PP algorithm is shown in [9].

4.2 Lookahead Ratio for Preemptive Priority Systems

In the remainder of this section, we analyze the lookahead ratio for the PP algorithm. We first derive the expected lookahead value $E[\Delta]$, then derive the expected departure time $E[T]$, and finally $\Theta = E[\Delta]/E[T]$.

We assume that type i jobs arrive in an independent Poisson stream with rate λ_i, and the service times are exponentially distributed with mean $1/\mu_i$. Thus, $\rho_j = \lambda_j/\mu_j$ is the probability that a job of type j is receiving service, or the load for type j. We further assume that the system is not saturated, i.e., $\rho_1 + \rho_2 + ... + \rho_N < 1$. Then Theorem C.1 in Appendix C gives the expected lookahead value $E[\Delta_i]$ given that an arrival job is of type i:

$$E[\Delta_i] = \frac{1 - \sum_{k=1}^{i} \rho_i}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i-1} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k} \tag{9}$$

and the expected lookahead value $E[\Delta]$ can be expressed as (cf. Corollary C.1 in Appendix C):

$$E[\Delta] = \sum_{i=1}^{N} \frac{\lambda_i}{\sum_{N=1}^{N} \lambda_n} \left(\frac{1 - \sum_{k=1}^{i} \rho_k}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k} \right) \tag{10}$$

If we further assume that $\lambda_1 = \lambda_2 = ... = \lambda_N = \lambda$, $\mu_1 = \mu_2 = ... = \mu_N = \mu$, and let $\rho = \lambda/\mu$, then

$$E[\Delta] = \frac{1}{N\mu} \sum_{i=1}^{N} \left(\frac{1}{i} + \sum_{j=2}^{i} \frac{\rho}{j} \right) \tag{11}$$

Based on (11), Figure 7 shows the effect of μ, ρ (or λ), and N on $E[\Delta]$:

- The mean service time $1/\mu$ has positive effect on $E[\Delta]$ in two ways: The main effect is that when μ decreases, the lookahead value yielded from (7) or (8) increases. μ also has indirect effect on $E[\Delta]$ via ρ which is described next.

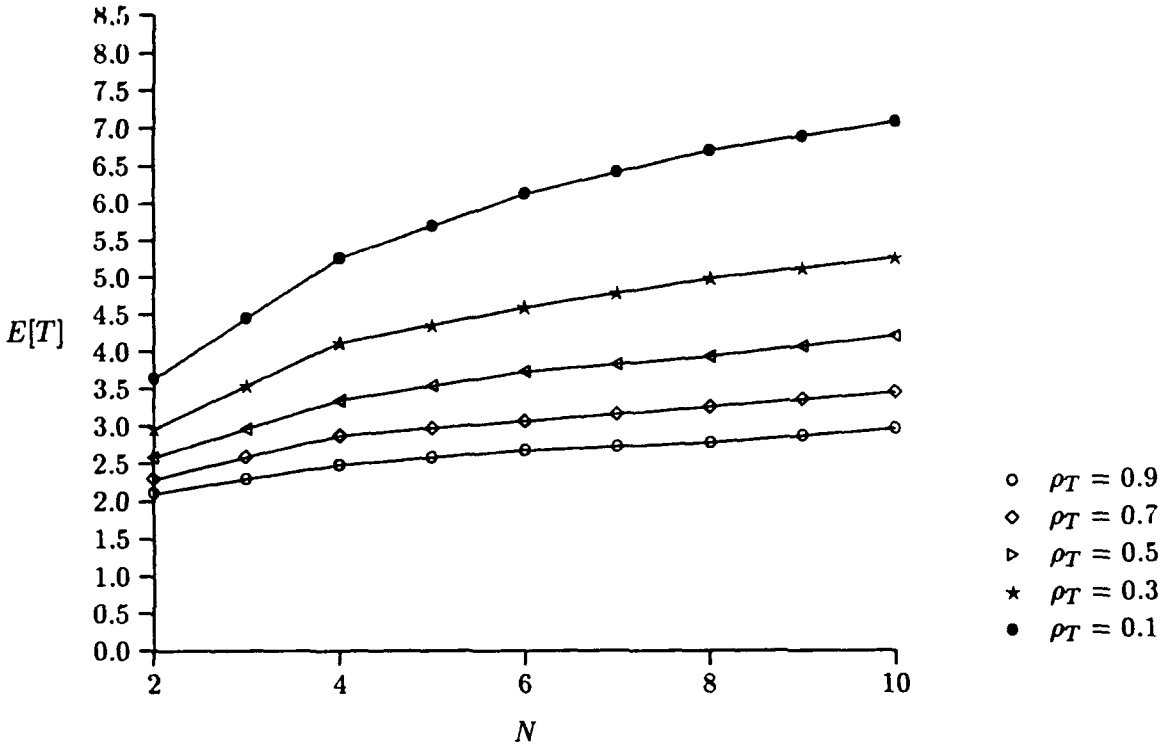


Figure 7: Expected lookahead value for the PP systems ($\mu = 0.5, \rho_T = \rho_1 + \rho_2 + ... + \rho_N = N\rho$).

- The probability that a type j job is receiving service, $\rho_j = \rho = \lambda/\mu$, has positive effect on $E[\Delta]$. When λ increases and/or μ decreases, ρ increases, and $E[\Delta]$ increases. For large ρ, an arrival job is likely to see a higher priority job receiving service, and the lookahead value estimated from (8) is higher (because less random variables are involved).
- Effect of heterogeneous arrival rates: It is obvious (or it can be observed in (9)) that $E[\Delta_i] \geq E[\Delta_j], 1 \leq j \leq i \leq N$. Thus, from (10) we expect that the larger the arrival rates of higher priority jobs, the better the estimated lookahead values.
- Effect of N. When $N = 1$, we have a FCFS system, and $E[\Delta] = 1/\mu$ which is equal to the expected lookahead value yielded in [8]. As N increases, $E[\Delta]$ decreases (because the number of random variables involved in (7) or (8) increases).

The expected time $E[T]$ between the arrival/resumption of an arbitrary job and the next departure is derived in Appendix C.2 (Theorem C.2, Corollary C.3 and C.4). The relationship among $E[T]$, ρ, and N is shown in Figure 8, assuming that $\lambda_1 = \lambda_2 = ... = \lambda_N = \lambda$, $\mu_1 = \mu_2 = ... = \mu_N = 0.5$, and $\rho_T = \rho_1 + \rho_2 + ... + \rho_N$. We first make the following claim:

Claim 4.1: Let $\lambda_1 = \lambda_2 = ... = \lambda_N = \lambda$, $\mu_1 = \mu_2 = ... = \mu_N = 0.5$, and $\rho_T = \rho_1 + \rho_2 + ... + \rho_N$. Suppose J_i gains service at time t. Let $t + D(J_i)$ be the time of the next departure. Then $E[D(J_i)] \geq E[D(J_j)], j < i$.

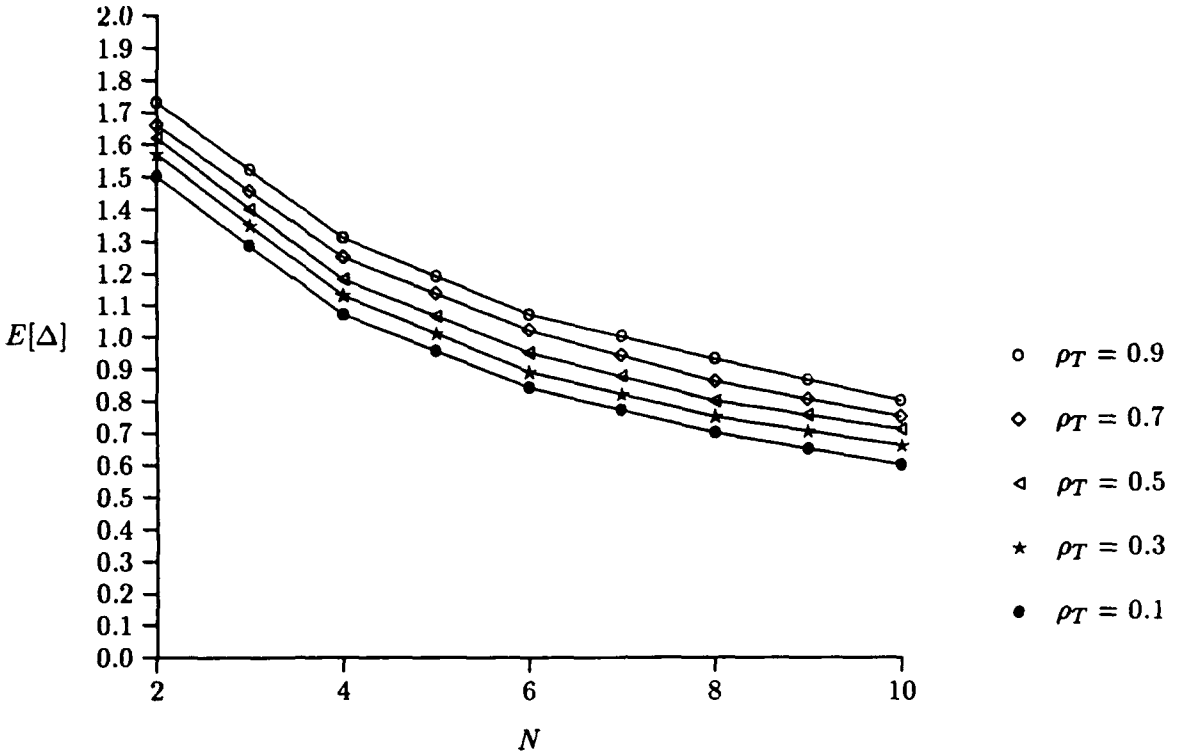


Figure 8: Expected departure time for the PP systems ($\mu = 0.5, \rho_T = \rho_1 + \rho_2 + ... + \rho_N = N\rho$).

Claim 4.1 is the direct result of Theorem C.2. Intuitively, a lower-priority job is more likely to be interrupted, and the next departure time is expected to be longer. The effects of ρ and N on $E[T]$ are described as follows:

- Effect of ρ: When a job J_i arrives p, there are two possibilities: (i) there is no higher-priority job receiving service, and J_i is served immediately; or (ii) there is a higher-priority job J_j receiving service. The probability that (i) is true is $(1 - \sum_{k=1}^{i} \rho_k)$, and the probability that (ii) is true is ρ_j. The expected departure time for (i) is longer than that for (ii), according to Claim 4.1. Thus, when $\rho_j = \rho$ increases, ρ_T increases, and $(1 - \sum_{k=1}^{i} \rho_k)$ decreases, and the expected departure time decreases.

- Effect of N. For a large N, it is likely that the job receiving service is of type $i >> 1$. Then from Claim 4.1, long departure time is expected.

- Effect of heterogeneous arrival rates: The larger the arrival rates of higher priority jobs, the shorter the expected departure times (Claim 4.1).

Finally, $\Theta = E[\Delta]/E[T]$. Figure 9 shows the relationship among Θ, ρ, and N. From the discussions about $E[\Delta]$ and $E[T]$, it is apparent that Θ increases if N decreases and/or ρ increases. Thus, we conclude that large Θ value can be obtained when N is small, and ρ_T is large. Also, the larger the arrival rates of higher priority jobs, the better the lookahead ratio.

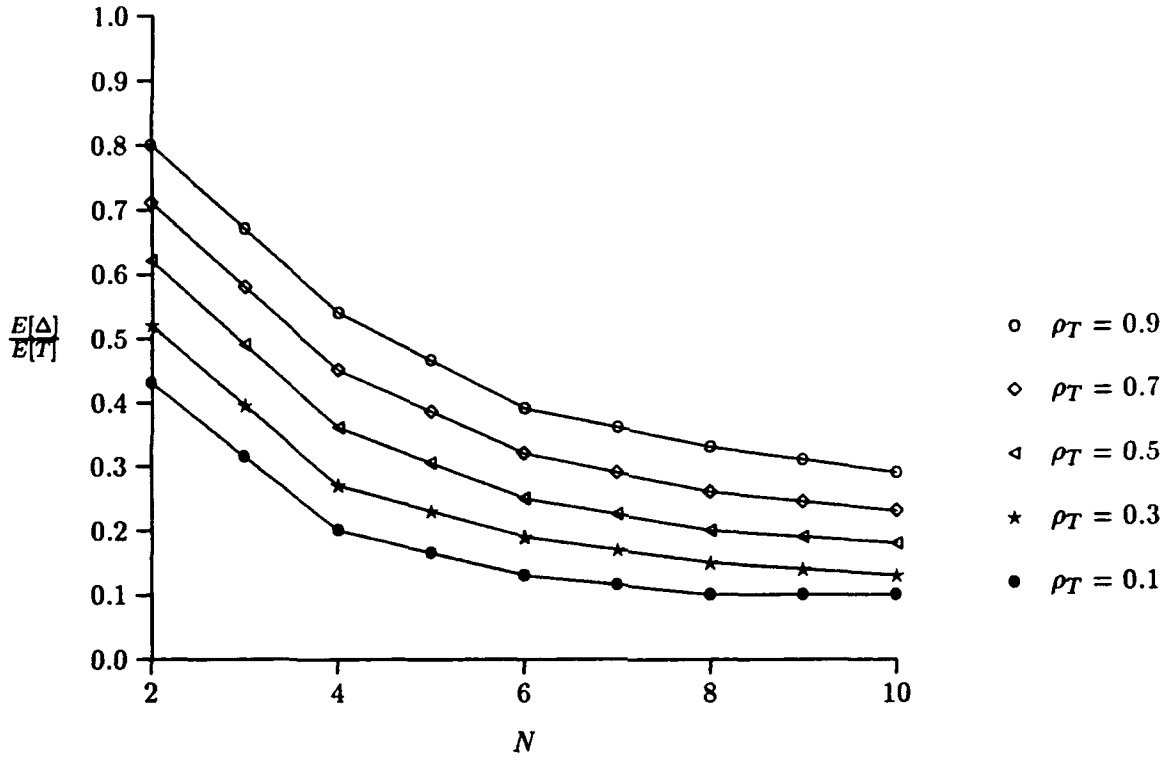


Figure 9: The lookahead ratio for the PP systems ($\mu = 0.5, \rho_T = \rho_1 + \rho_2 + ... + \rho_N = N\rho$).

5 Conclusions

In Chandy-Misra deadlock avoidance (*DA*) simulation, lookahead must exist in the simulated system to ensure the absence of deadlock [7]. Fujimoto recognized that, independent of its effect on deadlock, lookahead also has significant effect on the performance (speedup) of a *DA* simulation [3, 2].

In Fujimoto's study, *explicit* lookahead is assumed, i.e., the lookahead is known before the simulation, and it does not change during the simulation. Another kind of lookahead, called *implicit* lookahead, was introduced by Nicol [8]. Nicol showed that by exploiting implicit lookahead, the speedup of a *DA* simulation of a FCFS system is significantly improved.

In this paper, we showed the feasibility of exploiting lookahead for two non-FCFS system simulations, the round-robin (RR) and preemptive priority (PP) systems. We proposed several lookahead exploiting techniques for the RR systems. Technique 1 (Lemma 3.1) exploits lookahead of a process when more than one job is in service. An analytical model (Theorem A.1) showed that good performance (lookahead ratio) can be achieved when the service time quantum, Q, and/or the system load, ρ, are large. Technique 2 and Technique 3 exploit the lookahead of a process when no job is in service. Experiments and an analytical model (Theorem A.2) showed that the lookahead ratio increases when Q and/or ρ increase. We also constructed a lookahead generation algorithm using Technique 2 (Theorem B.1). The time complexity of the algorithm is $O(1)$ with a reasonably small constant (Theorem B.2). Technique 1 and Technique 2 (or 3) can be combined (Theorem 3.3) to

yield larger lookahead ratio for the range $0.2 \leq \rho \leq 0.6$. Note that the feasibility of Technique 1 does not depend the future list technique, showing that the future list technique is not the only way to exploit lookahead.

A lookahead exploiting technique for PP systems was proposed by Wagner and Lazowska [9]. We analyzed this technique by an analytical model (Appendix C). The model showed that the lookahead ratio increases when the number of job classes in the system increases and/or the system load increases. We also observed that the larger the arrival rates of higher priority jobs, the better the lookahead ratio. Our analyses showed that exploiting lookahead significantly improves the lookahead ratios of the RR and PP system simulations.

In this paper, the lookahead ratio (cf. (1)) is used as the performance measure. Intuitively, larger lookahead ratio results in higher speedup. However, the detailed relationship between the lookahead ratio and speedup is seldom studied in the literature. Our future research direction is to establish a concrete relationship between the lookahead ratio and speedup.

Acknowledgement

We would like to thank David Wagner: his work concerning techniques for exploiting lookahead motivated this research.

References

[1] Ajmone, M., Balbo, G., and Conte, G. *Performance Models of Multiprocessor Systems.* The MIT Press, 1986.

[2] Fujimoto, R.M. Lookahead in Parallel Discrete Event Simulation. *International Conference on Parallel Processing*, 1988.

[3] Fujimoto, R.M. Performance Measurements of Distributed Simulation Strategies. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 14–20, 1988.

[4] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[5] Kleinrock, L. *Queueing Systems: Volume 2 - Computer Applications.* New York: Wiley, 1976.

[6] Lin, Y.-B., Lazowska, E.D., and Baer, J.-L. Conservative Parallel Simulation For Systems With No Lookahead. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[7] Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[8] Nicol, D.M. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137, 1988.

[9] Wagner, D.B., and Lazowska, E.D. Parallel Simulation of Queueing Networks: Limitations and Potentials. 1989 ACM SIGMETRICS and Performance '89, 1989.

[10] Wagner, D.B., Lazowska, E.D. and Bershad, B. Techniques for Efficient Shared-Memory Parallel Simulation. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 29–37, 1989.

A Lookahead Exploiting Techniques for RR System Simulations

In this appendix, the lookahead ratios yielded by Technique 1 and Technique 3 are derived using analytical models.

A.1 Lookahead Ratio of Technique 1

In this subsection we derive the lookahead ratio of Technique 1. we assume that the service quantum size Q is small compared with the service time. We also make the following assumptions:

- Jobs arrive at a process p in a Poisson stream with rate λ.
- Jobs' required service times have some general distribution function with mean $\frac{1}{\mu}$. Note that the service times is no less than Q, and $\frac{1}{\mu} >> Q$.
- $E[\Delta]$: The average lookahead prediction at a particular timestamp t.
- $E[T]$: The average time interval between t and the next departure.
- $\Theta = E[\Delta]/E[T]$: The lookahead ratio.

Some performance measures of the simulated system come out immediately:

- $\rho = \lambda/\mu$: the load. We assume that $\rho < 1$, i.e., the system is not saturated.
- λ: the throughput.
- n: the steady-state average number of jobs in p. Since Q is small, we borrow the result of a processor sharing system that $n = \rho/(1-\rho)$ [5].
- $q = \lambda Q$: the probability that a departure occurs in the time period Q. Since $\lambda/\mu < 1$, and $Q << 1/\mu$, we have $q << 1$.

Theorem A.1: The lookahead ratio Θ yielded by Technique 1 is

$$\Theta = \frac{1-(1-\lambda Q)^{n+2}}{2-\lambda Q-(1-\lambda Q)^{n+1}}, \quad \text{where } n = \frac{\rho}{1-\rho}$$

Proof: Suppose job J arrives at p at timestamp t_J, and p will start the next service time quantum at time t. Note that $t - t_J < Q << 1/\mu$. Since $t - t_J$ is small, we ignore its effect and assume that $t_J \approx t$. Then at time t, J finds an average of n jobs already present (the random observer property of a Poisson process). Thus, from Lemma 3.1 we have full knowledge about p's behavior on the timestamp interval $[t, t+(n+1)Q]$ (Note that J is also in service at t, so there are $n+1$ jobs in service).

Let p_i be the probability that the latest departure occurs in $[t, t+(n+1)Q]$ is at time $t+iQ$. Then $p_i = q(1-q)^{n+1-i}, 1 \leq i \leq n+1$, ($p_0 = (1-q)^{n+1}$ is the probability that no departure occurs in $[t, t+(n+1)Q]$) and the average lookahead prediction is

$$E[\Delta] = \sum_{i=0}^{n+1}(n+1-i)Qp_i + Q$$

Note that the last "Q" term is due to the fact that after J receives its first time quantum, no job can depart *within* the next quantum. Finally,

$$E[\Delta] = \frac{1}{\lambda}\left[1-(1-\lambda Q)^{n+2}\right]$$

$E[T]$ is derived as follows: Consider any time instant t, the probability that the next job departs at the next *ith* quantum is $q(1-q)^{i-1}$, so

$$E[T] = E[\Delta] - Q + \sum_{i=1}^{\infty}(iQ)q(1-q)^{i-1} = E[\Delta] - Q + \frac{1}{\lambda}$$

Thus, the lookahead ratio is

$$\Theta = \frac{E[\Delta]}{E[T]} = \frac{1-(1-\lambda Q)^{n+2}}{2-\lambda Q-(1-\lambda Q)^{n+2}}, \quad \text{where } n = \frac{\rho}{1-\rho} \quad \blacksquare$$

A.2 Lookahead Ratio of Technique 3

In this subsection, we derive the lookahead ratio of Technique 3. We first derive the expected lookahead value in Lemma A.1, then show the lookahead ratio in Theorem A.2.

Lemma A.1: Suppose the service times are i.i.d. exponentially distributed random variables with parameter μ, and $Q << 1/\mu$. Then the expected lookahead value $E[\Delta]$ yielded by Technique 3 is

$$E[\Delta] \simeq -\frac{1}{\mu \ln(2\mu Q)}$$

Proof: From Lemma 3.3 and because Q is very small, we have

$$\delta \simeq \min_{0 \leq i \leq j}[(i+1)s_i], \quad \text{where } j = \left\lceil \frac{s_0}{2Q} \right\rceil$$

Since $E[j] = E\left[\left\lceil \frac{S_0}{2Q} \right\rceil\right] \simeq \left\lceil \frac{1}{2\mu Q} \right\rceil$,

$$E[\Delta] \simeq \frac{1}{\mu \sum_{i=1}^{E[j]} \frac{1}{i}} \simeq \frac{1}{\mu \int_{i=1}^{E[j]} \frac{1}{i} di} = \frac{1}{\mu \ln(E[j])} \simeq \frac{1}{\mu \ln \frac{1}{2\mu Q}} = -\frac{1}{\mu \ln(2\mu Q)} \quad \blacksquare$$

Theorem A.2: Suppose the service time is an exponentially distributed random variable X with parameter μ, and $Q << 1/\mu$. Let the job arrival intervals be an exponentially distributed random

variable Y with parameter λ, and $\rho = \lambda/\mu << 1$. Then the lookahead ratio yielded by Technique 3 is

$$\Theta \simeq -\frac{\rho}{(1+\rho)\ln(2\mu Q)}$$

Proof: When there is no job in service, the residual arrival interval of the next arrival job J is a random variable with the same distribution as that of Y (the random observer property of a Poisson process). Since $\rho << 1$, we expect that J will finish service without being interrupted. Thus,

$$E[T] = \frac{1}{\lambda} + \frac{1}{\mu}$$

and from Lemma A.1

$$\Theta = \frac{E[\Delta]}{E[T]} \simeq -\frac{\rho}{(1+\rho)\ln(2\mu Q)} \quad \blacksquare$$

Note that Lemma A.2 under estimates $E[T]$ when ρ is large.

B Computational Aspects of Technique 2

In this section, we show an algorithm that generates lookahead values via Technique 2. To generate a lookahead value, the time complexity is $O(1)$ with a reasonably small constant.

Definition B.1: Let s_i be the service time for the i^{th} arriving job. For $n \geq i$, define

$$\delta_{i,n} \triangleq \min_{0 \leq k \leq n-i} (kQ + s_{k+i})$$

and for each $i \geq 0$ define

$$\delta_i \triangleq \delta_{i,j}, \text{ where } j = \left\lfloor \frac{s_i}{Q} \right\rfloor$$

Note that if J_{i-1} and J_i are $(i-1)^{th}$ and i^{th} arrival jobs that arrive at times $t_{J_{i-1}}$ and t_{J_i} respectively, then δ_i is the lookahead yielded by Technique 2 at time t where $t \in (t_{J_{i-1}}, t_{J_i}]$. This is a general form of (3) in Lemma 3.2.

From (3) and Definition B.1, we have the following lemma:

Lemma B.1: For $i \geq 0$
(a) Let $j = \lfloor s_i/Q \rfloor$, then for all $k \geq j$, we have $\delta_i = \delta_{i,i+k}$.
(b) For all $i \leq l < m$, we have $\delta_{i,m} = \min[\delta_{i,l}, (l-i)Q + \delta_{l+1,m}]$.

Definition B.2: For all $i \geq 0$ define α_i as

$$\alpha_i \triangleq \max_{0 \leq k \leq i} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right)$$

From Definition B.1, it is apparent that $s_0, s_1, ..., s_{\alpha_i}$ are those service times required to determine $\delta_0, \delta_1, ..., \delta_i$.

```
    procedure A(γ,β):
Inputs: γ,β;
Outputs: a, δ_{γ,β}, δ_{γ+1,β}, ...., δ_{β,β}
1.      Generate s_β;
2.      δ_{β,β} = s_β;
3.      a = β + ⌊s_β/Q⌋;
4.      for k = β − 1 to γ do
5.          Generate s_k;
6.          δ_{k,β} = min(s_k, Q + δ_{k+1,β});
7.          a = max(k + ⌊s_k/Q⌋, a);
        end for;
    end A.
```

```
    program Main:
Outputs: δ_0, δ_1, ....., δ_n;
1.      Generate s_0;
2.      β = ⌊s_0/Q⌋,   γ = 0;
3.      call A(0,β);
4.      output(δ_{0,β});
5.      i = 1;
6.      while i < n do
7.          γ = β + 1;
8.          β = a;
9.          call A(γ,β);
10.         while (i < γ) do
11.             if ⌊s_i/Q⌋ + i < γ then
12.                 output(δ_{i,γ−1}),
                else
13.                 output(min[δ_{i,γ−1}, (γ − i − 1)Q + δ_{γ,β}]);
                end if;
14.             i = i + 1;
            end while;
        end while;
    end Main.
```

Figure 10: Lookahead generation algorithm of Technique 2.

Figure 10 shows a program, Main, which generates n lookahead values $\delta_0, \delta_1, ..., \delta_{n-1}$. A procedure, A, is called by Main. The correctness of procedure A and Main are shown in Lemma B.2, Lemma B.3, and Theorem B.1.

Lemma B.2: Consider the outer while loop in Main (Lines 6-14). Let $a(i), \beta(i)$, and $\gamma(i)$ be the values of a, β, and γ at the end of the i^{th} iteration. Initially, $\beta(0) = \left\lfloor \frac{s_0}{Q} \right\rfloor$, and $\gamma(0) = 0$ (Line 2 of Main), and $a(0)$ is assigned by invoking procedure A at Line 3. Then we have $\beta(0) = \alpha_0$, and for $i \geq 1$,

$$a(i) = \alpha_{\beta(i)}, \quad \gamma(i) = \beta(i-1) + 1, \quad \text{and} \quad \beta(i) = \alpha_{\gamma(i)-1}$$

Proof: In every iteration i, $\beta(i)$ is calculated at Line 8 of Main, γ is calculated at Line 7 of Main, and $a(i)$ is calculated at Lines 3 and 7 of procedure A:

$$a(i) = \min_{\gamma(i) \leq k \leq \beta(i)} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right) \tag{12}$$

A modified program segment of Main for proving Lemma B.2 is shown in Figure 11. Consider the while loop in Figure 11 (Line 3-6). In each iteration, γ, β and a are replaced by new values. The values of $\gamma(0), \beta(0)$, and $a(0)$ are determined at Lines 1 and 2. We prove by induction on i that the hypotheses are true.

Basis: We need to prove the cases that $i = 0$ and $i = 1$. When $i = 0, \gamma(0) = 0$ (Line 1), and $\beta(0) = \alpha_{\gamma(0)}$ (from Definition B.2 and Line 1, and because $\gamma(0) = 0$), and $a(0) = \alpha_{\beta(0)}$ (From Definition B.2 and Line 2).

When $i = 1$, $\gamma(1) = \beta(0) + 1$ (from Line 4), and $\beta(1) = a(0) = \alpha_{\beta(0)} = \alpha_{\gamma(1)-1}$ (from Line 5 and because $\beta(0) = \gamma(1) - 1$). At Line 6, $a(1)$ is computed by (12):

$$a(1) = \max_{\beta(0)+1 \leq k \leq \beta(1)} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right)$$

Since

$$\max_{0 \leq k \leq \beta(0)} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right) = \alpha_{\beta(0)} = \beta(1)$$

we have

$$a(1) = \max_{\beta(0)+1 \leq k \leq \beta(1)} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right) = \max \left[\beta(1), \max_{\beta(0)+1 \leq k \leq \beta(1)} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right) \right] = \alpha_{\beta(1)}$$

Inductive step: Suppose the hypotheses are true for the first k iterations, we need to prove that in the $i + 1^{th}$ iteration, the hypotheses are also true. The proof is similar to the one for $i = 1$ in the basis. This part is omitted. ∎

Lemma B.3: Consider the outer while loop in Main (Line 6-14). At the end of each iteration, Main generates $\delta_{k,\beta}, \gamma(i) \leq k \leq \beta(i)$.

Proof: Lemma B.2 ensures that $\gamma(i) \leq \beta(i)$. Line 2 of procedure A generates $\delta_{\beta,\beta}$ (Definition B.1). Line 6 in the while loop generates $\delta_{k,\beta}, \gamma(i) \leq k < \beta(i)$ (Lemma B.1 (b)). ∎

1.	$\gamma = 0, \beta = \left\lfloor \frac{s_0}{Q} \right\rfloor$;	(Line 2,3, Main)
2.	$a = \max_{\gamma \le k \le \beta} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right)$;	(Line 7, A)
3.	**while** $\gamma < n$ **do**	(Line 6, 10, Main)
4.	$\gamma = \beta + 1$;	(Line 7, Main)
5.	$\beta = \alpha$;	(Line 8, Main)
6.	$a = \max_{\gamma \le k \le \beta} \left(k + \left\lfloor \frac{s_k}{Q} \right\rfloor \right)$;	(Line 7, A)
	end while;	(Line 14, Main)

Figure 11: A modified program segment of Main.

Theorem B.1: The program Main generates n lookahead values $\delta_0, \delta_1, ..., \delta_{n-1}$.

Proof: Consider the outer while loop of Main (Lines 6-14). Let $\gamma(i), \beta(i)$, and $a(i)$ be the values for the i^{th} iteration. We prove by induction that at the i^{th} iteration, $\delta_0, \delta_1, ..., \delta_{\gamma(i)-1}$ are outputted, and $\delta_{\gamma(i),\beta(i)}, \delta_{\gamma(i)+1,\beta(i)}, ..., \delta_{\beta(i),\beta(i)}$ are generated.

Basis: $i = 0$. Because $\beta(0) = \left\lfloor \frac{s_0}{Q} \right\rfloor$ we have $\delta_0 = \delta_{0,\beta(0)}$, and $\delta_{1,\beta(0)}, \delta_{2,\beta(0)}, ..., \delta_{\beta(0),\beta(0)}$ are generated at Line 3 (Lemma B.3). Then δ_0 is outputted at Line 4.

Let $i = 1$. $\gamma(1) = \beta(0) + 1$ (Line 7), and $\beta(1) = \alpha_{\beta(0)}$ (Line 8 and Lemma B.2). At Line 9, $\delta_{\gamma(1),\beta(1)}, \delta_{\gamma(1)+1,\beta(1)}, ..., \delta_{\beta(1),\beta(1)}$ are generated (Lemma B.3). Now we show that $\delta_1, \delta_2, ..., \delta_{\gamma(1)-1}$ are outputted at Line 12 or Line 13 in the inner while loop (Line 10-14): For all $k, 0 \le k < \gamma(1)$, $\delta_{k,\gamma(1)-1}$ are generated at Line 3. If $k + \left\lfloor \frac{s_k}{Q} \right\rfloor < \gamma(1)$ then $\delta_{k,\gamma(1)-1} = \delta_k$ (Lemma B.1 (a)), and δ_k is outputted at Line 12. If $k + \left\lfloor \frac{s_k}{Q} \right\rfloor \ge \gamma(1)$ then $k + \left\lfloor \frac{s_k}{Q} \right\rfloor < \beta(1) = \alpha_{\gamma(1)-1}$ (Lemma B.2 and Definition B.2). Thus, $\delta_k = \delta_{k,\beta(1)} = \min \left[\delta_{k,\gamma(1)-1}, (\gamma(1) - k - 1)Q + \delta_{\gamma(1),\beta(1)} \right]$ (Lemma B.1 (a) and (b)), and δ_k is outputted at Line 13.

Inductive step: The proof for the inductive step is similar to the one for $i = 1$ in the basis, and we omit this tedious work. To summary, the outer while loop (Lines 6-14) outputted $\delta_0, \delta_1, ..., \delta_{\gamma(i)-1}$ *in the first i iterations. This process is repeated until $\gamma(i) > n$.* ∎

Theorem B.2: The time complexity of generating one lookahead is $O(1)$.

Proof: The time complexity of Main is determined by the outer while loop (Line 6-14). The dominant operations of the outer while loop are at Line 9 and Line 10-14 (the inner while loop). At the i^{th} iteration of the outer while loop, there are $\gamma(i+1) - \gamma(i)$ operations performed at Line

9, and there are $\gamma(i) - \gamma(i-1)$ operations performed at the inner while loop, and the "γ" value (Line 7) is increased by the amount $\gamma(i) - \gamma(i-1)$. The process stops at the i'^{th} iteration where i' is the smallest integer such that $\gamma(i') > n$. Thus, the total number of operations contributed by Line 9 is $\gamma(i'+1) \approx n$(where $n >> \gamma(i'+1) - \gamma(i')$), and the total number of operation contributed by the inner while loop is $\gamma(i') \approx n$, and the time complexity of generating n lookahead values is $O(n)$. In other words, the time complexity of generating one lookahead value is $O(1)$. Note that the time constant is reasonably small. ∎

C Lookahead Ratio of PP Systems

In this appendix, we first derive the expected lookahead values $E[\Delta]$ for a PP system, then the expected departure time $E[T]$, and finally $\Theta = E[\Delta]/E[T]$.

C.1 Expected Lookahead Value

We first introduce some notations/assumptions:

- λ_i : type i jobs arrive in an independent Poisson stream with rate λ_i.
- $1/\mu_i$: the service time of a type i job is exponentially distributed with mean $\frac{1}{\mu_i}$.
- $\rho_i = \lambda_i/\mu_i$: the probability that a job of type j is receiving service, or the load for type i. Assume that the system is not saturated, i.e., $\rho_1 + \rho_2 + ... + \rho_N < 1$.

Theorem C.1: Suppose an arrival job is of type i. Then the expected lookahead value $E[\Delta_i]$ yielded by the PP algorithm (Section 4.1) is

$$E[\Delta_i] = \frac{1 - \sum_{k=1}^{i} \rho_i}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i-1} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k}$$

Proof: For the arrival of an type i job J_i, let A be the event that no job with higher priority over J_i is receiving service, i.e., Case I of the algorithm is true (cf. Section 4). Let A^c be the event that there is a job with higher priority over J_i is receiving service (i.e., Case II of the algorithm is true). Then

$$E[\Delta_i] = E[\Delta_i; A] + E[\Delta_i; A^c]$$

Consider Case I of the algorithm. Since Δ_i and A are independent, and $\Pr[A] = 1 - \sum_{k=1}^{i} \rho_k$, we have

$$E[\Delta_i; A] = E\left[\min_{1 \le j \le i}(S_j)\right] \Pr[A] = \frac{1 - \sum_{k=1}^{i} \rho_k}{\sum_{j=1}^{i} \mu_j}$$

Consider Case II of the algorithm. We have

$$E[\Delta_i; A^c] = E\left[\min\left(R_j, \min_{1 \le k \le j-1}(S_k)\right); 1 \le j \le i\right] = \sum_{j=1}^{i} \rho_j E\left[\min\left(R_j, \min_{1 \le k \le j-1}(S_k)\right)\right]$$

Because of the memoryless property of the exponential distribution, and because of the random observer property of the Poisson process, the residual life r_j has the same distribution as s_j. Hence,

$$E[\Delta_i; A^c] = \sum_{j=1}^{i} \rho_j E\left[\min_{1 \le k \le j}(S_k)\right] = \sum_{j=1}^{i} \frac{\rho_j}{\sum_{k=1}^{i} \mu_k}$$

Thus, the expected lookahead value for an arrival job J_i is

$$E[\Delta_i] = E[\Delta_i; A] + E[\Delta_i; A^c] = \frac{1 - \sum_{k=1}^{i} \rho_k}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k} \quad \blacksquare$$

Corollary C.1: The expected lookahead value $E[\Delta]$ yielded by the PP algorithm for an arbitrary arrival job is

$$E[\Delta] = \sum_{i=1}^{N} \frac{\lambda_i}{\sum_{n=1}^{N} \lambda_n} \left(\frac{1 - \sum_{k=1}^{i} \rho_k}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k} \right)$$

Proof: The probability, q_i, that an arrival job is of type i is

$$q_i = \frac{\lambda_i}{\sum_{n=1}^{N} \lambda_n}$$

Thus, from Theorem C.1

$$E[\Delta] = \sum_{i=1}^{N} q_i E[\Delta_i] = \sum_{i=1}^{N} \frac{\lambda_i}{\sum_{n=1}^{N} \lambda_n} \left(\frac{1 - \sum_{k=1}^{i} \rho_k}{\sum_{j=1}^{i} \mu_j} + \sum_{j=1}^{i} \frac{\rho_j}{\sum_{k=1}^{j} \mu_k} \right) \quad \blacksquare$$

C.2 Expected Departure Time and Lookahead Ratio

In this subsection, we derive the expected departure time $E[T]$, then obtain the lookahead ratio Θ by the definition $\Theta = E[\Delta]/E[T]$. We use the Poisson arrival assumption and the exponential service time assumption made in the previous subsection. We first define some notations:

- $p_i^*(t)$: the probability that a job J_i with residual service time t finishes its service without interruption.
- $p_i^{(j)}(t)$: the probability that a job J_i with residual service time longer than t is interrupted by a job $J_j (j < i)$ at time t.
- $r_i(t)$: the probability that the residual service time of a job J_i is t.
- $\phi_i(t)$: the probability that the residual arrival interval of a job J_i is t.

Since the service time and arrival interval for J_i are both exponential distributed with mean $1/\mu_i$ and $1/\lambda_i$ respectively, the residual service time and residual arrival interval also have the same distributions (The memoryless property). The arrival job streams of different types are independent of each other. When J_i arrives (remember that the arrivals of type i jobs are a Poisson process,

and J_i is considered as a random observer of the other arrival streams), it sees $i-1$ higher-priority arrival streams with residual arrival intervals that have distribution $\phi_j, 1 \leq j \leq i-1$. Suppose that J_i is receiving service and its residual service time is t. J_i finishes its service without interruption if and only if the next arrival of type j job, where $1 \leq j < i$, has a residual arrival interval $t_j > t$. Thus, we have $p_1^*(t) = r_1(t)$, and for $i \geq 2$,

$$p_i^*(t) = r_i(t) \prod_{j=1}^{i-1} \left(\int_{t_j=t}^{\infty} \phi_j(t_j) dt_j \right)$$

Consider another case that J_i's residual time is longer than t. Let t_j be the residual arrival interval of the next arrival job J_j, where $1 \leq j < i$. J_i is interrupted by J_k at t if and only if $1 \leq k \neq j < i$, $t_k = t$, and $t_j > t$. Thus, for $i = 1, p_i^{(j)}(t) = 0$, and for $i \geq 2$

$$p_i^{(j)}(t) = \prod_{1 \leq k \neq j \leq i-1} \left(\int_{t_k=t}^{\infty} \phi_k(t_k) dt_k \right) \phi_j(t) \int_{t_i=t}^{\infty} r_i(t_i) dt_i$$

Replace $r_i(t)$ by $\mu_i e^{-\mu_i t}$ and $\phi_i(t)$ by $\lambda_i e^{-\lambda_i t}$, and let $\Lambda_i = \lambda_1 + \lambda_2 + ... + \lambda_{i-1} + \mu_i$, we have for $i \geq 2$

$$\begin{aligned} p_i^*(t) &= \mu_i e^{-\mu_i t} \int_{t_1=t}^{\infty} \int_{t_2=t}^{\infty} .... \int_{t_{i-1}=t}^{\infty} \lambda_1 e^{-\lambda_1 t_1} \lambda_2 e^{-\lambda_2 t_2} .....\lambda_{i-1} e^{-\lambda_{i-1} t_{i-1}} dt_{i-1} ... dt_2 dt_1 \\ &= \mu_i e^{-\mu_i t} e^{-(\lambda_1 + \lambda_2 + ... + \lambda_{i-1})t} = \mu_i e^{-\Lambda_i t} \end{aligned}$$

and

$$p_i^{(j)}(t) = e^{-(\sum_{1 \leq k \neq j \leq i-1} \lambda_k)t} \lambda_j e^{-\lambda_j t} \int_{t_i=t}^{\infty} \mu_i e^{-\mu_i t_i} dt_i = \lambda_j e^{-\Lambda_i t}$$

Theorem C.2: The expected time interval, $E[R_i]$, between a job J_i starts (or resumes) it service and the next departure is $E[R_1] = 1/\mu_1$, and for $i \geq 2$,

$$E[R_i] = \frac{\mu_i}{\Lambda_i^2} + \sum_{j=1}^{i-1} \frac{\lambda_j}{\sum_{k=1}^{N} \lambda_k} \left(\frac{\lambda_j}{\Lambda_i^2} + \frac{E[R_j]\lambda_j}{\Lambda_i} \right), \text{ where } \Lambda_i = \sum_{k=1}^{i-1} \lambda_k + \mu_i$$

Proof: Consider the case that $i \geq 2$. Let A be the event that a J_i starts and finishes the service without interruption. Let A^c be the event that J_i starts the service but is interrupted (i.e., the next departure is a job $J_j, j < i$). Then

$$E[R_i] = E[R_i; A] + E[R_i; A^c]$$

Consider the case that A is true. Since the probability that J_i with residual service time t finishes its service without interruption is $p_i^*(t)$, we have

$$E[R_i; A] = \int_{t=0}^{\infty} t p_i^*(t) dt = \int_{t=0}^{\infty} t \mu_i e^{-\Lambda_i t} dt = \frac{\mu_i}{\Lambda_i^2}$$

Consider the case that A^c is true. If J_i is interrupted at time t by a job $J_j, j < i$, then the expected next departure time is $t + E[R_j]$. Since the probability that an arrival is of type i is

$q_i = \lambda_i/(\sum_{k=1}^{N} \lambda_k)$, we have

$$\begin{aligned} E[R_i; A^c] &= \sum_{j=1}^{i-1} q_j \int_{t=0}^{\infty} (t + E[R_j])\, p_i^{(j)}(t)dt \\ &= \sum_{j=1}^{i-1} q_j \left(\int_{t=0}^{\infty} \lambda_j t e^{-\Lambda_i t} dt + E[R_i] \int_{t=0}^{\infty} \lambda_j e^{-\Lambda_i t} dt \right) \\ &= \sum_{j=1}^{i-1} \frac{\lambda_j}{\sum_{k=1}^{N} \lambda_k} \left(\frac{\lambda_j}{\Lambda_i^2} + \frac{E[R_j]\lambda_j}{\Lambda_i} \right) \end{aligned}$$

Thus,

$$E[R_i] = \frac{\mu_i}{\Lambda_i^2} + \sum_{j=1}^{i-1} \frac{\lambda_j}{\sum_{k=1}^{N} \lambda_k} \left(\frac{\lambda_j}{\Lambda_i^2} + \frac{E[R_j]\lambda_j}{\Lambda_i} \right)$$

When $i = 1$, $E[R_1] = E[R_1; A] = 1/\mu_1$ ∎

Corollary C.3: The expected time $E[T_i]$ between the arrival/resumption of a job J_i and the next departure is

$$E[T_i] = \left(1 - \sum_{k=1}^{i} \rho_k \right) E[R_i] + \sum_{j=1}^{i} \rho_j E[R_j]$$

Corollary C.4: The expected time $E[T]$ between the arrival/resumption of an arbitrary job and the next departure is

$$E[T] = \sum_{i=1}^{N} \frac{\lambda_i}{\sum_{n=1}^{N} \lambda_i} \left[\left(1 - \sum_{k=1}^{i} \rho_k \right) E[R_i] + \sum_{j=1}^{i} \rho_j E[R_j] \right]$$

The Optimal Checkpoint Interval in Time Warp Parallel Simulation

Yi-Bing Lin and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The Time Warp mechanism is the most common "optimistic" parallel simulation protocol. A process executes every message as soon as it arrives. If a message with a smaller timestamp subsequently arrives, the process rolls back its state to the time of the earlier message and re-executes from that point.

Clearly, the state of each process must be saved (checkpointed) regularly in case rollback is necessary. Although most existing Time Warp implementations checkpoint after every state transition, this is not necessary, and the checkpoint interval is in reality a tuning parameter of the simulation.

This paper shows how to derive the optimal frequency of checkpointing in Time Warp simulation. We express the optimal checkpoint interval as a function of the average rollback distance, the average state saving overhead, the ratio of the total number of executed messages to the total number of undone messages, and the average cost of a simulation step. The result is consistent with the following intuition: A large checkpoint interval should be chosen if and only if the state saving overhead is large, and/or a large number of events are executed, on the average, between two consecutive rollbacks.

1 Introduction

The virtual time paradigm [5, 6] is a method of organizing and synchronizing distributed systems. An implementation of this paradigm, called the *Time Warp mechanism*, is one of the most important parallel simulation protocols.

In a parallel simulation [1, 12], the simulated system is partitioned into a set of sub-systems that interact through the scheduling of events[2]. The set of sub-systems are simulated by a set of processes that communicate by sending/receiving timestamped messages. The scheduling of an event for a sub-system at time t is simulated by sending a message with timestamp t to the corresponding process. The global event list and global clock of a sequential simulation do not exist in the parallel counterpart. Each process has its own input message queue and local clock. To correctly simulate a sub-system, the corresponding process must execute arriving messages in their timestamp order, as opposed to their real-time arrival order. To satisfy this causality constraint, a synchronization mechanism is required.

The Time Warp synchronization mechanism takes an optimistic approach in which a process executes every message as soon as it arrives. If a message with a smaller timestamp subsequently arrives, the process must roll back its state to the time of the earlier message and re-execute from that point. Thus, the state of each process must be saved regularly (regardless of whether or not rollbacks actually occur).

It has been shown [8, 9] that, under the assumption that the state saving overhead is negligible, Time Warp outperforms conservative approaches to synchronization [3] such as Chandy-Misra [1]. In other words, rollbacks and concomitant re-executions *per se* do not represent a

[2] In this paper, the terms "event" and "message" have the same meaning.

[3] Conservative approaches ensure that all messages are initially executed in timestamp order.

serious liability. Instead, the performance of Time Warp is determined by the efficiency of state saving. Thus, it is important to reduce the state saving overhead.

To our knowledge, the only study in this area is Fujimoto's work [3] which developed special-purpose hardware to support fast state saving. A complementary approach, which we explore in this paper, is to reduce the frequency of state saving. We show how to find the optimal frequency of state saving in a Time Warp simulation. Informally, an optimum will exist because frequent checkpointing will result in large total checkpointing overhead, whereas infrequent checkpointing will require the re-execution of a large number of events when a rollback occurs (because it is necessary to roll back to the first checkpoint prior to the timestamp of the offending message and re-execute forward from that point).

2 Checkpointing in Time Warp

We assume that a process saves its state every N event executions. The operation of saving the process state is called *checkpointing*. An event that immediately follows a state saving operation is called a *checkpoint*. For simplicity, we assume that rollbacks only occur at the end of an event execution; in other words, there is no *message preemption* [7]. The number of events being undone in a rollback is called the *rollback distance*. The following notation is used (cf. Figure 1):

- N: the checkpoint interval; the number of events between two checkpoints.
- m_1: the rollback distance; the number of undone events in a rollback.
- $\frac{1}{p} = E[m_1]$: the expected value of m_1.

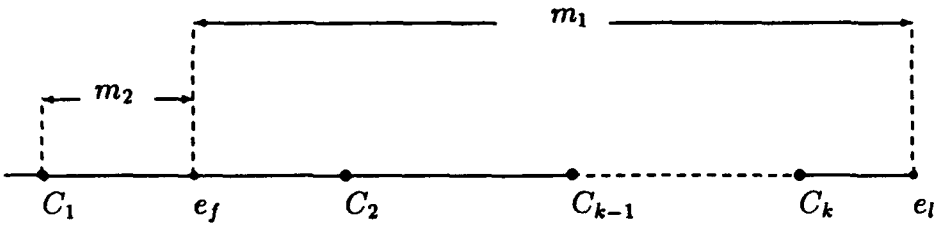


Figure 1: A rollback in Time Warp

- m_2: the number of events that must be re-executed because of a mismatch between the rollback distance and the checkpoint interval.
- $\delta(e_i)$: the execution time of the event e_i.
- δ_{sv}: the overhead of saving process state, assumed to be a constant.

Figure 1 shows a case where a rollback requires the undoing of m_1 events, starting from the event e_f, and ending at the event e_l. If the earliest undone event e_f is not a checkpoint, then there are $k-1$ undone checkpoints ($C_2, ..., C_k$, cf. Figure 1), and the m_2 events between the previous checkpoint C_1 and e_f must be re-executed. If e_f is a checkpoint, then $m_2 = 0$, and there are k checkpoints ($C_1, C_2, ..., C_k$) among the m_1 events. Thus, it is apparent that

$$k = \left\lceil \frac{m_1 + m_2}{N} \right\rceil \tag{1}$$

Now we define the overhead to restore the process state after a rollback. Consider Figure 2. Let $e_{f,i}$ ($e_{f,i-1}$) and $e_{l,i}$ ($e_{l,i-1}$) be the earliest and the latest undone events in the i^{th} ($i-1^{st}$) rollback respectively. Then the number of events executed between the $i-1^{st}$ rollback and the i^{th} rollback is m' (not including $e_{f,i-1}$) plus m_1 (note that m' can be a negative integer but $m_1 + m' > 0$). The number of checkpoints among $m' - m_2$ events is $k' = \left\lfloor \frac{m' - m_2}{N} \right\rfloor$ (cf.

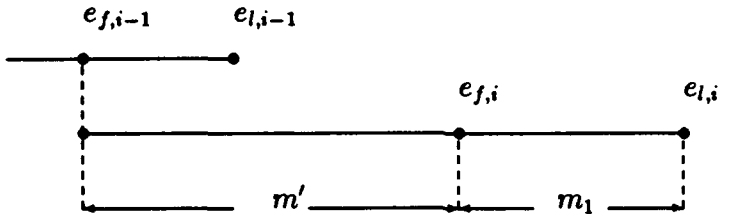


Figure 2: The executed events between two consecutive rollbacks.

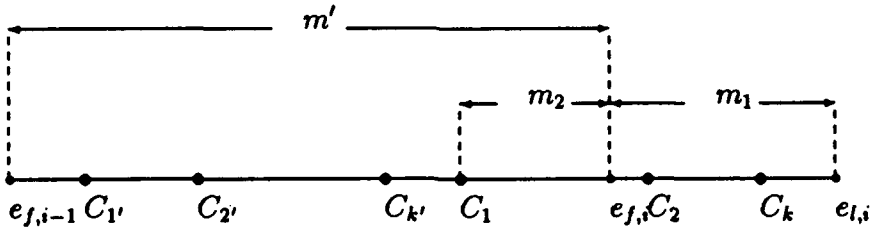


Figure 3: The checkpoints between two rollbacks.

Figure 3). Hence, the overhead to restore the process state after a rollback can be considered as the summation of (i) the overhead of checkpointing in those $m_1 + m'$ events (which is $(k' + k)\delta_{sv}$) and (ii) the overhead of re-executing those m_2 events (which is $\sum_{i=1}^{m_2} \delta(e_i)$). Let T_N be the overhead to restore the process state after a rollback when the checkpoint interval is N. Then T_N can be expressed as

$$T_N = (k' + k)\delta_{sv} + \sum_{i=1}^{m_2} \delta(e_i) = \left(\left\lfloor \frac{m' - m_2}{N} \right\rfloor + \left\lceil \frac{m_1 + m_2}{N} \right\rceil \right) \delta_{sv} + \sum_{i=1}^{m_2} \delta(e_i) \tag{2}$$

where

$$1 \leq m_1 \leq \infty \quad \text{and} \quad 0 \leq m_2 \leq N - 1 \tag{3}$$

When $N = 1$, we have $m_2 = 0$, and

$$T_1 = (m' + m_1)\delta_{sv} \tag{4}$$

When a large (small) N is chosen, we expect a small (large) $k' + k$ value and a large (small) m_2 value in Equation (2). Thus, the optimal value of N should balance the $(k' + k)\delta_{sv}$ term against the $\sum_{i=1}^{m_2} \delta(e_i)$ term such that T_N is minimized. In the next section, we derive an expression for $E[T_N]$, and find the value of N that minimizes it.

3 Performance Analysis of Checkpointing

This section derives an expression for $E[T_N]$, the mean value of the overhead to restore the process state after a rollback when the checkpoint interval is N, and finds the value of N that minimizes $E[T_N]$.

We first let $k' = 0$, and derive the mean value of the quantity

$$T'_N = k\delta_{sv} + \sum_{i=1}^{m_2} \delta(e_i) = \left\lceil \frac{m_1 + m_2}{N} \right\rceil \delta_{sv} + \sum_{i=1}^{m_2} \delta(e_i) \tag{5}$$

In other words, we temporarily ignore the events that are not rolled back. In that case (4) is re-written as

$$T_1' = m_1 \delta_{sv} \tag{6}$$

Then we approximate $E[T_N]$ as

$$E[T_N] = E\left[\frac{k'+k}{k}\right] E[k\delta_{sv}] + E\left[\sum_{i=1}^{m_2} \delta(e_i)\right] \tag{7}$$

From (1) and (3), we have

$$\begin{aligned} (k-1)N - m_2 \leq \ & m_1 \ \leq kN - m_2, \quad \text{if } k > 1 \\ 1 \leq \ & m_1 \ \leq N - m_2, \quad \text{if } k = 1 \end{aligned}$$

If m_1, m_2, and k are values of discrete random variables M_1, M_2 and K respectively, then the distribution of K can be derived from the distributions of M_1 and M_2. The conditional probability $\Pr[K = k | M_2 = m_2]$ is

$$\Pr[K = k | M_2 = m_2] = \begin{cases} \displaystyle\sum_{m_1=(k-1)N-m_2+1}^{kN-m_2} \Pr[M_1 = m_1], & k > 1 \\ \displaystyle\sum_{m_1=1}^{N-m_2} \Pr[M_1 = m_1], & k = 1 \end{cases} \tag{8}$$

Studies have shown that short rollback distances are observed more often than long rollback distances [2]. To be consistent with this observation, we assume that the random variable M_1 has a geometric distribution. The probability mass function is

$$\Pr[M_1 = m_1] = p(1-p)^{m_1 - 1}, \quad m_1 > 0 \tag{9}$$

Substituting (9) into (8) gives

$$\Pr[K = k | M_2 = m_2] = \begin{cases} \dfrac{(1-p_N)p_N^{k-1}}{(1-p)^{m_2}}, & k > 1 \\ 1 - \dfrac{p_N}{(1-p)^{m_2}}, & k = 1 \end{cases} \tag{10}$$

where $p_N = (1-p)^N$.

We assume that the first undone event e_f in a rollback is equally likely to fall any place between two checkpoints. In other words, the random variable M_2 has a discrete uniform distribution on $\{0, 1, .., N-1\}$ corresponding to the probability mass function

$$\Pr[M_2 = m_2] = \begin{cases} \frac{1}{N}, & \text{for } m_2 = 0, 1, ..., N-1 \\ 0, & \text{elsewhere} \end{cases} \tag{11}$$

For $i > 0$, assume that the $\delta(e_i)$ are i.i.d. with mean μ. From (5), (10), and (11), we obtain

$$\begin{aligned} E[T'_N] &= \sum_{m_2=0}^{N-1} \Pr[M_2 = m_2] \left(\sum_{k=1}^{\infty} \Pr[K = k | M_2 = m_2] k\delta_{sv} + E\left[\sum_{i=1}^{m_2} \delta(e_i) \right] \right) \\ &= \sum_{m_2=0}^{N-1} \left(\frac{1}{N} \right) \left\{ \left[1 - \frac{p_N}{(1-p)^{m_2}} + \sum_{k=2}^{\infty} \frac{(1-p_N)p_N^{k-1}}{(1-p)^{m_2}} k \right] \delta_{sv} + m_2\mu \right\} \\ &= \left[\frac{1 + (N-1)p}{pN} \right] \delta_{sv} + \left(\frac{N-1}{2} \right) \mu \end{aligned} \tag{12}$$

When $N = 1$, (12) is re-written as

$$E[T'_1] = \frac{\delta_{sv}}{p} \tag{13}$$

According to (6) and (9), we have

$$\begin{aligned} E[T'_1] &= \sum_{m_1=1}^{\infty} \Pr[M_1 = m_1] m_1 \delta_{sv} \\ &= \sum_{m_1=1}^{\infty} p(1-p)^{m_1-1} \delta_{sv} \\ &= \frac{\delta_{sv}}{p} \end{aligned}$$

which is the same as (13).

Let $\alpha = E\left[\frac{k' + k}{k} \right]$. When $m_2 << m_1, m'$, α can be approximated as $\alpha \simeq E\left[\frac{m' + m_1}{m_1} \right] = \frac{N_T}{N_R}$, where N_T is the total number of events executed in the simulation, and N_R is the total

number of undone events. The Time Warp mechanism guarantees $\frac{N_T}{N_R} > 1$ [4, 5, 6, 10, 13]. From (12) and (7), $E[T_N]$ is expressed as

$$E[T_N] = \alpha \left[\frac{1+(N-1)p}{pN}\right] \delta_{sv} + \left(\frac{N-1}{2}\right) \mu$$

To find the optimal value of N, we differentiate $E[T_N]$ with respect to N, and equate the result to zero:

$$\frac{dE[T_N]}{dN} = -\frac{\alpha(1-p)\delta_{sv}}{pN^2} + \frac{\mu}{2} = 0$$

which when solved yields the optimal value of N

$$N_{opt} = \sqrt{\frac{2\alpha(1-p)\delta_{sv}}{p\mu}} \approx \sqrt{\frac{2\alpha(E[m_1]-1)\delta_{sv}}{\mu}} \tag{14}$$

Since $N_{opt} \in \{1, 2, ...\}$, (14) is written as

$$N_{opt} = \left\lceil \sqrt{\frac{2\alpha(E[m_1]-1)\delta_{sv}}{\mu}} \right\rceil$$

The optimal value of N is plotted against $E[m_1]$, α, δ_{sv}, and μ in Figure 4. The figure shows that a large checkpoint interval should be chosen if and only if the state saving overhead is large, and/or a large number of events are executed, on the average, between two consecutive rollbacks. (Note that the product $\alpha E[m_1]$ is interpreted as the average number of events executed between two consecutive rollbacks). This result is consistent with intuition. When the state saving overhead is large, we should checkpoint infrequently. Also, when the number of events executed between two consecutive rollbacks is large, the extra overhead due to long checkpoint intervals is not significant, and a large N should be chosen. Besides, Figure 4 indicates that the larger the $E[m_1]$ value, the greater the sensitivity of N_{opt} to $\frac{\delta_{sv}}{\mu}$. (When $E[m_1] = 2$, $\frac{\Delta N_{opt}}{\Delta(\delta_{sv})/\mu} = \frac{4-1}{2.5-0.1} = 1.25$. When $E[m_1] = 6$, $\frac{\Delta N_{opt}}{\Delta(\delta_{sv})/\mu} = \frac{9-1}{2.5-0.1} \simeq 3.33$.)

A variance analysis of T_N also can be performed easily in our model (cf. Appendix A).

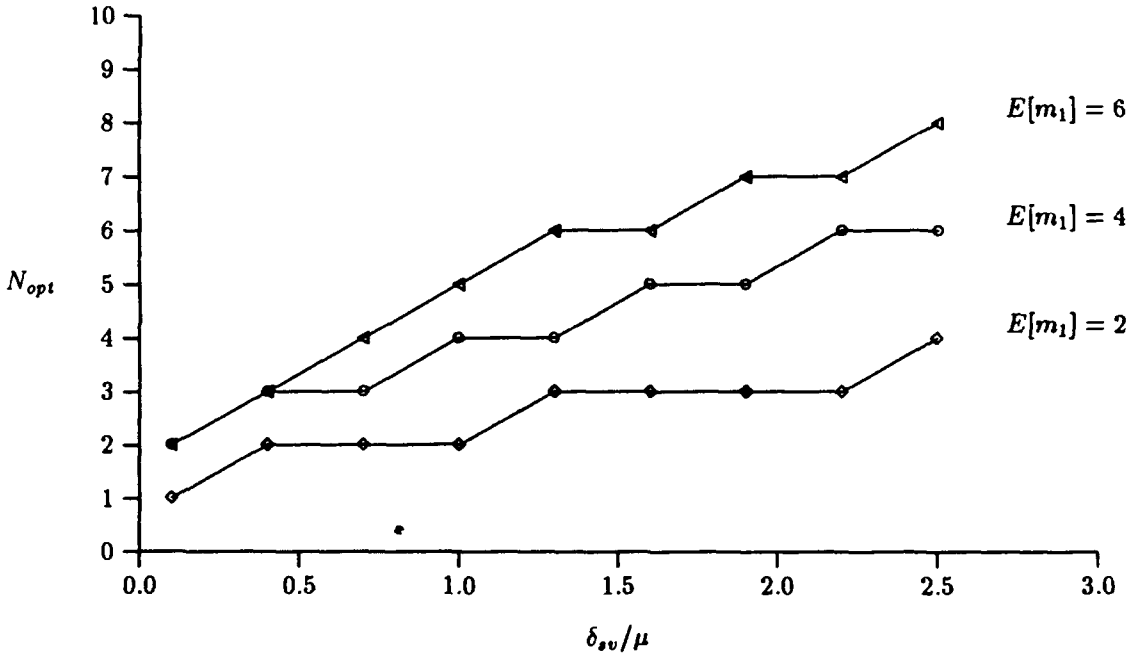


Figure 4: The optimal checkpoint interval N_{opt} as the function of the average rollback distance, $E[m_1]$, the state saving overhead, δ_{sv}, the ratio of the total number of executed messages to the total number of undone messages, α, and the average event execution time μ. ($\alpha = 2$)

4 Other Applications of Checkpointing

Checkpointing is used in other application areas, such as backing up databases, updating differential files, checkpointing programs, and database reorganization [11]. In these contexts, the checkpointing model can be described as follows.

The computation of the system is considered as time intervals between checkpoints. When a failure occurs, the execution is resumed at the previous checkpoint. The process of checkpointing begins anew after the occurrence of a failure. In other words, the progress of the system can be considered as *cycles of failure*. As shown in Figure 5, each failure cycle starts at a checkpoint C_1. After an execution time interval of t_1, a failure occurs. Then the next cycle starts at C_k. Let T_c be the time interval between checkpoints, then the effort is to find

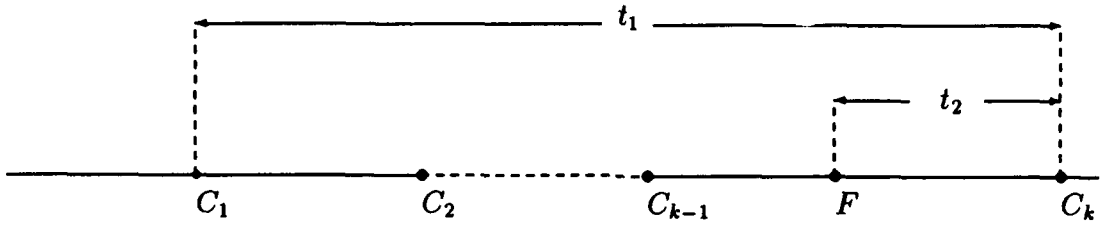


Figure 5: A failure cycle

the optimal T_c value which minimizes the total *lost time* T_l:

$$T_l = t_2 + k\delta_{sv} \tag{15}$$

where t_2 is the time interval between the failure point and the restarting point, and δ_{sv} is the state saving overhead of a checkpoint.

Young [14] has derived a first order approximation to the optimal checkpoint interval:

$$T_c \simeq \sqrt{2E[t_1]\delta_{sv} - \delta_{sv}^2} \simeq \sqrt{2E[t_1]\delta_{sv}} \tag{16}$$

Equation (16) is very similar to our equation (14). Despite the similarity, (16) is an approximation that hinges on the assumption that $\delta_{sv} << E[t_1]$ while our model make no assumption about δ_{sv}.

Beyond the similar results lie significant differences between our model and Young's model (and other fault recovery models):

- In the fault recovery model, each failure cycle starts at a checkpoint. In our model, the first and the last undone events are not necessarily checkpointed.
- In the fault recovery model, the time interval between two checkpoints, T_c, can be arbitrary. In our model, checkpointing only occurs after the execution of an event

finishes. Thus, the interval between two checkpoints is counted by the number of events. In Young's model, T_c is a fixed interval. In our model, N is fixed, but $\sum_{i=1}^{N} \delta(e_i)$ may have an arbitrary distribution.

- In the fault recovery model, a fault may occur during either an interval T_c or δ_{sv}. We assume no message preemption in our model, and rollbacks are handled at the end of event execution.

- The quantities to be minimized are different. (Cf. (15) and (7).) Young assumes t_1 to be exponentially distributed. We assume m_1 has a geometric distribution, but $\sum_{i=1}^{m_1} \delta(e_i)$ may have an arbitrary distribution.

Because of these differences, the technique developed to solve our problem is different from the techniques for the fault recovery systems. A general performance analysis of checkpointing for fault recovery systems can be found in [11].

5 Summary

It has been shown that, under the assumption that the state saving overhead is negligible, Time Warp outperforms conservative approaches such as Chandy-Misra. In other words, rollbacks and concomitant re-executions *per se* do not represent a serious liability. Instead, the performance of Time Warp is determined by the efficiency of state saving. Thus, it is important to reduce the state saving overhead.

This paper derived the optimal checkpoint interval for Time Warp simulation. We found that the optimal checkpoint interval can be expressed as a simple function of the average rollback distance, $E[m_1]$, the ratio of the total number of executed messages to the total

number of undone messages, α, the average state saving overhead, δ_{sv}, and the average event execution time, μ. This function can be interpreted intuitively: A large checkpoint interval should be chosen if and only if the state saving overhead is large, and/or a large number of events are executed, on the average, between two consecutive rollbacks.

The same analysis applies to the space complexity of Time Warp. Let s be the amount of memory required to save process state, and let s_N be the average amount of memory for state saving per event execution. Then $s_N = \frac{s}{N}$. When s is large, we should choose large N to avoid memory exhaustion. Since we expect that s is proportional to δ_{sv}, (14) may give a value of N which economizes the space requirement of Time Warp (when s is large, we expect a large δ_{sv} and (14) will choose a large N which results in a small s_N, as we desire).

As a final remark, we note that the use of (14) is very easy. The value of δ_{sv} is usually a constant, and can be pre-determined. The values of $E[m_1]$, α and μ can be collected during the simulation. (The parameter $\frac{N_T}{N_R} \simeq \alpha$ was considered an important output metric in many studies [4, 10, 13]. When m_2 is of the order of m_1 and m', we may directly measure α as $E\left[\frac{k'+k}{k}\right]$.) In fact, we may collect these values and re-calculate N regularly in the hope that dynamic modification of N may yield better performance.

References

[1] Chandy, K.M. and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[2] Fujimoto, R.M. Time Warp on a Shared Memory Multiprocessor. Technical Report UUCS-88-021a, Computer Science Department, University of Utah, January 1989.

[3] Fujimoto, R.M., Tsai, J.-J. and Gopalakrishnan, G. Design and Performance of Special Purpose Hardware for Time Warp. *Proc. 15th Symp. on Computer Architecture*, 1988.

[4] Hontalas, P., Beckman, B., Diloreto, M., Blume, L., Reiher, P., Sturdevant, K., Warren, L., Wedel, J., Wieland, F., and Jefferson, D. Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior & Sectoring). *In Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 3–7, 1989.

[5] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[6] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[7] Lin, Y.-B. and Lazowska, E.D. A Study of Time Warp Rollback Mechanisms. Technical Report 89-09-07, Department of Computer Science, University of Washington, 1989.

[8] Lin, Y.-B. and Lazowska, E.D. Comparing Synchronization Protocols for Parallel Logic-Level Simulation. Technical Report 89-09-06, Department of Computer Science, University of Washington, 1989.

[9] Lin, Y.-B. and Lazowska, E.D. Optimality Considerations for "Time Warp" Parallel Simulation. Technical Report 89-07-05, Department of Computer Science, University of Washington, 1989.

[10] Presley, M., Ebling, M., Wieland, F., Jefferson, D. Benchmarking the Time Warp Operating System with a Computer Network Simulation. *In Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 8–13, 1989.

[11] Tantawi, A.N. and Ruschitzka, M. Performance Analysis of Checkpointing Strategies. *ACM Transactions on Computer Systems*, 2(2):123–144, May 1984.

[12] Wagner, D.B., and Lazowska, E.D. Parallel Simulation of Queueing Networks: Limitations and Potentials. 1989 ACM SIGMETRICS and Performance '89, 1989.

[13] Wieland, F., Hawley, L., Feinberg, A., Loreto, M., Blume, L., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson, D. Distributed Combat Simulation and Time Warp: The Model and Its Performance. *In Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 14–20, 1989.

[14] Young, J.W. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM*, 17(17):530–531, September 1974.

A The Standard Deviation of T_N

This appendix derives the standard deviation for T_N. We assume that the event execution times are i.i.d. with exponential distribution:

$$f(t) = \lambda e^{-\lambda t} \quad \text{where} \quad \lambda = \frac{1}{\mu}$$

Then the summation, S_{m_2}, of m_2 event execution times has the Erlang distribution

$$f_{m_2}(t) = \frac{(\lambda t)^{m_2-1}}{(m_2-1)!} e^{-\lambda t}$$

The second moment of S_{m_2} is

$$\begin{aligned} E[S_{m_2}^2] &= \int_{t=0}^{\infty} t^2 f_{m_2}(t)dt = \int_{t=0}^{\infty} \left[\frac{m_2(m_2+1)}{\lambda_2}\right] f_{m_2+2}(t)dt \\ &= \frac{m_2(m_2+1)}{\lambda^2} = m_2(m_2+1)\mu^2 \end{aligned}$$

By using a similar approximation for (7), we derive $E[T_N^2]$ as follows:

$$\begin{aligned} E[T_N^2] &= \sum_{m_2=0}^{N-1} \Pr[M_2 = m_2] \left(\sum_{k=1}^{\infty} \Pr[K = k | M_2 = m_2](\alpha k \delta_{sv})^2 + E\left[\sum_{i=1}^{m_2} \delta(e_i)^2\right] \right) \\ &= A + B \end{aligned} \tag{17}$$

where A is

$$A = \sum_{m_2=0}^{N-1} \left(\frac{1}{N}\right) \left[1 - \frac{p_N}{(1-p)^{m_2}} + \sum_{k=2}^{\infty} \frac{(1-p_N)p_N^{k-1}}{(1-p)^{m_2}} k^2\right] (\alpha\delta_{sv})^2 = \left[\frac{(1-p)(3-p_N)}{Np(1-p_N)} + 1\right] (\alpha\delta_{sv})^2 \tag{18}$$

and B is

$$\begin{aligned} B &= \sum_{m_2=0}^{N-1} \left(\frac{1}{N}\right) E\left[\sum_{i=1}^{m_2} \delta(e_i)^2\right] = \sum_{m_2=0}^{N-1} \left(\frac{1}{N}\right) E[S_{m_2}^2] \\ &= \left(\frac{1}{N}\right) \left[\frac{(N-1)N(N+1)}{3}\right] \mu^2 = \frac{(N^2-1)\mu^2}{3} \end{aligned} \tag{19}$$

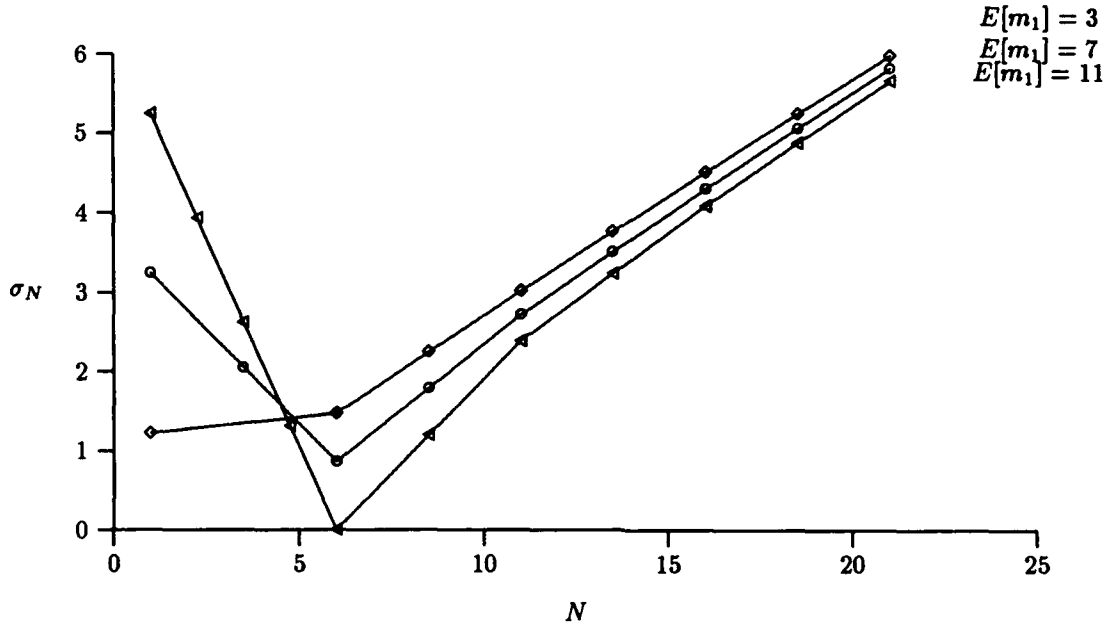


Figure 6: The standard deviation of T_A where $(\alpha\delta_{sv})/\mu = 0.5$ (Unit: μ).

Substituting (19) and (18) in (17)

$$E[T_N^2] = \left[\frac{(1-p)(3-p_N)}{Np(1-p_N)} + 1\right](\alpha\delta_{sv})^2 + \frac{(N^2-1)\mu^2}{3}$$

and the standard deviation of T_N, $\sigma_N = \sqrt{E[T_N^2] - (E[T_N])^2}$, can be expressed as

$$\sigma_N = \sqrt{\left[\frac{(1-p)(3-p_N)}{Np(1-p_N)} + 1\right](\alpha\delta_{sv})^2 + \frac{(N^2-1)\mu^2}{3} - \left[\frac{1+(N-1)p}{pN}\alpha\delta_{sv} + \frac{N-1}{2}\mu\right]^2} \tag{20}$$

Based on (20), Figures 6, 7, and 8 plot σ_N against N, $E[m_1]$, δ_{sv} and μ. We note that as N increases, the variance decreases then increases. This phenomenon can also be observed directly from (17): $\lim_{N\to\infty} A = 0$, and $\lim_{N\to\infty} B = \infty$. The effect of A dominates when N is small. On the other hand, the effect of B dominates when N is large. This leads to a more general conclusion. If the variance of the distribution for the event execution time is sufficiently small, then the σ_N curve is concave upward. Otherwise, σ_N is an increasing function of N.

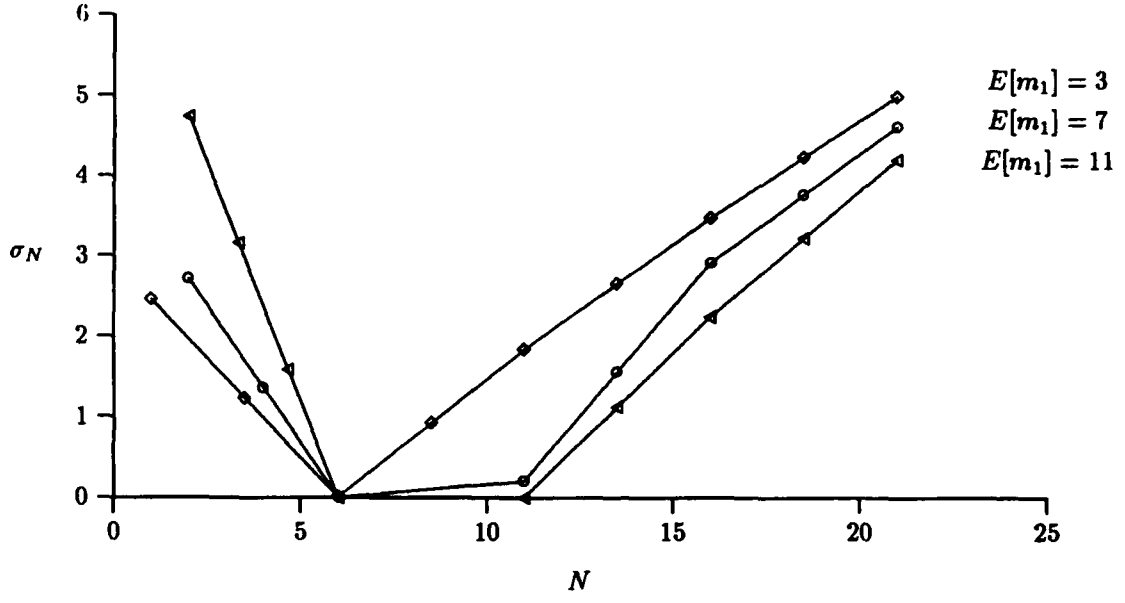


Figure 7: The standard deviation of T_A where $(\alpha\delta_{sv})/\mu = 1$ (Unit: μ).

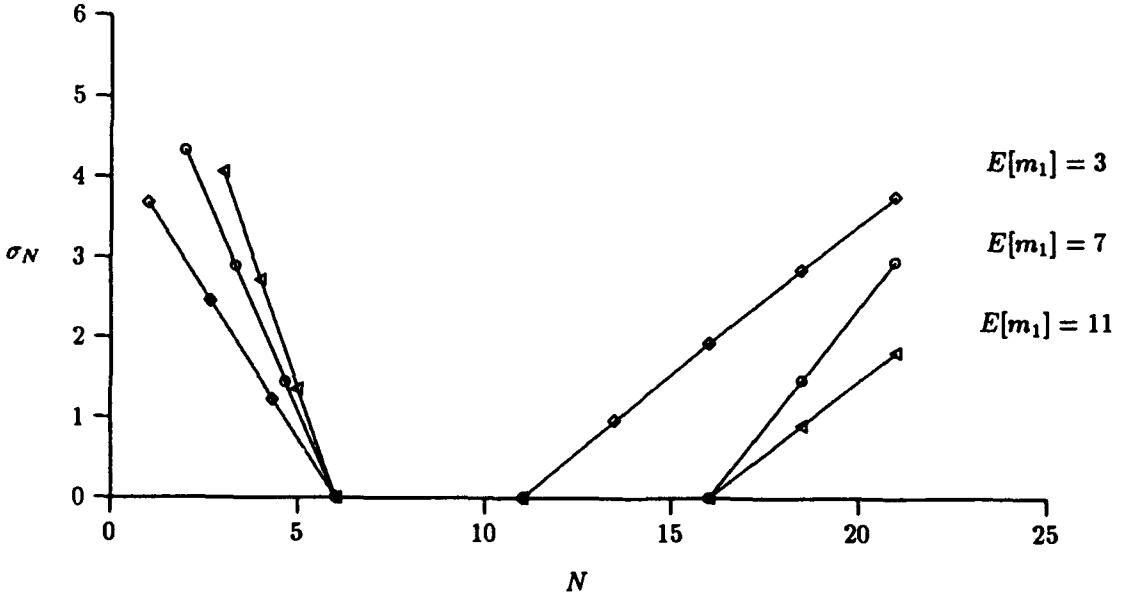


Figure 8: The standard deviation of T_A where $(\alpha\delta_{sv})/\mu = 1.5$ (Unit: μ).

A Study of Time Warp Rollback Mechanisms [1]

Yi-Bing Lin and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The Time Warp "optimistic" approach is one of the most important parallel simulation protocols. Time Warp synchronizes processes via *rollback*. The original rollback mechanism of Time Warp is called *aggressive cancellation*. Recently, a new rollback mechanism called *lazy cancellation* has aroused great interest. This paper studies these rollback mechanisms.

We begin by discussing the general tradeoffs between aggressive and lazy cancellation, and by defining a conservative-optimal simulation for comparative purposes.

Within the framework of aggressive cancellation, we develop algorithms and evaluate performance for various queueing system simulations, and analyze the rollback behavior of non-FCFS tandem systems.

Using a metric called the *sensitivity of output message*, we study the lazy cancellation mechanism. We observe that both aggressive and lazy cancellation work well for a process with a small simulated load intensity.

Finally, an analytical model is given to analyze *message preemption*, an important factor that affects the performance of rollback mechanisms. Our study shows that message preemption has a significant effect on performance when (i) the processor is highly utilized, (ii) the execution times of messages have high variance, and (iii) rollbacks occur frequently.

1 Introduction

The virtual time paradigm [11] is a method of organizing and synchronizing distributed systems. An implementation of the paradigm, called the *Time Warp mechanism*, is one of the most important parallel simulation protocols.

In a parallel simulation [23], the simulated system is partitioned into a set of sub-systems that interact through the scheduling of events[2]. The set of sub-systems are simulated by a set of processes that communicate by sending/receiving timestamped messages. The scheduling of an event for a sub-system at time t is simulated by sending a message with timestamp t to the corresponding

[1] This work was supported in part by the National Science Foundation (Grants CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

[2] In this paper, the terms "event" and "message" have the same meaning.

process. The global event list and global clock of a sequential simulation do not exist in the parallel counterpart. Each process has its own input message queue and local clock. To correctly simulate a sub-system, the corresponding process must execute arriving messages in their timestamp order, as opposed to their real-time arrival order. To satisfy this causality constraint, a synchronization mechanism is required. This paper concentrates on Time Warp synchronization mechanisms.

Denote by $ts(p, m)$ the timestamp of a message m executed at process p. Time Warp takes an optimistic approach in which a process executes every message as soon as it arrives. If a message m' with a smaller timestamp $ts(p, m')$ subsequently arrives, the process must roll back its state to the timestamp of the earlier message and re-execute from that point.

The rollback operation (i) undoes all executed messages in p that have timestamp larger than $ts(p, m')$, (ii) annihilates all output messages that have been scheduled by the execution of the undone messages (an *antimessage* is sent to annihilate each output message), and (iii) resumes the progress of p at time $ts(p, m')$ (i.e., executes messages in p's input message list according to their timestamp order).

To support rollback, a snapshot of the state of each process is taken after the execution of each input message. These states are stored in a queue associated with the process and are reinstated whenever it is necessary to roll back.

Most studies of Time Warp simulation have been experimental [6, 7, 10, 13, 20, 24]. Some studies have compared the performance of Time Warp with Chandy-Misra (a "conservative " protocol") [5] experimentally [7] or analytically [18]. Despite these studies, the *rollback mechanism* of Time Warp still is not well understood. The original rollback mechanism is called *aggressive cancellation* [11]. Recently, a new rollback mechanism called *lazy cancellation* [3, 9, 12, 18] has aroused great interest. The difference between the two mechanisms hinges on the time when an antimessage is sent.

Definition 1.1: In a Time Warp simulation with *aggressive cancellation* (abbreviated AC), the output messages are annihilated immediately after the corresponding out-of-order executed messages are undone. Then the progress of the process is resumed.

Definition 1.2: In a Time Warp simulation with *lazy cancellation* (abbreviated LC), after the out-of-order executed message is undone, the corresponding output message is not annihilated. Instead, the progress of the process p is resumed. When m is re-executed, the content of the message it generates is compared to the message generated during the earlier out-of-order execution. If they are the same, no action is taken. Otherwise, an antimessage is sent to annihilate the earlier message, and then the replacement message is sent.

The execution time of a Time Warp simulation with aggressive cancellation, as well as of conservative parallel simulations, is bounded below by the *critical path* [4] (cf. Section 3). Any simulation that achieves this lower bound is called a *conservative-optimal simulation* [18]. The term "conservative-optimality," instead of "optimality," is used because there are cases in which Time Warp simulations with lazy cancellation outperform the conservative-optimal simulation.

This paper studies Time Warp rollback mechanisms. We begin by discussing the general tradeoffs between aggressive and lazy cancellation, and by defining a conservative-optimal simulation for comparative purposes.

Within the framework of aggressive cancellation, we develop algorithms and evaluate performance for various queueing system simulations, and analyze the rollback behavior of non-FCFS tandem systems.

Using a metric called the *sensitivity of output message*, we study the lazy cancellation mechanism. We observe that both aggressive and lazy cancellation work well for a process with a small simulated load intensity.

Finally, an analytical model is given to analyze *message preemption*, an important factor that affects the performance of a rollback mechanism. Our study shows that message preemption has a significant effect on performance when (i) the processor is highly utilized, (ii) the execution times of messages have high variance, and (iii) rollbacks occur frequently.

2 Anachronous Phenomena of Time Warp

The behavior of Time Warp simulation is difficult to analyze. Besides the normal messages (called *true messages*) executed in the simulation, Time Warp generates two other types of messages: *false messages* and *antimessages*. False messages are created by *anachronism* (out-of-order executions of messages). Antimessages are created to annihilate false messages.

Definition 2.1: Consider a Time Warp simulation. A message is said to be *true* if execution of the message has an effect on the simulation. A message is said to be *false* if execution of the message has no effect on the simulation. A true message is possibly rolled back several times, but the effect of its *final execution* is preserved. A false message is possibly rolled back several times, and is eventually *annihilated* by an antimessage.

Another important property of Time Warp is that rollbacks may *propagate*. Rollback propagation is initiated by a rollback due to out-of-order message execution in a process (the rollback is called the *root* of the propagation). This rollback may result in antimessages being sent to other processes, which in turn cause the rollback of these processes. The propagation eventually terminates.

The executions of false messages and out-of-order true messages cause two anachronous phenomena in Time Warp simulations [18]:

Phenomenon 2.1. It is possible that a true message will be sent prematurely for the wrong reasons [12]. The lazy cancellation mechanism was developed to take advantage of this phenomenon. (In Definition 1.2, if the earlier message and the later message are the same, then the earlier message is a "true" message sent for wrong reason, and in effect, the message is correctly executed ahead of its scheduled time.)

Phenomenon 2.2. It is possible that a correct computation will be delayed (either rolled back or blocked) by a false message. If lazy cancellation is used, this phenomenon is heightened. In Definition 1.2, assume that the earlier message is a false message (i.e., the earlier message and the later message are not the same), and it rolls back a true message m in another process. In aggressive cancellation, the computation for m is resumed immediately after the earlier message is rolled back. On the other hand, if lazy cancellation is used, the re-computation for m is delayed until the later message is scheduled.

It should be obvious that while lazy cancellation may improve the performance of Time Warp by taking advantage of Phenomenon 2.1, it may degrade the performance of Time Warp by heightening the effect of Phenomenon 2.2.

3 Performance Bounds of Time Warp

An important concept called *conservative-optimality* is introduced in this section. Our definition of the conservative-optimal simulation is based on the *critical path analysis* of a simulation [4]: Let Ψ be the set of messages (events) executed by a group of processes in the simulation. Then there are two fundamental sequential constraints:

Constraint 3.1: If two messages m_1 and m_2 are scheduled for the same process p with timestamps $ts(p, m_1)$ and $ts(p, m_2)$ respectively, where $ts(p, m_1) < ts(p, m_2)$, then m_1 must be executed before m_2.

Constraint 3.2: If a message m_1 at process p causes message m_2 to be sent to process q (and therefore $ts(p, m_1) < ts(q, m_2)$), then m_1 must be executed before m_2.

Based on the aforementioned constraints, an *event precedence graph* is built for each parallel simulation. Each vertex of the graph represents an event (message), and each edge represents a communication. A cost which represents the *message execution time* (the execution time of a message) is associated with each vertex. A message sending delay is associated with each edge. Since the graph is feedforward, a maximal weighted path can be found. This path is called the *critical path*, and the cost is the minimal time required to finish the execution of the parallel simulation. This minimal execution time is a lower bound for any conservative parallel simulation as well as for the Time Warp simulation with AC. Any simulation that achieves this lower bound is called a *conservative-optimal simulation*.

Somewhat counter-intuitively, a Time Warp simulation with LC may outperform the conservative-optimal simulation. From the discussion in the previous section, LC does not necessarily obey Constraint 3.2. Thus, an LC simulation may outperform the conservative-optimal simulation. The theoretical bounds of execution times for LC can be derived by relaxing Constraint 3.2. Let P be the set of processes in the simulation, Ψ_p be the set of true messages executed by the process $p \in P$, and $t_{exe}(m)$ be the execution time of message m. Then a lower bound Γ_{LC} for the execution time of an LC simulation is

$$\Gamma_{LC} = \max_{p \in P} \left(\sum_{m \in \Psi_p} t_{exe}(m) \right)$$

Let Γ_{AC} be the execution time of the the conservative-optimal simulation, then Γ_{AC} is a lower bound for the corresponding AC simulation. Although $\Gamma_{LC} \leq \Gamma_{AC}$, studies [22] report examples where AC outperforms LC (cf. the discussion for Phenomenon 2.2).

4 Aggressive Cancellation

This section discusses aggressive cancellation. Section 4.1 derives some properties of AC simulations for FCFS tandem systems. Section 4.2 proposes a conservative optimal AC algorithm for simulating stochastic FCFS queueing networks. Section 4.3 analyzes the performance of AC for simulating non-FCFS tandem systems. We first introduce some assumptions:

Assumption 4.1: Each process in the simulation is assigned to a dedicated processor.

Assumption 4.2: Execution of a message m_1 is rolled back immediately upon the arrival of a message m_2 if m_1's timestamp is larger than m_2's. This is called *message preemption*. Message preemption is considered as a requirement for Time Warp. We will elaborate more on message preemption in Section 6.

Assumption 4.3: The Time Warp operational overhead (such as state saving/restoration, global virtual time calculation) is 0.

The operational overhead mentioned in Assumption 4.3 merits further discussion. If we assume that the global virtual time calculation is done in the background, then the main overhead of the Time Warp mechanism is due to the domino effect of antimessage transmissions instead of the operational overhead described above. Thus, Assumption 4.3 is reasonable when message execution time and message sending delay are long, or the size of the process state is small. Several techniques [7, 8, 12, 17] have been developed to efficiently reduce the overhead of the Time Warp operations.

It is important to know the effect of rollback propagation; it is also important to know when an AC simulation is conservative-optimal. The following two lemmas provide such information. Denote $t(p, m)$ as the real time when a message m is received (and thus executed) by a process p.

Lemma 4.1: Consider an AC simulation. Let rollback Rb_p be the root of a rollback propagation. If Rb_p does not roll back any correct computation, then the subsequent rollbacks in this rollback propagation do not roll back any correct computation.

Proof: Let m be the message that causes the rollback Rb_p. Let S be the set of messages being rolled back. If every $m' \in S$ is a false message, then its output messages are also false. The rollback propagation rooted at Rb_p that eliminates m''s effect does not roll back any correct computation. If there exists a true message $m' \in S$, then there are two possibilities: (i) If m is a true message then $ts(p, m') > ts(p, m)$. Any effect of m' should occur after m's execution; (ii) if m is a false message, then p will receive a true message m'' such that $ts(p, m'') \leq ts(p, m)$ (otherwise, m rolls back correct computation, a contradiction). In both cases, the effect of m' before m's arrival is not correct. That is, the rollback propagation only eliminates incorrect computation. ∎

Lemma 4.1 states that no matter how long a rollback propagation is, it does not degrade the progress of the AC simulation if the root of the propagation does not rollback correct computation.

Lemma 4.2: (cf. [18]) Consider an AC simulation. Let m be any false message sent from a process p to another process q. Suppose that m is annihilated at real time t. The simulation is said to satisfy the *sufficient conservative-optimal condition* if and only if there exists a message m' sent

from p to q after t such that $ts(q,m') < ts(q,m)$. An AC simulation is conservative-optimal if it satisfies the above condition. Note that this condition is sufficient but not necessary.

4.1 FCFS Tandem Systems

For some simulation applications, no rollback occurs during a Time Warp simulation (for either AC or LC). An example is FCFS tandem systems, which consist of N FCFS processes in tandem. Each process receives a job (an input message), simulates a service time, then sends the job (as an output message) to the next process. In other words, the execution of an input message schedules exactly one output message. (Note that the following discussion generalizes for the case in which various numbers of output messages are scheduled.) Observation 4.1 and Observation 4.2 below show that Time Warp simulation of a FCFS tandem system does not create any rollback.

Observation 4.1: Consider a stage (a process) in the FCFS tandem system. If all input messages are received in non-decreasing timestamp order, then all output messages are sent in non-decreasing order.

Based on Observation 4.1, the following observation can be proved easily by induction:

Observation 4.2: For every stage in the FCFS tandem system, all input messages are received in non-decreasing timestamp order, and all output messages are sent in non-decreasing timestamp order. *In other words, no rollback occurs in the system.*

Observation 4.2 can be generalized as follows:

Corollary 4.1: No rollback occurs in the Time Warp simulation of a feedforward FCFS system where every process has exactly one input channel.

If the state saving overhead is negligible, then the Time Warp simulation of the systems described in Corollary 4.1 is optimal. We note that the same performance can be achieved if the conservative method such as Chandy-Misra approach is used. (Performance figures were reported in [1], [15], and [21].)

Consider an FCFS system with an arbitrary network topology. If every message travels through the network in the same simple path, then the Time Warp simulation is optimal. (Note that the parallel simulation based on the conservative method may not achieve the same performance.) If the message travel patterns show some *locality*, then rollbacks may occur only when the path locality changes.

4.2 Stochastic FCFS Queueing Networks

This subsection presents a conservative-optimal AC technique for simulating stochastic FCFS queueing networks. Nicol [19] developed an efficient conservative parallel simulation technique for stochastic FCFS queueing networks. However, due to the limitations of the conservative approach, this technique requires static network topology, and may not be feasible/efficient for a network with dynamic network topology (in such a case, the conservative approach must assume that the network is fully connected).

Message	m_0	m_1	m_2	m_3	m_4	m_5
Arrival time	ts_0	ts_1	ts_2	ts_3	ts_4	ts_5
Service time	s_0	s_1	s_2	s_3	s_4	s_5
Next destination	r_0	r_1	r_2	r_3	r_4	r_5
Departure time	d_0	d_1	d_2	d_3	d_4	d_5

Figure 1: An example of the input/output queue.

Message	m_0	m	m_1	m_2	m_3	m_4	m_5
Arrival time	ts_0	ts	ts_1	ts_2	ts_3	ts_4	ts_5
Service time	s_0	s_1	s_2	s_3	s_4	s_5	s_6
Next destination	r_0	r_1	r_2	r_3	r_4	r_5	r_6
Departure time	d'_0	d'_1	d'_2	d'_3	d'_4	d'_5	d'_6

Figure 2: The input/output queue after the arrive of the message m.

We propose an AC approach which satisfies the sufficient conservative-optimal condition given in Lemma 4.2. The success of our technique (as with Nicol's technique) is based on the stochastic property of the simulated service times and the routing pattern. The algorithm is described as follows: Figure 1 shows the data structure of the input/output queue of a process (since each input message schedules exactly one output message, the input queue and output queue are combined into the same data structure for simplicity). Note that only those fields related to our algorithm are described in Figure 1. Other fields used in Time Warp simulation are not shown. The ith entry represents that a job arrives at the process with timestamp ts_i [3]. The job requires simulated service time (timestamp interval) s_i, and departs with timestamp d_i to process p_{r_i}.[4] Suppose that the local clock of the process is ts_5 (i.e., messages m_0, m_1, m_2, m_3, m_4 and m_5 have been processed). Our technique works as follows: When a message m with timestamp ts arrives, a rollback occurs. There are two possibilities:

Case I. The message m is a positive message (and for example, $ts_0 \leq ts < ts_1$): The messages m_1, m_2, m_3, m_4 and m_5 are rolled back, and the new input/output queue is shown in Figure 2. Instead of generating a new simulated service time and destination for the message m, the simulated service time s_1 and destination p_{r_1} are assigned to m. Similarly, we assign s_{i+1} and $p_{r_{i+1}}$ to m_i. Finally, the last message in the queue is assigned a new generated service time

[3] ts_i is an abbreviation for $ts(p, m_i)$.

[4] d_i is an abbreviation for $ts(p_{r_i}, m_i)$.

Message	m_0	m_2	m_3	m_4	m_5
Arrival time	ts_0	ts_2	ts_3	ts_4	ts_5
Service time	s_0	s_2	s_3	s_4	s_5
Next destination	r_0	r_2	r_3	r_4	r_5
Departure time	d'_0	d'_2	d'_3	d'_4	d'_5

Figure 3: The input/output queue after m_1 is annihilated.

and the next destination (e.g., m_5 in Figure 2 is assigned s_6 and p_{r_6}). The departure times are recomputed.

Case II. The message m is an anti-message (for example, m is the anti-message of m_1, and its timestamp is ts_1): Remove m_1's entry, and recompute the departure times for m_2, m_3, m_4, and m_5 (cf. Figure 3).

Statistically, the result of simulation is the same as other simulation approaches for stochastic FCFS queueing networks.

Lemma 4.3: Our technique for stochastic FCFS queueing network simulations is conservative-optimal.

Proof: It suffices to prove that the sufficient conservative-optimal condition is satisfied, that is, if a process p sends a false message m to another process q, then at a later time p will send another message m' to q such that $ts(m', q) < ts(m, q)$.

Consider Case I mentioned above (cf. Figures 1 and 2). The sufficient conservative-optimal condition is satisfied if d_i' (in Figure 2) is less than d_i (in Figure 1, where $1 \leq i \leq 5$ in this case). Since

$$d_i = \max(d_{i-1}, ts_i) + s_i \quad \text{and} \quad d_i' = \max(d_{i-1}', ts_{i-1}) + s_i$$

and because $ts_{i-1} < ts_i$, we have

$$d_i' < d_i \quad \text{if} \quad d_{i-1}' < a_{i-1}$$

Since $d_1' < d_1$ (because $d_0 = d_0'$, and $ts < ts_1$), from an inductive proof, we have $d_i' < d_i$. Similarly, we can prove that $d_i' < d_i$ in Case II. ∎

Lemma 4.3 showed that, statistically, our technique always outperforms every conservative parallel simulation approach. Besides, no restriction about network topology is required (dynamic or reconfigurable topologies are usually assumed for applications such as communication network simulation). Finally, our technique is simple. It does not change the AC mechanism.

4.3 Non-FCFS Tandem Systems

This subsection analyzes the rollback behavior of non-FCFS tandem systems. This class of systems is chosen for study because of its mathematical tractability and because we expect that the path locality mentioned in Section 4.1 exists, and this study may shed light on the local behavior of general systems.

In a non-FCFS tandem system, rollbacks may occur because of out-of-order messages sent from a process. To illustrate this effect, consider a tandem system in which each process represents a station with an infinite number of servers. (We note that any non-FCFS process can be analogized as a $G/G/\infty$ queue.) For convenience, assume that the simulated service times of the servers in the first process have an identical exponential distribution with parameter μ_1, and the simulated job arrival stream of the first process is a Poisson process with parameter λ_f. Let m_i be the ith arrival job, and p_2 be the second process. It is possible that $ts(p_2, m_i) \leq ts(p_2, m_j)$ where $i > j$.

λ_f/μ_1	$P(0)$	$P(1)$	$P(2)$	$P(3)$	$P(4)$	$P(5)$
0.1	0.9545	0.0454	0.0001	0.0000	0.0000	0.0000
0.2	0.9162	0.0833	0.0005	0.0000	0.0000	0.0000
0.3	0.8831	0.1154	0.0015	0.0000	0.0000	0.0000
0.4	0.8538	0.1429	0.0032	0.0001	0.0000	0.0000
0.5	0.8274	0.1667	0.0056	0.0003	0.0000	0.0000
0.6	0.8032	0.1875	0.0087	0.0006	0.0000	0.0000
0.7	0.7805	0.2059	0.0125	0.0010	0.0001	0.0000
0.8	0.7590	0.2222	0.0169	0.0017	0.0002	0.0000
0.9	0.7383	0.2368	0.0221	0.0025	0.0003	0.0000

Figure 4: $P(n)$: The probability that m_i is rolled back by m_{i+n} at the second process.

However, we note that $t(p_2, m_i) > t(p_2, m_j)$. The message (job) m_i is rolled back by m_j at p_2, if $ts(p_2, m_i) < ts(p_2, m_k)$ and $ts(p_2, m_i) > ts(p_2, m_j)$, where $i < k < j$. The probability, $P(n)$, that m_i is rolled back by m_{i+n} at p_2 is (cf. Appendix A)

$$P(n) = \begin{cases} \dfrac{\lambda_f}{2(\mu_1 + \lambda_f)}, & n = 1 \\ \dfrac{\mu_1^2 \lambda_f^{n+1}}{[2\mu_1 + (n-2)\lambda_f][3\mu_1 + (n-2)\lambda_f][2\mu_1 + (n-1)\lambda_f] \prod_{k=1}^{n-1} (k\mu_1 + \lambda_f)}, & n \geq 2 \end{cases} \quad (1)$$

Denote $P(0)$ as the probability that m_i is never rolled back at the second process, then

$$P(0) = 1 - \sum_{n=1}^{\infty} P(n)$$

The probability $P(n)$ is shown in Figure 4. We observe that (i) when $\dfrac{\lambda_f}{\mu_1}$ is small, the probability that a rollback occurs at p_2 is very small; and (ii) for $n > 1, P(n) < 0.03$. This implies that when an out-of-order message execution occurs, it is usually detected promptly.

Consider the rollback effect at the second process p_2. Suppose that p_2 receives N messages from p_1 during the simulation. Arrange these messages, m_k, such that $ts(p_2, m_k) \leq ts(p_2, m_l)$, where $1 \leq k \leq l \leq N$. Since p_1 may send out-of-order messages, it is possible that $t(p_2, m_k) > t(p_2, m_l)$, where $k < l$. In that case, a rollback occurs at real time $t(p_2, m_k)$. Since m_{k+n} is a true message executed out-of-order, it is possible that m_{k+n}'s execution results in a false message m'_{k+n}, which rolls back or delays the correct computation in the subsequent process. Let p_3 be the third process. If $ts(p_3, m_k) < ts(p_3, m'_{k+n})$ for every rollback in p_2, then Lemma 4.2 guarantees that m'_{k+n} never rolls back or delays correct computation in p_3. Suppose that the simulated service times of a server in p_2 is exponentially distributed with parameter μ_2. Let $\overline{\Phi(n)}$ be the probability that $ts(p_3, m_k) > ts(p_3, m'_{k+n})$, where m'_{k+n} is a false message due to the out-of-order execution of m_{k+n} at p_2. Then from Appendix B

$$\overline{\Phi(n)} = \frac{1}{2} \left(\frac{\lambda_f}{\lambda_f + \mu_2} \right)^n$$

$\frac{\lambda_f}{\mu_1}$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Φ	0.998	0.993	0.987	0.979	0.972	0.964	0.957	0.949	0.941

Figure 5: The probability, Φ, that a false message does not roll back correct computation.

where λ_f is the rate of the simulated job arrival stream to the first process. The probability Φ that a false message does not roll back or delay correct computation in p_3 is

$$\Phi = 1 - \sum_{n=1}^{\infty} P(n)\overline{\Phi(n)} \tag{2}$$

Based on (2), Figure 5 shows Φ as a function of λ_f and $\mu_1 = \mu_2$.

Figure 5 indicates that when $\lambda_f > \mu_2$, the probability that a false message rolls back correct computation at a subsequent process is very small. We also note that no correct computation is rolled back at p_2 (cf. [18] for detailed proof).

The analysis in this subsection shows that when $\lambda_f > \mu_1$, the rollback effect is not significant. This implies that good performance of an AC simulation can be achieved if the activities inside a process are more frequent than the activities outside that process.

5 Lazy Cancellation

It might seem natural to consider LC as merely an "optimization" of AC. However, Jefferson [12] argued that LC is the "correct" mechanism, instead of an optimization. In other words, AC should not be used even if it happens to perform better under some circumstances. The reason is two-fold.

1. Lazy cancellation allows outperforming the conservative-optimal simulation, whereas AC does not.

2. Lazy cancellation works better, and is *not* equivalent to AC, when messages can be sent with receive time equal to send time. There are cases when a cycle of such messages yields infinite rollback with AC, but ordinary correct progress with LC. This is a *semantic* difference.

The performance of LC depends on how events affect each other. Suppose that a message m arrives at a process p after another message m' has been executed, where $ts(m',p) > ts(m,p)$, and the execution of m' causes an output message m'' to be sent to another process. The success of LC is based on the idea that the scheduling of m'' may not be affected by m, and no annihilation is required. To measure the performance of LC, we introduce a metric called the *sensitivity of output message*.

Definition 5.1: Let m_i be an input message of a process p, the execution of which results in the scheduling of an output message m_o. The *sensitivity* of m_o, θ, is defined as the probability that m_o is affected by any message m where $ts(m,p) < ts(m_o,p)$.

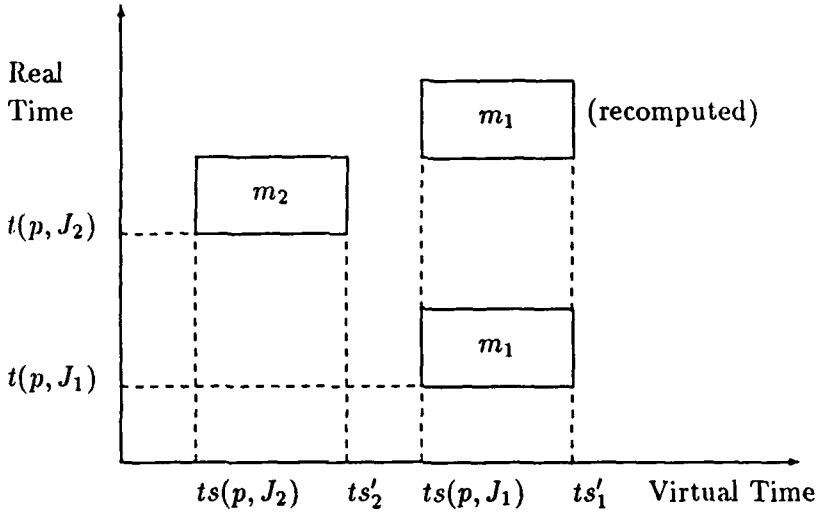


Figure 6: Real-virtual time diagram of an FCFS process – Case I.

When a rollback occur, θ is the probability that LC fails. In the remainder of this section, we analyze θ by examples.

5.1 FCFS Processes

Consider an FCFS process p in a system. If p is isolated from the rest of the system, it can be modeled as a single queue. The behavior of p is shown in Figure 6. At real time $t(p, J_1)$, a job J_1 arrives at p with timestamp $ts(p, J_1)$. Let the simulated service time of J_1 be Δs_1. Then J_1 departs with timestamp $ts'_1 = ts(p, J_1) + \Delta s_1$. At real time $t(p, J_2) > t(p, J_1)$, a job J_2 arrives at p with timestamp $ts(p, J_2) < ts(p, J_1)$. The simulated service time of J_2 is Δs_2. For an AC simulation, the event (an output message) that "J_1 departs with timestamp ts'_1" (we call it event A) must be annihilated immediately. However, it is possible that $ts(p, J_2) + \Delta s_2 \leq ts(p, J_1)$. In such a case, J_1 still departs with timestamp $ts(p, J_1) + \Delta s_1$, and no annihilation is required. In other words, the departure of J_1 is not affected by J_2, and the execution orders of these two events are not pertinent. Thus LC never annihilated an event (output message) A if the incoming events do not affect A. On the other hand, if $ts'_2 > ts(p, J_1)$, (cf. Figure 7) then J_1 departs with timestamp $ts'_1 = ts'_2 + \Delta s_1$ instead of $ts(p, J_1) + \Delta s_1$. In this case, LC delays the annihilation of the "wrong event", which represents the departure of J_1 with timestamp $ts(p, J_1) + \Delta s_1$. (This wrong message is annihilated at real time t_3 (cf. Figure 7)).

In this example, the sensitivity of output message, θ, can be defined as the probability that the queue length in the process is non-zero (and the departure time of a job is determined by the jobs in the queue). For a $G/G/1$ queue, $\theta = \lambda E[s]$ where λ is the simulated job arrival rate, and $E[s]$

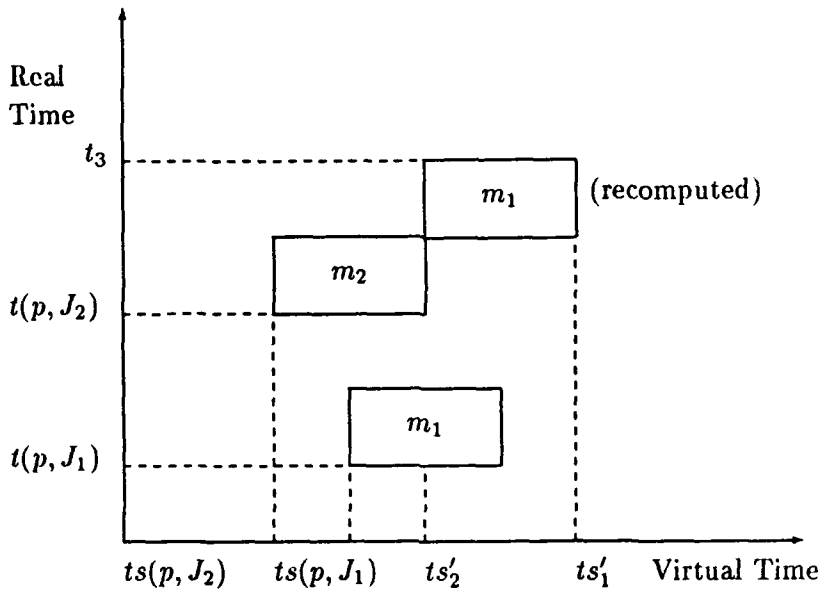


Figure 7: Real-virtual time diagram of an FCFS process – Case II.

is the expected simulated job service time. If there are m servers in the process (i.e., we have a $G/G/k$ queue), then

$$\theta = \frac{\lambda E[s]}{k} \tag{3}$$

The derivation of (3) is independent of architecture parameters (i.e., message sending delay, message execution time), and is independent of the simulated arrival time distribution and the timestamp increment (the simulated service time) distribution. Figure 8 plots θ as an function of $\lambda E[s]$ and k.

In this type of application, LC works well when the simulated load traffic is small because the simulated queueing effect is small, and the probability that a departure event is affected by other events is small. In the extreme case that the simulated queue length is 0 (i.e., $\theta = 0$), any job J arriving at p with timestamp ts always departs with timestamp $ts + \Delta s$. It is obvious that no departure event is affected by any arrival event, and no annihilations are required in the simulation.

Note that similar techniques can be used to evaluate θ in many applications. Examples are listed below.

- A multi-class job system with priority.
- The colliding pucks simulation [10] in which a process represents a region, and input/output messages represent the pucks moving in and out of the region. The move-out direction of a puck depends on whether it collides with other pucks (if there is no collision, then it moves out in the same direction as it moves in). Thus, θ is the probability that a puck collides before it moves out an region.

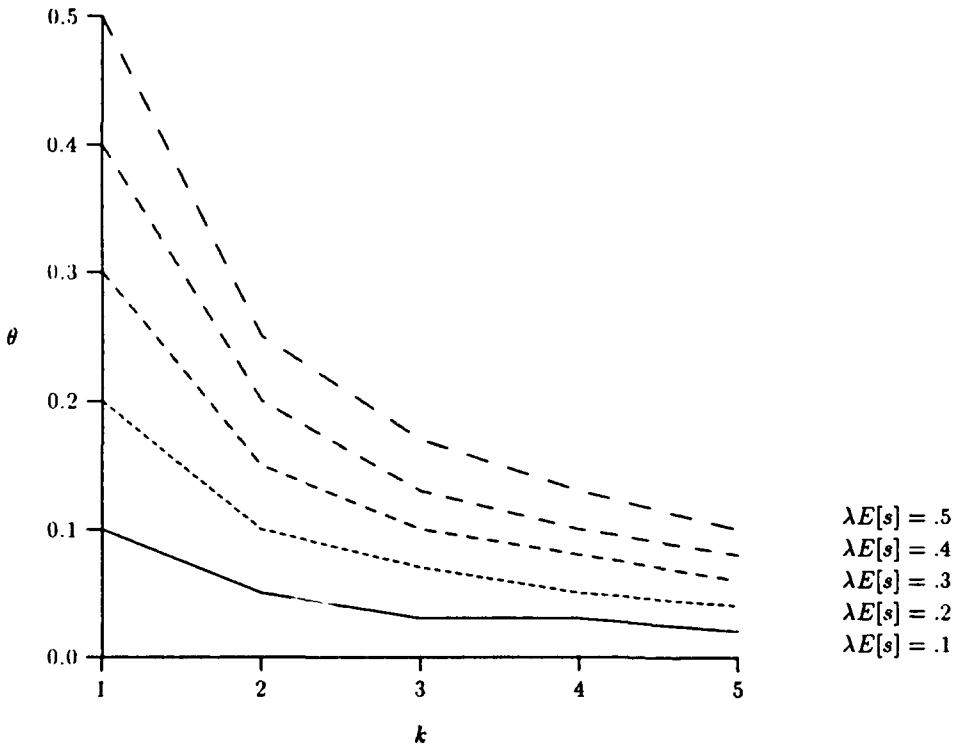


Figure 8: The sensitivity of output messages in Example 1.

- The war game simulation [13] in which a process represents a region, and input/output messages represent objects moveing in/out of the region. For example, the movement of an object (say, a tank) depends on whether it is hit by a missile. In this case, θ is the probability that an object is hit by a missile.

5.2 The Producer/Co umer Model

Consider a producer/consumer model. In this type of application (for example, the Antopia model [6]), the jobs (messages) that arrive at a process are producers, and the process is a consumer (or vice versa). The process p has a storage of capacity N. A job carries an item, and leaves it in p's storage slot, then departs with some timestamp increment. If the storage is full, the job is blocked at p until there are empty slots. Consider the following scenario. At real time $t(p, J_1)$, a job J_1 arrives at p with timestamp $ts(p, J_1)$. Suppose that the number of empty slots is $n \leq N$. Then J_1 departs with timestamp $ts(p, J_1) + \Delta s_1$

(we call it event A), and the number of empty slots is $n-1$. At real time $t(p, J_2) > t(p, J_1)$, a job J_2 arrives at p with timestamp $ts(p, J_2) < ts(p, J_1)$. If $n - 1 > 0$, then the event that represents J_1's departure (we call it event B) is not affected by the arrival of J_1. Under LC, B is not annihilated after A is recomputed. If $n - 1 = 0$ and p does not consume any item in the timestamp interval $[ts(p, J_2), ts(p, J_1)]$, then LC delays the annihilation of B. (If AC is used, then an antimessage is sent at real time $t(p, J_2)$ to annihilate B.)

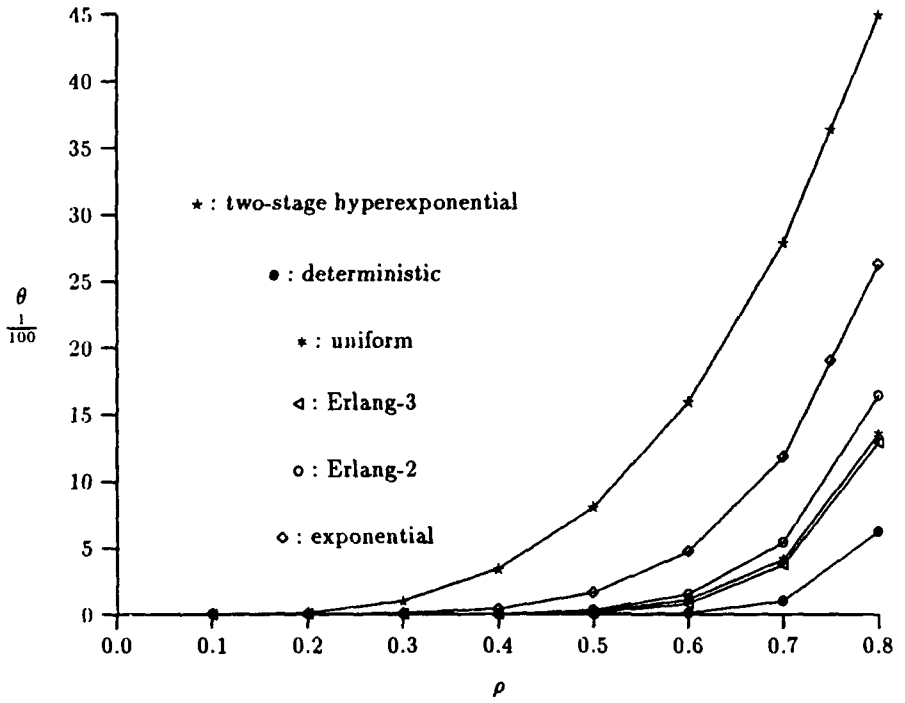


Figure 9: The relationship between θ and ρ ($N = 5$): The producer/consumer model.

To derive θ, we view the producer/consumer model as a $G/G/1$ queue where λ is the simulated job arrival rate μ is the simulated service rate and π_i is the probability that the queue length is i. In the producer/consumer model, π_i is interpreted as the probability that there are i items in the storage slots. When $i > N$, π_i represents the probability that the storage is full, and there are $i - N$ jobs waiting for empty slots. Thus, the output message sensitivity, θ, is expressed as

$$\theta = \sum_{i=N+1}^{\infty} \pi_i \tag{4}$$

Note that π_j cannot be easily obtained by analytical methods unless either the simulated job interarrival times or the simulated service times are exponentially distributed. The values of θ for $G/M/1$ queues are derived in Appendix C, and Figures 9, 10, and 11 plot the relationship among θ, ρ, and N. When $\rho \leq 0.5$, we have $\theta \simeq 0$ (except for two-stage hyperexponential distribution), and good performance of LC can be expected. In both Examples 1 and 2, we observe that for a process, θ is small when the arrival activities are less frequent than the activities inside the process.

6 The Effect of Message Preemption

In a Time Warp simulation, it is possible that at the arrival of a message m, a process p is executing another message m' such that $ts(m,p) < ts(m',p)$. The message m *preempts* m' if p stops executing m' (i.e., a rollback occurs immediately at the arrival of m; cf. Assumption 4.2). Message preemption is considered necessary for the correctness of the Time Warp mechanism. For instance, without

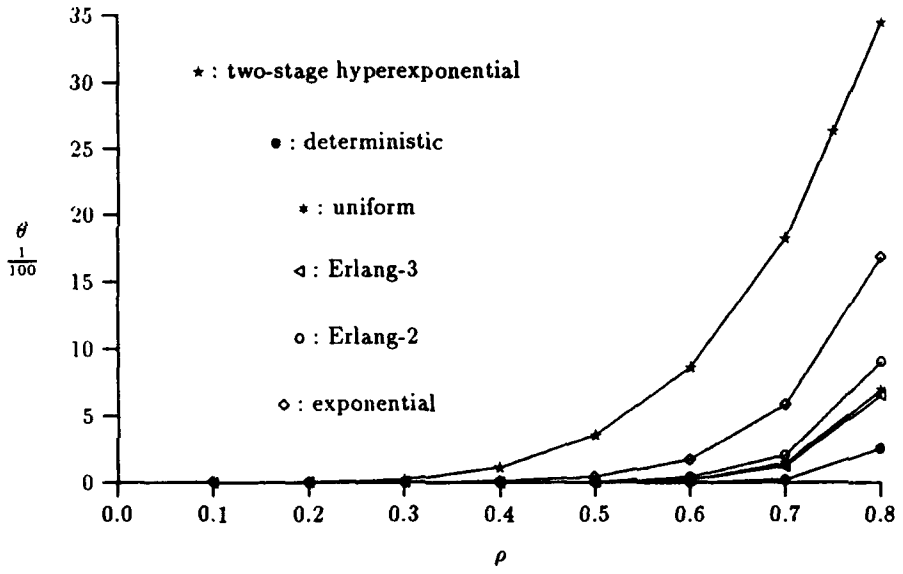


Figure 10: The relationship between θ and ρ ($N = 7$): The producer/consumer model.

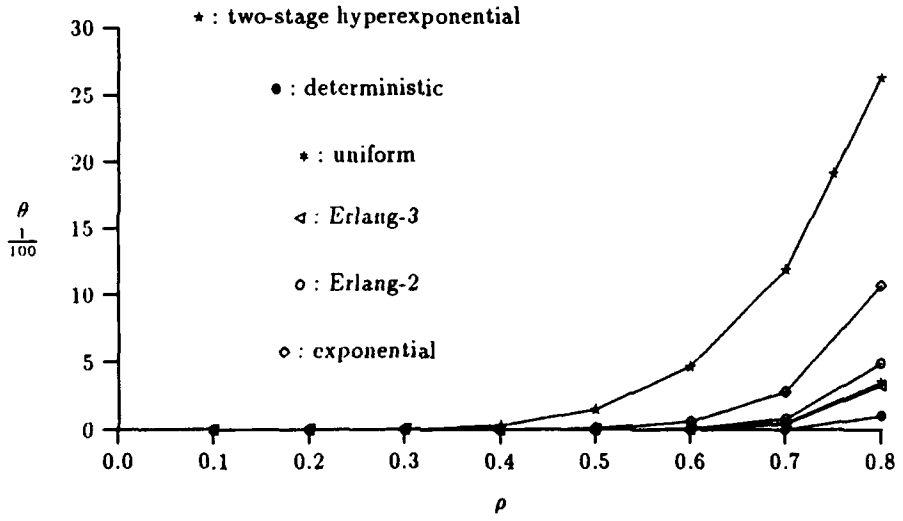


Figure 11: The relationship between θ and ρ ($N = 9$): The producer/consumer model.

message preemption the computation of m' may be on the wrong path and may be trapped in an infinite loop.

Many current implementations of Time Warp [1, 2, 7] do not support message preemption. These implementations assume that incorrect computation never causes a correctness problem. This section shows that even when there is no correctness problem, message preemption may be still desirable for the purpose of performance.

Suppose that no message preemption is allowed in the Time Warp simulation. Consider a process p in which the message execution time has the density function $f(t)$ with mean $\frac{1}{\mu}$, and the second moment M_2. Let the message arrival rate be λ.

When a message arrives at process p, the probability that p is busy is $\frac{\lambda}{\mu}$. The busy times of p are considered as renewal periods where the renewal process is observed by the arrival message m at random over a long time. Let the observation point be t_0 (i.e., the message m arrives at p at real time t_0) and the two renewal points between which it falls be t_1 and t_2. The remainder of the observed renewal period, $\Delta t = t_2 - t_0$, is the time between the arrival time of m and the time when p is available for the next computation. From the standard technique [14], $\Delta t = \frac{\mu M_2}{2}$. If the computation done in $[t_1, t_2]$ is not correct, then Δt is the extra time that m waits (With message preemption, m is immediately executed at real time t_0). Let γ be the probability that a message arrives at p with a timestamp smaller than p's local clock. Then, without message preemption, the expected time $E[t_w]$ that an arrival message waits for an incorrect computation to finish is

$$E[t_w] = \gamma \left(\frac{\lambda}{\mu}\right) \Delta t = \gamma \left(\frac{\lambda}{\mu}\right) \frac{\mu M_2}{2} = \frac{\gamma \lambda M_2}{2}$$

Let $t = t_2 - t_1$ be the message execution time, $\rho_r = \frac{\lambda}{\mu}$, and $\beta = \mu^2 M_2$. Define $\delta = \frac{E[t_w]}{E[t]}$, then

$$\delta = E[t_w]\mu = \frac{\gamma \beta \rho_r}{2}$$

The parameter γ depends on the behavior of the simulation application. Experimental studies reported γ values of 0.32-0.45 (with 32 processors) [10], 0.51 (with 32 processors) [20], and 0.54 (with 64 processors) [24]. We note that when the number of processors increases, γ increases (If one processor is used in the simulation, then $\gamma = 0$). The parameter β is always larger than 1 (from the definition of the second moment that $M_2 \geq \frac{1}{\mu^2}$). The experimental study [20] reported a β value of 1.87. The parameter ρ_r represents the utilization of the processor. Thus, $0 \leq \rho_r \leq 1$. Experience [12] showed high processor utilization of Time Warp simulation. Figure 12 shows the relationship among β, ρ_r, and δ, the expected time that a message waits because of non-preemption (assuming that $\gamma = 0.45$). We observed that message preemption has a significant effect on performance when (i) the processor is highly utilized, (ii) the execution times for messages have high variance, and (iii) rollbacks occur frequently.

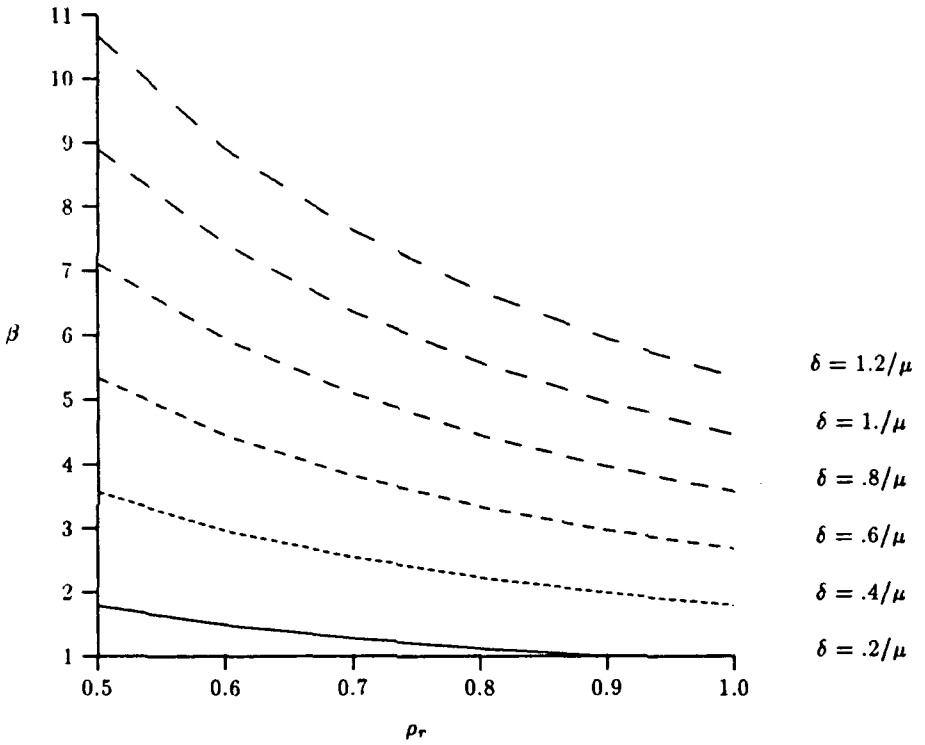


Figure 12: The relationship among β, ρ_r, and δ ($\gamma = 0.45$).

7 Conclusion

This paper has studied the rollback mechanisms of Time Warp simulation. We offered some observations about aggressive cancellation (AC). Based on these observations, we proposed a technique for stochastic FCFS queueing network simulations that outperforms any conservative parallel simulation approach. We also gave an analysis of the rollback behavior of non-FCFS tandem system simulations. A metric called the *sensitivity of output message* was used to study lazy cancellation (LC).

From the examples we studied, we observed that both AC and LC work well for a process in which the simulated arrival rate is significantly less than the simulated service rate. This is different from what we observed in the lookahead exploration for the Chandy-Misra simulation [16]. To exploit a larger lookahead for the Chandy-Misra simulation, the simulated arrival rate must be close to the simulated service rate (i.e., good performance is achieved when the simulated system is nearly saturated). Since most systems under study are not saturated, they favor Time Warp simulations. We also noted that a parallel simulation of a lightly-loaded system does not mean that less parallelism can be exploited. The simulated service time of an event is not directly related to the execution time of that event. When an event is simulated in great detail, high processor utilization can be expected.

Finally, we analyzed the impact of message preemption on the performance of rollback. We observed that message preemption has a significant effect on performance when (i) the processor is

highly utilized, (ii) the execution times for messages have high variance, and (iii) rollbacks occur frequently.

References

[1] Abrams, M. The Object Library for Parallel Simulation (OLPS). *Proc. 1988 Winter Simulation Conference*, 1988.

[2] Baezner, D. and Lomow, G. Sim++: The Transition to Distributed Simulation. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[3] Berry, O. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* PhD thesis, University of Southern California, May 1986.

[4] Berry, O. and Jefferson, D. Critical Path Analysis of Distributed Simulation. *Proc. 1985 SCS Multiconference on Distributed Simulation*, pages 57–60, 1985.

[5] Chandy, K.M. and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[6] Ebling, M., Loreto, M., Presley, M., Wieland, F., and Jefferson, D. An Ant Foraging Model Implemented on the Time Warp Operating System. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 21–26, 1989.

[7] Fujimoto, R.M. Time Warp on a Shared Memory Multiprocessor. Technical Report UUCS-88-021a, Computer Science Department, University of Utah, January 1989.

[8] Fujimoto, R.M., Tsai, J.-J. and Gopalakrishnan, G. Design and Performance of Special Purpose Hardware for Time Warp. *Proc. 15th Symp. on Computer Architecture*, 1988.

[9] Gafni, A. Rollback Mechanisms for Optimistic Distributed Simulation. *Proc. 1988 SCS Multiconference on Distributed Simulation*, 1988.

[10] Hontalas, P., Beckman, B., Diloreto, M., Blume, L., Reiher, P., Sturdevant, K., Warren, L., Wedel, J., Wieland, F., and Jefferson, D. Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior & Sectoring). *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 3–7, 1989.

[11] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[12] Jefferson, D. Private communication. 1989.

[13] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 77–93, November 1987.

[14] Kleinrock, L. *Queueing Systems: Volume I - Theory.* New York: Wiley, 1976.

[15] Lakshmi, M.S. A Study and Analysis of Performance of Distributed Simulation. Technical Report 87-32, Department of Computer Sciences, The University of Texas at Austin, August 1987.

[16] Lin, Y.-B. and Lazowska, E.D. Exploiting Lookahead in Parallel Simulation. Technical Report 89-10-06, Department of Computer Science and Engineering, University of Washington, 1989.

[17] Lin, Y.-B. and Lazowska, E.D. The Optimal Checkpoint Interval in Time Warp Parallel Simulation. Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, 1989.

[18] Lin, Y.-B. and Lazowska, E.D. Optimality Considerations for "Time Warp" Parallel Simulation. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[19] Nicol, D.M. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 124–137, 1988.

[20] Presley, M., Ebling, M., Wieland, F., Jefferson, D. Benchmarking the Time Warp Operating System with a Computer Network Simulation. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 8–13, 1989.

[21] Reed, D.A., and Malony, A. Parallel Discrete Event Simulation: The Chandy-Misra Approach. *Proc. 1988 SCS Multiconference on Distributed Simulation*, pages 8–13, 1988.

[22] Reiher, P., Fujimoto, R., Bellenot, S., and Jefferson, D. Cancellation Strategies in Optimistic Execution Systems. *Proc. 1990 SCS Multiconference on Distributed Simulation*, 1990.

[23] Wagner, D.B., and Lazowska, E.D. Parallel Simulation of Queueing Networks: Limitations and Potentials. 1989 ACM SIGMETRICS and Performance '89, 1989.

[24] Wieland, F., Hawley, L., Feinberg, A., Loreto, M., Blume, L., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson, D. Distributed Combat Simulation and Time Warp: The Model and Its Performance. *Proc. 1989 SCS Multiconference on Distributed Simulation*, pages 14–20, 1989.

A The Derivation of $P(n)$

This appendix derives $P(n)$, the probability that a message m_i is rolled back by m_{i+n} at process p_2. $P(n)$ is used to analyze non-FCFS tandem systems in Section 4. Assume that the simulated service time (i.e., the timestamp interval) of the servers in p_1 have an identical exponential distribution with parameters μ_1, and the simulated job arrival stream to p_1 is a Poisson process with parameter λ_f. The probability $P(1)$ that m_i is rolled back by m_{i+1} is derived as follows: Let

$$t_0 = ts(p_2, m_i) - ts(p_1, m_i), \quad t_1 = ts(p_1, m_{i+1}) - ts(p_1, m_i),$$

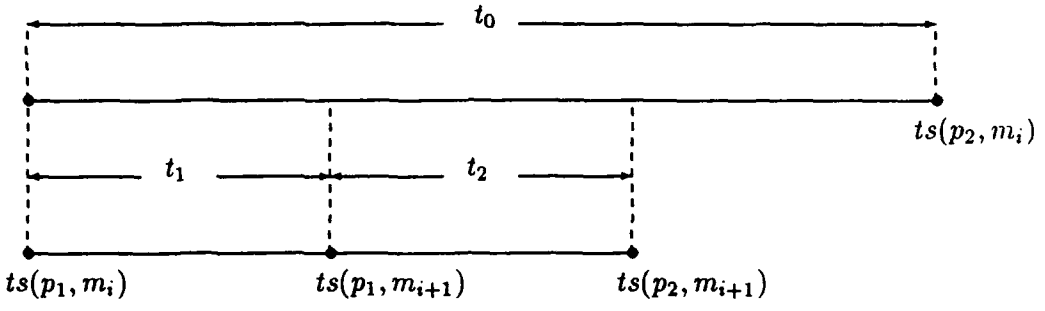


Figure 13: The timing diagrams used in the derivation of $P(1)$.

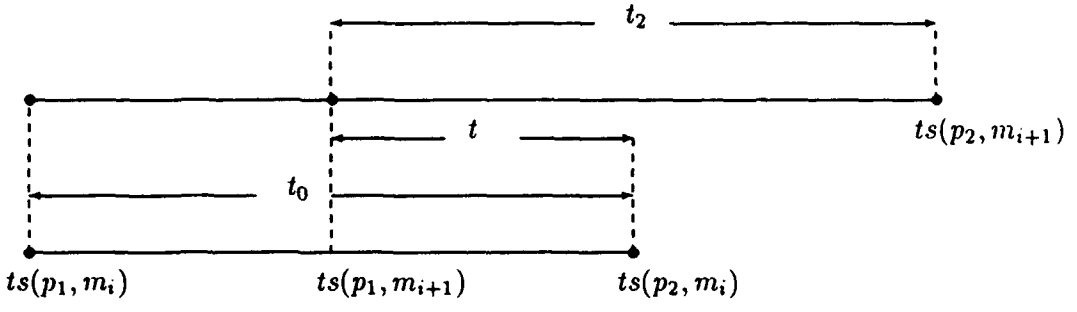


Figure 14: The timing diagrams used in the derivation of $\eta_{t_0}(2, t)$.

$$\text{and} \quad t_2 = ts(p_2, m_{i+1}) - ts(p_1, m_{i+1})$$

The message m_i is rolled back by m_{i+1} if and only if $t_0 - (t_1 + t_2) > 0$ (cf. Figure 13). Both t_0 and t_2 are exponentially distributed with parameter μ_1, and t_1 is exponentially distributed with parameter λ_f. Thus,

$$\begin{aligned} P(1) &= \Pr[t_0 - (t_1 + t_2) > 0] \\ &= \int_{t_1=0}^{\infty} \int_{t_2=0}^{\infty} \int_{t_0=t_1+t_2}^{\infty} \lambda_f e^{-\lambda_f t_1} \mu_1 e^{-\mu_1 t_2} \mu_1 e^{-\mu_1 t_0} dt_0 dt_2 dt_1 \\ &= \frac{1}{2}\left(\frac{\lambda_f}{\lambda_f + \mu_1}\right) \end{aligned} \tag{5}$$

For $n \geq 2$, a probability η_{t_0} is defined to to derive $P(n)$. Let $\eta_{t_0}(n, t)$ be the probability that (i) m_i is not rolled back by m_j, where $i < j < i + n$, (ii) $ts(p_2, m_i) - ts(p_1, m_i) = t_0$, and (iii) $ts(p_2, m_i) - ts(p_1, m_{i+n-1}) = t$. Note that when (i), (ii), and (iii) are satisfied, m_i is rolled back by

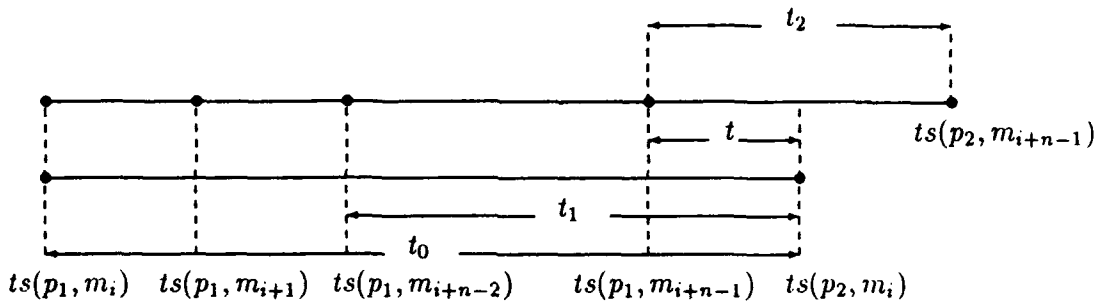


Figure 15: The timing diagrams used in the derivation of $\eta_{t_0}(n,t)$.

m_{i+n} if and only if $ts(p_2, m_{i+n}) - ts(p_1, m_{i+n-1}) < t$ (in other words, $ts(p_2, m_{i+n}) < ts(p_2, m_i)$). The probability $\eta_{t_0}(n,t)$ is derived as follows. Consider the case that $n = 2$. From Figure 14, we have

- $ts(p_2, m_i) - ts(p_1, m_i) = t_0$, which is exponentially distributed with parameter μ_1.
- $ts(p_2, m_i) - ts(p_1, m_{i+1}) = t$. This implies that $ts(p_1, m_{i+1}) - ts(p_1, m_i) = t_0 - t$, which is exponentially distributed with parameter λ_f.
- $ts(p_2, m_{i+1}) - ts(p_1, m_{i+1}) = t_2$, which is exponentially distributed with parameter μ_1.

Thus,

$$\begin{aligned} \eta_{t_0}(2,t) &= \int_{t_2=t}^{\infty} \mu_1 e^{-\mu_1 t_2} \lambda_f e^{-\lambda_f (t_0 - t)} dt_2 \\ &= e^{-\mu_1 t} \lambda_f e^{-\lambda_f (t_0 - t)} \end{aligned} \tag{6}$$

When $n > 2$, $\eta_{t_0}(n,t)$ can be expressed as (cf. Figure 15)

$$\begin{aligned} \eta_{t_0}(n,t) &= \Pr[ts(p_2, m_{i+n-1}) > ts(p_2, m_i)] \int_{t_1=t}^{t_0} \eta_{t_0}(n-1, t_1) \\ &\times \Pr\{t_1 - [ts(p_1, m_{i+n-1}) - ts(p_1, m_{i+n-2})] = t > 0\} dt_1 \\ &= \int_{t_1=t}^{t_0} \int_{t_2=t}^{\infty} \mu_1 e^{-\mu_1 t_2} \eta_{t_0}(n-1, t_1) \lambda_f e^{-\lambda_f (t_1 - t)} dt_2 dt_1 \\ &= \int_{t_1=t}^{t_0} e^{-\mu_1 t} \eta_{t_0}(n-1, t_1) \lambda_f e^{-\lambda_f (t_1 - t)} dt_1 \end{aligned} \tag{7}$$

The recurrence equation (7) solves to yield

$$\eta_{t_0}(n,t) = \begin{cases} e^{-\mu_1 t} \lambda_f e^{-\lambda_f (t_0 - t)}, & n = 2 \\ \dfrac{\lambda_f^n}{\mu_1^{n-1}(n-1)!} (e^{-\mu_1 t} - e^{-\mu_1 t_0})^{n-1} e^{-(\mu_1 - \lambda_f) t} e^{-\lambda_f t_2}, & n > 2 \end{cases}$$

Let $t_3 = ts(p_1, m_{i+n}) - ts(p_1, m_{i+n-1})$ and $t_4 = ts(p_2, m_{i+n}) - ts(p_1, m_{i+n})$, then the probability $P(n)$ that m_i is rolled back by m_{i+n} at p_2 is

$$P(n) = \int_{t_3=0}^{\infty} \int_{t_4=0}^{\infty} \int_{t=t_4+t_3}^{\infty} \int_{t_0=t}^{\infty} \eta_{t_0}(n,t)\mu_1 e^{-\mu_1 t_4} \lambda_f e^{-\lambda_f t_3} dt_0 dt dt_4 dt_3 \tag{8}$$

Integrate the solution of (8) and (5), we have

$$P(n) = \begin{cases} \dfrac{\lambda_f}{2(\mu_1+\lambda_f)}, & n = 1 \\ \dfrac{\mu_1^2 \lambda_f^{n+1}}{[2\mu_1+(n-2)\lambda_f][3\mu_1+(n-2)\lambda_f][2\mu_1+(n-1)\lambda_f]\prod_{k=1}^{n-1}(k\mu_1+\lambda_f)}, & n \geq 2 \end{cases} \tag{9}$$

B The Derivation of $\overline{\Phi(n)}$

This appendix derives the probability $\overline{\Phi(n)}$ that a message m_k is rolled back by a false message m'_{k+n}, where m'_{k+n} is scheduled due to the out-of-order execution of the message m_{k+n}. $\overline{\Phi(n)}$ is used to analyze non-FCFS tandem systems in Section 4. Assume that the simulated service times of a server in the second process p_2 are a random variable Y that has an exponential distribution with parameter μ_2. The simulated interarrival time distribution of p_2 is the same as the interdeparture time distribution of the first process. From the following theorem, the interdeparture time distribution of the first process can be obtained:

Theorem B.1 [Burke's theorem] [14]: The steady-state output of a stable $M/M/k$ queue with input parameter λ_f and service-time parameter μ_1 for each of the k servers is a Poisson process at the same rate λ_f.

Applying Theorem B.1 with $k \to \infty$, the interdeparture times are a random variable X which has an exponential distribution with parameter λ_f. In other words, the interarrival time of n jobs is a random variable $S_n = X_1 + X_2 + ... + X_n$ which has an Erlang distribution with the density function

$$f_n(t) = \frac{\lambda_f}{(n-1)!}(\lambda_f t)^{n-1} e^{-\lambda_f t} \tag{10}$$

Suppose that p_2 receives N messages from p_1 during the simulation. Arrange these messages m_k, m_l, where $1 \leq k \leq l \leq N$, such that $ts(p_2, m_k) \leq ts(p_2, m_l)$. Since p_1 may send out-of-order messages, it is possible that $t(p_2, m_k) > t(p_2, m_l)$, where $k < l$. In that case, a rollback occurs at real time $t(p_2, m_k)$. For $n > 0$, $P(n)$ (cf. (1)) is the probability that m_{k+n} is rolled back by m_k, and the distribution for $t = ts(p_2, m_{k+n}) - ts(p_2, m_k)$ is characterized by the density function $f_n(t)$ given in (10). Since m_{k+n} is a true message executed out-of-order, it is possible that m_{k+n}'s execution results in a false message m'_{k+n}, which rolls back or delays the correct computation in the subsequent processes. Let p_3 be the third process. If $ts(p_3, m_k) < ts(p_3, m'_{k+n})$ for every rollback in p_2, then Lemma 4.3 guarantees that m'_{k+n} never rolls back or delays correct computation in p_3 [18]. Let $\overline{\Phi(n)}$ be the probability that $ts(p_3, m_k) > ts(p_3, m'_{k+n})$, then it can be derived as follows: Let

$$t_1 = ts(p_2, m_{k+n}) - ts(p_2, m_k), \quad t_2 = ts(p_3, m_{k+n}) - ts(p_2, m_{k+n}),$$

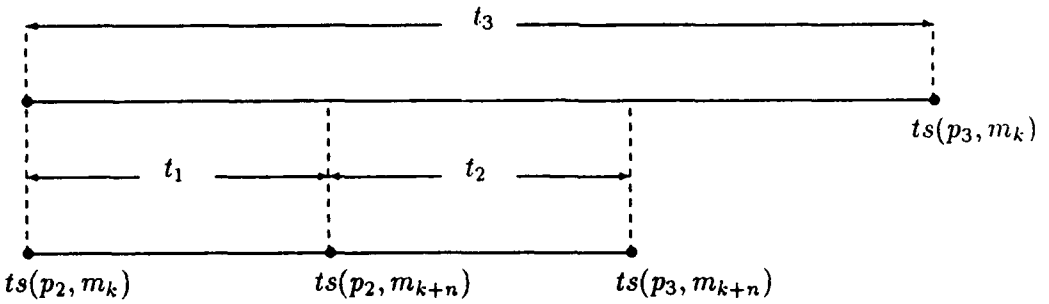


Figure 16: The timing diagrams used in the derivation of $\overline{\Phi(n)}$.

$$\text{and} \quad t_3 = ts(p_3, m_k) - ts(p_2, m_i)$$

then $ts(p_3, m_k) - ts(p_3, m_{k+n}) = t_3 - (t_1 + t_2)$ (cf. Figure 16).

Since the values of t_1 and t_3 are drawn from the random variable Y, and the value of t_2 is drawn from the random variable S_n, the probability that $ts(p_3, m_k) > ts(p_3, m_{k+n})$ can be expressed as

$$\begin{aligned} \overline{\Phi(n)} &= \int_{t_1=0}^{\infty} \int_{t_2=0}^{\infty} \int_{t_3=0}^{t_1+t_2} f_n(t_1) \mu_2 e^{-\mu_2 t_2} \mu_2 e^{-\mu_2 t_3} dt_3 dt_2 dt_1 \\ &= \frac{1}{2} \left(\frac{\lambda_f}{\lambda_f + \mu_2} \right)^n \end{aligned}$$

C The Derivation of θ for Example 2

This appendix derives θ, the sensitivity of output message for the producer/consumer model in Section 5.2. Consider a $G/M/1$ queue. Let π_i be the probability that the queue length is i. It has been shown that [14]

$$\pi_n = \pi_0 (1 - \pi_0)^n, \quad n = 0, 1, 2, ..., \tag{11}$$

and π_0 is the unique solution of the equation

$$1 - \pi_0 = F^*(\mu \pi_0) \tag{12}$$

where $F(t)$ is the distribution of the simulated inter-arrival time intervals, and F^* is the *Laplace-Stieltjes transform* of $F(t)$. Figure 17 lists the solution of (12) for the following distributions: Exponential distribution (M), Erlang-2 distribution (E_2), Erlang-3 distribution (E_3), uniform distribution (U), deterministic distribution (D), and two-stage hyperexponential distribution (H_2).

Distribution	π_0
M	$1-\rho$
E_2	$-2\rho+0.5+\sqrt{2\rho+0.25}$
E_3	$1-\left(\frac{3\rho}{3\rho+\pi_0}\right)^3$
U	$1-\frac{\rho(1-e^{-2\pi_0/\rho})}{2\pi_0}$
D	$1-e^{-\pi_0/\rho}$
H_2	$0.5-\rho+0.5\sqrt{(1-2\rho)^2+16\rho\alpha_1(1-\alpha_1)(1-\rho)}$

Figure 17: The π_0 values for various distributions, where $\rho = \frac{\lambda}{\mu}$, $\alpha_1 = \frac{1}{2}\left[1-\left(\frac{C^2-1}{C^2+1}\right)^{1/2}\right]$, and C^2 is the squared coefficient of variation.

From (11), θ can be expressed as

$$\theta = \sum_{j=N}^{\infty} \pi_j = (1-\pi_0)^{N+1} \tag{13}$$

Substitute π_0 in Figure 17 into (13), the values of θ can be easily obtained. (For E_3, U, and D, the π_0 values are compiled by numerical methods.)

Design of Parallel Computer Memory Interconnects

An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols

M. K. Vernon

Department of Computer Science
University of Wisconsin-Madison
Madison, WI 53706

E. D. Lazowska
J. Zahorjan

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

A number of dynamic cache consistency protocols have been developed for multiprocessors having a shared bus interconnect between processors and shared memory. The relative performance of these protocols has been studied extensively using simulation and detailed analytical models based on Markov chain techniques. Both of these approaches use relatively detailed models, which capture cache and bus interference rather precisely, but which are highly expensive to evaluate. In this paper, we investigate the use of a more abstract and significantly more efficient analytical model for evaluating the relative performance of cache consistency protocols. The model includes bus interference, cache interference, and main memory interference, but represents the interactions between the caches by steady-state mean collision rates which are computed by iterative solution of the model equations.

We show that the speedup estimates obtained from the mean-value model are highly accurate. The results agree with the speedup estimates of the detailed analytical models to within 3%, over all modifications studied and over a wide range of parameter values. This result is surprising, given that the distinctions among the protocols are quite subtle. The validation experiments include sets of reasonable values of the workload parameters, as well as sets of unrealistic values for which one might expect the mean-value equations to break down. The conclusion we can draw is that this modeling technique shows promise for evaluating architectural tradeoffs at a much more detailed level than was previously thought possible. We also discuss the relationship between results of the analytical models and the results of independent evaluations of the protocols using simulation.

1. Introduction

High-speed local memory that operates as a cache for a larger or more distant main memory can significantly increase the effective execution rate of a processor. When this technique is used in a general-purpose shared-memory MIMD multiprocessor, the problem of maintaining consistency among the data stored in multiple caches arises.

A number of dynamic cache consistency protocols have been developed for the case where the multiprocessor interconnection network is a shared bus. In this case, each cache controller monitors the traffic on the bus, and takes appropriate actions to maintain data consistency. The simplest protocol is a write-through protocol, in which all writes to a cache are written through to main memory, causing other caches that have the data to update or

Our program of research in computer system performance analysis is supported by the National Science Foundation (Grants No. DCR-8352098, DCR-8451405, CCR-8619663, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, Digital Equipment Corporation (the Systems Research Center and the External Research Program), CRAY Research, the AT&T Foundation, Bell Communications Research, Boeing Computer Services, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company.

invalidate their copies [Smit82]. In 1983, Goodman proposed a copy-back multi-cache consistency protocol that has since become known as the Write-Once protocol [Good83]. Since that time, a number of more sophisticated protocols have been proposed. These include the Synapse protocol [Fran84], the Illinois protocol [PaPa84], the RWB protocol [RuSe84], the Dragon protocol [MCCr84], and the Berkeley protocol [KEWP85].

The relative performance of the above protocols has been studied extensively using simulation [ArBa86]. In another approach, the key modifications to the Write-Once protocol that have been included in each of the five successor protocols were identified, and the contribution of each to overall performance was evaluated [VeHo86]. This second study used an analytic technique called Generalized Timed Petri Nets (GTPNs) [HoVe85]. (Results of the GTPN model agreed very well with results of independent evaluations in the various protocol proposals.)

Both of the above studies used relatively detailed models, which capture cache and bus interference rather precisely, but which are highly expensive to evaluate. In this paper, we investigate the use of a more abstract and significantly more efficient analytical model for evaluating the relative performance of the cache consistency protocols. We use the abstract model to reproduce and extend the results in [VeHo86]. The model includes bus interference, cache interference, and main memory interference, but represents the interactions between the caches by steady-state mean collision rates which are computed by iterative solution of the model equations.

One might expect the more abstract mean-value model to be less accurate than the detailed GTPN model. However, we show that the speedup estimates obtained from the mean-value model are surprisingly accurate. The results agree with the speedup estimates of the GTPN model to within 3%, over all modifications studied and over a wide range of parameter values. The validation experiments include sets of reasonable values of the workload parameters, as well as sets of unrealistic parameter values for which one might expect the mean-value equations to break down. The conclusion we can draw is that this modeling technique shows promise for evaluating architectural tradeoffs at a much more detailed level than was previously thought possible.

Greenberg and Mitrani have also recently developed an analytical model of the Write-Through, Write-Once, and Dragon protocols [GrMi87]. Our mean-value model of cache, main memory, and bus interference is more comprehensive and more firmly grounded in queueing network theory than their model. Furthermore, we are able to consider deterministic bus access times for cache block transfers, rather than the exponential access times required in their model. However, our workload model is less sophisticated than their workload model and the workload model in [ArBa86]. We comment on this further in Section 2.3.

The remainder of this paper is organized as follows. Section 2 contains a brief review of the Write-Once protocol, the four modifications to Write-Once that have been proposed, and the workload model in [VeHo86], which is also used in this paper. Section 3 presents our mean-value analytical model of Write-Once and the protocol modifications. Section 4 compares the estimates obtained using the mean-value model with estimates obtained using the GTPN model, and with estimates published in independent studies, for a variety of parameter settings. This section also contains some new results, including the asymptotic performance of the modifications. Section 5 contains the conclusions of this work.

2. Background

The multiprocessor configuration assumed for the snooping cache consistency protocols is illustrated in Figure 2.1. Each processor is connected directly to its local cache. Each cache is connected through the shared bus to main memory and to all other caches.

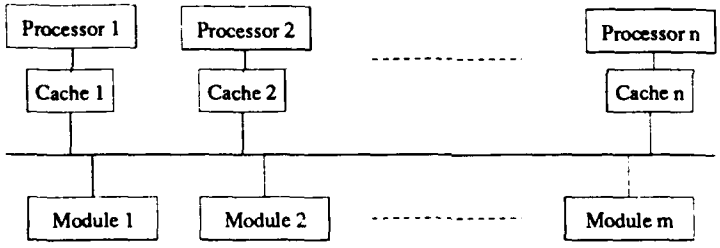


Figure 2.1: Multiprocessor Configuration

2.1. System Assumptions

The system assumptions made in the model developed in this paper are those in [VeHo86]. These assumptions are very similar to the assumptions in [ArBa86], and include the following.

A processor executes for a variable number of cycles, assumed to be exponentially distributed with mean τ, between memory requests. Useful execution is not overlapped with fetching data from memory. Thus, once the request is made, the processor is idle until the request is satisfied. The cache takes one unit of time to satisfy the processor request, either immediately, or following a required bus transaction.

Bus transactions may be one of five types: *read*, *read-mod* (i.e., read-with-the-intent-to-modify), *invalidate*, *write-word*, or *write-block*. The first and second request types are issued when processor read and write requests

miss in the cache, respectively. The third or fourth request type is used by the consistency protocol when a processor write request hits in the cache, and the cache block is clean (i.e., unmodified). The fifth request type is used to write a modified data block back to main memory.

Bus requests are served in random order in the GTPN model [VeHo86], but are assumed to be scheduled in first-come first-served order in the mean-value model developed in this paper. Both scheduling disciplines have the same mean waiting time, and thus yield the same predicted speedup measures. Bus requests have priority over processor requests for service in a cache. Dual directories are assumed, so processor requests are only delayed by bus requests that require some action on the part of the cache.

The main memory is divided into m modules, where m is the cache block size, assumed to be four in this paper. Main memory latency is assumed to be three cycles.

Cache block states are assumed to be defined by three bits of state information. (Not all bits are used in exactly the same way by each protocol.) The first bit denotes whether the block is *valid* or *invalid*. The second bit indicates whether the cache knows that it has the only copy of a block, i.e., state *exclusive*, or does not know that it has an exclusive copy, i.e., state *non-exclusive*. The third bit (*wback/no-wback*) denotes whether or not the processor must write back the block when it is purged from the cache. Note that this third bit indicates whether or not the block is modified relative to main memory.

2.2. Review of Snooping Cache Protocols

Below we review briefly the Write-Once protocol, and each of the key modifications that have been proposed to improve performance. In some cases we comment on the potential advantages and disadvantages of the proposed modifications. The reader is referred to [ArBa86], [VeHo86], and the original papers for further detail.

Write-Once

A bus read request loads the cache block in state *non-exclusive* and *no-wback*. A bus read-mod request invalidates all other copies of the block, and loads the block in state *exclusive* and *wback*.

The key idea in the Write-Once cache consistency algorithm is that the *first* time a processor writes a word to a *non-exclusive* block in its cache, the word is written through to main memory. When the word is broadcast on the bus, any cache containing the block invalidates its copy. The write operation changes the state of the block to *exclusive* and *no-wback*.

Writes to a block in state *exclusive* in the cache are written only locally, changing the state to *wback*. If another cache requests the block, indicated by a read or read-mod operation on the bus, a cache containing the block in state *wback* interrupts the bus transaction and writes the block to main memory, thereby updating the contents of main memory before main memory supplies the requested data. The state of the block changes to *no-wback* if the bus request is of type read.

In this protocol, if a cache contains a block in state *wback*, it is the only cache containing the block.

Modification 1

In the first of the proposed modifications, a cache containing a copy of a block requested by a read or read-mod bus operation must raise a *shared* line on the bus. If this line is not raised, the cache block can be loaded in state *exclusive* in the requesting cache. Writes to this block by the requesting cache will not require bus operations to notify other caches. (However, note that writing the block to main memory will be required when the block is purged from the cache.)

This modification reduces the total number of bus operations required by an amount that depends on 1) the workload characteristics and 2) which, if any, of the other three modifications are also implemented. If this is the only modification to the Write-Once protocol, the number of operations required is reduced in the following case: 1) a requested block is not resident in another cache, and 2) the block is written more than once during its tenure in the cache. Modification 1 is included in the Illinois, Dragon, and RWD protocols.

Modification 2

In the second of the proposed modifications, a cache that has a requested block in state *wback*, supplies the copy directly to the requesting cache and does not update main memory. This saves memory traffic and bus operations in the case that the requesting cache modifies the block before purging it.

If the bus operation is a read request, the supplying cache takes responsibility (sometimes called *ownership*) for writing back the block when it is purged. In other words, the supplying cache sets the state to *non-exclusive* and *wback*, and the requesting cache sets the state to *non-exclusive* and *no-wback*, if the bus request is a read operation. Modification 2 is included in the Berkeley and Dragon protocols. The Illinois protocol assumes the data is written to memory and supplied to the requesting cache in the same bus operation, which is another optimization similar to this modification.

Modification 3

In the third modification to Write-Once, a bus *invalidate* operation is performed, instead of the write-word operation, on the first write to a non-exclusive data block. This potentially reduces memory traffic, and eliminates the need for "partial write" operations on the bus. There is potentially a reduction in bus traffic in the case that *write-word* requires two bus cycles and *invalidate* requires one cycle. On the other hand, there is the potential for increased bus traffic with this modification, since the write-word operation sometimes takes the place of writing the entire block to main memory when the block is purged. Modification 3 is included in all five protocols proposed as improvements to Write-Once.

Modification 4

The final modification allows multiple copies of a cache block to remain valid even in the presence of write operations. In this case, all writes to a block in state *non-exclusive*, are broadcast on the bus. All caches update their copies, and main memory is updated by the broadcast write. Cache blocks remain in state *no-wback*.

Note that this modification alone reduces the Write-Once protocol to a write-through protocol. Thus, this modification is only practical when implemented together with modification 1. Modification 4 is included in the RWB and Dragon protocols. Furthermore, the RWB protocol includes the capability to switch between invalidation and broadcast write operations.

Summary

We have presented the above modifications, except as noted, as independent modifications to the Write-Once protocol. The modifications can be implemented in any combination, if the following observation is noted. If modifications 3 and 4 are implemented together, the effect is to broadcast all write operations to non-exclusive blocks, but not to update main memory. In this case, as in modification 2, some cache has to take responsibility for writing back the block when it is purged. We assume the cache performing the broadcast takes this responsibility.

2.3. Our Workload Model

The workload model we use to evaluate the above protocols is the same as the workload model in [VeHo86]. This model was based on the model in [DuBr82], which views the memory reference stream as the merging of two streams, one for private and shared read-only blocks, and one for shared-writable blocks. The first stream was decomposed into two substreams in [VeHo86]. All three streams are treated probabilistically in our model. The

probabilistic treatment is very similar to the treatment of the private stream in [ArBa86]. However, less locality is assumed in the shared-writable stream.

The following basic parameters are specified for our workload model:

- $p_{private}$, p_{sro}, and p_{sw}, are the probabilities that a memory reference is to a private, shared read-only (sro), and shared-writable (sw) block, respectively.
- $h_{private}$, h_{sro}, and h_{sw}, are the hit rates for private, sro, and sw streams, respectively.
- $r_{private}$ (r_{sw}) is the probability that the processor request is a read request, given that the reference is of type private (sw).
- $amod_{private}$ ($amod_{sw}$) is the probability that a reference to a private (sw) block that hits in the cache finds the block already modified.
- $csupply_{sro}$ ($csupply_{sw}$) is the probability that a copy of a requested sro (sw) block exists in at least one other cache.
- $wb_{csupply}$ is the probability that the cache supplier contains the data in state *wback*.
- rep_p (rep_{sw}) is the probability that a private (sw) block must be written back to memory when it is purged.

From these parameters, the following model inputs can be computed [VeHo86]:

- p_{local}, the probability that a memory request can be satisfied locally in the cache,
- p_{bc}, the probability that the memory request requires a broadcast write or invalidation,
- p_{rr}, the probability that the memory request requires a remote read or read-mod operation,
- t_{read}, the mean bus access time for a remote read or read-mod operation, which includes main memory write-back by another cache and/or by the requesting cache, if necessary,
- $p_{csupwb|rr}$, the probability that another cache must write the block to main memory in response to a remote read request, and
- $p_{reqwb|rr}$, the probability that the requesting cache must write back a replaced block on a remote read operation.

We note that our probabilistic treatment of the shared data reference stream treats the relationship between system size and *actual* sharing of data more approximately than the workload models in [ArBa86] and [GrMi87]. The workload submodel of both the mean value model in this paper and the GTPN model should be improved to treat the shared references more similarly to the model in [GrMi87]. However, this should not change the conclusions of this paper with regard to the relative accuracy of the mean value model. Furthermore, we show in Section 4 that results of the model agree well with independent evaluations of the protocols, in spite of the approximate workload representation.

3. Mean Value Models of the Cache Consistency Protocols

In this section, we develop a mean-value model of cache, memory, and bus interference for the Write-Once protocol. We then briefly discuss the iterative model solution technique, and the required model modifications for

each of the four potential improvements outlined in the previous section. The results of the mean value models are compared with the results of the corresponding GTPN models in Section 4.

The idea in Mean Value Analysis, which has been used with great success to solve queueing network models of computer system performance, is to construct a set of equations that compute the mean values of various performance quantities in terms of the mean values of various model inputs – frequently resorting to iteration when a direct calculation is not possible. What is important to bear in mind (and will be discussed in Section 3.2) is that our mean value approach to analyzing multi-cache consistency protocols yields a dramatic improvement in the time required to obtain performance measures from the model – a reduction from hours of computing to seconds of computing for large numbers of processors – with a negligible loss in accuracy, as will be shown in Section 4.

The notation used in this section is defined in the development of the equations.

3.1. The Write-Once Protocol

We first consider response times for memory requests, then mean bus waiting time, memory interference, and cache interference.

Response Time Equations

The mean total time between memory requests issued by a processor, R, is the sum of the processor execution time between requests, τ, the weighted mean delays for each of the three ways in which memory requests are handled by the cache (locally, broadcast write-word, or remote read), and the cache supply time ($T_{supply} = 1.0$):

$$R = \tau + R_{local} + R_{broadcast} + R_{RemoteRead} + T_{supply}. \quad (1)$$

The weighted mean response time for a memory request that can be handled locally in the cache, R_{local}, is the product of the probability that the request can be handled locally, the mean number of consecutive bus requests that delay the cache response to the processor request ($n_{interference}$), and the mean number of cycles that each interfering bus request requires in the cache ($t_{interference}$):

$$R_{local} = p_{local} \times n_{interference} \times t_{interference}. \quad (2)$$

The calculation of $n_{interference}$ and $t_{interference}$ is discussed in the subsection on cache interference below.

The mean response time for broadcast write operations is estimated as the sum of the mean waiting time for the bus (w_{bus}), the mean waiting time for the main memory module (w_{mem}), and the fixed bus access time for the

write-word operation (T_{write} = 1.0). Thus, the weighted mean response time is given by:

$$R_{broadcast} = p_{bc} \; (w_{bus} + w_{mem} + T_{write}). \tag{3}$$

Equations for w_{bus} and w_{mem} are developed in the following subsections on mean bus waiting time and memory interference, respectively.

Finally, the mean response time for a remote read (or read-mod) operation, is approximately the sum of the mean bus waiting time, and the mean bus access time for the read operation (t_{read}):

$$R_{RemoteRead} = p_{rr} \; (w_{bus} + t_{read}). \tag{4}$$

Memory interference is not an important factor in the response time for remote reads. This is due to the fact that main memory latency is assumed to be fixed and small (i.e., 3.0 cycles). Thus, the mean wait for memory after the request is served by the bus, is negligible. Note that t_{read} includes the mean time for one and possibly a second and third cache block transfer on the bus, as defined in Section 2.3.

Mean Bus Waiting Time

To complete the response time calculations in equations (1)-(4), we need to compute w_{bus}, w_{mem}, $n_{interference}$, and $t_{interference}$. The equations for w_{bus} are developed next.

An arriving request will wait for the mean remaining bus access time (i.e., the mean *residual life*, $t_{res,bus}$) of the request in service, plus one mean bus access time (t_{bus}) for every other request in the queue when it arrives. Let $\bar{Q}_{bus}$ represent the mean number requests found in the queue by the arrival, and $p_{busy,bus}$ represent the probability an arriving request finds the bus busy. The equation for w_{bus} is thus:

$$w_{bus} = (\bar{Q}_{bus} - p_{busy,bus}) \, t_{bus} + p_{busy,bus} \, t_{res,bus}. \tag{5}$$

Applying techniques from Product Form queueing networks [LZGS84] in an approximate way, the mean queue length seen by an arriving request is estimated by the steady state mean queue length in the system if the requesting cache were removed. This is approximately the product of the average fraction of time each cache spends in the bus queue, and the number of other caches ($N-1$):

$$\bar{Q}_{bus} = (N-1) \, \frac{R_{bc}+R_{rr}}{R}. \tag{6}$$

Bus utilization is estimated by the product of the number of caches in the system, and the fraction of time each cache uses the bus:

$$U_{bus} = N \times \frac{p_{bc}\,(w_{mem} + T_{write}) + p_{rr} \times t_{read}}{R}, \tag{7}$$

from which we can estimate the probability that an arriving request finds the bus busy:

$$p_{busy,bus} = \frac{U_{bus} - \frac{U_{bus}}{N}}{1 - \frac{U_{bus}}{N}}. \tag{8}$$

Finally, the mean bus access time, and mean residual life of the request in service, are given by the following weighted sums:

$$t_{bus} = \frac{p_{bc}}{p_{bc} + p_{rr}}\,(T_{write} + w_{mem}) + \frac{p_{rr}}{p_{bc} + p_{rr}}\,t_{read}, \tag{9}$$

and

$$\begin{aligned} t_{res,bus} = {} & \frac{p_{bc}\,(T_{write} + w_{mem})}{p_{bc}\,(T_{write} + w_{mem}) + p_{rr}\,t_{read}} \times \frac{T_{write} + w_{mem}}{2} \\ & + \frac{p_{rr}\,t_{read}}{p_{bc}\,(T_{write} + w_{mem}) + p_{rr}\,t_{read}} \times \frac{t_{read}}{2}. \end{aligned} \tag{10}$$

Memory Interference

The mean time that a broadcast write operation waits for the main memory module (w_{mem}) is the product of the probability that the request finds the module busy ($p_{busy,mem}$) and the mean remaining memory latency. Letting d_{mem} represent the mean total memory latency, assumed to be 3.0 in this paper, the mean memory waiting time is given by:

$$w_{mem} = p_{busy,mem} \times \frac{d_{mem}}{2}. \tag{11}$$

The probability the request finds the module busy is computed from the utilization of the memory module, in the same way that $p_{busy,bus}$ was estimated from the bus utilization. The utilization of the memory module is estimated as the fraction of time each cache uses the memory module, times the number of caches in the system:

$$U_{mem} = N \times \frac{1}{4} \times \frac{[\,p_{bc} + p_{rr}\,(p_{csupwb|rr} + p_{repwb|rr})\,] \times d_{mem}}{R}. \tag{12}$$

Note that the above equation assumes four memory modules, but can easily be modified for some other number of modules.

Cache Interference

The cache interference submodel involves computing $n_{interference}$ and $t_{interference}$.

The mean number of consecutive bus requests that interfere with a processor request, $n_{interference}$ is computed assuming, approximately, that the maximum number of requests that can interfere with the processor request is equal to the number of requests in the bus queue when the processor request is issued (i.e., $\bar{Q}_{bus}$).

Using the model input parameters, and assuming that a block supplied by a cache is equally likely to be supplied by any of the other caches, it is straightforward to compute the following probabilities (see Appendix B): 1) the probability, p, that a cache must service a bus request, and 2) the probability, $p' < p$, that the cache must service the bus request for the entire duration of the bus transaction. An example of an event of the second type is a broadcast write operation on a block contained in the cache. An example of an event of the first type, but not of the second type, is a read-mod operation where the cache has the block in state *no-wback*. The cache must respond by invalidating the block, which is of shorter duration that the bus transaction.

$n_{interference}$ is easily computed from p and p', as follows:

$$\begin{aligned} n_{interference} &= \bar{Q}_{bus}\, p'^{\bar{Q}_{bus}-1}\, p + \sum_{k=1}^{\bar{Q}-1} k\, p'^{k-1}[\,(p-p') + p'\,(1-p)\,] \\ &= p\left[\frac{1-p'^{\bar{Q}}}{1-p'}\right]. \end{aligned} \tag{13}$$

$t_{interference}$, the mean cache interference time per request that blocks a processor request, is computed from model inputs in the same way that p and p' are computed (see Appendix B).

3.2. Solution of the Model Equations

The above equations contain cyclic interdependencies, in which R depends on the mean bus and memory waiting times, which in turn depend on R. Thus, the equations must be solved iteratively. We do so, starting with all waiting times set to zero. Solution of the equations converged within 15 iterations in all experiments reported in this paper, yielding results in under one second of cpu time, independent of the size of the system analyzed. In contrast, the time to solve the GTPN model increases exponentially with the number of processors analyzed. With ten processors, the GTPN model requires on the order of one hour of CPU time on a DEC MicroVAX-II with eight megabytes of memory. We note that simulation is equivalently expensive.

3.3. Models of the Four Protocol Modifications

The changes required in the mean-value model for each of the protocol modifications in Section 2.2, are a subset of the required changes in the GTPN model [VeHo86], and mostly involve changes in computing the model inputs.

For modification 1, the calculation of $p_{broadcast}$ no longer includes a term for write hits to private blocks. This term is instead added to p_{local}. The equation for p, the probability of cache interference, must be modified, since write hits to private blocks are no longer broadcast (i.e., p increases slightly). Furthermore, the input parameter rep_p must be increased (from 0.2 to 0.3 in the workload of [VeHo86]), which causes a small increase in t_{read}. For modification 2, the calculations of $t_{contention}$ no longer includes the term for cache supply write-back, and the input parameter rep_{sw} increases (from 0.5 to 0.6 in the workload of [VeHo86]), causing a slight increases in t_{read} and a slight decrease in p'. For modification 3, the term for broadcast writes is removed from equation (12), and the probability rep_{sw} increases, by an amount assumed to be comparable to the effect of modification 2. Finally, for modification 4, changes are required in the calculation of p_{bc}, p_{local}, and p.

4. Results and Accuracy of the Mean Value Models

Speedup is computed from the mean-value models using the formula: $N \times \frac{\tau + T_{supply}}{R}$. The input parameter values specified in [VeHo86] are reproduced in Appendix A. The mean value analysis speedup estimates for these parameter values are plotted in Figure 4.1 and tabulated in Table 4.1 for three protocols: 1) the Write-Once protocol, 2) a protocol that includes modification 1 only, and 3) a protocol that includes modifications 1 and 4. Speedups for modifications 2 and 3 are nearly indistinguishable from the results for the protocols without these modifications, and are thus not shown. Results for each of the three levels of sharing considered in the GTPN study (1%, 5%, and 20%), are given for the first two protocols. For the third protocol, only the 5% sharing curve is drawn, since the other two curves are nearly identical (see Table 4.1(c).)

Below we discuss the results of our analysis, the accuracy of the mean value analysis estimates as compared with the GTPN results, and the relationship between the results in this study and results of independent evaluations of the protocols.

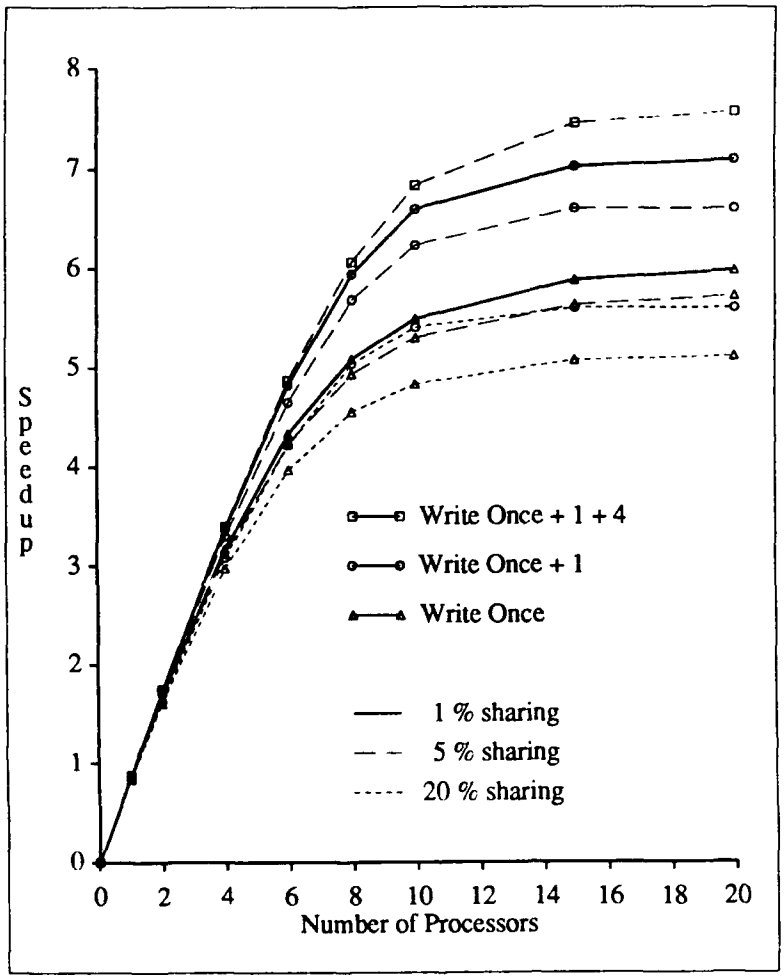


Figure 4.1: The Mean Value Analysis Performance Results

4.1. Model Results

As discussed in the next section, the mean-value analysis results are nearly identical to the results of the GTPN model. Furthermore, we are able to analyze the speedup for arbitrarily large systems using the MVA equations. (Solution of the GTPN model is impractical for more than ten or twelve processors.) Table 4.1c includes the MVA results for 100 processors, to verify that the performance does not change appreciably beyond twenty processors, for each curve shown in the figure. The asymptotic results indicate a greater potential gain for modification 4 than was evident from previous results for ten processors.

The conclusions we draw from figure 4.1 are as follows. Modification 1 is clearly advantageous for the workloads we have studied. Modification 4 is more advantageous as system size and the level of sharing increase,

assuming this modification substantially increases the value of h_{sw}, as we have assumed in our input parameter values. Modifications 2 and 3 have little effect for the workload we investigated. We comment on this further in Section 4.4.

4.2. Agreement Between the Mean Value Analysis and GTPN Results

In this section we compare the performance estimates obtained from the mean-value equations with estimates obtained from the (expensive) solution of the detailed steady-state equations of the GTPN model.

Table 4.1(a) contains the numerical speedup estimates for the Write-Once protocol derived from the two models. Results for each of the three levels of sharing considered in the GTPN study are shown in the table. We find the speedup estimates of the mean value analysis are in excellent agreement with the speedup estimates of the GTPN for each sharing level. Nearly all MVA estimates are within 1% of the GTPN estimates, and the maximum relative error is 2.6%.

We also find very good agreement between the models (i.e., typically less than 5% relative error) for other performance measures, such as bus utilization and mean bus waiting time, which are not shown in the table. For example, in the 6-processor case, the GTPN and MVA estimates of bus utilization are approximately 81% and 77%, respectively. We note, however, that the approximate MVA equations generally underestimate bus utilization and overestimate memory and cache interference relative to the GTPN model.

Table 4.1(b) compares the speedup results from the MVA and GTPN models for the Write-Once protocol plus modification 1 of Section 2.2. Here again we find excellent agreement between the estimates, with most MVA results within 1% of the GTPN values, and a maximum relative error of 4.25%.

We investigated the accuracy of the MVA model further by validating it against the GTPN for each of the other three enhancements. In every case, the MVA model estimates agreed nearly exactly with the GTPN results. We thus conclude that, for the workload parameters in this set of experiments, the MVA model is as accurate as the more detailed GTPN model for assessing the relative merits of all of the proposed modifications to Write-Once. Table 4.1(c) further illustrates the point by giving the estimates from both models for the Write-Once protocol with modifications 1 and 4.

Table 4.1: Comparison Between MVA and GTPN Estimates

(a) Speedups for the Write Once Protocol

Sharing Level	Solution Method	Number of processors 1	2	4	6	8	10	15	20	100
1%	MVA	0.86	1.68	3.17	4.33	5.08	5.49	5.88	5.98	6.07
	GTPN	0.86	1.69	3.20	4.41	5.21	5.60			
5%	MVA	0.855	1.67	3.12	4.23	4.93	5.30	5.63	5.72	5.79
	GTPN	0.855	1.67	3.14	4.30	5.04	5.37			
20%	MVA	0.84	1.61	2.97	3.97	4.55	4.83	5.07	5.12	5.16
	GTPN	0.84	1.62	3.02	4.07	4.67	4.87			

(b) Speedups for Enhancement 1

Sharing Level	Solution Method	Number of processors 1	2	4	6	8	10	15	20	100
1%	MVA	0.875	1.73	3.37	4.82	5.94	6.59	7.02	7.09	7.04
	GTPN	0.875	1.73	3.37	4.84	6.00	6.72			
5%	MVA	0.87	1.71	3.30	4.65	5.68	6.23	6.59	6.64	6.60
	GTPN	0.86	1.71	3.31	4.71	5.76	6.31			
20%	MVA	0.85	1.63	3.08	4.22	5.03	5.40	5.63	5.66	5.62
	GTPN	0.85	1.65	3.15	4.39	5.19	5.58			

(c) Speedups for Enhancements 1 and 4

Sharing Level	Solution Method	Number of processors 1	2	4	6	8	10	15	20	100
1%	MVA	0.88	1.75	3.40	4.90	6.06	6.83	7.49	7.58	7.56
	GTPN	0.88	1.75	3.41	4.91	6.13	6.91			
5%	MVA	0.88	1.75	3.40	4.87	6.06	6.83	7.46	7.57	7.57
	GTPN	0.88	1.75	3.41	4.92	6.16	6.98			
20%	MVA	0.88	1.74	3.35	4.75	5.90	6.70	7.47	7.64	7.70
	GTPN	0.88	1.75	3.39	4.87	6.09	6.93			

These preliminary results indicate not only that the MVA model is quite accurate, but also that it is *as suitable as the GTPN for evaluating the potential performance gains of the various protocol modifications*. This result is surprising, given that the distinctions among the protocols are quite subtle. This is the first result known to the authors that indicates mean-value queueing analysis techniques may be applied to the evaluation of rather detailed architectural trade-offs. The computational efficiency of the MVA approach allows a wide range of design alternatives to be interactively investigated.

4.3. Accuracy of the Model Under Stress Tests

In the next set of experiments, we modified the workload parameters, in some cases assigning unrealistic values, in an attempt to find cases where the MVA equations are inaccurate. In particular, we experimented with cases that have a large amount of cache interference, since cache interference is represented much less precisely in the MVA model than in the GTPN.

In one experiment, we set the values of rep_p, rep_{sw}, and $amod_{sw}$ to 0.0, $csupply_{sro}$ and $csupply_{sw}$ to 1.0, p_{sw} to 0.2, and hit_{sw} to 0.1. The speedup estimates of the MVA model agreed, within 5% relative error, with the speedup estimates in the GTPN. This was the case in all of the experiments we performed in attempting to stress-test the MVA model. It appears that the MVA model is quite robust.

4.4. Agreement Between the Mean Value Model and Independent Evaluation Studies

It is generally difficult to compare results of the MVA and GTPN models with results of independent protocol evaluation studies, for two reasons. First, the parameter values used in experiments reported in the literature are not always fully specified. Second, if a different workload model is used, the mapping between parameters in the different workload models is generally not straightforward. In spite of these difficulties, we are able to compare our results with the results of independent studies in three cases.

The first comparison we are able to make is in the estimate of *processing power* for the protocol with modifications 1,2, and 3. Processing power is defined as the sum of the processor utilizations, over all processors in the system. Processing power can be computed from the MVA results by taking $\frac{\tau}{R} \times N$. Alternatively, processing power can be computed from the product of speedup and $\frac{\tau}{\tau+T_{supply}}$, which is $\frac{2.5}{3.5}$ or approximately 0.7143 for the workload assumptions in this paper. In either case, we compute a processing power of 4.32 for the protocol with modifications 1, 2, and 3, nine processors, 5% sharing, and parameter values equal to those in the appendix. The GTPN predicts a processing power of 4.1 for this case. Both results agree reasonably well with results of the simple analytical model in [PaPa84], for cache block size equal to four.

The second comparison we are able to make is in the estimate of relative bus utilization for Write-Once and a protocol with modifications 2 and 3. In this case, if the probability that a block is unmodified on a write hit decreases significantly in the protocol with modification 2, the MVA models predict a 10% increase in bus utiliza-

tion for the Write-Once protocol, 99% sharing, and total loads which do not saturate the bus. This result agrees well with the trace-driven simulation results of Katz et. al. [KEWP85].

The final comparison we make is to the simulation results of Archibald and Baer's study [ArBa86]. An important discrepancy between the results of that study and the results in Figure 4.1 is in the performance estimates for modification 1 relative to modification 2. The simulation study shows a nearly equal performance benefit for each of these modifications. (For example, the Berkeley and Illinois protocols have nearly equal performance in most of their experiments.) Careful examination of their parameter values reveals that the value they use for $amod_p$ in most of their experiments is substantially higher than the value we assumed. If we set $amod_p$ to 0.95, as in many of their experiments, we also find the performance of modification 2 to be roughly equal to the performance of modification 1 for the 1% sharing case in Figure 4.1.

The agreement between the mean-value estimates and estimates from independent studies further increases our confidence in the accuracy of the MVA model.

5. Conclusion

Efficient, accurate tools for studying the performance implications of architectural design decisions are critically important.

In this paper we consider a family of dynamic cache consistency protocols for shared bus multiprocessor systems. The performance of these protocols has been studied extensively using detailed simulation models and Generalized Timed Petri Net models. These modeling techniques, while accurate, have running times measured in hours on 1 MIPS processors, for models of systems of only modest size.

We have devised a new modeling approach, based upon the specification and the iterative solution of sets of equations that express the mean values of interesting performance measures in terms of the mean values of certain model inputs. The equations are intuitive, in the sense that each can be explained simply in terms of the mechanics of the architecture being modeled. The solution technique is extremely efficient, requiring on the order of one second of CPU time for systems of arbitrary size. This makes it possible to explore a large design space quickly and interactively. The results are essentially as accurate as those of the previously existing techniques, which are dramatically more expensive.

We believe that we have convincingly demonstrated the surprising result that simple and efficient models can be used to study the performance of architectural alternatives that differ from one another in only quite subtle ways. The model can be put to good use for evaluating the protocols more thoroughly – all that is needed are workload measurement studies to aid in the assignment of parameter values.

The demonstration of the accuracy of our model is the principal thrust of our paper. Along the way, we have used the model to reproduce and extend the results in [VeHo86], in one case showing a greater benefit of protocol enhancement 4, which could not be anticipated from the existing results due to the inability to solve the more expensive models for large systems.

We believe that computer architects should seriously consider our "customized mean value equation" approach conducting future architectural performance studies. The approach is certainly applicable to the performance analysis of larger and more complex cache-coherent multiprocessors [Wils87, GoWo87]. It is most likely also applicable to realms other than cache consistency protocols.

We also wish to emphasize the utility of the more detailed GTPN and simulation tools for validating the results of the mean-value analysis for small systems. The authors are aware that there are cases where mean value analys' is inaccurate. Thus, validation against more detailed models is critically important when applying the technique to new problem domains.

References

[ArBa86] Archibald, J., and J.-L. Baer, "An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors," *ACM Transactions on Computer Sysytems*, Vol. 4, No. 4, November 1986.

[DuBr82] Dubois, M., and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Trans. on Computers*, Vol. C-31, November 1982, pp. 1083-1099.

[Fran84] S.J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory Access Times," *Electronics*, Vol. 57, no. 1, January 1984, pp. 164-169.

[Good83] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. of 10th Int. Symp. on Computer Architecture*, June 1983, pp. 124-131.

[GoWo87] Goodman, J.R., and P. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," to appear in *Proc. 15th Ann. Int'l. Symp. on Computer Architecture*, Honolulu, Hawaii, May 30 - June 2, 1988.

[GrMi87] Greenberg, A. G., and I. Mitrani, "Analysis of Snooping Caches," to appear in *Proc. of Performance 87, 12th Int'l. Symp. on Computer Performance*, Brussels, December 1987.

[HoVe85] Holliday, M. A., and M. K. Vernon, "A Generalized Timed Petri Net Model for Performance Analysis,"
Proc. Int'l. Workshop on Timed Petri Nets, Torino, Italy, July 1985.

[KEWP85] Katz, R., S. Eggers, D.A. Wood, C. Perkins, and R.G. Sheldon, "Implementing a Cache Consistency Protocol,"
Proc. of 12th Int. Symp. on Computer Architecture, June 1985, pp. 276-283.

[LZGS84] Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984.

[MCCr84] McCreight, E., "The DRAGON Computer System: An Early Overview,"
NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy, July 1984.

[PaPa84] Papamarcos, M., and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories,"
Proc. of 11th Int. Symp. on Computer Architecture, June 1984, pp. 348-354.

[RuSe84] Rudolph, L., and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,"
Proc. of 11th Int. Symp. on Computer Architecture, June 1984, pp. 340-347.

[Smit82] Smith, A.J., "Cache Memories,"
Computing Surveys, Vol. 14, no. 3, pp. 473-530, September 1982.

[Smit85a] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice,"
Proc. of 12th Int. Symp. on Computer Architecture,, June 1985.

[Smit85b] Smith, A. J., "Line (Block) Size Choice for CPU Cache Memories," Technical Report CSD 85/239, Computer Science Division, Univ. of Calif. at Berkeley, 1985.

[VeHo86] Vernon, M. K., and M. A. Holliday, "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," *Proc. of Performance 86 and ACM SIGMETRICS 1986 Joint Conf. on Computer Performance Modeling, Measurement, and Evaluation*, Raleigh, N.C., May 1986, pp. 9-17.

[Wils87] Wilson, A. W., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. 14th Annual Int'l. Symp. on Computer Architecture*, Pittsburgh, PA, June 2-5, 1987, pp. 244-252.

Appendix A

The following workload parameter values are used in the experiments in Section 4:

Parameter	Value		
τ	2.5		
$p_{private}$	0.99	0.95	0.80
p_{sro}	0.01	0.03	0.15
p_{sw}	0.00	0.02	0.05
$h_{private}$	0.95		
h_{sro}	0.95		
h_{sw}	0.5		
$r_{private}$	0.7		
r_{sw}	0.5		
$amod_{private}$	0.7		
$amod_{sw}$	0.3		
$csupply_{sro}$	0.95		
$csupply_{sw}$	0.5		
$wb_{csupply}$	0.3		
rep_p	0.2		
rep_{sw}	0.5		

Note that the value of rep_p is increased to 0.3 for Modification 1; rep_{sw} is increased to 0.6 for Modifications 2 or 3, and to 0.7 for a protocol with both modifications; and, finally, hit_{sw} is set to 0.95 for the protocol with modifications 1 and 4.

Appendix B

In this appendix we give the formulas for p, p', and $t_{interference}$, used in Section 3.1. We assume PSRWM, PSWHumod, SRMiss, SWMiss, SWHumod, SRMiss, SWMiss, and SWCSup, are defined as in [VeHo86]. The equations are as follows:

$$p = p_a + p_b,$$

where:

$$p_a = \frac{PSRWM}{PSRWM + PSWHumod} \times (SRMiss + SWMiss) \times 0.5$$

and

$$p_b = \frac{SWHumod}{PSRWM + PSWHumod} \times 0.5.$$

$$p' = p_b + p_a \times \frac{1}{\frac{n-1}{2}} \times (csupply_{sro} \times SRMiss + csupply_{sw} \times SWMiss)$$

$$\times [1-(rep_p \times p_{private} + rep_{sw} \times p_{sw})]$$

$$t_{interference} = 1.0 + \frac{p_a}{p_a + p_b} \left\{ \frac{1}{\frac{n-1}{2}} \times (csupply_{sro} \times SRMiss + csupply_{sw} \times SWMiss) \right.$$

$$\left. \times [4.0 + (wb_{csupply} + SWCSup) \times 4.0] \right\}$$

Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks

Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, and John Zahorjan

Department of Computer Science
University of Washington
Seattle, Washington 98195

206-543-1695

Abstract

As VLSI technology continues to improve, circuit area is gradually being replaced by pin restrictions as the limiting factor in design. Thus, it is reasonable to anticipate that on-chip memory will become increasingly inexpensive since it is a simple, regular structure than can easily take advantage of higher densities.

In this paper we examine one way in which this trend can be exploited to improve the performance of multistage interconnection networks (MINs). In particular, we consider the performance benefits of placing significant memory in each MIN switch. This memory is used in two ways: to store (the unique copies of) data items and to maintain directories. The data storage function allows data to be placed nearer processors that reference it relatively frequently, at the cost of increased distance to other processors. The directory function allows data items to migrate in reaction to changes in program locality. We call our MIN architecture the Memory Hierarchy Network (MHN).

In a preliminary investigation of the merits of this design [8] we examined the performance of MHNs under the simplifying assumption that an unlimited amount of memory was available in each switch. We found that despite the longer switch processing times of the MHN, system performance is improved over simpler, conventional schemes based on caching.

In this paper we refine the earlier model to account for practical storage limitations. In particular, we study ways to reduce the amount of directory storage required by keeping only partial (rather than complete) information regarding the current location of data items. The price paid for this reduction in memory requirement is more complicated (and in some circumstances slower) protocols. We obtain comparative performance estimates in an environment containing a single global memory module and a tree-structured MIN. Our results indicate that the MHN organization can have substantial performance benefits and so should be of increasing interest as the enabling technology becomes available.

Keywords: multistage interconnection networks, shared memory multiprocessors, performance

This material is based on work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, CCR-8702915, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, The Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the System Research Center). Additional partial support for this work was generously provided by Bell Communications Research, Boeing Computer Services, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company. The Centre National de la Recherche Scientifique, France, and Laboratoire MASI, University of Paris 6, provided generous support and resources for Zahorjan for the year sabbatical leave during which this work was performed.

1 Introduction

In this study a new architecture for shared memory multiprocessors, the Memory Hierarchy Network (MHN), is introduced and analyzed. The systems under study are medium scale multiprocessors where processors are connected to memory modules by a multistage interconnection network. The approach advocated here extends the memory hierarchy into the interconnection network, tailoring it to the specific needs of accessing shared variables. A hierarchy of memory elements is built into the switches of the interconnection network, and dynamic data positioning and routing protocols are introduced. The study focuses on the performance and cost of various realistic implementations of this idea.

The motivation for this study is that there may be a considerable performance penalty in existing systems when shared variables are heavily referenced. This problem will be aggravated by the need for bigger systems. Trends in VLSI technology will allow faster and denser CPUs and memory designs. The inter-chip communication times, however, will not be sped up in the same proportion. Because of these developments, traditional measures of cost, such as total memory space, will be replaced by design constraints such as interconnection costs and pin count. Thus, adding memory and some logic to the interconnection switches will become an attractive way to enhance system performance.

A key observation is that keeping coherent multiple copies of shared variables, in the context of a multistage interconnection network, is very traffic intensive. Therefore, the proposed scheme is based on keeping only a single copy of each shared writable variable in the system, and dynamically moving it in the extended memory hierarchy to adapt to changing access patterns. As there is only one copy of shared writable data, there cannot be any inconsistent data, i.e., the coherency problem does not exist. Therefore, this study does not provide "yet another coherency protocol". Rather, the main questions addressed here are when and where to migrate a data object, as opposed to when to create a new copy or when to invalidate or update an existing one.

The material presented here has two major parts. The first part reviews the background and recent relevant studies, discusses the motivation for a new architecture, and briefly presents the main ideas. In the second part, the performance of a tree network with practical amounts of memory in switches is analyzed in relation to the implementation cost.

2 Architecture Models

In this section the models of the parallel systems used in our study are presented. First, for the purposes of our performance comparisons, we define a baseline system in which shared writable data is not cached. We then present a "standard improvement" to that system that allows caching and uses a single, central directory to maintain coherency. Finally, we present an overview of the general MHN architecture and identify four specific versions of particular interest.

2.1 The baseline Processors-Caches model

The baseline architecture will be referred to as the Processors-Caches (PC) model. It contains N (a power of two) identical processors and associated local caches, a global memory, and a multistage interconnection network of depth log N.

In our model the caches are used exclusively for private and read-only shared data, as well as (non-writable) code. Accesses to local caches always result in hits satisfied in a single processor cycle. In other words, we assume a fast, conflict-free conventional multistage network that is transparent to the architecture.

Global memory is used for shared writable data. It is accessed through the network, and a cache is placed between each memory module and the network to speed accesses. (Note that there is no coherency problem associated with a single cache located at this port.)

To simplify our feasibility study, we decompose the model by assuming that there is only a single memory module. Therefore, our interconnection network is a complete binary tree with the global memory (and its cache) at the root and processors at the leaves [1]. Because the single module case is not preferential to either the MHN or conventional network designs, the comparative results obtained for this model should be applicable to systems with larger numbers of memory modules.

We assume that read requests to global memory are synchronous, that is, the issuing processor waits for the result. The use of the interconnection network, memory access latency time, and potential memory contention cause the processor to remain idle for some period of time during this request. In contrast, write requests are assumed to be asynchronous: the issuing processor immediately resumes computing after the request packet is placed on the network.

We define the processor cycle time to be equal to one. The relatively simple switches in our PC model are also assumed to have unit cycle times. The global memory module has an access time of four.

2.2 The directory-based scheme

A clear weakness of the PC architecture is that all accesses to writable shared data must be sent to the global memory module. Performance may be improved through the use of "directory schemes" [1]. This involves caching data items at the processors and using a single, central directory located at the root to maintain coherence. In the simplest directory scheme, which we call DIR, only a single copy of each shared data block is kept in the system. (Note that this policy corresponds to policy "Dir1NB" from [1].) As will be seen shortly, the MHN also maintains only a single copy of each data item. Thus, the comparisons between DIR and the MHN serve to isolate the contribution of the dynamic routing capabilities of the latter.

[1] A binary tree results from conventional assumptions about switch fan-out. However, other assumptions are possible [6].

Because the switches required by the DIR scheme are not substantially more complicated than those required by PC, we assume the same cycle times for them. This is a slightly optimistic assumption for DIR, and so serves to understate somewhat the performance advantages of the MHN design demonstrated in Section 4.

2.3 The MHN architecture

As in the DIR scheme, data items in the MHN architecture may move dynamically from one memory to another (although only a single copy of a data item can be present in the system at a time). However, the MHN extends the DIR scheme in two ways. Firstly, switches of the MHN can hold data. Thus, data can be present not only in the global memory module and the caches located at the processors, but also at intermediate stages of the interconnection network. Secondly, MHN switches contain directories indicating the location of items stored in the subtrees for which they are the roots. Thus, the information in the single directory of the DIR scheme is partially replicated and distributed among the interconnection network switches of the MHN.

The exact manner in which data movement takes place in an MHN is controlled by a data migration policy. In the selection of an appropriate migration policy for use in the MHN there are two dimensions to be considered: "when should data be migrated?" and "how far toward the referencing processor should it be migrated?". A family of answers to the question of "when" is given by "each time the last j references are from the same processor" for differing values of j. For example, for j=1 data items are moved on every reference, while for j=2 two successive references to the item must come from the same processor before migration takes place. Similarly, a family of answers to the question of "how far" is given by "k steps" for various values of k. Here obvious choices for k are 1 (one step toward the referencing processor) and ∞ (all the way to the referencing processor). We introduce the notation MHN/j/k to denote the MHN policy that moves a data item k positions after j consecutive references by the same processor.

Intuitively, appropriate values for parameters j and k of the migration policy relate to the assumptions made about the "burstiness" of workload. A workload is considered bursty if it exhibits alternating periods of high and low frequency of access to individual data items. (In contrast, the workload behavior is considered to be "random" if the frequency of access to an individual data item is relatively constant over time.) A workload becomes increasingly bursty when, other factors (in particular, overall average reference rate) held fixed, either the length of the high frequency periods increases or the access rate during the low frequency periods decreases.

Parameter k of the migration policy relates to the assumed length of a burst. The longer a burst is likely to be, the more advantageous it is to move data towards the referencing processor despite the fact that this moves it away from many other processors. Thus, parameter k should be large for bursty workloads and small for random workloads.

Parameter j relates to the low frequency reference rate. It is used to detect when a burst has begun. Some references to a data item are made even during periods of overall low frequency. It is counter-productive to migrate a data item in response to these accesses. For a very bursty workload, however, these low frequency period accesses are rare. Thus, for a bursty workload j can be small, that is, it is safe to assume that (nearly) all references to the data item indicate the beginning of a high access frequency period.

In the work presented here, we have chosen to evaluate four specific policies:

1. MHN/1/1: Move the data one step on each reference. Here a step is one edge in the path from the current location towards the processor that issued the request.

   Consider the example shown in Figure 1. The data block x, located in switch $S5$, will move to switch $S2$ if the next reference to it is made by a processor in the set $\{P1, P2, P5, P6, P7, P8\}$. If the reference is made from $P3$ or $P4$, it will move to the local cache of the referencing processor. This policy is reasonable if successive references made to the same data are made mainly by a subset of the processors; data items would gradually migrate to the node of the network that is the "center of reference" among this set of processors.

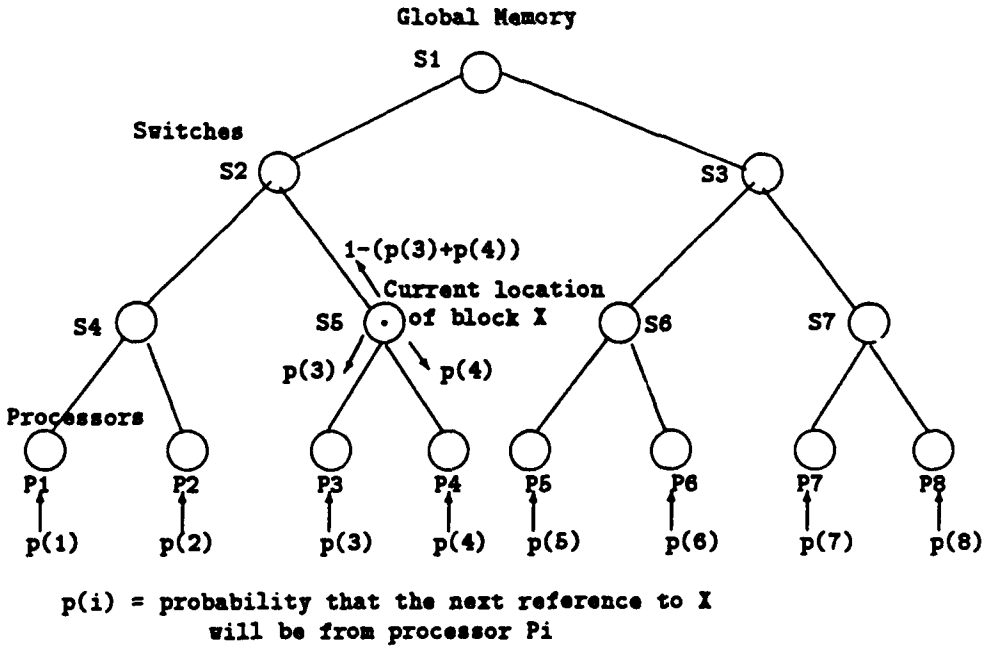


Figure 1: Data Migration in a Tree MHN/1/1 Architecture

2. MHN/2/2: Move the data two steps after two successive references from the same processor. This policy is reasonable if a mixture of "random" and "burst" behavior is presumed. It assumes that a burst is reliably indicated by two successive requests from the same processor, and that

otherwise a reference is simply a random reference. It has almost the same "speed of migration" as the previous policy (one step per access made within a burst, when an even number of successive references have been encountered), but avoids unnecessary migrations and the overhead associated with them.

3. MHN/1/∞: Move the data all the way to the requesting processor on each reference. This policy resembles the DIR scheme. It is based on the observation that since the data must be propagated alo..g the entire path to the processor in response to the access request regardless of the final placement of the item, it can be moved into the the final placement of the item. Note, however, that it is more expensive to move the item all the way because there is some overhead in updating the directories along the way. Finally, while a requested data item is never migrated into a switch memory other than at the processors, all switches contain some data storage. This storage is used to "bubble up" replaced data items that arise when a lower level memory is full and a new item must be migrated there.

4. MHN/2/∞: Move the data all the way to the requesting processor on each two successive references from the same processor. This policy resembles the previous one, but is more conservative in deciding when a burst has begun.

The performance evaluation of these alternatives is presented in Section 4. For brevity in what follows, whenever a detailed description is needed we will demonstrate the operation of the MHN/1/1 policy.

As will be seen in the next section, implementation of an MHN requires switches that are more complicated than those needed for the PC or DIR designs. Thus, in our performance evaluation of MHN we assume a switch cycle time of two, i.e., twice the cycle time of the switches in the simpler networks. (The one exception is that the MHN switches connected directly to the processors are relatively simple, acting as normal caches. Thus, we keep the unit cycle time assumption for those switches.)

3 The Design of MHN Networks

3.1 Switch protocols

In this section possible designs for MHN switches are discussed and their relative complexity and performance are compared. The complexity of these switches can be substantial, mainly as a result of the dynamic routing capabilities. Therefore, attention is focused on cost/performance tradeoffs in the implementation of the directories. The discussion will be presented at two levels: (1) the logical level, where the switch protocols are presented, and (2) the implementation level, where the switch structure is discussed. First, a general description of the switch structure with the simplest switch protocol, assuming unlimited data memory and full knowledge of routing information in each switch, is presented. For the practical case, where the data memory in the switches is limited, two approaches for

designing directories with full routing information are presented. These require either a large amount of memory or a high degree of associativity. A possible organization for directories with partial routing information is then described. The effect of finite data memory and partial routing knowledge on system performance is evaluated in the next section.

The basic operation of a switch consists of three tasks:

1. As part of the global but distributed shared *memory*, the switch controls accesses to the data currently held in its local store. The switch data memory acts as a conventional memory module in a global memory system. However, because the position of data changes dynamically, the local data memory is accessed associatively, like a cache.

2. As part of a global but distributed *directory*, the switch performs routing of requests from processors to memory. Arriving request packets are routed according to information on the current location of the requested data. Since routing decisions are made locally, a switch needs to hold enough information to uniquely select an appropriate port to forward the request. The routing information is initialized according to the initial data placement and is updated dynamically during execution. In contrast to a global directory approach, the routing information in a switch need not be updated on every movement of the data, but only when there is a change in the port through which the data is accessible, i.e., when the data passes through the switch.

3. As part of a conventional multistage interconnection network, the switch performs "static" routing of packets on the return path from memory to processors. (The routing is static because each packet is designated for a specified processor, and the routing is based on the processor identifier.) This type of routing is relatively fast since it does not require a directory access. Note, however, that for those packets that migrate data (rather than simply transferring it) through the switch a directory access is still required to update routing information.

3.2 Switch structure

Figure 2 depicts a schematic block diagram of an MHN switch. The major components are:

1. Input and output buffers, implemented as First In First Out (FIFO) queues. The external inter-switch links can carry only one packet at a time. Whenever a destination buffer at a switch is full, transfers to it are stalled.

2. I/O ports. Each switch has six unidirectional ports, two for each of its neighbors ("Up", "Left" and "Right"), allowing full-duplex communication. The data paths of the inter-switch links are wide enough to handle one packet in a cycle.

3. Internal busses. The internal busses and logic can transfer a packet from each input port to any output queue in a switch cycle. Separate parts of the output queue allow insertion of multiple packets into it, thus avoiding contention when more than one packet is routed to the same

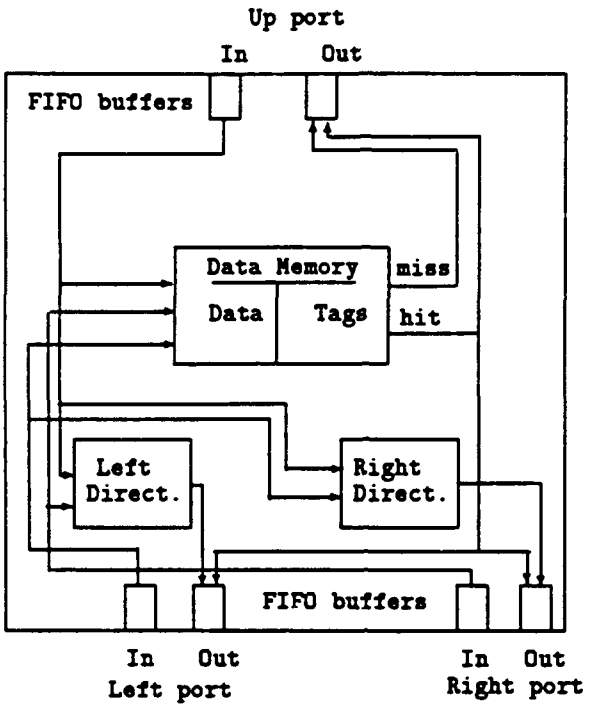


Figure 2: Basic MHN Switch Structure

port. The head of the output queue is one of the heads of its sub-queues, selected randomly by arbitration logic. (Similar assumptions have been made for almost all interconnection network performance evaluations, and a switch design that achieves this degree of parallelism was designed for the NYU architecture [3].)

4. Data memory. This is the part of the global shared memory that is currently located in the switch. It is organized like a cache. The size of the data memory depends on the size of the global memory, the technology used, and the switch's level in the network. The appropriate amount of memory that should be allocated to each level depends on the migration policy. For the MHN/(1 or 2)/∞ policies, for example, it will be reasonable to allocate a substantial portion of the memory to the leaves (processors) and to the levels near the root (which serve more descendant nodes and of which there are relatively few in the system). In the MHN/1/1 and MHN/2/2 policies more memory should be allocated at intermediate levels.

5. Routing directory. This part of the switch holds dynamically updated routing information. As routing is performed locally in each switch, the relative position ("Up", "Left", or "Right") of the data is sufficient. An important aspect of this work is to assess the complexity and the performance implications of various directory implementations. Two possible approaches are valid: (1) a single directory, which is accessed for routing all the packets through the switch, or, (2) separate subdirectories, each holding routing information for the data accessible through a

particular external port. (Figure 2 depicts the latter organization.) Note that a separate directory for data stored locally is not necessary, as "hits" or "misses" on the local data store provide this information explicitly.

3.3 Basic switch operation and protocol under optimistic assumptions

The simplest switch protocol is based on two assumptions: that the data memory in each switch is large enough to contain all the shared data, and that the switches' directories hold "full knowledge" (i.e., they include routing information for *all* data blocks that are located in all descendants of the switch). Under these optimistic conditions: (1) data entries are never replaced, (2) routing information is never lost because of lack of space in a directory, (3) broadcasting is not needed, and (4) requests that cannot be serviced locally are forwarded through one port only. The protocol assures that whenever a data block is stored in a switch or transferred through it, the appropriate entry in the directory at the switch is updated. For data blocks that have never been transferred through a switch, the default information ("Up", as initialized) correctly indicates that packets referring to this data should be forwarded upwards.

The "full knowledge directory" switch protocol distinguishes among the following five types of packets:

1. Read: Issued when a processor loads a shared writable data item.
2. Write: Issued when a processor stores a shared data item.
3. Answer: Issued by a switch to forward data in response to a Read request.
4. Answer Migrate: Issued by a switch in response to a Read request in order to cause the migration of data along the return path of the Answer. To execute migration of data under MHN/k/j migration policies when $k > 1$, Answer Migrate packets contain a migration counter. It is initialized in the switch that generates the Answer Migrate packet and is decremented thereafter at each switch on its path to the processor. When the migration counter is zeroed, the migration ends, the data is stored in the local memory, and the packet is transformed into an Answer packet.
5. Write Migrate: Similar to an Answer Migrate packet, it is issued by the switch that responds to an original Write request. It contains the data and a migration counter, which is decremented on each step. Again, when this counter is zeroed it signifies that the data has reached its destination. The switch stores the data in its local memory and removes the packet.

On each clock cycle, each switch routes packets and updates its directory. A single packet can be served in each cycle from each non-empty input queue. Conflicts between packets arriving from different incoming queues are resolved at random (or in the order specified by some priority policy). The internal switch structure (see Section 3.2 above) is such that if no contention arises, a packet will be dequeued from each input queue and either be inserted in the appropriate output queue (e.g., for Answer packets)

or gain access to the local data and directory. When contention arises, multiple reads and insertions to an output queue proceed in parallel. Directory updates and local data memory writes are, however, serialized. Whenever a queue is full, it blocks the operations that attempt to insert into it.

The assumption of unlimited memory space on which this straightforward protocol is based is not tractable for large systems. In practice, only limited memories are available and therefore misses on both the data store and the directory must be addressed. Replacement policies for data and directory entries, and routing in cases when information on the location of the data is not available, need to be specified. These modifications to the basic protocol are discussed in the next section.

3.4 Alternative directory organizations

In this section we discuss possible directory organizations. The possibilities fall into two groups. In the first, the "full knowledge" approach, each directory keeps information sufficient to forward on the correct port an arriving request for any data item. These organizations admit relatively simple routing protocols but require extensive directory memory.

The second approach requires that directories keep information on only some subset of the global data items. Here memory requirements are reduced at the price of more complicated protocols in the event that no information on the requested data item exists in the directory.

Dividing the directory organizations into these two groups serves not only to illustrate the possible approaches but also to highlight individually the effects of limiting data and directory memory. We do this in the performance evaluation section that follows by comparing the unlimited memory MHN results to those for the unlimited data, limited directory memory MHN to isolate data memory effects, and subsequently comparing the unlimited data, limited directory memory MHN to the limited data, limited directory memory MHN.

3.4.1 Full Knowledge Bit Map directories (FKBM)

A simple way to keep full routing information is to hold in each switch a full bit-map table, with an entry for each block in the shared memory subsystem. For the dynamic routing protocol in a tree network two bits are enough to encode the three possible relative positions of each data block ("Up", "Left", or "Right"). A fourth logical location information, "Local", is not kept explicitly in the directory, but is available to the router by checking the local data memory.

The obvious advantages of this approach are the simplicity of the protocol and the overall performance that all full knowledge schemes exhibit. By keeping a full bit map, the directory entry corresponding to each data block can be directly accessed and there is no need for associative search. By keeping full routing information for every block, no broadcasting is ever needed. The disadvantages of this approach are the size of the directories needed and the lack of scalability. The directory size in each switch grows

linearly with the size of the global memory for shared data. Bigger directories will not only increase the cost of the implementation but will also mean slower switches, as the time to perform a memory access is a function of the memory size.

3.4.2 Full Knowledge Set Associative directories (FKSA)

An alternative full knowledge organization can be obtained by observing that each switch need keep track only of those data items that currently reside in the memories of descendant switches. Given this "inclusion property" [2], a very simple protocol suffices to effect correct routing: on a directory "hit", the packet is routed either "Left" or "Right" according to the routing information in the directory; on a "miss", the request is forwarded "Up".

As compared with FKBM, under FKSA the directory memory requirement of each switch is reduced from linear in the total amount of global memory to linear in the amount of memory contained in its descendants. Nonetheless, this still represents a substantial number of directory items in at least some switches (those near the root). To make manageable the task of building hardware to support these directories, a set associative scheme is used.

Let us denote by $A_{dir}(i)$ the associativity of the directories of switch i at the l_i^{th} level in the tree, by $A_{data}(i)$ the associativity of the data memory, and by $S_{dir}(i)$ and $S_{data}(i)$ the number of sets in the directory and data memory respectively. Then it can be proven ([2], [7]) that the inclusion condition requires

$$A_{dir}(i) \geq \sum_{j \in Des(i)} A_{data}(j) * \max(1, \lceil \frac{S_{data}(j)}{S_{dir}(i)} \rceil) \quad (1)$$

Note that the number of sets in the data memory and the directories need not be fixed over the entire network. The last factor in the equation above takes into account the ratio of the number of data sets to the number of directory sets. If the number of sets in the (parent) directory is bigger than the number of sets in the (descendant) data memory, the ratio is less than 1, and the associativity of the directory "includes" room for all the entries in the data memory. If the directory has fewer sets than a descendant data memory, several sets of the descendant will fall into the same set in the parent directory, and the associativity is increased to reserve enough room for all the data blocks. Note that the maximum in the equation is taken for each level independently, guaranteeing that there is enough room for the routing information of each data block in each descendant level independently, regardless of the organization in other descendant levels.

3.4.3 The full knowledge data replacement protocol

Both the FKBM and FKSA directory organization are based on finite data memories in the switches. This finiteness causes a "data replacement" whenever a migration packet tries to write into a filled data set. Some modifications to the basic protocol are therefore necessary to handle the cases of data

migration. The only cases when a directory entry is replaced is when a *local data store* in a switch has no place in the appropriate set.

The protocol makes use of an additional type of packet, labeled "DataMigrate" packets, that will be used to migrate a data block when a replacement occurs. Writing new routing information in an unfilled set of a directory poses no problems. Writing new routing information into a filled set requires more care. To avoid any loss of routing information when the location information of a migrating packet needs to replace an existing entry in the directory, the protocol limits the migration of data blocks into its sub-trees. Since there is always enough room in each directory for all data blocks in its descendant switches, a full directory set implies that the corresponding sets in the data memories in the descendant switches are also full, and local data replacement will take place. In these cases, the data replacement protocol will free a line (an entry in the local data set), making room for the new data entry. The protocol makes use of an "overflow" buffer to temporarily store routing information for added entries. During this time, additional data migrations whose directory entries fall in the same set are disallowed. The overflow buffer holds the routing information for data blocks that are currently being inserted into the sub-tree, until a data block is migrated out of the sub-tree to free a directory entry. When the "overflow" buffer is full, no additional migrations are allowed. (Overflow buffer entries are removed when the replaced data arrives at the switch).

3.4.4 Partial Knowledge Set Associative directories (PKSA)

In this section, the full knowledge conditions are relaxed, allowing switch directories to be smaller at the cost of more complicated request routing protocols. This leads to the design of limited set associative directories, holding only partial routing information. The underlying premise is that small associativity can "capture" most of the references because of locality in the access pattern, so that only relatively infrequent directory misses will have to be dealt with in a more expensive way.

The organization of PKSA follows that of FKSA, with the difference that the associativity is not required to increase with the level in the tree. This means that a PKSA switch is incapable of keeping directory entries for all data items in descendant switches. When a switch decides to add a new directory entry and the set into which that item falls is full, the switch simply discards one of the existing entries. There is no need for the switch to notify either the parent or descendant nodes of this action.

Clearly, PKSA does not satisfy the inclusion or full knowledge conditions. However, a modified form of the inclusion property does hold: although blocks can be located in the sub-tree and not have a valid entry in the directory, all the (valid) entries in the directories refer to data blocks located in the sub-tree. Therefore, no blocks that are not located in the sub-tree can have a valid entry in the directory.

An additional requirement of PKSA networks needed to make the data location protocol work is that the root directory *is* inclusive, that is, it contains routing information for all data items held by its

descendants. In this sense, PKSA resembles the DIR scheme. However, there are two important differences. First, while the number of directory entries at the root under PKSA is the same as that for DIR, the amount of information per item under PKSA is substantially smaller. This is because PKSA stores only *routing* information (i.e., "Left", "Right", or "Up"), while DIR must store *location* information (i.e., a processor address). Second, because DIR stores location information, the root must be notified *every* time a data item changes location. In contrast, the root of a PKSA network needs to update its routing information only when the data item is migrated through the root (from one subtree to the other). This potentially results in substantially less root traffic due to data migrations.

The protocol used to handle directory misses takes advantage of the modified inclusion property of PKSA networks. Recall that in full knowledge networks a directory miss guaranteed that the data item was not located in any descendant switch memory. PKSA networks are more complicated, since a directory miss provides no information at all about the location of the data item.

The protocol we use to locate data items on directory miss has two phases: an initial search up the tree until a directory is located that contains information on the data item, followed by a search down the tree to locate the item itself. The search up the tree is performed using SearchUpRead and SearchUpWrite packets. A switch receiving such a packet and neither storing the data item nor having a directory entry for it simply forwards the packet to its parent. We are guaranteed to reach eventually a directory holding routing information for the data item because of the pure inclusion property of the root directory.

Once routing information has been encountered, a downward search for the data item begins. If the switch with the routing information does not actually contain the data item, it sends a Read or Write packet as appropriate down the port indicated by its directory entry. Each subsequent switch receiving a Read or Write packet (and not containing the data item) interrogates its directory for further routing information. If such information is present, the packet is simply forwarded. If there is no directory entry present for the data item, we cannot know which subtree the data item is in. (Although because of the partial inclusion property, the routing information in the switch's parent guarantees that the item is located in one of the two subtrees.) Thus, in this case the switch sends a BroadcastRead or BroadcastWrite packet down both the Left and the Right port.

Finally, a switch receiving a BroadcastRead or BroadcastWrite packet also broadcasts it if it contains no routing information for the data item, converts it into a Read or Write packet if it contains routing information of either "Left" or "Right", and annihilates the packet if it contains routing information "Up". (Leaves not containing the data item always annihilate broadcast packets.)

The broadcast mechanism assures that the data referred to in the packet will be located so long as it is resident in the sub-tree where the broadcast takes place. A problem may arise when a broadcast packet (bp) references a data block (d) that currently is being migrated out of the sub-tree to which bp is being sent. Consider the case in which bp is arriving at switch i from its parent switch j (which implies that there was a miss in the directory at j) and the directory in i points upwards. This can happen if i has

just migrated the data d to switch j and j has not yet updated its directory. A possible way for bp to meet d is by sending bp back to j. The fact that at switch i there is no routing information for bp implies that i would broadcast (alternatively, forward) the request bp to its descendants and to its parent j. But the protocol does not allow backward broadcasting. This can be solved by searching through a buffer holding the broadcasts (or migration packets) recently sent from i. Each broadcast (migrating packet) is placed in the buffer as soon as it is inserted in the output queue, and is removed from this buffer only after an acknowledgement message (BroadcastAck) is received. BroadcastAck is issued by the receiving switch when the broadcast (migration packet) has been received. This handshaking mechanism assures that broadcasts and migration packets do not miss each other even when they are queued or are in the process of being transferred between adjacent switches.

This protocol is more complex than the preceding ones because we cannot guarantee that the set associative directories will satisfy the inclusion rule. Since it is desired to make the replacement of routing information in the directory at the root of a subtree as infrequent as possible, practical implementations should make the associativity as large as possible, and the capacity of the directories should be at least as large as the sum of the capacities of its descendant directories. Nonetheless, the PKSA will require less memory than a comparable FKSA configuration (with the same amount of switch data storage) because PKSA does not have to conform to the pessimistic conditions on the associativity and the number of sets required of FKSA. Still, since the size of directories is increasing as we move up the tree toward global memory, the time to search the directories will increase with their height in the tree network.

4 The Performance of MHN Networks

In this section, the performance of systems built with limited data memories and partial directories is examined. A controlled set of experiments, in which the original assumptions concerning the memories in the systems are gradually relaxed, is conducted. By comparing the performance of the different designs, an assessment of the major consequences of the various organizations is possible.

1. The effect of *data* misses and replacements is evaluated by comparing the performance of the full knowledge MHN network with unlimited memory space to the performance of full knowledge MHN networks with limited data memory.
2. The effect of limited *directory* memory is evaluated by comparing the performance of FKSA to PKSA.

4.1 Workload characterization

It is clear that the workload used in comparing interconnection network architectures can have a strong influence on the results. For example, a (perhaps artificial) workload exhibiting little or no locality of

reference will tend to favor a very simple network built out of fast, dumb switches over a network with smarter, slower switches.

Unfortunately, measurement data about the behavior of real workloads is scarce [4], and so it is not possible to make performance comparisons using "a typical, real workload". Because of this, a flexible abstract model of reference patterns is adopted that allows, through varying parameterizations of a single basic workload model, the exploration of the interconnection network performance over a wide spectrum of possible program behaviors.

The analytical workload model is specified by five parameters:

$$< Shared, AverBurst, NoObjects, Contention, Write >$$

The first parameter, *Shared*, is defined as the fraction of a processor's references that are to shared writable data out of the total number of memory references (both private and shared) that it makes. As only shared references are placed on the MHN network, this parameter controls the overall load on the network.

In comparing conventional interconnection networks and the MHN, we are particularly interested in the impact of locality. Our locality measure, *AverBurst*, reflects the tendency of a processor to reference repeatedly shared variables during a relatively short period of time. It specifies the average length of a burst of references (that is, the average number of consecutive shared references to the same data item.) When one burst ends, the next burst to begin is for a data item that is chosen at random among all the shared writable data items. The number of such items is given by *NoObjects*. While real workloads undoubtedly exhibit more complicated behavior than this simple burst model, we feel that it captures the most salient aspects of those workloads, especially when employed in small models required to obtain performance estimates.

The third parameter, *Contention*, is defined as the fraction of memory references generated by a processor to randomly chosen shared data objects. Each processor is considered to have at all times a set of current "burst variables" which it references with probability $(1 - Contention)$ on each shared variable reference. With probability *Contention* a shared variable reference is made to some other data item, chosen uniformly. These accesses are not included in the burst produced by this processor, but rather represent occasional references to other global data items which are accessed from within a burst.p

Parameter *Write* is defined as the fraction of accesses to shared data that change the value of the accessed data item.

While it is possible to choose different values for these parameters for each data item and processor (as appropriate), a homogeneous system, in which a single parameter value applies to all processors and data items, is assumed. This simplifies the interpretation of results, as well as the construction and manipulation of the models.

To summarize, we briefly describe the processor behavior in terms of workload model parameters. On

each cycle, each processor generates a single reference (provided that it is not waiting for a pending memory request). This reference is either to a private or to a shared data object, in the proportion specified by the *Shared* parameter. Each shared request is either a read or a write, with proportions that are specified by the *Write* parameter. To generate the address of the request, a data item is first selected following the *Contention*, and *AverBurst* parameters. When generating a *Contention* request, this item is picked at random uniformly from the appropriate set of data items. When generating a burst request, the likelihood of continuing the one of the current bursts from the given processor is specified by *AverBurst*.

4.2 Performance evaluation methodology

We have used simulation to obtain performance estimates because of the complexity of the mechanisms being investigated. However, there is a basic limitation to this approach in this domain. So long as we have infinite memory in the switches, the simulation results are insensitive to the number of data objects in the global shared memory. For the infinite memory case, different data objects do not interact and the presence of a data item in a given switch is not affected by references made to other data items. In the finite memory case this is no longer true, since the migration of one data item can cause the replacement of another one. Because of a practical limitation on the simulation run's CPU time, we could not simulate systems with realistic memory sizes. However, it is common practice in studies of this type to run simulations with a limited number of data objects to efficiently produce approximate results. A "scaled down" system is used in the simulation in which the size of the data memory and directory in the simulated system is proportional to the size of real systems. We have checked the sensitivity of our results to the number of data objects in the global shared memory over the range in which it was still feasible to simulate a 128 processors system. The changes in the performance metrics were minor. Unless otherwise stated, the results reported below are a system with 128 processors and 512 shared writable data items.

4.3 The effect of limited data memory

In this section we evaluate the effect of limited data memory. The question to be answered is the extent to which the limitation of the data memory degrades performance due to longer access times (since less data can be stored close to the processors) and the increased network traffic generated by data migrations needed for replacements. We assess this impact by comparing limited data memory versions to unlimited data memory versions, with unlimited directory memory in each case. (The performance of MHN with both limited data and directory memories, as well as a comparison of MHN performance with the simpler PC and DIR designs, is considered subsequently.)

Figure 3 displays the effect of finite data memories on the four MHN architectures for two locality patterns ($AverBurst = 2, 20$) as a function of memory size. (A memory size of '1' indicates that a total storage equivalent to 1024 data objects is distributed among all the switches in the network.) In our

examples there are 512 shared writable data objects (i.e., *NoObjects*=512). Directories are organized as FKSA. The total data memory has been allocated among the MHN switches in an attempt to equalize their utilizations, that is, the ratio of the number of valid entries to the capacity.

The Y-axis of Figure 3 is the ratio of the effective processing power of the system with limited data memory to that of the same system with unlimited data memory. We call this ratio a "speedup gain".

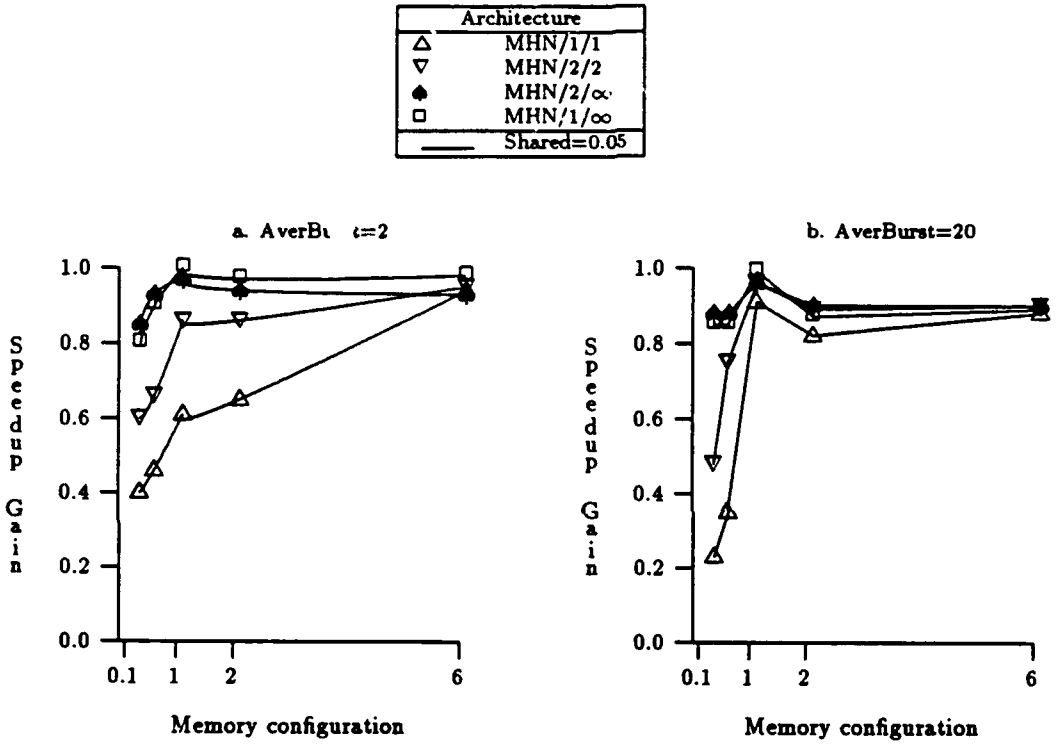


Figure 3: The Effect of Limited Data Memory

It is interesting to compare the sensitivity of different MHN architectures to the limitation of memory space, as a function of the locality in the access pattern. As can be seen, the MHN/1/1 architecture is the most sensitive to the allocation of memory in the switches, while MHN/2/∞ and MHN/1/∞ are the less sensitive . This is expected, as these latter policies do not allocate data in the memories of the network switches. Thus, data objects which are no longer in use percolate slowly towards the root, becoming more accessible to a new user (processor). In fact, because of this phenomenon the performance of more limited memory space systems can be better than that of less limited ones. This is observed for the unit memory configurations, which outperform configurations richer in memory space. Generally, the degradation in performance due to limited memory exhibits a sharp knee. While the 6 memory unit configuration has essentially the same performance as the unlimited memory case, and the degradation due to a 2 memory unit limitation is between about 40% of speedup in the MHN/1/1 case, reducing the data memory further, as 1.0 or 0.1 units, causes a steep degradation.

4.4 The effect of limited directory memory

In this section the effect of limited associativity in the directories is studied. There are three main reasons for performance degradation in the PKSA case:

1. The additional latency on directory misses due to the "search and broadcast" process.
2. The additional traffic introduced by the searching/broadcasting.
3. The additional traffic caused by the BroadcastAck packets used to avoid "data in migration" misses.

To assess the relative effect that the restricted directory organization has on performance, we checked the performance under the PKSA protocol, which includes the search, broadcast and broadcast-acknowledgement phases. We measured the degradation relative to the FKSA case. We also recorded the number of broadcast packets, the level at which they were introduced, and the level at which they were consumed.

Figure 4 displays the expected performance of MHN/1/1 and MHN/2/∞ in their PKSA implementation. (Other MHN policies behave similarly.) The speedup is compared to that of the same architectures with full knowledge directories and the same data memory capacity. We note that for small localities (small burst lengths), performance is almost unaffected by limited directory space. This insensitivity is the result of the fact that for both policies data items never migrate very far from the root (in the case of MHN/1/1 because they cannot migrate very far before being referenced by a processor that causes them to move back toward the root, and in the case of MHN/2/∞ because there are seldom two consecutive references from the same processor to cause migration.)

For higher localities (longer burst lengths), data is more spread out throughout the MHN. Thus, directory information is more important, resulting in some noticeable differences between the limited and unlimited directory memory schemes. The degradation caused is quite modest, however, never exceeding 20%.

Comparing the different migration policies, we observe that the MHN/1/1 architecture is much more sensitive to the organization of the directories than MHN/2/∞. This is intuitive, as MHN/x/∞ policies make much less usage of the data memories of the network switches. The MHN/2/∞ policy performs well because it is selective about which data objects should be placed in the (limited) data memories. Data blocks that are randomly accessed are not migrated, and therefore, the data memory holds only those objects that have a high probability of being accessed in bursts. So long as the capacity is sufficient to contain this active working set the performance is good.

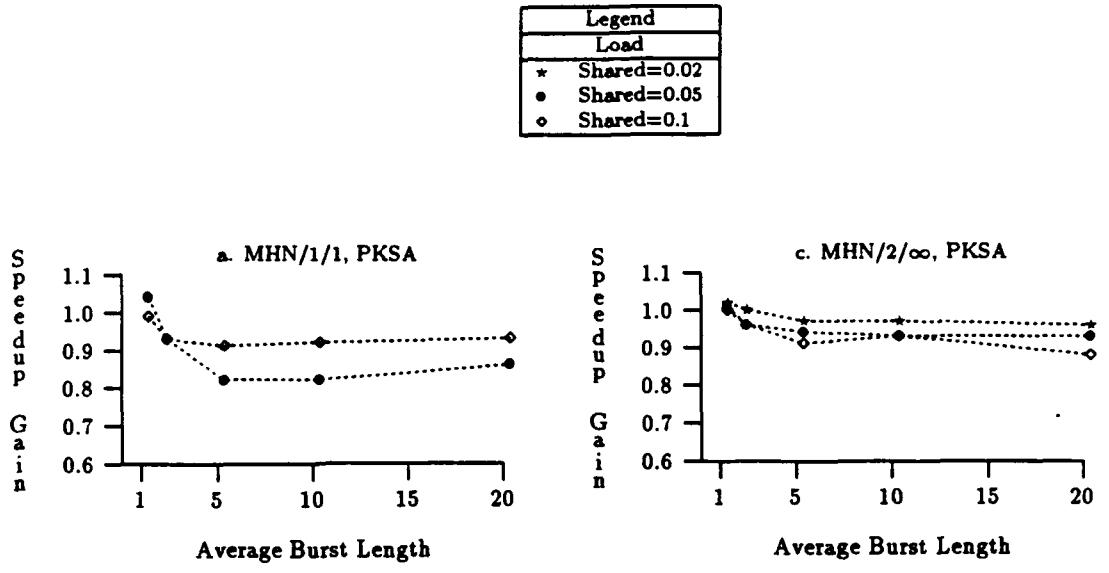


Figure 4: The Effect of Limited Directory Memory

4.5 The bottom line

We conclude this section by exploring the performance of MHN architectures with limited memory relative to the performance of PC and DIR architectures. This will be the "bottom line" of the expected performance, incorporating the degradation due to all the restrictions implied by limited data memories and practical partial directories. All the sources of overhead discussed above (data replacements, searching and broadcasting) are included in this evaluation.

Figure 5 shows the ratio of the effective processing power under a number of organizations to that obtained by the basic PC scheme. As can be seen, except for workloads exhibiting essentially no locality, the expected performance of MHN/1/1, MHN/2/2, and MHN/2/∞ is very good compared to both PC and DIR. All of the MHN architectures exhibit substantial performance gains, but the MHN/x/∞ schemes are clearly superior. This is an indication that much of the performance benefit of the MHN comes from the dynamic routing capability provided by the switch directories rather than from the ability to store data in intermediate levels of the network. Quantitatively, MHN/2/2 exhibits a factor of 2.2 to 3.2 improvement over PC, and MHN/2/∞ a factor of 2.2 to 4.0 (for the longer burst length). However, the MHN/1/1 scheme presents little advantage (and even a potential degradation at light loads) over the much simpler DIR scheme, making the cost effectiveness of this policy questionable.

5 Conclusions

The main conclusion of the feasibility study is that the MHN approach, based on the inclusion of data memories and dynamic routing capabilities in the switching elements of the interconnection network,

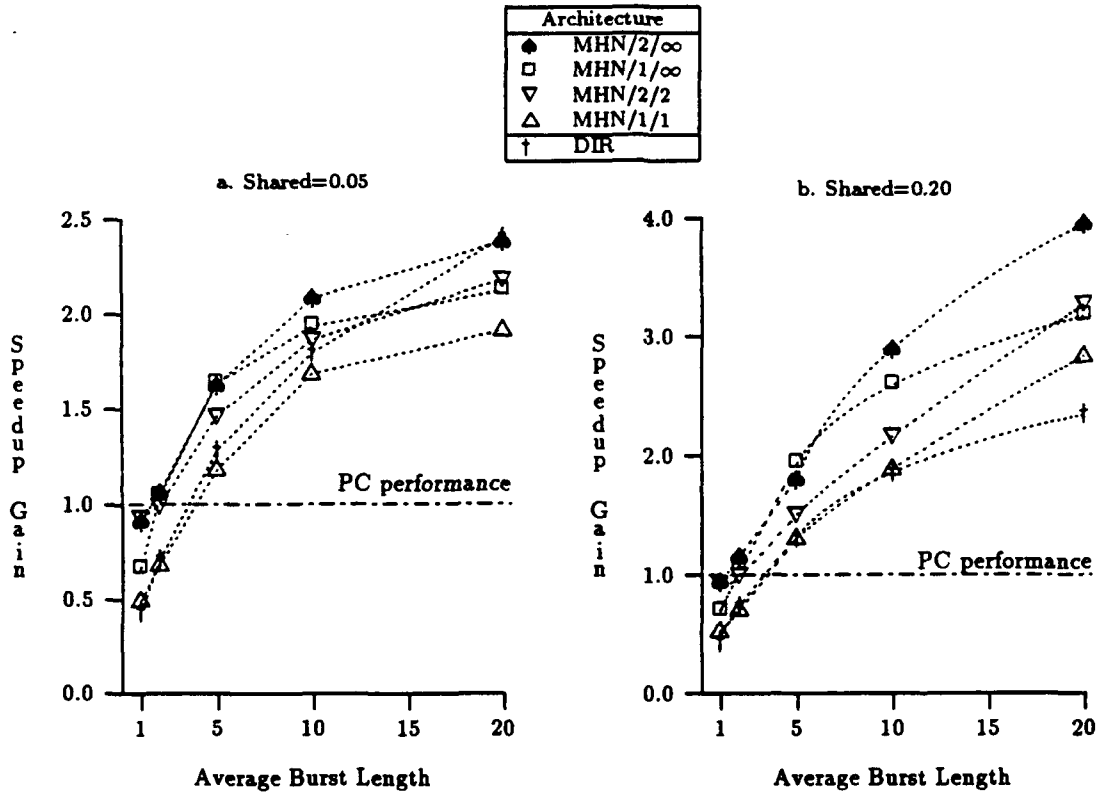


Figure 5: "Bottom line" (PKSA) Performance

is a promising one. The use of sophisticated switches in the implementation of the network for shared data was demonstrated to offer a significant performance improvement.

The price for this performance gain is more complex switches, with a substantial amount of memory. The implementation of such switches requires an ambitious VLSI design. In particular, large and fast memories are required in each switch for directory (and possibly data) storage. Based on the trends in VLSI technology, we believe that such switches will become feasible in the near future.

Specifically for the MHN, an obvious observation is that most of the complexity of the switch structure stems from the directories used to locate the data items. Recall that the dynamic routing capability is the major reason for observed performance gain. In that sense, the price paid for implementing the dynamic routing cannot be avoided. The simulation results show that, under the wide set of assumptions made in the workload model, the dynamic routing capability of the switches contributed most of the performance gain, while the introduction of memory in the switches was found to be beneficial in only a limited domain.

Another important result can be inferred from the performance of the deferred migration policies (MHN/2/x), which performs substantially better than the commonly used "copy on the first reference". We claim that this has general applicability in the domain of multistage networks, where the cost of indiscriminate copying is high. Whenever there is only a single copy for each shared variable, its

location becomes very important. Therefore, the decisions of whether to migrate a data item, and where to migrate it, should correlate with workload behavior. The workload used in our study exhibits bursts that are reliably indicated by two successive references from the same processor. Mapping this as a decision rule into the migration logic of each switch has little ramification on the hardware, but requires additional memory space for state information. The results show that this additional cost is well rewarded, in terms of improved performance.

We have shown several possible implementations and directory organization of MHN with finite memory in a switch and discussed the related switch protocols. The complexity of the protocols is still manageable, and can be mapped into a hardware design. The expected performance loss due to the limited memories is noticeable, especially in the more naive migration policies, but is not enough to put the overall performance gain in doubt. We conclude that building an MHN network is feasible, and that these networks will exhibit considerable improvement in performance. Furthermore, this improvement is monotonic in memory capacity, hence the more memory invested, the better the expected performance becomes. In our study, tree MHN networks were explored for systems having up to 256 processors. Building larger systems is possible, but the performance is expected to be reasonable only for moderate degrees of sharing (e.g., $Shared < 0.05$).

Future research should investigate other topologies such as $NlogN$ multistage interconnection networks and Multicube [5]. Validation of the basic assumptions made in formulating the workload model via measurement of real workloads could enhance the confidence in the results considerably. An approximate analytical model that can predict the performance of MHN networks could be constructed and validated. Such a tool might be useful in extending the study to other topologies and workloads. Implementation of MHN switches and networks, along the principles and guidelines presented here, and implementing the data migration and dynamic routing protocols must be addressed.

References

[1] Agarwal, A., Simoni, R., Hennessy J. and Horowitz, M. "An Evaluation of Directory Schemes for Cache Coherence". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 280–289, 1988.

[2] Baer,, J.-L. and Wang, W.-H. "On the Inclusion Property for Multi-Level Cache Hierarchies". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 73–80, 1988.

[3] Dickey, S., Kenner, R., Snir, M. and Solworth, J. "A VLSI Combining Network for the NYU Ultracomputer". *Ultracomputer Note 85*, June 1985.

[4] Eggers, J., and Katz, Randy H. "A Charaterization Of Sharing In Parallel Programs And its Applicability to Coherency Protocol Evaluation". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 373–382, 1988.

[5] Goodman, J.R., and Woset, P.J. "The Wisconsin Multicube: A New Large-Scale Data-Coherent Multiprocessor". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 422–433, 1988.

[6] Kruskal, C.P. and Snir, M. "The Performance of Multistage Interconnection Networks for Multiprocessors". *IEEE Trans. on Computers*, pages 1091–1098, December 1983.

[7] Mizrahi, E. Haim. *"Extending Memory Hierarchy into Multiprocessor Interconnection Networks"*. University of Washington, Dept. of Comp. Sci., Ph.D. Dissertation, Technical Report 88-11-03, November 1988.

[8] Mizrahi, E.H., Baer, J.-L., Lazowska, E.D., and Zahorjan, J. *"Extending the Memory Hierarchy into Multiprocessor Interconnection Networks"*. University of Washington, Dept. of Comp. Sci., Tech. Report 88-11-10, November 1988.

Extending the Memory Hierarchy into Multiprocessor Interconnection Networks: A Performance Analysis

Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, and John Zahorjan[1]

Department of Computer Science
University of Washington
Seattle WA 98195

December 1988

Abstract

One of the critical problems in shared-memory multiprocessors is to reduce memory latency. In small scale systems where all processors are attached to a single bus, this problem is solved by associating private caches with each processor. Cache coherence is enforced by protocols relying on a fast broadcast mechanism. In larger systems where processors and memory modules are connected by a multistage interconnection network (e.g., an Omega network), this solution is not as practical since broadcasting is cumbersome. Caching can still be present, although in a more restricted fashion and under software control.

In this paper, we explore a *new approach* to reducing memory latency in medium to large scale systems: the use of the interconnection network itself as a component of the memory hierarchy. Instead of allowing copies of an item of shared data to be present in several caches, a single copy of the item migrates in the network according to the reference patterns of the individual processors. Switches in the network contain directories to indicate where the items are stored and local memories to store some of these data items.

The performance of this new architecture – and of several associated migration policies – is compared to more classical architectures under various loads. This evaluation is performed via simulation. Our main results are that indeed this new architecture can improve performance markedly, and that the introduction of directories in the switches is the most important reason for this improvement.

Keywords: multiprocessor, interconnection network, memory hierarchy, performance

1 Introduction

This investigation deals with accesses to shared data in shared memory multiprocessor systems. Attention is focused on medium scale, high performance systems with on the order of several

[1]Our work is supported by the National Science Foundation (Grants No. DCR-8352098, CCR-8619663, CCR-8702915, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

hundreds of processing elements. Because of the number of processors, communication between processors and memory is based on a *multistage interconnection network*. These systems preserve the simple paradigm of programming in a shared memory environment, while potentially providing powerful computing capabilities.

In systems with multistage interconnection networks, the memory reference delay can be a major determinant of system performance. One component of this delay is simply distance. In typical multistage "uniform distance" networks (e.g., the Omega network [13]) each memory module is located at the same distance from each processor. Thus, each memory access exhibits the worst case latency. One solution to this problem is to build a system in which not all memories are equally distant from all processors, thus allowing data of special interest to a particular processor to be profitably located near it even when this increases the distance to many other processors. Hierarchical systems with local memories such as Cm* [19], hierarchical cache/bus architectures [10] [23], and modified single-stage shuffle-exchange networks [21] are examples of this strategy. The Memory Hierarchy Network proposed here, labeled MHN, expands this idea. It is a non-uniform distance network built on top of a conventional uniform distance network topology. Only a single copy of a (sharable) data item exists in the system. Its location is not fixed prior to execution, but may change dynamically during execution in response to processor-memory reference patterns. Dynamically changing the placement of data can be a significant advantage in situations where programs exhibit phases of computation, each of which has a distinct reference pattern, or where the behavior of the computation depends heavily on the data and so cannot be determined accurately *a priori*.

Systems employing local "snoopy" caches [11] [22] attached to a system bus [20] are the primary example of the use of non-uniform distance to improve performance. These systems allow caching of shared data and apply a cache coherency protocol. Extending these protocols to a multistage interconnection network environment is not straightforward. Schemes that use global directories with or without broadcasting (broadcasting is a very expensive operation in these environments) have been proposed [3] [4] [5]. In contrast to these systems, since there is only a single copy of each data item in our proposed architecture, there is no need for the coherency enforcement protocols and the ensuing traffic in the network.

Distance is one reason for memory reference delays. A second reason is contention, which consists of both network contention and memory contention. Network contention is an inevitable consequence of the reduced complexity of the networks. It is avoided in a cross-bar by using $O(N^2)$ switches, but it is because of this complexity that crossbar switches are essentially not scalable. In a multistage interconnection network with $O(N\ log N)$ switches, memory references from two or more processors can compete for a switch, with the result that one or more is delayed [12].

Memory contention for shared variables exists both in cross-bar and multistage interconnection network based systems. It is caused by concurrent accesses to a memory module, and therefore is an inevitable consequence of the access pattern. For shared variable contention, one can further distinguish between contention for synchronization variables ("hot spots" [14] [18]), which can be resolved by combining networks [17], and contention for "data" shared variables. One solution for the latter would be caching, but we argue against it since maintaining coherence is very costly in an interconnection network environment.

The thrust of this study is to handle contention for shared variables using a novel architecture. In the MHN, both network and memory contention problems are addressed in the same manner as the latency problem: by placing data near the referencing processors. This reduces network contention by reducing the number of switches involved in a typical access, resulting in an increase in the effective capacity of the network. It reduces memory contention by distributing data blocks across the switches' memories.

There are two basic questions that one might pose about the MHN: how does it perform, and what is the cost of building it? The construction of the MHN is justified only if the potential performance benefits are sufficiently large. This paper focuses on evaluating these potential benefits. A companion paper [16] is devoted to the design of the switches. It is intuitively clear that the increased flexibility of the MHN (data and directories in the switches) will allow it to outperform conventional multistage interconnection network (MIN) architectures. However, there is a runtime cost associated with MHN that must be taken into consideration. This cost is the dynamic overhead involved in locating a data item referenced by a processor. Because data items may move, it is necessary to keep a set of directories in the switches so that items can be located. Searching these directories causes additional runtime delay over standard networks where data items are at fixed locations. The results that we present in this paper show that despite the runtime overhead, the performance of the MHN is superior to that of currently proposed networks. More specifically, the following questions are addressed:

- How well does the MHN perform under various assumptions about workload behavior? Because measurement data about the behavior of real workloads on shared memory multiprocessors are scarce (examples are [2], [6] and [9]), a broad but abstract model of reference patterns is adopted. It allows the exploration of performance over a wide spectrum of assumptions about sharing and locality of reference. MHN performance is compared to architectures both with and without caching capabilities.
- What is the performance penalty associated with the switch overhead required to implement the dynamic migration function of the MHN? The MHN trades overhead for improved performance if there are dynamic changes in the reference locality of the workload. Therefore applications that exhibit neither static nor dynamic locality are expected to run slower on MHN than on conventional architectures. The extent of the runtime overhead of the MHN is evaluated quantitatively.
- What is the contribution to performance of each of the salient features of the MHN: data storage in the network switches, dynamic placement capabilities, and choice of dynamic migration policy?

The additional functionality of the MHN has its cost in the hardware required to implement the switch functions, which can be quite complicated. Our study is based on the belief that the trends in VLSI design argue for compromising some of the internal switch simplicity (measured by part count), for overall system simplicity and improved performance. In particular, the MHN approach requires the introduction of substantial amounts of memory in each switch.

The paper is organized as follows. Section 2 presents the architecture models of the MHN and of other architectures that will be used for comparison purposes. Section 3 introduces the workload model. Section 4 answers the performance questions raised above. Section 5 presents conclusions and suggestions for further study.

2 Architecture models

In this section the models of the parallel systems used in our study are presented. The systems that we will consider have much in common, including:

- N (a power of two) identical processors and associated local caches used only for code, private and non-writable shared data

- a global memory
- a standard interconnection network (e.g., an Omega network) for access to code, private and non-writable shared data
- a second interconnection network, described below for each individual architecture, for access to shared (writable) variables.

Accesses to local caches always result in hits satisfied in a single processor cycle. This can be interpreted as having "infinite caches" or as having a fast, conflict-free multistage network. Therefore in our performance study, we will not have to model the conventional multistage network.

To simplify further our study, we assume that there is a single memory module storing shared data. Therefore our second interconnection network will be a (binary) tree rooted at the memory and with processors at the leaves. Because the single module case is not preferential to either MHN or the architectures that will be compared to it, the results obtained for this model should be applicable to systems with larger numbers of memory modules.

Read accesses to shared data are synchronous, i.e., the processor waits for the result. The use of the interconnection network, memory (and/or switch for MHN) access latency time, and potential contention for a data item cause a processor to remain idle for some period of time during this request. On a write request, however, the processor immediately resumes computing after placing the request packet on the network.

The unit of time will be the processor cycle time. Access to the local caches also requires a single cycle. We assume switches with "infinite" buffers (so that the non-MHN architectures will not be penalized). When two packets attempt to enter a switch port (there are 6 unidirectional, fully multiplexed ports, 3 input and 3 output to the parent and two children of each switch), one of them, chosen randomly, is served. In the non-MHN architectures, internal busses and logic can transfer a packet from each input port to any output queue in one cycle (see [7] for a possible design). The time to transfer a packet from one switch to its neighbor will also be one cycle.

We now describe enhancements to this model for a baseline architecture and one possible extension. Then, four variants of the MHN architecture are specified. We note that throughout this paper we assume unlimited memory capacity wherever it is needed, both for the MHN as well as for the architectures with which MHN is compared. This assumption, which considerably simplifies the analysis and its interpretation, is analogous to the "infinite cache" performance studies commonly done for shared memory systems [1] [9]. This evaluation is important since:

- It gives an upper bound on performance, thus presenting a first-cut evaluation of the new architecture.
- The performance evaluation of restricted memory implementations of the suggested architecture is necessarily based on an extended set of assumptions, design trade-offs, and protocol choices.
- Memory costs are constantly decreasing, and are predicted to continue to do so. Practical constraints on memory size posed today might not be valid in the near future.

2.1 The baseline Processors-Caches model (PC)

The baseline architecture that we call the Processors-Caches (PC) model is essentially the architecture described above. In order to have a fair performance comparison between PC and more

sophisticated architectures, we introduce a global ("infinite") cache placed between the memory module (root of the tree) and the network. This global cache will have an access time of four cycles. Therefore, in the absence of contention, a read request for a shared variable will take slightly over $2log_2N$ cycles to be serviced.

2.2 Global directory of shared data blocks (DIR)

In this architecture (referred to as DIR), we allow single copy caching of shared variables in the local caches. The cache at the root of the tree network contains not only shared data items but also a central directory of the locations of all shared blocks in the system.

Shared data requests from a processor either are serviced at the local cache (hit) or are forwarded to the root (miss). In the former case, the request is serviced in one cycle. In the latter case (assuming a read), the data is either sent directly to the processor from the root if it is not cached in another local cache (this takes slightly over $2log_2N$ cycles), or otherwise it is sent from the local cache (where it is invalidated) to the root and then to the processor (the trip through the root is always required in order to update the central directory). In absence of contention, this request is serviced in $4log_2N$ cycles. This model corresponds to policy "Dir1NB" from [3], the simplest of directory-based solutions since it does not require cache coherency protocols.

We still assume a switching time of one cycle since the switches are not significantly more complex than in PC.

2.3 The MHN architecture (MHN)

As in DIR, the MHN architecture allows single copy caching of shared data. However, MHN extends DIR in two ways. Firstly, MHN switches can hold data so that now read accesses will be in the range from 1 to $4log_2N$ switch traversals since the data can be present in intermediate stages of the network. Secondly, MHN switches contain directories indicating the location of items ("Up", "Right", or "Left") stored in the subtrees of which they are the roots (local memories in the switch are searched in parallel; "Up" is the initial condition for an item that has never been seen by the switch). Thus, the central directory of DIR is now distributed and partially replicated in the switches. However, since the network is a tree and we have single copy caching, there is no directory consistency problem.

Clearly, the switches are now more complex. Since this performance study is a feasibility study, we assume that the switches have sufficient memory so that there is never need for data replacement (data however migrates, see below, and does not remain always at the same place). Similarly, we assume that the directories are sufficiently large for keeping the full knowledge of what items are stored in their subtrees. A discussion of these assumptions, protocols for accessing and locating data when these assumptions are not held, and implementation of realistic switches are discussed in a companion paper [16]. For this study, we assess a penalty to the switching time, with switches at lower levels of the tree having a one cycle switch time, those in the intermediate levels taking two cycles, and those near - and including - the root taking four cycles. This is summarized in Table 1. The reasons for these various delays will become clearer after we discuss migration policies.

Since data is allowed to migrate and since the MHN architecture allows data to be stored at intermediate levels of the network, a data migration policy is required. This policy must consider two dimensions: "when should data be migrated?" and "how far toward the referencing processor should it be migrated?".

Architecture	Switch delays			
	Proc-level	Intermediate	Global/Full	Remarks
PC	1	1	4	BaseLine
DIR	1	1	4	GlobalDir
MHN/1/1	1	2	4	
MHN/2/2	1	2	4	
MHN/1/∞	1	2	4	
MHN/2/∞	1	2	4	

Table 1: Delays in the various switches

A family of answers to the question of "when" is given by "each time the last j references are from the same processor" for differing values of j. For example, for j=1 data items are moved on every reference, while for j=2 two successive references to the item must come from the same processor before migration takes place. Similarly, a family of answers to the question of "how far" is given by "k steps" for various values of k. Here obvious choices for k are 1 (one step toward the referencing processor) and ∞ (all the way to the referencing processor). We introduce the notation MHN/j/k to denote the MHN policy that moves a data item k positions after j consecutive references by the same processor.

Intuitively, the values assigned to parameters j and k of the migration policy correspond to the assumptions made about the "burstiness" of workload. A workload is considered bursty if it exhibits *alternating periods of high and low frequency* of access to individual data items. In contrast, the workload behavior is considered to be "random" if the frequency of access to an individual data item is relatively constant over time. A workload becomes increasingly bursty when, other factors (in particular, overall average reference rate) held fixed, either the length of the high frequency periods increases or the access rate during the low frequency periods decreases.

Parameter k of the migration policy reflects the assumed length of a burst. The longer a burst is likely to be, the more advantageous it is to move data towards the referencing processor despite the fact that this moves it away from many other processors. Thus, parameter k should be large for bursty workloads and small for random workloads.

Parameter j reflects the low frequency reference rate. It is used to detect when a burst has begun. Some references to a data item are made even during periods of overall low frequency. It is counter productive to migrate a data item in response to these accesses. For a very bursty workload, however, these low frequency period accesses are rare. Thus, for a bursty workload, j can be small, that is, it is safe to assume that (nearly) all references to the data item indicate the beginning of a high access frequency period.

In this paper we evaluate four migration policies:

- MHN/1/1: Move the data one step on each reference. Here a step is one edge in the path from the current location towards the processor that issued the request. This policy is reasonable if successive references made to the same data are made mainly by a subset of the processors; data items would gradually migrate to the node of the network that is the "center of reference" among this set of processors.
- MHN/2/2: Move the data two steps after two successive references from the same processor. This policy is reasonable if a mixture of "random" and "burst" behavior is presumed. It assumes that a burst is reliably indicated by two successive requests from the same processor.

and that otherwise a reference is simply a random reference. Setting k=2 results in MHN/2/2 having almost the same "speed of migration" as MHN/1/1.

- MHN/1/∞: Move the data all the way to the requesting processor on each reference. This policy resembles DIR except that the requested item does not pass through the root if there is a shorter path. On the other hand, there is a penalty incurred in updating all directories on the transfer path.
- MHN/2/∞: Move the data all the way to the requesting processor on each two successive references from the same processor. This policy resembles the previous one, but is more conservative in deciding when a burst has begun.

To illustrate these policies, consider the example of Figure 1 (assuming a MHN/1/1 policy). The data item x, located in switch $S5$, will move to switch $S2$ if the next reference to it is made by a processor in the set $\{P1, P2, P5, P6, P7, P8\}$. If the reference is made from $P3$ or $P4$, it will move to the local cache of the referencing processor.

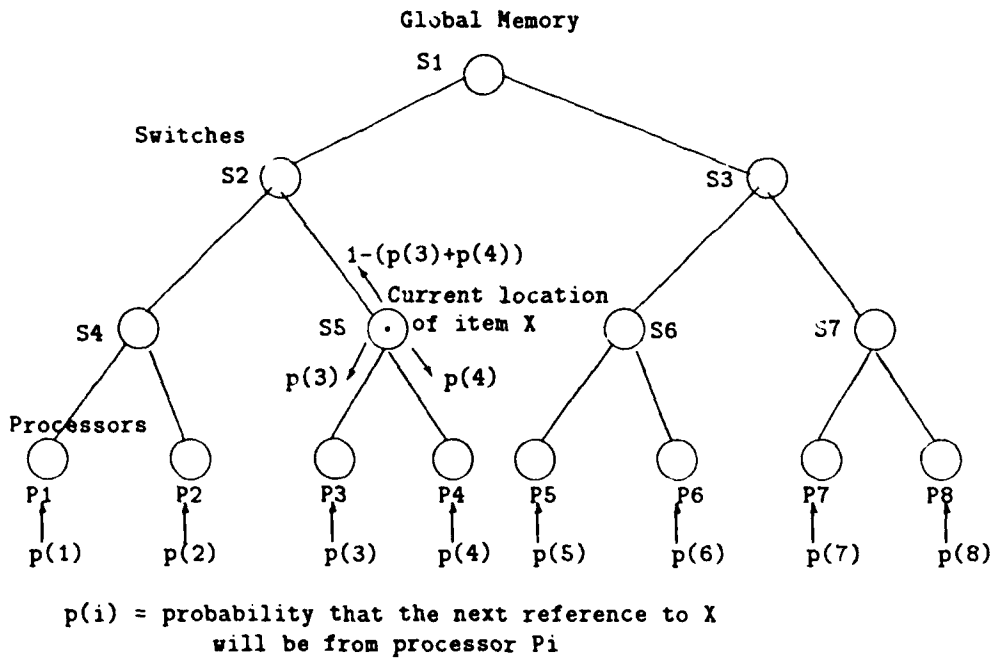


Figure 1: Data migration in a tree MHN/1/1 architecture

Returning now to Table 1, we can give a rationale for the switching times. The MHN switches connected directly to the processors work essentially as local caches. Consistent with the assumptions made in the baseline architecture they are assumed to have a "hit" service time of one cycle. In the intermediate levels of the network the switches have to perform a directory access prior to any routing. This search can take time comparable to the actual data transfer. We thus have assumed a service time of two cycles for those switches. For the root, the existence of a larger directory is reflected by assuming an access time of 4 cycles as in the other architectures.

3 Workload characterization

3.1 The general model

It is clear that the workload used in comparing interconnection network architectures can have a strong influence on the results. For example, a (perhaps artificial) workload exhibiting little or no locality of reference will tend to favor a very simple network built out of fast, dumb switches over a network with smarter, slower switches.

Unfortunately, measurement data about the behavior of real workloads is scarce (see [6] for a source of data; the measurements reported in [2] [9] are for small-scale multiprocessors only) and so it is not possible to make performance comparisons using "a typical, real workload". Because of this, we adopt a flexible abstract model of reference patterns that allows, through varying parameterizations of a single basic model, the exploration of interconnection network performance over a wide spectrum of possible program behaviors.

The analytical workload model is specified by six parameters:

$$< Shared, Write, AverBurst, NoObjects, Contention, Synch >$$

The first parameter, *Shared*, is defined as the fraction of a processor's references that are to shared data out of the total number of memory references (both private and shared) that it makes. As only shared references are placed on the tree network, this parameter controls the load on the network.

The parameter *Write* is defined as the fraction of accesses to shared data that change the value of the accessed data item.

In comparing conventional interconnection networks and the MIIN, we are particularly interested in the impact of locality. By limiting the history dependence of the address generation process to the most recent generation only, we are able to use a simple Markov chain to specify and generate various types of locality patterns. (See [1] and [8] for similar approaches.) Our locality measure, *AverBurst*, reflects the tendency of a processor to reference repeatedly shared variables during a relatively short period of time. It specifies the average length of a burst of references (that is, the average number of consecutive shared references to the same data item).

When one burst ends, the next burst will be for a data item chosen at random among all shared data items (there are *NoObjects* of those).

The fifth parameter, *Contention*, is defined as the fraction of memory references generated by a processor to randomly chosen shared data objects. Each processor is considered to have at all times a set of current "burst variables" which it references with probability $(1 - Contention)$ on each shared variable reference. With probability *Contention* a shared variable reference is made to some other data item, chosen uniformly. These accesses are not included in the burst produced by this processor, but rather represent occasional references to other global data items which are accessed from within a burst.

The last parameter, *Synch*, is used to distinguish between accesses to synchronization variables (that cannot be referenced in bursts) and accesses to shared data objects. *Synch* specifies the fraction of shared data accesses that are made to synchronization variables.

While it is possible to choose different values for these parameters for each data item and processor (as appropriate), a homogeneous system, in which a single parameter value applies to all processors and data items, is assumed. This simplifies the interpretation of results, as well as the construction and manipulation of the models.

To summarize, we briefly describe the processor behavior in terms of workload model parameters. On each cycle, each processor generates a single reference (provided that it is not waiting for a pending memory request). This reference is either to a private or to a shared data object, in the proportion specified by the *Shared* parameter. Each shared request is either a read or a write, with proportions that are specified by the *Write* parameter. To generate the address of the request, a data item is first selected following the *Synch*, *Contention*, and *AverBurst* parameters. When generating a *Synch* or *Contention* request, this item is picked at random uniformly from the appropriate set of data items. When generating a burst request, the likelihood of continuing the current burst from a given processor is specified by *AverBurst*. If a new burst is to start, it will be for one of the other *NoObjects* -1 data items chosen at random.

3.2 Static and dynamic workloads

The workload model parameters allow a wide range of workload behaviors to be represented. To vary each of these parameters independently over a wide range would yield a study so complex that it would be impossible to draw conclusions. Thus, in our studies of the MHN architecture we have restricted ourselves to two classes of workload behavior that are obtained from the general model through appropriate parameterizations. Both classes represent reasonable assumptions about the behavior of real workloads. Identifying results with these workload types allows us to provide a higher-level interpretation of interconnection network performance than if we examined the effect of each workload model parameter individually.

Each workload class in fact represents a family of related behaviors obtained by fixing all but one parameter. By varying that one parameter we are able to evaluate the sensitivity of the results to the parameter that dominates the characterization of the workload class.

The two workload classes are:

- Static workloads. A static workload is one whose future reference behavior does not depend on the results of previous computations. For such workloads, a static analysis of data references can be used to exploit the tendency of certain processors to access predefined data items by placing them near their referencing processors. This is reflected in our model by setting *AverBurst* to 3000. In this way the set of burst variables of a processor changes only very infrequently. A small set of burst variables represents the set of data items of the real workload referenced more frequently by one processor than the others.

  The relative proportion of requests to the preferred and non-preferred subsets of data items is controlled by the setting of *Contention*. We ran our simulation with *Contention* in the range [1.0 - 0.01], which corresponds to the range of of [0.0 - 0.99] for probabilities of referencing a data item within the favorite set of data objects.

- Dynamic workloads. A dynamic workload is one that contains "bursts" of references to shared variables, and exhibits preference for different data items at different times. The frequency at which the preference changes, as well as the burst length, is controlled by the parameter *AverBurst*. In our experiments we have varied *AverBurst* in the range [1 - 20].

The setting of the parameters for these workloads is summarized in Table 2. Both models have a variable load as defined in the *Shared* column. As explained above, while a range of *AverBurst* is evaluated in the dynamic model, only very long bursts are evaluated in the static model. The variability in the static model is introduced by the range of the *Contention* parameter. The *Write* parameter is held fixed at a value that is representative.

Workload	Shared	AverBurst	Contention	NoObjects	Write	Synch
Dynamic	0.01-0.2	1-20	0.1	2	0.2	0.01-0.1
Static	0.01-0.2	3000	0-0.99	2	0.2	0.05

Table 2: Parameter settings for the different workloads

The results reported in the next section actually concern almost entirely the dynamic workload, which we consider to be more interesting and more realistic.

4 Performance analysis

In this section we present the results of our performance study. We compare the various architectures based mainly on two performance measures:

- Speedup, defined as the ratio of the expected elapsed time to perform a computation on a uniprocessor to that on a multiprocessor.
- Efficiency, which is the speedup measure normalized by the number of processors in the system, and therefore is always in (0, 1].

Because our model includes complex characteristics that are not easily evaluated analytically, the performance estimates are obtained via simulation. The simulation model includes the processors and the interconnection network switches. For simplicity, synchronous operation of the system is assumed. Each cycle is partitioned into 2 logically distinct phases: internal updates within switches and processors, and the transferring of packets on the data links. On each cycle each active processor generates a memory reference, which can be either a local or a global memory request. On each cycle, each switch can receive from 0 to 3 requests from different sources as explained previously. Each request can be a "Read" a "Write", or an "Answer". Packets referring to data items that are not stored in the switch are forwarded according to the routing information in the directory. For data stored locally, data access is performed, and migration of data is performed according to the migration policy. Thus a switch can reply with an "Answer" (provides the data for a "Read" request), "Answer Migrate" (like "Answer" + migration according to the migration policy), and "Write Migrate" (in response to a "Write" that causes a migration). The details of the protocol are given in [15].

4.1 Performance of the baseline PC architecture

In the PC architecture, shared data is always located at the root of the interconnection network tree. Thus, the *AverBurst* parameter does not affect performance, and the most important factor on network performance is overall load as reflected in the *Shared* parameter.

Figure 2 presents the efficiency and speedup of the baseline architecture for 32 and 128 processor systems, as a function of *Shared*. It can be seen that while this baseline system performs reasonably well for low degrees of sharing (e.g., when *Shared* is less then 0.05), its performance degrades substantially for heavier loads. This is obviously more pronounced for the bigger system. Note that while the efficiency of the smaller system is higher, both systems exhibit almost the same speedup for *Shared* > 0.10. At these loads, system throughput is dictated by the rate at which

the root can satisfy requests, and thus does not increase as the system size is increased. This phenomenon is well known for tree networks and is the cause of hot spot degradation in multistage interconnection networks [17].

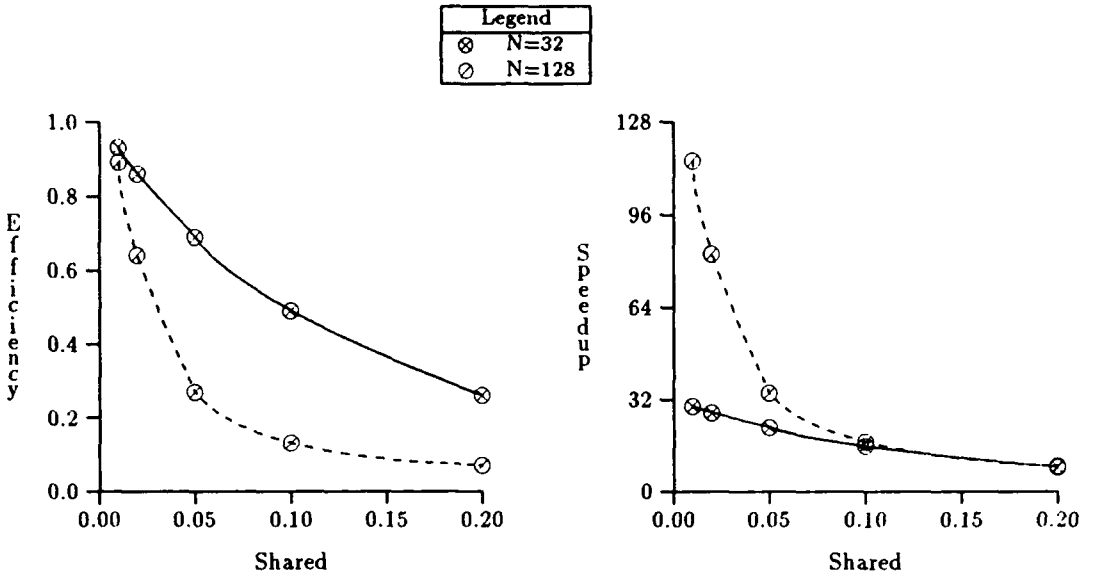


Figure 2: The efficiency and speedup of the PC architecture

Because all global references in the PC architecture must access the root memory module, it is intuitively clear that the major cause of the performance problem in this architecture is contention for resources at the root. Thus, alternative architectures can improve performance only if the dominant bottleneck at the root of the tree can be alleviated.

4.2 Relative performance of other architectures

As we are interested in the relative performance of the various architectures, we plot the ratio of their speedups rather than the absolute values. Such a ratio is called a *speedup gain*. Figure 3 presents the speedup-gain of the DIR and several representatives of the MHN family of architectures over the PC architecture. The results shown are for a 128 processor systems. In each graph, the load in the network, as defined by the *Shared* parameter, is fixed. The left graph in Figure 3 represents the case of a moderately light load (*Shared*=0.05). (Loads that are even lighter exhibit quite similar behavior.) The right graph of Figure 3 shows the case of heavy load (*Shared*=0.2). The gain in speedup is plotted as a function of the average burst length. Depending on the load, we can partition the range of average burst lengths into different subranges, in which a different policy performs best.

From the left graph in Figure 3, we conclude that under a moderately light load, the maximum gain in speedup over the simplest (PC) architecture is significant but is less than 3 (for $AverBurst \leq 20$). This is explained by the fact that even the PC architecture performs reasonably well under such a light workload [2], and that the additional run time overhead of the more complex MHN switches is barely paying off.

[2]The absolute speedup evaluated for a 128 processors PC system with *Shared* =0.05 is 34.1, and thus the maximum possible gain is 128/34.1=3.75.

Thus, when the locality is low (*AverBurst* in $\{1 \ldots 5\}$), there is little incentive to invest in upgrading the network. In fact, PC outperforms DIR, MHN/1/1 and MHN/1/∞ and performs slightly better than MHN/2/2 and MHN/2/∞ when there is no locality (*AverBurst* = 1). This indicates that for very low locality workloads it is best to leave the data at or near the root. Over almost all the range of burst lengths, however, there is a clear advantage for MHN/2/∞ and MHN/1/∞, with the former becoming more and more effective with longer bursts.

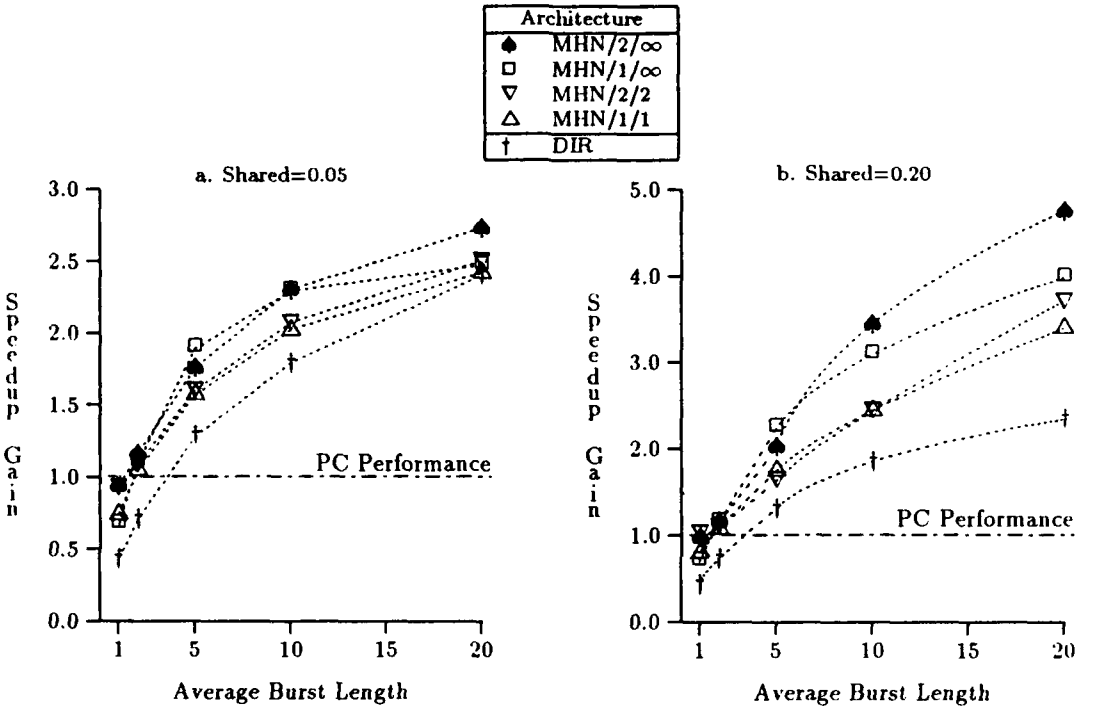


Figure 3: Comparing architectures under dynamic workloads, N=128

The picture changes somewhat when the load intensifies (Figure 3 on the right). Still, when the locality is limited, neither DIR nor MHN/1/1 or MHN/1/∞ pay off, and PC performs slightly better than all the others. But the most noticeable result is that for a very wide range of moderate to long bursts the MHN/2/∞ policy performs the best[3]. Also note that the DIR architecture is outperformed by all the other alternatives over the entire range of workload burst behaviors, is even outperformed by the basic PC architecture for *AverBurst* less than about 4, and proportionally is worse when the sharing is higher.

We can provide some intuitive explanations for these phenomena. First, we focus our attention on the range of applications that exhibit no locality, i.e., those with *AverBurst* of 1. Under such circumstances it is preferable to keep the data at the root than to move it closer to some of the processors at the expense of others. In particular, in a depth $n = logN$ network, if a data item is at level k of the network (where level 0 consists of the switches directly connected to the processors), moving it to level $k-1$ brings it one switch closer to $2^{(k-1)}$ processors but moves it one switch further from $2^n - 2^{(k-1)} \geq 2^{(k-1)}$ processors. Since when there is no locality of reference in the workload each processor is equally likely to access all global data elements, there can be no net

[3]We don't expect to gain by deferring migrations any further by using MHN/n/∞ policies with $n > 2$, as two successive references from the same processor are a very good indicator of the beginning of a burst, and deferring the migration is expected to cause a loss in performance.

gain in average access distance obtained by moving a data item to a lower level of the network. This effect, coupled with the fact that switches in the simple PC architecture can be faster than in more complex architectures, results in better performance for the PC architecture in this case.

When locality increases, as reflected by an increase in the average burst length, DIR outperforms the PC architecture because it allows a processor to access data from its local memory after the first access in a burst, as opposed to accessing it in the global memory all the time (as in the PC network). However, in DIR all the non-local traffic is transferred through the root, which soon becomes a bottleneck for the whole system. In contrast, MHN/1/1 distributes the load more evenly on the network among the switches because of its dynamic routing policy. This feature allows packets to be routed to their destination without passing through the root. In particular, under the uniform random pattern of external references and locality changes assumed in the model, the traffic routed through the root in the MHN is cut approximately in half when compared to DIR. Thus, the network performs better than PC in spite of its slower switches and better than MHN/1/∞ in spite of a slower reaction to a change of locality. However, as soon as the burst length increases beyond a small value (just above 3 in these graphs), MHN/1/∞ and MHN/2/∞ are the policies of choice. This reflects their faster response to the onset of a new burst of references by a processor, since they move the data all the way to that processor in a single move. For the range of moderate length bursts, MHN/1/∞ has a slight advantage over MHN/2/∞ due to its reaction to the first reference in a burst. For longer bursts, the loss of reaction to the first reference in a burst is amortized, and MHN/2/∞ benefits from the avoidance of unnecessary migration present in MHN/1/∞. Consequently, it performs better than all the other policies. At the extreme of very long bursts, i.e., a static load (not shown in this graph), the various MHN/j/k policies perform nearly equally well because they make different decisions only at the beginning of bursts, and this initial behavior is amortized over a large number of succeeding references. Thus, for very long burst behavior, the simplest MHN architecture (MHN/1/1) can be employed.

As can be seen from the above discussion, there are several factors that can contribute to the performance of the interconnection network architecture. Depending on the workload, the contributions of the different architectural features and the policies may have contradictory effects. To provide more insight, we attempt to isolate the respective benefits of the basic features of MHN that can make it advantageous compared to the PC architecture:

- The presence of a directory that enables "caching" of shared data; this directory (global or distributed) keeps track of the location of the single copy of each shared variable.
- The dynamic routing capabilities of a switch, which allows dynamic data migration.
- The deferred migration policy.
- The introduction of memory into the switches.

The first issue will be illustrated by comparing the DIR with the PC model. The benefits of the second feature are assessed by comparing the MHN/1/∞ model with the DIR model. The usefulness of the introduction of memory into the switches is examined by comparing the performance of MHN/1/1 with that of MHN/1/∞. The effect of the migration policy is analyzed by comparing the different MHN variants to one another, in particular, MHN/1/1 with MHN/1/∞ and MHN/2/∞.

4.3 The effect of a global directory

In the basic PC architecture, each data item is at distance $logN$ from each processor. By introducing a directory to this architecture (i.e., the DIR architecture), a data item can be located at distance

0 from one processor and distance $2logN$ from the others. The distance $2logN$ results from the fact that the directory is located at the root, while all data items are located in memory modules directly attached to processors. Thus, as long as the processor at distance 0 is more than twice as likely to access the data item as the aggregate of the other processors, as is the case for sufficiently bursty workloads, there is a net reduction in average access distance to this item. This improves overall performance by reducing the latency involved in accessing the data item and by reducing overall network traffic and its subsequent contention.

This effect is clearly demonstrated in Figure 4 which compares the performance of the PC and DIR architectures. The left graph of Figure 4 presents the relative speedup (gain or loss) of DIR over PC as a function of network load (i.e., the value of *Shared*) for various assumptions about workload burstiness (i.e., for various values of *AverBurst*).

The results corresponding to values 1 and 2 of *AverBurst* show the penalty of moving the data to a processor that has a low future probability of referring to it. Under these conditions it is best to keep the data at the root, equally distant from all the processors. As the average burst length increases, the gain of moving the data to the processor actively referencing it also increases. This gain is relatively insensitive to network load for other than very light loads, where interconnection network performance is not a major factor in speedup.

The right graph in Figure 4 gives a complementary view of the same data, and shows clearly that the crossover point where DIR becomes preferable to PC is near an average burst length of 4 and that for very light loads the speedup gain is small even for long bursts.

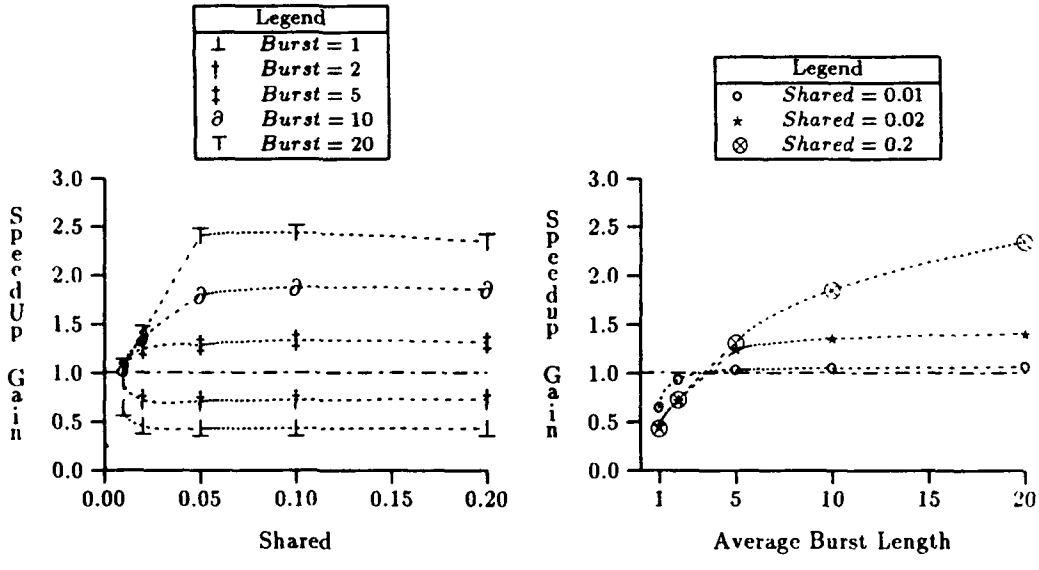


Figure 4: The speedup of DIR relative to PC

4.4 The effect of dynamic routing

The benefits of dynamic routing (i.e., directories in each switch rather than a single, centralized directory) and dynamic data migration are illustrated by comparing the DIR and MHN/1/∞ architectures. Figure 5 plots the speedup gain of MHN/1/∞ over DIR for various network loads and workload burst behaviors. For very light loads, even DIR performs well, and dynamic routing within the network does not result in added performance. As the load intensifies, however, the MHN

network has better performance because it avoids the bottleneck of a central directory exhibited by DIR. For short bursts (*AverBurst=1*), this increase in the speedup gain continues up to a point where accesses to the root create a bottleneck even in the MHN/1/∞ network.

Another interesting effect is the relationship between burst length and speedup gain. The longer the bursts are, the more substantial is the effect of local accesses made in DIR and MHN, which eliminates parts of the network traffic. Thus, for long bursts (static loads) and low rates of access to global data, there is little leverage offered by the dynamic routing to enhance performance. For short bursts, the MHN dynamic routing pays off immediately while DIR performance actually degrades in this range.

4.5 The effect of the deferred data migration policy

The effect of the part of the migration policy that estimates when a burst has been detected is analyzed by comparing the MHN/1/∞ and MHN/2/∞ architectures.

Figure 5 depicts the speedup gain of MHN/2/∞ over MHN/1/∞ as a function of average burst length for various global data access rates. For short average bursts (roughly below 4), MHN/2/∞ performs much better than MHN/1/∞. This is due to the fact that MHN/2/∞ is more reluctant to move a data item, and thus avoids the data movement for short bursts. In MHN/2/∞, once the data is located in a processor, it will seldom move again before the burst is finished. This makes it costly to access the data from the other processors. Still, it is better than MHN/1/∞ which will pay a price of migrating the data on each reference without amortizing it over a long "burst". (Recall from Figure 4 that for short burst lengths it is best to leave all data items at the root.)

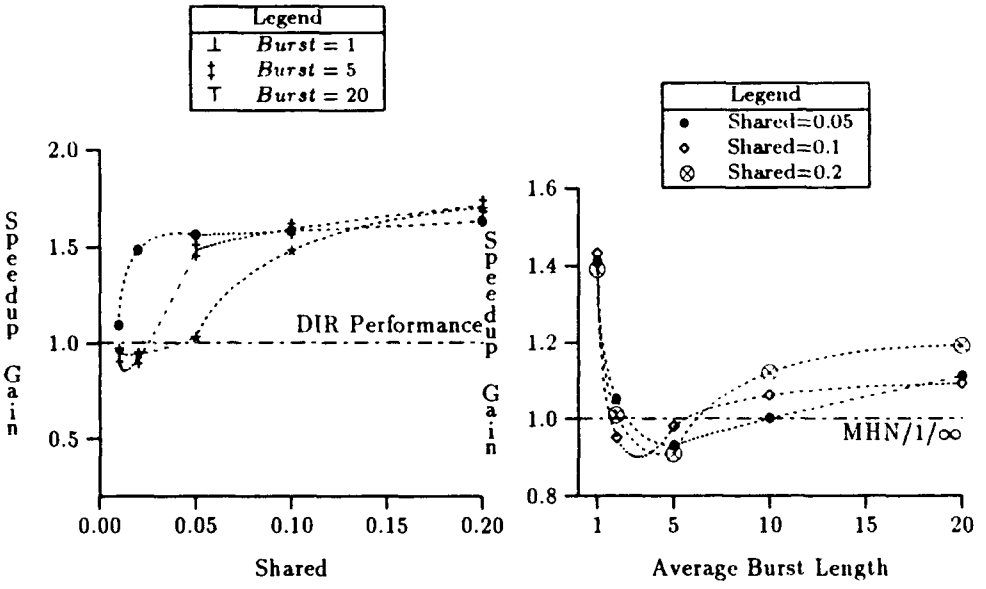


Figure 5: Comparing MHN/1/∞ to DIR, MHN/2/∞ to MHN/1/∞

For moderate burst lengths (*AverBurst* from 4 to 10) the policies yield comparable performance as the faster reaction of MHN/1/∞ to the onset of a burst pays off. However, for longer bursts MHN/2/∞ becomes increasingly advantageous, as it avoids the spurious movements of data that occur under MHN/1/∞ because of occasional references to a data item by processors other than the one currently in a burst. Notice the linear behavior of the gain in speedup in this region.

This reflects the approximately linear increase in the probability that an external reference will fall within an active burst as the burst length increases.

4.6 The effect of introducing memory into the switches

The left graph of Figure 6 plots the speedup gain of MHN/1/1 over MHN/1/∞ and the right one plots the speedup gain of MHN/2/2 over MHN/2/∞ for various network loads and workload burst behaviors. For very short bursts, MHN/1/1 performs better than MHN/1/∞, as the latter pays the price of migrating data without amortizing the cost over several references to the data that was migrated. In these cases, the existence of memory in the switches yields an improvement of 20%. For intermediate values of *AverBurst*, MHN/1/1 performs worse than MHN/1/∞ due to the fact that its slower migration process does not succeed in moving the data into the processors. This advantage of holding data only in the processors is not as significant, but still exists, as the average burst length is increased. Thus it can be seen that the introduction of memory into the switches does not have a clearly beneficial effect on performance. The same lack of significant difference in performance can be observed when comparing MHN/2/2 with MHN/2/∞ (right hand side of Figure 6).

Thus, under the workload we are considering, the introduction of memory into the switches does not contribute significantly to performance. However, we expect that under different workloads, this feature will be more utilized. For example, when data is actively shared only between groups of processors (clusters), the allocation of data to a switch that serves as the "center of gravity" between these processors would make sense. These ideas are further explored in [15], which also includes a detailed analysis of the sensitivity of MHN performance to switch delays and system size, and an analysis of MHN performance under static loads.

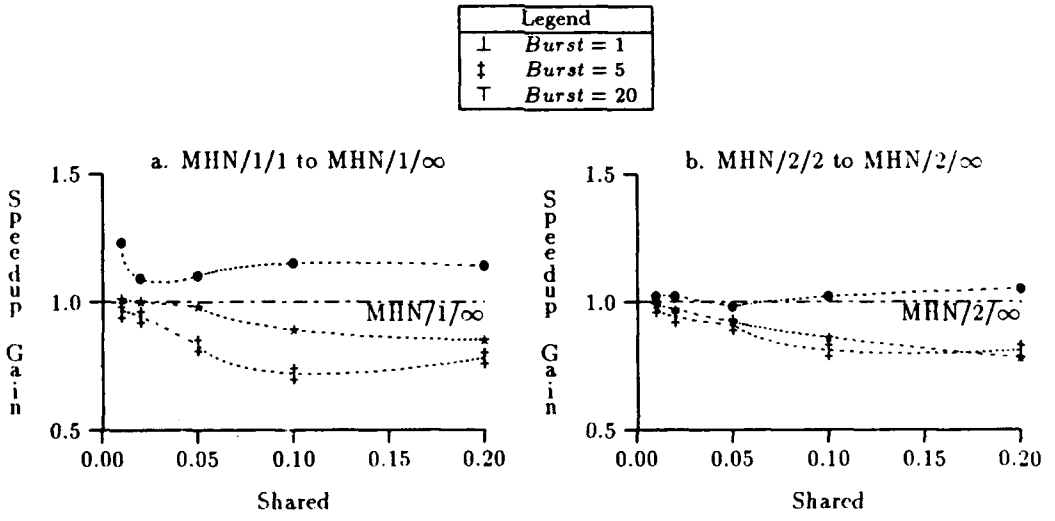


Figure 6: Comparing MHN/1/1 to MHN/1/∞ and MHN/2/2 to MHN/2/∞

5 Conclusions

We divide our conclusions into two parts: those related to the relative performance of the various architectures, and those related to design alternatives for the MHN architecture.

In terms of the relative performance of the various architectures, we note:

- The PC architecture (caching non-shared data only) performs well only when the locality is very low. In fact, when the locality is low it is better not to cache shared data at the processors, in order to avoid "cache-interference", i.e., caching the data at one processor and fetching it from another processor the next time it is referenced.
- The central directory solution, as represented by the DIR architecture, performs much better than the PC architecture when the locality is above a certain threshold. Under the assumptions made in our evaluation, this threshold is an average burst length of about 4. However, accesses to the global directory cause a bottleneck in these systems, and limit the performance in cases where the locality is low or where the rate of globally shared data accesses is high.
- The MHN architecture performs well over the entire range of localities and network loads that we have examined. In particular, under moderate and heavy workloads exhibiting locality it substantially outperforms the PC architecture, and under dynamic workloads it offers better performance than the directory based scheme (DIR).

In other words, our principal conclusion is that the MHN architecture, characterized by the inclusion of data memories and dynamic routing capabilities in the switching elements of the interconnection network, is a promising one. The use of sophisticated switches in the implementation of the network for shared data was demonstrated to offer a significant performance improvement. The price for this performance gain is more complex switches, with a substantial amount of memory. The implementation of such switches requires an ambitious VLSI design. In particular, large and fast memories are required in each switch for directory (and possibly data) storage. Based on the trends in VLSI technology, we believe that such switches will become feasible in the near future.

Looking now at design alternatives for the MHN architecture, we note:

- The dynamic routing capability has a major effect on the performance. This is the main conclusion of the comparison between DIR and MHN/1/∞. We note that it is also the major determinant in the cost of implementation of the MHN. The amount of memory needed for the directories can be an order of magnitude larger than the data memory in the switches.
- The "deferred migration" policy of MHN/2/∞ offers improved performance. This indicates that conventional caching schemes, which create a copy of a data item on every access to it and which were appropriate in bus architectures, are not applicable to multistage networks. In the latter, the cost of data movements and coherence traffic is so large that unless there is a very high likelihood that the data is going to be used heavily, it is better never to move it (or to create a copy of it elsewhere).
- The storage of shared data in the network itself does not improve performance in a significant way for the workload model used in this study. This feature of the network is introduced in conjunction with the selection of a migration policy, which has a dominant effect. Thus, MHN/1/∞ performs better than MHN/1/1, and MHN/2/∞ performs better than MHN/2/2. This is due, at least in part, to the assumptions made in the specification of the workload -in particular, the uniform distribution for external references as well as the uniform distribution of migration destinations. Thus, there is no notion that a group of processors may simultaneously exhibit an unusually high rate of access to certain items, and thus no incentive to store data items at intermediate locations in the network. Note however that there is a rich class of problems, such as block matrix multiplication (where the matrices are partitioned into

blocks) and grid-related numerical techniques to solve partial differential equations (Jacobi iterations or SOR), in which the geometrical (physical) characteristics of the problem induce a geometrical locality in the access pattern. For such applications, the placement of data in intermediate positions in the network will be more beneficial.

In other words, the introduction of memory in the switches was found to be beneficial in only a limited domain, while the dynamic routing capability is the major reason for the observed performance gain.

We claim that the fact that the deferred migration policy performs substantially better than the commonly used "copy on the first reference" policy has general applicability in the domain of multistage networks, where the cost of indiscriminate copying is high. Whenever there is only a single copy for each shared variable, its location becomes very important. Therefore, the decisions of whether to migrate a data item, and where to migrate it, should correlate with workload behavior.

Beyond the architecture itself, we believe that our work will contribute also to the research effort in modeling multistage interconnection networks, specifically those that exhibit non-uniform access costs. We have shown a way to adapt workload characterization to the domain of these networks by introducing new definitions of locality parameters, which are not present in existing models (predominantly geared to bus based architectures). In particular, by distinguishing between different aspects of "locality", we were able to evaluate possible extensions to the basic network design.

References

[1] Agarwal, A. *"Analysis of Cache Performance for Operating Systems and Multiprogramming"*. Ph.D. Dissertation, Computer Systems Lab, Stanford University, 1987.

[2] Agarwal, A. and Gupta, A. "Memory-Reference Characteristics of Multiprocessor Applications under Mach". In *Proc. 1988 ACM SIGMETRICS Conf.*, pages 422-433, 1988.

[3] Agarwal, A., Simoni, R., Hennessy J. and Horowitz, M. "An Evaluation of Directory Schemes for Cache Coherence". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 280-289, 1988.

[4] Archibald, J. and Baer, J.-L. "An Economical Solution to the Cache Coherence Problem". In *Proc. 11th Int. Symp. on Computer Architecture*, pages 355-362, 1984.

[5] Censier, M. and Feautrier, P. "A New Solution to Coherence Problems in Multicache Systems". *IEEE Trans. on Computers*, C-27(12):1112-1118, December 1978.

[6] Darema-Rogers, F., Pfister, G.F. and So, K. "Memory Access Patterns of Parallel Scientific Programs". In *Proc. 1987 ACM SIGMETRICS Conf.*, pages 46-58, 1987.

[7] Dickey, S., Kenner, R., Snir, M. and Solworth, J. "A VLSI Combining Network for the NYU Ultracomputer". *Ultracomputer Note 85*, June 1985.

[8] Dubois, M. and Wang, J.-C. "Shared Data Contention in a Cache Coherence Protocol". In *1988 Int. Conf. on Parallel Processing*, 1988.

[9] Eggers, J., and Katz, Randy H. "A Charaterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 373-382, 1988.

[10] Goodman, J.R., and Woest, P.J. "The Wisconsin Multicube: A New Large-Scale Data-Coherent Multiprocessor". In *Proc. 15th Int. Symp. on Computer Architecture*, pages 422–433, 1988.

[11] Hill, M., et al. "Design Decisions in SPUR". *Computer*, 19(11):8–22, November 1986.

[12] Kruskal, C.P. and Snir, M. "The Performance of Multistage Interconnection Networks for Multiprocessors". *IEEE Trans. on Computers*, pages 1091–1098, December 1983.

[13] Lawrie, D.H . "Access and Alignment of Data in an Array Processor". *IEEE Trans. on Computers*, pages 1145–1155, December 1975.

[14] Lee, G., Kruskal, C.P. and Kuck, D.J. "The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot-Spots' ". In *Proc. 1986 Int. Conf. on Parallel Processing*, pages 11–12, August 1986.

[15] Mizrahi, E. Haim. *"Extending the Memory Hierarchy into Multiprocessor Interconnection Networks"*. University of Washington, Dept. of Comp. Sci., Ph.D. Dissertation, Technical Report 88-11-03, November 1988.

[16] Mizrahi, H.E., Baer, J.-L., Lazowska, E.D., and Zahorjan, J. *"Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks"*. University of Washington, Dept. of Comp. Sci., Tech. Report 88-11-10, November 1988.

[17] Pfister, G.F. and Norton, V.A. "Hot Spot Contention and Combining in Multistage Interconnection Networks". *IEEE Trans. on Computers*, pages 943–948, October 1985.

[18] Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A. and Weiss, J. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture". In *Proc. 1985 Int. Conf. on Parallel Processing*, pages 764–771, 1985.

[19] Swan, R.J., Fuller, S.H. and Siewiorek, D.P. "Cm* - A Modular Multiprocessor" . In *AFIPS 1977 Nat. Comp. Conf.*, pages 637–644, 1977.

[20] Sweazey, P. and Smith A.J. "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus". In *Proc. 13th Int. Symp. on Computer Architecture*, pages 414–423, 1986.

[21] Tan, X-N. and Sevcik, K.C. "Reduced Distance Routing in Single-Stage Shuffle-Exchange Interconnection Networks". In *Proc. 1987 ACM SIGMETRICS Conf.*, pages 95–110, 1987.

[22] Thacker, C.P., Stewart, L.C. and Satterthwaite, E.H. Jr. "Firefly: A Multiprocessor Workstation". In *IEEE Trans. on Computers*, pages 909–920, August 1988.

[23] Wilson, Jr., A.W. "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors". In *Proc. 14th Int. Symp. on Computer Architecture*, pages 244–252, 1987.

Distributed and Heterogeneous Computer Systems

A Heterogeneous Distributed File System

C. Brian Pinkerton, Edward D. Lazowska, David Notkin, and John Zahorjan

Department of Computer Science and Engineering
University of Washington

October 1989

Abstract

The increase in diversity of computing hardware and software has made heterogeneity a fact of life in today's distributed systems. The overall heterogeneous nature of such systems is of benefit to users when working individually, since they are free to choose the environment best suited to their work. On the other hand, this same flexibility makes sharing more difficult.

The work presented here addresses the system service most critical to sharing: the file system. A distributed file system in a heterogeneous environment must provide remote access between machines with different hardware, operating systems, native file systems, and application programs. To be most useful, it must provide convenient access to remote files, which implies that no special preprocessing of files should be required to make them available system-wide. Furthermore, to be successful in an evolving heterogeneous environment, the file system just be easy to build and maintain as well as easy to use.

Our goals for this project were to elucidate the demands on a heterogeneous distributed file system, and to design and implement a prototype to meet these demands. Our prototype, the Heterogeneous File System (HFS), provides a network-wide file system supporting a simple record-oriented file model. Through this standard file model, the HFS provides global access to files stored locally in many different file types, from byte stream to ISAM.

The HFS is implemented as a set of HFS servers, one running on each participating host. Each HFS server extends its host's local file system by fielding remote requests for files stored locally, translating those requests into the appropriate local file system calls, and returning any information so obtained.

Our prototype HFS implementation has been used on a network composed of VAX systems running Ultrix, Sun systems running 4.2BSD UNIX, and Xerox Dandelions running XDE.

1. Introduction

True distributed file systems, those offering access to files in a location-independent way, are a necessity in today's computing environments. The convenient sharing they make possible has decreased the administrative cost of maintaining large, distributed systems and has facilitated co-operative work in these environments. Users of these systems deal with a single logical file space, and are able to access files physically resident on a remote

Our work is supported by the National Science Foundation (Grants No. DCR-8352098, DCR-8420945, CCR-8611390 and CCR-8858804), U S WEST Advanced Technologies, Digital Equipment Corporation's External Research Program, and a Xerox Corporation University Equipment Grant.

machine as easily as files resident locally. The most successful file systems (e.g., Sun's NFS [SGK85] and Carnegie-Mellon's Andrew File System [SHN85]) transparently integrate the remote systems with the local ones, effectively extending the local file system to other machines.

While these distributed file systems are quite successful in homogeneous distributed systems, they have not been designed to deal with the truly heterogeneous environments that are increasingly commonplace. These heterogeneous environments arise because different users have different requirements, with each most naturally met by a particular kind of system. These diverse computer systems generally are not compatible: they have differing hardware platforms, operating systems, file systems, and application software. Because these systems are distributed, they exhibit the same basic need for sharing that motivated the construction of homogeneous distributed file systems. But existing distributed file systems are inappropriate since they rely on system homogeneity.

Fundamental to our approach is that the heterogeneous file system deals only with file accesses between system types, while the existing native homogeneous file systems handle intra-system requests. This division of labor has two benefits. First, because heterogeneity imposes some cost in efficiency and convenience, preserving the homogeneous file systems allows optimal service within each system type. Second, by taking advantage of the existing homogeneous file systems the amount of work required to integrate a new system type into the distributed environment is greatly reduced over that which would be required if the heterogeneous file system controlled file accesses at all levels.

The appropriate measures of success for homogeneous distributed file systems are access transparency and performance. These file systems are optimized for use in environments that present high request volumes for program executables, as well as data. In contrast, we expect the volume of traffic supported by heterogeneous distributed file systems to be much smaller, in part because executables cannot be shared across system type boundaries and in part because in our design the underlying homogeneous file systems take care of the most common case of data sharing, sharing within a system type. With this in mind, we have placed a premium on convenience over efficiency in sharing data files in a heterogeneous environment, where convenience means both the amount of work required of a user to access remote files and the amount of work imposed on the system manager to incorporate new system types into the environment.

The work described in this paper began as part of the Heterogeneous Computer Systems (HCS) project at the University of Washington [NBL88]. The purpose of this work was to elucidate the demands placed on a

heterogeneous distributed file system, and to address these demands in ways that would produce a usable system to supplement existing file systems. Our prototype Heterogeneous File System (HFS) integrates existing file systems by providing an access mechanism allowing users operating on one system to access files stored on another.

Certain realities of distributed, heterogeneous computing environments imposed constraints on the design of the HFS. First, we wanted a design that minimized the difficulty of incorporating new types of systems into the environment, so that the file system would be easily scalable in the heterogeneous dimension. This implies that the introduction of a new system type cannot necessitate the creation or modification of large amounts of code either on the new system or on previously integrated ones, nor the "registration" of files stored on one system with another system. Second, we wanted to avoid modifying system software on our heterogeneous machines, because source code availability is not guaranteed and because of the long-term problems associated with maintaining modified system code. Finally, we wanted a design that would coexist with other file systems; in particular, changes to a file using any method of access should be immediately visible using the other methods of access. Such coexistence improves the ease with which files can be shared since no additional semantics are imposed on the use of globally accessible files over locally accessible ones.

2. Different Types of Heterogeneity

Designing a heterogeneous distributed file system is difficult for a variety of reasons. Some of the problems that must be addressed (e.g., security and authorization, replication and consistency, and full file versus block transfer) are properties of all distributed file systems, including homogeneous ones. A different set of issues relate specifically to heterogeneity. Since our focus is on heterogeneity, we address only these problems, and rely on previous work in homogeneous distributed file systems for established solutions to set of problems held in common.

To understand the impact of heterogeneity on a distributed file system, it is necessary to examine the kinds of heterogeneity that can occur, and how each impedes file sharing. We do this using an example.

Imagine that a client wishes to access a set of images maintained by a server. Each image is stored in a separate file, with each file representing a monochromatic bitmap encoded as two 16 bit integers giving the size of the bitmap, followed by a sequence of 16 bit unsigned integers giving the picture data in row-major order. (Each integer represents a 16 bit wide by 1 bit high slice of the picture.) If the client is operating in a heterogeneous environment, so that the file may reside on any of several system types, then to correctly access the data in the file

the client must contend with the effects of heterogeneity at four distinct levels:

Language Heterogeneity

Some languages use different representations for data stored in a file. For example, a Pascal program might read the file with a series of `READ (NEXT_INT)` instructions. The Pascal runtime expects the integers in the file to be encoded as character strings (so that each integer requires up to 6 bytes of file storage), and to be separated with spaces or newlines. On the other hand, a C program using raw UNIX system calls might expect binary coded integers (each integer requires 2 bytes), with the fields in a record aligned on word boundaries.

Note that in both cases the data's final representation once read into main memory is as a binary coded integer. For the Pascal program, the conversion between the data coding in the file and the internal coding is performed by library routines that lie between the application and the file system. Our heterogeneous file system uses a similar approach: library routines that translate between a specific language's view of data encoding in the file and the internal format.

File System Heterogeneity

There are three primary problems introduced by heterogeneity among file systems. First, different file systems name files differently. To follow the name syntax rules of the system on which the client runs, the client may have to use a name for a remote file that differs from that file's name on its local system. For example, a file residing on a UNIX machine in `/usr/bp/pictures/rhine` might be expressed as `USR$DSK:[BP.PICTURES]RHINE.DAT` using VMS's naming rules. Clearly, having different names for files depending on where the name is issued can be quite confusing, and impedes sharing.

The second problem is that file systems differ in the type and degree of file access control they provide. Some provide a simple owner/other dichotomy while others maintain highly specific access lists. A distributed file system must have a protection scheme that interacts with the local ones. Unfortunately, local file systems usually cannot handle network-wide protection.

The final and most important problem is that different file systems have different models of files themselves. These differences are usually manifested in the operations the file systems define on files. Some systems provide a single file model, such as byte stream. UNIX is an example of this class of system. Other systems may provide several file types, such as sequential record and ISAM. VMS and MVS are examples in this class.

As an example, our bitmap image could be stored in one of several file types, depending on the particular file system involved. On a UNIX machine, the file would simply be stored as a sequence of bytes. On a VMS system, the bitmap might be stored as a one or two record file.

Operating System Heterogeneity

Operating systems differ in the methods of inter-process communication that they provide. Some support communication using complex transport protocols, while others communicate with simple remote procedure calls built upon a datagram transport. If such systems are to cooperate through a remote file system, a common form of communication must be established.

Hardware Heterogeneity

Files contain data implicitly typed by the applications that use the files. Different machines have diverse representations for these data, which include integers, real numbers, characters and even pointers to other file regions. In our example, existing systems represent the integers in the bitmap in four distinct ways, differing in the order of the bytes in the integer and the order of the bits in the bytes. A heterogeneous file system must ensure that these data are in the format appropriate for a client reading a file.

3. Related Work

A variety of distributed file systems accommodate heterogeneity to some degree. Sun's Network File System (NFS) [SGK85] and Carnegie-Mellon's Andrew File System (AFS) [SHN85] are good examples of systems that facilitate interoperability between hosts that have different kinds of hardware but that run basically the same software (UNIX). Because of this operating system homogeneity, neither of these systems must seriously address heterogeneity in file naming, file protection or the file model. (To be fair, recent efforts suggest that interfaces to slightly different file systems can be built using these models.) To truly accommodate hardware heterogeneity, it would be necessary to correctly transform the information in files, but neither system addresses this problem; for example, our bitmap image file would not be transferred correctly between a VAX and a Sun by either NFS or AFS.

The File Transfer, Access and Management ISO standard [III85] was designed specifically to address the problems of heterogeneity. Its approach is to dictate a standard file representation to be used in transferring files between system types. The large, general interface defined by FTAM must be provided on all participating system types. Inter-system access is available to only those files stored in a special FTAM file store. To register a file in

the FTAM store, a user must provide both the file data and a type description for that data. The type description indicates the data type (e.g., integer, real, string, etc.) of the file contents, information that is needed to convert the local data representation into FTAM's standard representation when the file is transferred. A similar design was explored in early phases in the early phases of our work [NHS87]. We created a centralized store for record-based files, and included type information as well as the data itself. Thus, as with FTAM, users had to register local files in a special store to make them globally accessible.

The requirement imposed by FTAM and the early HCS work that files be entered into special file stores has has at least two drawbacks. First of all, shared files are separate from conventional local files. Keeping the centralized copy consistent with the local copy (needed for use by standard utilities such as editors) imposes a significant management overhead. Secondly, globally sharable files must have an associated, often cumbersome, type description, and so to be available an explicit registration step is required. Both of these problems impede easy remote access to files, as well as constraining local access to them.

4. The Heterogeneous File System: Design

The philosophy behind the HFS is simple: to provide convenient access to files stored in a heterogeneous set of distributed file systems. Considering the constraints imposed by heterogeneous environments, and our desire to create a simple collaborative tool, we arrived at the following set of goals:

- The HFS should be simple enough for use on a small scale, with easy-to-learn yet powerful concepts.
- Existing files should be available to applications through the standard, homogeneous file system interfaces as well as being available to new clients via the HFS.
- To ensure consistency among these different file systems, the HFS should not keep any information about individual files. New files created through the HFS should actually reside in some standard homogeneous file system.
- Users should not have to perform any special action (e.g., registering files in a global store) to allow heterogeneous remote access to a file.
- No modifications to existing file systems should be necessary. Indeed, the implementation of the HFS should be simple enough that new file system types can be integrated easily.

The overall structure of the HFS is simple. The HFS makes the files stored in each homogeneous distributed file system available globally through a set of HFS servers (see Figure 1). There is an HFS server associated with each homogeneous file system; this server interfaces between the global set of HFS clients and the files available via that homogeneous file system. In this way, the information specific to each particular system type is encapsulated in

a single HFS server type. At the same time, because the HFS servers share a single file model, a uniform file interface is presented to HFS clients independently of the underlying homogeneous file system in which any particular file is stored. In this way a client can operate on a remote file in a standard fashion without knowing even the type of system on which the file is actually stored.

Because the HFS is a file system, its components mirror those of all other file systems. To specify the HFS fully requires making decisions about file naming, file organization, file protection and a file model. In the following sections, we describe the design of the HFS, separate from any implementation issues. A discussion of our implementation follows.

4.1. File Naming and Organization

Traditionally, there have been three approaches to naming files in distributed environments. In the first, and most simple, files are named by concatenating their host name with their local name on that host. This guarantees that all files have unique names while allowing each individual system to name files locally without consulting the

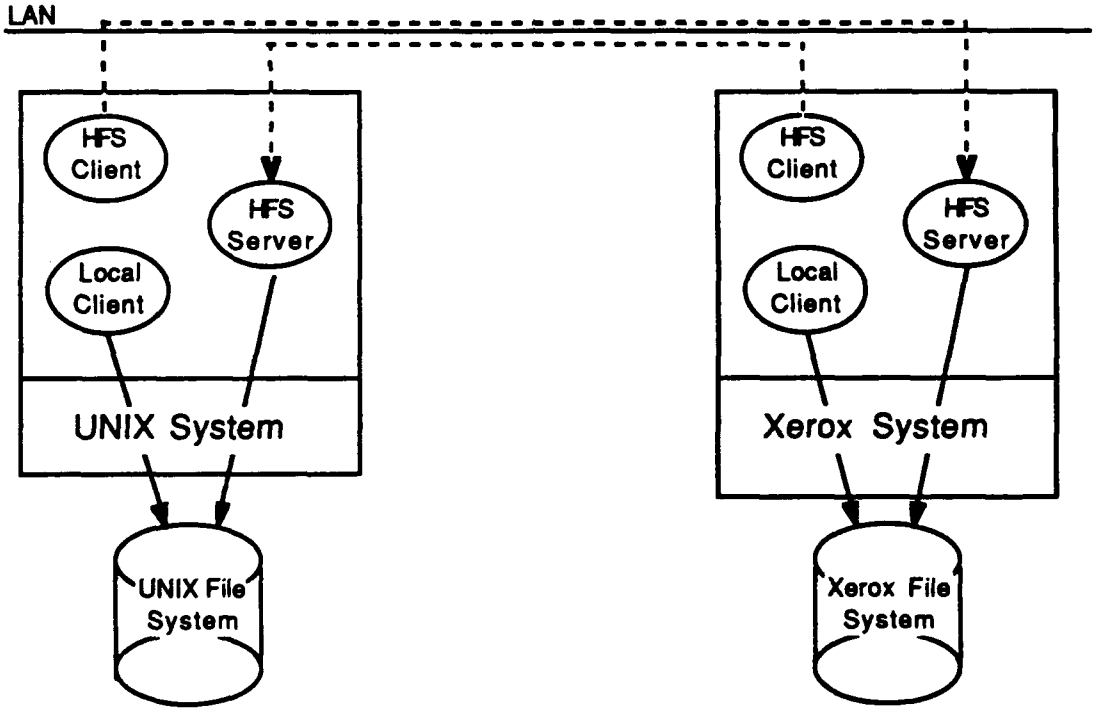


Figure 1: Overall HFS Structure

other systems. IBIS was one of the first systems to make use of this method [TiR84]. The second approach, popularized by Sun's NFS, allows users to "mount" remote directories into the local directory tree. Under this scheme, each user names both local and remote files according to a single, coherent naming tree, although this tree can vary from user to user. The last approach is that of the global directory tree, in which all files exit in a single name space. AFS has successfully used this method for organizing files on a campus-wide level.

Because the first approach most easily accommodates heterogeneity, we chose a variant of it for the HFS. Specifically, HFS names are composed of two pieces, a *name context* and a *local name*. Rather than identifying a host, the name context identifies the name space in which to interpret the local name. Sometimes this name context corresponds to a specific host, since most hosts maintain local file systems with their own name spaces. Often, however, name contexts refer to file systems that are not necessarily associated with any specific host, such as the distributed global tree name space of AFS.

By adopting this approach, the HFS makes no assumptions about the nature of specific file systems' name space organizations or file names. Under each of the other options, the HFS would have had to provide uniform semantics to extend different file systems, some of which are directory based, others of which are flat. Further, we expect that a user typically will obtain the name of a shared file from its creator (e.g, through a mail message or through documentation). Our naming scheme makes it easy for the creator to provide the appropriate HFS file name, since the name depends only on local information naturally known to the creator. Alternatives involving substantially different HFS and local file names would in fact be less convenient for sharing.

4.2. File Protection

Authorization and protection are difficult problems, and have not yet been adequately solved even in homogeneous distributed systems. In the long term, we expect solutions to be outgrowths of work like that of the Kerberos project [SNS88]. Given the goals of our effort, it was clearly not appropriate for us to attempt to solve these problems completely. Instead, consistent with the philosophy of the HFS, we chose to provide a global file protection mechanism resulting from integrating the file access controls available from the underlying homogeneous file systems. We also wanted to avoid maintaining any state information (e.g., file access lists) solely for the purpose of protection, since this would impose a very considerable management overhead, one that we have successfully avoided in all other aspects of the HFS.

Since most existing file systems have a protection class that includes all local users other than the owner, we decided to extend this class to all other HFS users in the computing environment. For example, UNIX and VMS both recognize three classes of users: owner, group, and all others. On systems running the HFS, this "other" class includes all HFS users in the computing environment.

Although extending local protection schemes in this manner may seem risky, we believe that the quantum jump in development investment required, as well as the additional inconvenience in use, resulting from more complicated schemes is not justified by the marginal increase in protection they could provide. This is especially true in the case of our prototype, which serves only a trusted set of users within our own department.

4.3. File Model

Perhaps the most difficult part of designing the HFS was choosing a file model. What should a remote file look like to users of the HFS? What should the operations on that file be? These questions are hard to answer because hosts participating in the HFS could support several different file types, ranging from unstructured byte stream files to highly structured ISAM or VSAM files.

FTAM has answered these questions by building a very general, and consequently very complicated, hierarchical file model. The simpler file structures actually supported by a existing file systems can almost certainly be derived as special cases of the general FTAM file by restriction, that is, by using only part of the full power of the FTAM model to describe them. We decided to take the opposite approach in the HFS, providing only a very simple file model that would be well suited to handling simple file structures and from which more complicated file structures could be built by composition. We believe that our approach has the advantages of being easier to understand (the HFS file model corresponds more closely to the file structures most users are already familiar with) and being faster in execution (again, because the HFS file model more nearly corresponds to what many file systems provide, so that less translation is required).

In the HFS files are viewed as a sequence of records, not necessarily all alike. As well as differing in size, records may vary in type. For example, our bitmap image could be stored in a single file consisting of two records: the first record containing two integers (the number of rows and columns in the image) and the second record containing the actual data. Records in the HFS model can include many data types, including short (16 bit) or long (32 bit) integers, 32 bit real numbers, bytes, characters, strings, arrays, sequences and records. Measurements indicate

that files are typically accessed sequentially [ODH85]. In this case, the varying record types pose no problem to the HFS. The HFS also provides direct access, but with the usual restriction that records be of uniform size and type.

The operations supported on this file model are:

- `open (name, flags, handle)`

  Open a file. `name` is the HFS file name. `flags` give information about the open (e.g., read-only or write-create). `handle` is an opaque return value used in subsequent calls to the HFS to identify the file and provide efficient access to its control information.

- `info (handle, data_pointer)`

  Obtain information (size, last date written, etc.) on a file. `data_pointer` is the address where the returned information should be written.

- `read (handle, record_type, data_pointer, N)`

  Read the next `N` records of the file. `record_type` is a string describing the data types contained in these records. (A description of the language used to encode this type information is given in the next subsection.)

- `write (handle, record_type, data_pointer, N)`

  Write the next N records.

- `seek (handle, record_type, N)`

  Position at the `N`th record (i.e., the next record to be read or written will be the `N`th).

- `close (handle)`

  Close a file.

The HFS translates these requests, which operate on the record-based file model of the HFS, into requests appropriate for the local file under consideration. Thus, for example, in accessing a file actually stored on a VAX running UNIX, an HFS request for one record consisting of ten long integers would be translated into a UNIX request for forty bytes. Similarly, if on a different system the file were stored with eighty byte records, the HFS would perform appropriate buffering and decomposition of records to satisfy the same request.

4.4. Data Typing

Most of the problems with heterogeneous filing mentioned so far have involved software heterogeneity. Hardware heterogeneity poses problems, too, since different machines have different internal representations for data. These differences extend from relatively high level questions of floating point format to very basic considerations of bit and byte ordering. The different representations used by the hardware bases are reflected in the files they store. For example, a bitmap image file stored on a Sun would be nonsense if displayed directly on a VAX. Thus, the HFS needs a way to ensure that data from files is ordered correctly for the client machine type.

Unfortunately, knowledge of the data types of the information stored in a file typically is not available through the local file system. Using our bitmap image file again as an example, there is no way the HFS can determine that it consists of two integers followed by a sequence of integers (as opposed to, say, one float followed by a sequence of characters) because this type information is not maintained in any way by most local file system. Thus, someone must describe to the HFS what kinds of data are in the file.

It is the need for this data type information that motivates in large part the central file store approach used by FTAM and our earlier file system effort. Under this scheme, files are explicitly registered, and both their contents and a description of the type of their contents are maintained by the heterogeneous file system. For the reasons mentioned earlier, we believe that this design discourages sharing. Thus, we rejected the idea of registering file data type information.

If data type information is not registered, it must be presented when a file is accessed, either at OPEN time or with each request. To justify this apparent inconvenience, the HFS makes use of the observation that readers of a file must always know what type of data is stored in it and how that data is encoded, even in completely homogeneous systems. Often, this information is encoded in the form of the read or write function provided in a particular programming language, for instance, the `FORMAT` statement in FORTRAN or the control string in a C `scanf` or `printf` statement. In other cases, the data types can be determined from the types of the variables listed to be read or written, as is done for instance in PASCAL. Thus, it is reasonable for the HFS to demand that its clients supply data type information with their requests. This means that, for instance, instead of simply making a read request that says "*give me the next record from file X*," applications using the HFS must say "*give me the next record from file X; it consists of two integers*". Note that this interface allows the HFS to support files in which the size and type of records varies from record to record, since the data type specification is given on a per-record basis rather than a per-file basis (as would be the case if the type information were given at file open time).

Relying on the client to supply the type information for the records of the file is a significant inconvenience to one class of applications, the so-called generic utilities, such as "copy" and "compare". In a homogeneous environment, the steps taken by these applications typically do not depend on the type or format of the data stored in the files they manipulate. Thus, they can function in local file systems because file types are homogeneous and the operations they perform degenerate to byte-copy and byte-compare. Under the HFS, files can be easily byte-copied and byte-compared, but the results may have little meaning. To copy or compare the information contained in

remote files, as opposed to their raw image, an HFS application specific to a file type could be written. Alternatively, one could imagine an HFS application that simply asked the user for the type of data in a file before operating on that data. Note, however, that this problem cannot be solved without requiring that creators of globally accessible files register their types. The massive overhead this involves in our opinion is not warranted by the marginal benefits it brings.

Given that the client supplies a description of the data type of each record of the file, a question remains as to the encoding used to specify that information. There are several ways in which this might be done. One is the notation currently being used in the ISO presentation layer, where users describe data using a powerful language called Abstract Syntax Notation 1 (ASN.1) [III86]. The Xerox systems use the Courier Interface Description Language (IDL) to describe types to its Courier RPC system. Unfortunately, the power of both of these languages make them difficult to interpret at run time. The HFS uses a simplified variant of the Courier IDL, called DataString, that is easy to parse on the fly.

4.4.1. DataString

DataString is a type description language that represents different data types using a one letter description. Its goal is to preserve just enough information to allow types to be converted between representations while remaining easy to parse by machine. The types DataString offers are the same as those used in the Courier IDL, and include:

B: Boolean
S: String
C: Cardinal (16 bits)
D: Long Cardinal (32 bits)
I: Integer (16 bits)
L: Long Integer (32 bits)
Y: Unspecified Byte (8 bits)
U: Unspecified (16 bits)
P: Long Unspecified (32 bits)

In addition to these basic types, DataString provides four type constructors: array (A), sequence (Q), record and variant. Arrays are ordered collections of known size, while sequences support ordered collections of unknown size. Variant record support is defined in DataString, but not currently supported by the HFS.

The following example illustrates some uses of DataString. Information about a student could be represented by a record containing her name, address, student number, and list of current classes. In conventional notation, the type information would look something like this:

```
Class: TYPE = RECORD [
        department:     STRING,
        class:  CARDINAL,
        passed: BOOLEAN];

Student: TYPE = RECORD [
        name:   STRING,
        address:        STRING,
        number: LONG CARDINAL,
        classes:        SEQUENCE OF Class];
```

In DataString, the type information representing `Class` is the string 'S C B'. The type specification for `Student` is derived from the basic types and the definition of `Class`, giving 'S S D Q {S C B}'.

5. The Heterogeneous File System: Implementation

A number of issues are unresolved in the design of the HFS given above. To explore these issues, we built a prototype HFS and experimented with it. In this paper, we consider some of these issues in the context of the sample HFS file session of Figure 2. That figure gives the HFS client calls needed to read a bitmap from a remote file called `A_HOST:rhine.pix`. In this example, the bitmap is stored as two long integers giving the number of rows and columns, in bits, followed by the picture data packed in short, unsigned integers. It is important that the client know that the data was actually stored in the file as a sequence of short integers so that it can access the data correctly. Were the picture data packed in bytes, for example, the client would need to use a different type to access the file.

5.1. Opening a File

When an application wishes to access a remote file using the HFS, it invokes the local (library) routine `hfs_open`, passing as a parameter the name of the file it wishes to open. To achieve access to the file for the

```
err = hfs_open("A_HOST:rhine.pix", RONLY, &handle);          open the file for read
...check for failure
result = hfs_read(handle, "LL", &buf, 1);                    read # of rows and cols
...compute bitmap size
result = hfs_read(handle, "I", &buf, num_recs);              read the bitmap
...display the bitmap
result = hfs_close(handle);                                  close the file
```

Figure 2: Example HFS Session: Reading a Bitmap from a Remote File.

application, `hfs_open` must do several things: separate the name context from the local path in the name, resolve the name context to the address of an HFS server for the local file system used to store files in that context, establish communication with that HFS server, and present the request to open the file.

Isolating the name context is a simple matter because of the syntax of HFS names. Resolving the context to an address of an HFS server is more difficult. To do so, the HFS makes use of the Heterogeneous Naming Service, or HNS [SZN87]. The HNS, also part of the HCS project, provides a global name space accessible in a uniform manner throughout the heterogeneous environment, and a facility to register and retrieve data associated with those names. In the case of names representing HFS name contexts, the data associated with them is the address of the HFS server managing files in that name context. The HFS thus simply presents the HNS with the name context to obtain the server address.

Once the client has the address of the appropriate HFS server, communication must be established. Once again heterogeneity is a problem, since the HFS client does not know the type of system on which the HFS server runs, and servers in different systems use different communication paradigms. Thus, the client does not know which paradigm to use in any particular call. To solve this problem, the HFS uses another HCS facility, the Heterogeneous Remote Procedure Call (HRPC) [BCL87] [NBL88]. HRPC provides a system independent facility to negotiate a communication protocol with a remote server, and thus insulates the client from this aspect of heterogeneity.

Once the HFS client has set up communication with the server, it is free to make remote procedure calls to it. To finish opening a file, the HFS client simply issues an `hfs_remote_open` remote procedure call to the server indicating the local name of the file to be opened. The server, running in the least privileged class of user on the remote system, tries to open the file. If the open succeeds, the file is publicly readable and hence can be exported. If the open fails, there is either a problem with the file or a protection problem and the appropriate error designation is returned to the client.

When the open succeeds, the server enters some state information into a table associated with the current HRPC conversation. That state information, which includes the file name, open file identifier, and current record number, is also returned to the client for use as a "handle" for future accesses to the file. Were the server to lose the state information in a crash, it would recover it from the parameters supplied by the client on its next call.

5.2. Reading a File

Reading is the most common operation on files [ODH85]. Using the HFS, files can be read either one or several records at a time. These records may vary within a file in size and type (although all records accessed in a single read request must be of the same type). The HFS system is responsible for returning the results in a format appropriate for use by the client independently of the type of the system on which the file is actually stored.

To read one or more records from a file, an application issues a local procedure call to the library routine `hfs_read`, passing in the file handle obtained from the file open operation, the type description of the records to be read, a buffer in which to place the result, and the number of records to be read. The type of the records is specified using DataString, as described in the previous section. In Figure 2, two reads are performed. The first reads a single record containing two long integers (specified by the type string "LL" and count 1) while the second reads multiple (`num_recs`) records, each of which contains a single short integer. Note that while clients need to know the type of the data they access, they are insulated from the details of how the data is actually stored in the remote file system.

To actually perform the read, the local library routine `hfs_read` makes an HRPC call, `hfs_remote_read`, to the remote HFS server. The server receives the request and makes the appropriate file system calls to get the data. When the server has enough data from the file to satisfy the request, it replies to the client. The requested data is sent back in a standard on-the-wire format. Because the type of the data is known only at call time, the server must interpret the DataString description of the records to send them in the correct format. To correctly receive the result, the `hfs_read` library routine acting on behalf of the client also must interpret the type string as it receives that data to convert it from the on-the-wire format to the local data format.

When the `hfs_read` procedure returns to the client application, it passes back the actual number of records read from the file as well as the updated state information returned by the HFS server.

5.3. Other File Operations

The write operation on files is handled in much the same way as the read operation. Interpreters run on both client and server to marshall and demarshall the data according to the record type specified by the user. Concurrent writes are handled in whatever way the remote system would handle them locally. Such handling is possible because there is no caching of writes.

The seek operation allows direct access to records in a file. For files that are record-based, the seek operation simply moves the file pointer to the indicated record. For sequential files, the type information provided with the call is used to compute the size of each record. This, together with the record number specified by the user, allows correct positioning of the file pointer.

When an application is finished with a file, it issues a close operation. This operation performs a close on the remote file, tears down the HRPC connection, and cleans up any local state.

5.4. Implementation Status

The HFS currently runs in server mode on VAX Ultrix machines and Sun 4.2BSD machines. HFS clients, including a text file previewer, a bitmap display program, and an extension of the UNIX finger program that displays digitized photos of the user fingered, run on the VAX and Sun systems as well as on a Xerox Dandelion running XDE.

6. Conclusions

The HFS is a system that differs considerably from current distributed file systems and file transfer programs. The unique requirements of filing in a heterogeneous environment mandate these differences. A heterogeneous file system must account for differences in file naming and protection schemes, file models, operating system communication protocols, and hardware specific data representations. Large installations containing different kinds of hardware and software make changing current systems difficult, and thus motivate a system design that builds on top of existing software.

The needs of the users of heterogeneous file systems will ultimately determine what such a system should include. We believe that the users of a heterogeneous system will generally share application specific files on a small scale basis, thus placing priority on coexistence with local file systems and other remote file systems. This type of use is not well suited to file systems such as FTAM that are based on a centralized store and require users to maintain and register information beyond that already in the file. The HFS provides the ability to meaningfully access any file in the heterogeneous distributed environment with no penalty in the ease of local use or access to that file. It achieves this with a simple, easy to implement interface that does not require any modification of existing systems, and so is makes integration of new system types over time as straightforward as possible. We have shown that a simple implementation of the HFS design is possible. It can be implemented on existing systems easily, and

provides a powerful tool to share files.

Further work is certainly needed in this area. The HFS file naming mechanism, in particular, should be enhanced to allow manipulation of the HFS file name space, instead of simply providing access to file data. In this sense, the HFS is currently a file access method, as it provides no means itself to organize files. Our work here has also ignored certain hard aspects of filing, such as reliability, availability, locking and transactions. Work in these areas with respect to heterogeneity is needed.

7. Acknowledgements

DataString was conceived by Kimi Gosney and formalized by John Maloney. Partial support for our work was generously provided by Bell Communications Research, Boeing Computer Services, Digital Equipment Corporation, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company.

8. References

[BCL87] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems", *IEEE Transactions On Software Engineering 13,8* (August 1987), 880-894.

[III85] *Information Processing Systems - Open Systems Interconnection - File Transfer, Access and Management.*, International Organization for Standardization, April, 1985. ISO/DIS 8571, parts 1-4.

[III86] *Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*, International Organization for Standardization, July, 1986. ISO/DIS 8824.

[NHS87] D. Notkin, N. Hutchinson, J. Sanislo and M. Schwartz, "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity", *Communications of the ACM 30*, 2 (February 1987), 132-140.

[NBL88] D. Notkin, A. P. Black, E. D. Lazowska, H. M. Levy, J. Sanislo and J. Zahorjan, "Interconnecting Heterogeneous Computer Systems", *Communications of the ACM 31,3* (March 1988), 258-273.

[ODH85] J. K. Ousterhout, H. DaCosta, D. Harrison, J. A. Kunze, M. Kupfer and J. G. Thompson, "A Trace Driven Analysis of the UNIX 4.2BSD File System", *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985, 15-24.

[SGK85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Proceedings of the 1985 Summer USENIX Technical Conference*, 1985, 119-130.

[SHN85] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector and M. J. West, "The ITC Distributed File System: Principles And Design", *Proceedings Of The 10th ACM Symposium on Operating System Principles 19,5* (December 1985), 35-50.

[SZN87] M. Schwartz, J. Zahorjan and D. Notkin, "A Name Service for Evolving Heterogeneous Systems", *Proceedings of the 11th ACM Symposium on Operating System Principles*, November 1987, 52-62.

[SNS88] J. G. Steiner, B. C. Neuman and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems.", *Proceedings of the 1988 Winter USENIX Technical Conference*, 1988, 191-202.

[TiR84] W. F. Tichy and Z. Ruan, "Towards a Distributed File System", *Proceedings of the 1984 Summer USENIX Technical Conference*, June 1984, 87-97.

Multi-Language Support for Heterogeneous Remote Procedure Call[1]

Sung K. Chung, Kimiko Gosney,
Edward D. Lazowska, and David Notkin

Department of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195

Abstract

A remote procedure call (RPC) facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports network communication among application programs while relieving programmers from concern for data encoding, transport protocol details, etc.

An RPC facility developed as part of the University of Washington's Heterogeneous Computer Systems Project dealt effectively with system heterogeneity, but not with language heterogeneity. We have overcome this limitation by extending the facility to allow users to write applications in Franz Lisp and Smalltalk-80 as well as in C. This multi-language support for the Heterogeneous Remote Procedure Call (HRPC) facility is the focus of the present paper. Our results include: (1) a multi-language RPC model, (2) type mappings between the HRPC interface description language and the languages in which HRPC applications are written, (3) stub structures for these languages, and (4) experiences in implementing the language subsystems.

1 Introduction

Remote procedure call (RPC) has become widely adopted as a distributed programming paradigm since its introduction by Birrell and Nelson in the early 1980s [2]. An RPC facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports network communication among application programs while relieving programmers from concern for data encoding, transport protocol details, etc.

To a first approximation, an RPC facility works as follows: The client (caller) and server (callee) modules are programmed as if they were intended to be linked together. A description of the server interface – that is, the names of the procedures that the server implements and the types of their arguments – is processed, yielding two *stubs*. The client stub is linked with the client; to the client this stub is indistinguishable from the server. The server stub is linked with the server; to the server

[1] This work supported in part by the National Science Foundation (Grants No. DCR-8420945, CCR-8611390, and CCR-8858804), Digital Equipment Corporation (the External Research Program), Tektronix Corporation, and the Xerox Corporation.

this stub is indistinguishable from the client. A call by the client to the server is fielded by the client stub, which transmits the arguments of the call to the server stub. The server stub then invokes the server, receives any results, and passes them back to the client via the client stub. In this way, the two stubs shield the client and server from the details of network communication.

As part of the Heterogeneous Computer Systems (HCS) project [3] at the University of Washington, we designed and implemented the Heterogeneous Remote Procedure Call (HRPC) facility [1], which emulates existing RPC systems at low implementation and run-time cost. The original HRPC facility permitted users to construct systems in two ways. Users could write clients or servers using native RPC facilities that the HRPC facility could emulate (including Xerox Courier, Sun RPC, etc.). In this case, any language-level support provided by the native facilities could be used. Alternatively, users could write HRPC clients and servers, which could in turn be connected either with any peer written in an emulated native RPC facility or else with any peer written using HRPC. Initial HRPC language support was for C.

Due to their greater generality, it is more attractive to write HRPC clients and servers than to write native clients and servers. But requiring HRPC systems to be written in C is a significant limitation. In other words, our initial HRPC facility dealt effectively with system heterogeneity, but not with language heterogeneity.

We have overcome this limitation by extending our facility to allow users to write HRPC clients in Franz Lisp [12] and Smalltalk-80 [7], and to write HRPC servers in Franz Lisp. This multi-language support is the focus of the present paper. Our results include (1) a multi-language RPC model, (2) type mappings between the HRPC interface description language and the languages in which HRPC applications are written, (3) stub structures for these languages, and (4) experiences in implementing the language subsystems.

2 A Multi-Language RPC Model

By a "multi-language RPC system", we mean an RPC facility in which clients and servers written in two or more different languages can communicate.

The HCS philosophy has been to emphasize modularity and reusability of code, accommodating existing systems without modification whenever possible. We have extended this philosophy to support multiple languages in the context of HRPC. Our effort has adopted the following constraints:

- A user of a given language-specific HRPC subsystem must be protected from having to understand anything about the underlying implementation of HRPC. For instance, a user of our Smalltalk subsystem need only manipulate the Smalltalk application program, written in terms of classes and objects; no knowledge of C is required. Furthermore, the flavor of an HRPC application language should be retained as much as possible in using the HRPC subsystem. For example, with the Lisp subsystem, a remote service entity should be expressed as a set of Lisp functions.
- The services by all HRPC application languages should be shared, thus minimizing initial development and continuing maintenance efforts. In general, support for language heterogeneity should be independent of support for system heterogeneity. In particular, the existing HRPC

run-time system should be used. One reason for this is to avoid the cost of porting the run-time support to another language. Another reason is that in most cases a rewritten version of the run-time system would be far less efficient than the original one, especially for languages such as Lisp or Smalltalk. A final reason is that, by using the existing run-time system, future changes with HRPC will be automatically adapted to other existing and future languages with virtually no additional cost.

- The compilers, linkers, and loaders for the various languages must not be modified.

Two key questions arise. First, how are the stubs in different HRPC application languages structured? Second, how do the invocations made by these stubs connect to the underlying HRPC run-time system, which (except for C) is written in a different language? The following four-level model demonstrates our approach to addressing these problems:

- The top level of the model is the *application level*, which represents user programs written in an HRPC application language. These user programs call procedures defined by the service, which is defined by the interface description. Thus all the problems at the application level involve the syntactic and semantic mapping between the interface description language and an HRPC application language. Among the problems to be considered at this level are: how to denote a remote procedure call with its parameters in the HRPC application languages, data type mapping, and error and exception handling.

- Next is the *stub level*, which is in the same language as the user program, and hides the interactions with lower levels. The choice of stub structure for an HRPC application language is critical. One of the criteria for stub structure alternatives [4] that is most affected by the multi-language aspect is whether to use compiled or interpretive stub structures. Our stubs for Lisp and Smalltalk are interpreted.

- The *language interconnection level* then maps the stub level into the underlying run-time system. This "bridge" level translates RPC invocations in the HRPC application language to calls in the language that defines the run-time system. This level is largely dependent on the HRPC application language, the system language, and the system's inter-language communication mechanism. We implemented the interconnection between the HRPC run-time support and the application language environments in two different ways, inter-language procedure call (`cfasl`) for Franz Lisp and cross address space communication (pipe) for Smalltalk. As discussed in section 4 and 5, this level also decides how data type mapping and checking is done.

- The bottom level is the *run-time level*. This level implements the RPC run-time support, which generally provides "helper" procedures for the underlying RPC mechanism, and which is usually written in a system language for reasons of efficiency and interface to the operating system. In our approach to multi-language RPC, the run-time level is considered to be *given* by the existing RPC run-time support. The model is independent of the implementation of the underlying RPC run-time support.

Figure 1 depicts the structure of the HRPC system extended with multi-language support according to our four-level model.

We call the top two levels the *language layer* since the HRPC application language defines that portion. The language interconnection level is trivial if the application language is the same as

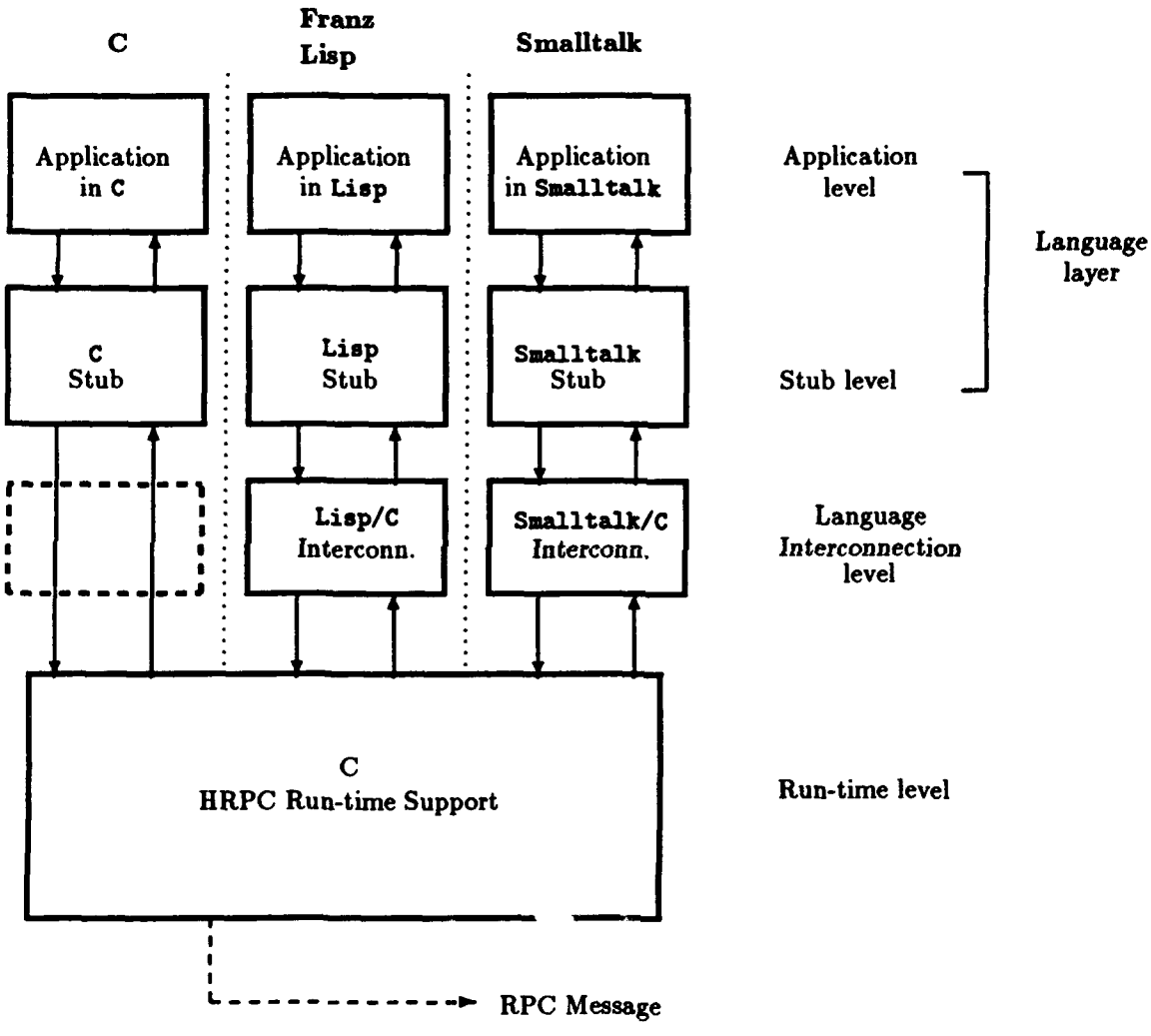


Figure 1: HRPC Structure Extended with Multi-Language Support

or very close to the system language. Otherwise, some inter-language communication mechanism, which naturally depends on both the application and system languages, must be provided.

Due to the differences between languages – in our case the interface description language and application languages – and their implementations, support for any application language must address several aspects of heterogeneity that must be either accommodated or eliminated: new data structures, or different restrictions on comparable data structures; layouts of data structures in memory; differences in calling conventions and call return semantics; error handling; and existence of a mechanism to access code in another language.

Our model does not address the case where the existing RPC run-time support cannot be ported to a system, e.g., a Lisp machine with no RPC facility. In such a case, there is no alternative but to implement a Lisp version of the RPC run-time facility first (assuming Lisp is the system language for the Lisp machine). Our model is independent of the system language the RPC run-time is written in. Thus, once an RPC run-time support is built, the four level model is applicable to building RPC subsystems for other languages.

HRPC Run-Time Support Numerous RPC facilities have been designed. In general, each facility has adopted a fixed set of conventions defining the key protocols and mechanisms: network standards for data representation, which define how integers, records, arrays, strings, and so on are serialized for transmission across the network; transport protocols, which actually pass data over the network; control protocols, which implement the procedure call protocol using low-level network messages; naming and binding mechanisms, which allow clients to find and link to servers; and activation characteristics, which determine whether servers must be running when they are called, or can be spawned dynamically when a call to them is made.

HRPC explicitly defines abstract interfaces for each of these, allowing multiple implementations in each dimension. In contrast to conventional stubs, which include information about the specific decisions made in these dimensions, HRPC stubs are defined in terms of the abstract interfaces provided by the HRPC run-time. These interfaces are represented by a group of related procedure pointers. At bind-time, when a client connects to a server, the matched implementations for the client-server pair are selected and bound to the procedure pointers in an explicit binding structure called an `HRPCBinding`. Hence, a single HRPC stub can be used to communicate using multiple native RPC facilities. By combining appropriate implementations, different RPC facilities can be easily emulated. The HRPC run-time support provides all the implementations of the abstract interfaces; it is also responsible for providing the binding mechanism.

In the following three sections, we describe HRPC subsystems for C, Franz Lisp, and Smalltalk. The descriptions follow the four level model.

3 C HRPC

This is a brief description of the original HRPC facility for C. Readers are referred to the paper by Bershad et al. [1] for further details of the internals of HRPC.

3.1 Application Level: C HRPC

An HRPC client calls a remote procedure of a server interface just like a local call. In fact, it is a local call to the corresponding *stub* procedure. Let's take an example of the FingerService specified as follows in HRPC-Courier.

```
SeqStr:  TYPE = SEQUENCE OF STRING;

WhoAll:  PROCEDURE [] RETURNS [names:  SeqStr] = 1;
                -- list of all users currently logged in the server host
WhoIs:   PROCEDURE [user:  STRING] RETURNS [info:  STRING] = 2;
                -- finger info of a user
```

The FingerService interface defines two procedures, **WhoAll** and **WhoIs**. The following code shows how those remote procedures are invoked from the C client side:

```
#include <FingerService.h>
                ....
        HRPCBinding * clientBinding;
        SeqStr fingerAll;
        String fingerInfo;
                ....
        FingerService_Import(clientBinding, "condor");
        WhoAll(clientBinding, &fingerAll);
        WhoIs(clientBinding, "John", &fingerInfo);
                ....
```

Although HRPC procedures are similar to ordinary C procedures, there are some differences: First, since Courier allows multiple return values, in contrast to C, the pointers to the return variables (**&fingerAll and &fingerInfo**) are given as part of the arguments. Second, the binding handle (**clientBinding**) is passed explicitly to a remote procedure call.[2] The binding of procedures to a particular instance of a service is done by **Import**. As the result, the binding structure is filled according to the control, data representation, and transport protocols selected for the client-server pair.

The differences between the interface description language and the C language are bridged in part by the programmer. The HRPC stub compiler generates a C version of data structures definition from an HRPC-Courier data type description in a header file (**FingerService.h** in the example). It is programmer's responsibility to properly use the HRPC-related data structures. Some types (e.g., integer subtypes, array, record, etc) are transparently mapped between HRPC-Courier and C, others are less transparent. For example, a Courier **SEQUENCE** (a variable size array) is mapped to a C record (**struct**) with a length field and a pointer to the array body. To fill a **SEQUENCE** structure in C, the length field should be assigned with the actual size of the array body pointed by the pointer field.

[2] Technically, the binding handle can be hidden under the stub level.

3.2 Stub Level: C HRPC

Application-specific stubs are generated from interface descriptions. The C stub compiler scans a Courier description file, creating a tree-structured symbol table.[3] As types and procedures are recognized the stub generator emits code into client and server stubs.

The C HRPC stubs are compiled-procedural stubs [4] in the sense that all the data type dependent information is *compiled* in the stub codes and (un)marshalling is performed by *procedure calls* to (un)marshalling procedures. For example, a part of the client stub routine for **WhoIs** procedure of the FingerService is:

```
/*#define WhoIs FingerService_WhoIs                    /* in FingerService.h */

FingerService_WhoIs(Bptr, user, info)
        HRPCBinding *Bptr;
        String * user;
        String * info;
{
        rpcControl * rpcDescr = Bptr->rpcDescr;
        otwControl * otwDescr = Bptr->otwDescr;
                .....
        (*rpcControl->InitOutgoing)(Bptr, procNum, versNum);
        (*otwControl->String)(Bptr, &user); /* marshall */
        (*rpcControl->FinishOutgoing)(Bptr);
        (*rpcControl->InitAnswer)(Bptr);
        (*otwControl->String)(Bptr, &info); /* unmarshall */
        (*rpcControl->FinishAnswer)(Bptr);
}
```

This shows the basic structure of HRPC stubs, which reflects a canonical abstraction of general RPCs. The rpc-control and (un)marshalling routines are invoked through procedure variables in the binding structure which are assigned with protocol specific procedure pointers at binding time. The stub codes for (un)marshalling are generated according to the data types of input and output parameters.

3.3 Language Interconnection Level: C HRPC

Since the language layer for C can be directly linked with the HRPC run-time support, there is no need for a special language interconnection level.

[3] Actually, it is a tree of such symbol tables. Any Courier description referenced by **DEPENDS UPON** is a branch of this parent tree, as is the main program.

4 Lisp HRPC

We wish to provide easy connections between Lisp clients and system services, especially the system services provided by HCS such as mail servers, file servers, remote computation servers, etc. The initial goal for providing Franz Lisp programs with HRPC capability was to explore, in our context, the difficulties arising from the classic inter-language problems and in preserving the flavor of a language of a different family.

Lisp's lack of static typing makes it difficult to map data into the structures supported by our original C HRPC run-time system. The mapping, along with the necessary limitations, must come from the supporting language layer, without explicit intervention by the Lisp user. Support must be automated so that the user does not need to know or deal with the limitations of the support services.

The kind of support needed for Lisp falls into two cases: the use of system services written in another language, and the use of Lisp services by Lisp clients. The first case requires that the data sent by the Lisp client be of the correct type for the server. For an Algol-family language, this would be checked at compile-time. Static checking is not, however, in the Lisp style, but it could be supported by a statically generated check executed at run-time. This approach would work for all except Lisp clients of Lisp servers. This situation requires dynamic typing, to support the syntactically untyped character of Lisp function arguments. The tension between these requirements led us to an interpreted Lisp-knowledgeable layer, which provides a ready opportunity to do run-time type checking of dynamically defined types.

4.1 Application Level: Lisp HRPC

We want to make our Lisp support *look and feel* like Lisp to Lisp users. It has normal Lisp call and return semantics, in so far as is possible.

The interface between Lisp HRPC stubs and a Lisp application is quite similar to that of the original HRPC facility. A Lisp client imports a wanted server using `Import` function, which returns a binding structure. The functions in the HRPC-Courier description can then be used as though they were local functions. (A Lisp server exports itself and serves RPCs in a similar way.) A Lisp version of the FingerService client introduced in section 3.1 can be written as follows.

```
(setq fsBinding (FingerService_Import "condor"))
(setq fingerAll (WhoAll fsBinding))
(setq fingerInfo (WhoIs fsBinding "John"))
```

This program segment imports the FingerService server on "condor" and calls two remote procedures, `WhoAll` and `WhoIs`, to get the results in the two variables `fingerAll` and `fingerInfo`. All the results of an RPC are returned in a single list.

By the interpretive nature of Lisp, once Lisp HRPC stubs are generated and loaded, they are runnable without any further modification or processing. All the necessary library routines are loaded by the stubs.

HRPC-Courier's four integer subtypes (INTEGER, LONG INTEGER, CARDINAL, and LONG CARDINAL) and the ENUMERATION type all map into Franz Lisp's `fixnum`. Booleans, strings,

and the simplest form of arrays are supported directly. Sequences and records are both implemented as lists, with run-time type checking of their components. Choice types, which are roughly the same as variant records, have been implemented by a matcher to support the Lisp style. Lisp symbols are the choices, and a matcher is employed to match the pattern of symbol types in the s-expression against the possible patterns of the argument in the Courier declaration.

Data types do not map readily from Lisp to HRPC. Lisp is untyped in the sense that a symbol may evaluate to anything. However, the internal representation of data is typed so that data can be appropriately manipulated. But the HRPC run-time assumes that types are known and fixed. The language interconnection level is used to handle these difficulties: it performs explicit type checkings of the Lisp values before they are passed to the HRPC run-time.

For Lisp clients using Lisp servers, the existing HRPC-Courier types were found to be too limiting. Too many types in Lisp are not supported by HRPC-Courier. For interfaces demanding greater flexibility, a new type, **Lispval**, was added to the HRPC-Courier language. This type represents any type, such as symbol, fixnum, string, or list. **Lispval** obviously makes sense only for Lisp clients and servers. The additional flexibility is not a burden to the ordinary HRPC user because it is loaded only on demand.

The environment for evaluation of free variables is another consideration. Franz Lisp is dynamically scoped, so that free variables are evaluated in the environment of the caller. In HRPC, Lisp free variables used in the server are evaluated in the server environment. However, we do not attempt to provide a transparent implementation. That is, the user is aware that a remote call is being evaluated in a remote server.

Error handling in Lisp HRPC depends on whether the error is from the client or server side, and whether the control is in the C HRPC run-time part or Lisp layer. When an exception is detected from the HRPC run-time, it prints an error message to the user, then passes the error to the standard Lisp error handling mechanism, which can be redefined by the user. For exceptions from the Lisp layer at the server side, the server stub sets an error handler that, on an exception, makes up an error return value and passes it to the HRPC run-time.

4.2 Stub Level: Lisp HRPC

We use an interpreted stub scheme mainly to provide type checking for the Lisp-to-C interface during marshalling. The type information necessary to drive the stub interpreter and run-time type checking is extracted from the Courier description by the Lisp HRPC stub compiler, and put into the Lisp stub. The type information is also used for the interpreter to drive (un)marshalling operations for arguments and results. Therefore, for each procedure, the HRPC Lisp stub contains the "arg-pattern" and "result-pattern" that describe data types for arguments and results, instead of direct procedure calls to (un)marshalling routines. For instance, for an HRPC-Courier procedure

```
Sum10:  PROCEDURE [a:  ARRAY 10 OF INTEGER] RETURNS[r:  INTEGER] = 1;
```

the following Lisp stub code is generated.

```
(setq Sum10_arg_pattern '((Array 10 Integer)) )
(setq Sum10_return_pattern '((Integer)) )
(defun Sum10 (fBinding a)
```

```
(clientTalkLisp fBinding Sum10_detect_jmp
    Sum10_arg_pattern Sum10_return_pattern
    Sum10_progN Sum10_verN 1 (list a) ) )
```

The type descriptions for the argument (**Sum10_arg_pattern**) and result (**Sum10_result_pattern**) are lists of the type names and structure sizes. The **clientTalkLisp** is the stub interpreter function for the client. It takes the type descriptions, list of argument variable(s), and other procedure-specific information such as the program number of the interface, the version number, and the procedure id number. It returns the results of an RPC in a single list.

Lisp stubs can be generated automatically, using command line switches indicating client and server languages. To minimize maintenance, only a single stub generator for Lisp and C exists. We split the procedure code-generating module into a language-independent module and language-knowledgeable modules. Language knowledge is encapsulated in procedures that are assigned to variables in the stub generator when the stub is compiled.

The multi-language stub compiler does essentially the same job as the initial C stub compiler. For Lisp, however, codes for types are emitted only for call and return parameters of procedures. The patterns used to drive the interpreter are generated by traversing the symbol table tree when a **PROCEDURE** is recognized. Since Lisp is untyped, any **TYPE** declarations in the Courier description are useless, except to describe what a compile-time type-checked partner would be expecting at run-time.

4.3 Language Interconnection Level: Lisp HRPC

Because Franz Lisp and C can communicate in the same address space (although with difficulty) using the **cfasl** function, relatively little code to bridge between the languages is needed. Each call to the C run-time requires a small repackaging function. These take care of: repackaging C procedures into C functions, sometimes with attendant awkwardness; converting all arguments either to values or to addresses, since **cfasl** can accommodate either, but not a mixture; and calling C procedures that are themselves variable (a consequence of another limitation of **cfasl**).

The differences in parameter passing between Lisp, which passes by-value pointers to the function arguments, and HRPC, which passes procedure arguments by-value (as in C), are straightforward to handle. The **cfasl** function, however, does not handle the differences in the way arguments are returned. Lisp functions always return a single pointer to an s-expression, which may represent a simple or complex Lisp construct. HRPC procedures may return more than one values. To be consistent, we define all HRPC Lisp procedures to return a list. Lisp stubs use individual function calls for each element of the return list, packaging them into a list in the Lisp bridging code. This list is returned to the caller. This portion of the code is highly implementation-dependent, because of the rigid requirements for the layout of data in memory for Franz Lisp.

As mentioned before, due to the differences between the type systems of Lisp and HRPC-Courier, all data from Lisp layers to HRPC should be checked for possible illegal uses. The type checking is done inside the stub interpreter for the input data to the Lisp functions, using the type description given by the stub.

Handling exceptions from the HRPC run-time is another source of language interconnection problem. HRPC run-time uses **setjmp** and **longjmp** for error handling. In a C stub, there is a

single **setjmp** that catches all the **longjmps** for the stub. But a Lisp stub makes several transitions between Lisp and C, and therefore, a single **setjmp** would not catch all potential **longjmps**. Thus, each Lisp transition routine must contain its own **setjmp**, and every call to a transition routine must contain a test for successful outcome of the call. Should an error be detected, it is transferred to a surrounding **errset**.

5 Smalltalk HRPC

With its versatile graphic interface and programming environment, Smalltalk is ideal for applications that handle bitmap images and user interfaces. A version of Finger that returns a bitmap (digitized) image of the user's face is one example of a natural application to define in Smalltalk. Smalltalk HRPC provides Smalltalk programs with the means to access remote resources in the HCS world.

5.1 Application Level: Smalltalk HRPC

To a Smalltalk client, an instance of a remote service is viewed as an instance of the corresponding service class. The service class for a server interface, as a part of the Smalltalk-HRPC stub, defines a set of methods that are mapped to the procedures defined in the HRPC-Courier interface description.

The following example shows how the FingerService described in section 3.1 is imported and called in Smalltalk.

```
aFinger ← FingerService newAt: 'condor'.
fingerAll ← aFinger whoAll.
fingerInfo ← aFinger whoIs: 'John'.
```

FingerService is the service class for the FingerService. Its instance creation method newAt:*host* imports the FingerService server at *host*. It has two instance methods whoAll and whoIs:*name*, which correspond to **WhoAll()** and **WhoIs(name)** procedures of the interface. An imported server is represented by a FingerService object, and RPCs to the server are done by sending *messages* to that object, as is natural in Smalltalk.

As in the case of Lisp HRPC, there should be proper data type mapping (and checking). Smalltalk provides an extremely flexible type system in the sense that there is no notion of a system-defined data type. Once a user data type (actually a *class*) is added to an existing class hierarchy, it is treated in the same way as any other classes in the system are and it inherits all the properties of its parent class. Given this flexibility in Smalltalk's type system, we decided to define new Smalltalk classes for HRPC-Courier data types rather than to map HRPC-Courier data types to existing classes. In fact, most of these new classes are relatively simple specializations of existing classes with a few HRPC-related (un)marshalling operations.

Having new classes for the data types of HRPC-Courier does not necessarily mean that the Smalltalk HRPC user should learn new operations (methods) for those new classes.[4] Actually most classes for HRPC data types provide or inherit the same methods as their compatible ordinary Smalltalk classes. But the HRPC-related classes guard against their usage beyond the limit of

[4] But the Smalltalk HRPC user still must live within the intrinsic limit of HRPC-Courier types.

what the HRPC-Courier type system allows. Let's take an example of HRPC-Courier ARRAY, which consists of a fixed number of elements of the same base type. Allowing variable number of elements with different base types, the ordinary Smalltalk array is more general than an HRPC-Courier ARRAY. A Smalltalk version of ARRAY object can be used in the same way as an ordinary Smalltalk array is in instance creation, accessing array elements, etc. But the validity of its use is checked before it is passed to the HRPC run-time at the language interconnection level.

5.2 Stub Level: Smalltalk HRPC

An HRPC-Courier interface description is mapped into a set of Smalltalk class definitions using two kinds of "stub classes", *service class* and *data type class*. A service class represents a server interface. It has a class method for instance creation and instance methods for each procedure in the interface. In addition to those methods, it contains all the interface-specific information pertinent to the service, e.g., program number, procedure arguments and results, data type names, etc.

All the data types defined in an interface have corresponding data type classes in Smalltalk. Smalltalk HRPC provides classes for the elementary HRPC-Courier types (INTEGER, LONG INTEGER, etc.) and type constructors (RECORD, ARRAY, SEQUENCE, and CHOICE). We call them the *base type classes* because the stub classes for the user-defined constructed types are based on them. All other data type classes are *constructed type classes*. A constructed type in an HRPC-Courier interface is mapped with a constructed type class in Smalltalk, which is defined as a subclass of the base type class for the corresponding constructor. For example, the Smalltalk data type class for an HRPC-Courier type constructed with ARRAY is a subclass of the ARRAY class.

The role of data type classes is the same as that of traditional stub procedures, i.e., marshalling and unmarshalling data. (Un)marshalling of constructed class's objects in Smalltalk HRPC uses the inheritance mechanism of Smalltalk. A constructed type class specifies only the type information of its components; for example, in the case of an array, the number and the class name of the elements. Constructed type classes inherit (un)marshalling methods from their super (base type) classes, not having their own. When an (un)marshalling operation is invoked on a constructed type class object, it eventually invokes an (un)marshalling method of its superclass. The (un)marshalling method of the base type class for a type constructor obtains type information from the target subclass object. For instance, for an array of 10 integers (described in Courier):

```
sampleArray:  TYPE = ARRAY 10 OF INTEGER;
```

the following stub class (written in "pseudo Smalltalk") is made.

```
ARRAY subclass: #SampleArray

class methods
    typeSpec
        ↑#( 10, INTEGER) "array of 10 integers"
```

Note that the symbol INTEGER in the definition of SampleArray is the literal name for the INTEGER class representing Courier INTEGER type. For a SampleArray object, (un)marshalling is done by its superclass ARRAY. The (un)marshalling method of ARRAY obtains the type information

of the SampleArray object's elements through the typeSpec method. Then it serializes the array contents, 10 integers, by invoking the INTEGER (un)marshalling method for each of them.

Smalltalk stubs are simple and compact. In contrast to traditional stub procedures, Smalltalk stubs for RPC invocation do not actually contain code to (un)marshal individual data components. Instead, once (un)marshalling for an argument or result object is initiated inside the RPC support class, all remaining activities are driven using the type information of the involved target objects. The polymorphism of Smalltalk is another factor that makes the stubs (and other portion of RPC support classes) compact. It eliminates, for instance, duplicated codes of marshalling procedure calls for each data object of different data types.

We have not implemented a stub compiler for Smalltalk HRPC. But due to the structural simplicity of Smalltalk stubs and the modularity of the HRPC stub generator, the current stub compiler could easily be extended for Smalltalk.

5.3 Language Interconnection Level: Smalltalk HRPC

Unlike Franz Lisp with `cfasl`, our target system, the Smalltalk system on Tektronix 4405 workstations, does not allow procedure calls to an external code body, and it was out of the question to modify the internals of the Smalltalk system because of the philosophy of HCS. This led to a design where an agent process runs in a separate address space, handling underlying RPC operations through the HRPC run-time support. We call the agent process a Stub Server. The Stub Server relays all RPC operations between the Smalltalk environment and the HRPC world. Unix pipes are used as the communication channel between a Smalltalk process and its Stub Server.

Contrary to conventional compiled stubs, the Stub Server should be able to perform RPCs without a priori knowledge of the interface description. It requires an interpretive stub inside the Stub Server. Instead of compiled knowledge of a specific interface description, it receives required information (data type specification, procedure call id, etc., collectively called a *signature*) on the fly with an RPC request. Then the signature drives the Stub Server's interpretation loop.

The existence of Stub Servers is known only to the lowest level of the Smalltalk HRPC support. It would be fairly easy to adapt the current system to a Smalltalk system with different means to communicate with external code body. And such an adaptation would not affect the existing user level Smalltalk programs (applications and stubs).

As in the case of Lisp HRPC, there should be a proper guard against "too liberal" uses of Smalltalk HRPC-Courier type objects. A form of type checking is implemented inside the *base type classes*. For example, the implementation of ARRAY in Smalltalk checks whether an instance is created with the designated number of objects of the designated class, and since the contents of an ARRAY object can be changed at run time,[5] it checks again when the ARRAY object is marshalled.

6 Evaluation

Our approach to multi-language support for heterogeneous remote procedure call was driven by the design decisions necessary to accommodate existing systems without modification. We have

[5] It does not perform other run-time type checking until it is marshalled.

developed two language support subsystems providing natural programming interfaces to application language layers while re-using large parts of the original HRPC system. This results in an RPC facility that effectively deals with both system and language heterogeneity. In this section, we discuss some interesting aspects of our new language subsystems.

Interface Description Language and Data Types We have two sources of inter-language communication problems, mapping between the interface description language and the HRPC application language, and connecting the application language layer to the HRPC run-time support.

Mapping data representations into an interface description language forces us to limit the degree to which an application language is supported. The benefit of an interface description language is that it is restricted enough that all of it can be supported, so there is no clash with user expectation. This is valuable in cases where the data types almost map. For example, both Franz Lisp and Smalltalk have a predefined data type called "array" that is more powerful than that of C or Pascal. But the HRPC-Courier array type is more restricted, and this is the type used in the interface descriptions for multi-language calls.

In the case of inter-language calls, the more general definition could not be handled by a server whose implementation language lacks that concept. This, and the dynamic typing nature of the application languages, required proper run-time type checking for data objects exported from the Franz Lisp or Smalltalk environment.

Stubs It comes as no surprise that the content of stubs reflects the character of the language for which the stub is written. We have no type declarations for Lisp, an untyped language. We have class definitions for Smalltalk. And procedure definitions occur for both. Some type information is supplied for Lisp. However, it is not for use by the Lisp client. Rather, it is to enable the Lisp layer to do run-time type checking on the input. Type information in Smalltalk is intrinsic. This helps abstract a large portion of functionality of conventional stubs, resulting in compact Smalltalk stubs.

Language Interconnection Language interconnection between the language layer and the underlying RPC run-time support depends on the application language implementation and available communication mechanisms to different languages. As we have seen in the case of Lisp HRPC, accomplishing inter-language connection can be nontrivial even when inter-language procedure call is permitted. When inter-language procedure call is not available, as in the case of Smalltalk HRPC, the connection is made across address spaces to an agent process, the Stub Server. The use of the interpretation technique is a key element in implementing the Stub Server.

Layering For support of multiple languages, we provided a language-specific layer for each new language. This layer segregates the language issues from the issues of transport, control, naming and binding. It also allows us to extract the cost of the additional support from the run-time, so that only users of a language subsystem pay the costs entailed in inter-language communication. It also insulates the language subsystem from any changes in or enhancements to the run-time services.

7 Related Multi-Language Systems

The problem of heterogeneous multi-language RPC has been addressed by several other groups. We discuss several of those efforts below.

Mixed Language Programming The University of Arizona's Mixed Language Programming (MLP) [8] emphasizes two different problems: accommodating polymorphic interfaces and utilizing existing subroutines without creating explicit interface descriptions. MLP utilizes an interface description language called UTS (Universal Type System) that allows underspecification (or no specification) of the interface. To support this, both client language and client compilers were extended, so that unification of argument types could be achieved and necessary packaging and unpackaging code could be inserted. MLP supports C, Pascal, and Icon, and plans to extend to Franz Lisp and possibly Fortran.

Matchmaker: Carnegie-Mellon's Matchmaker [9, 10] provides language-level support for the Accent and Mach operating systems. Matchmaker provides the necessary services for transport, tagging messages to tell the recipient what source-machine data representation is used in the message body. A degree of homogeneity is imposed on the system by the dependence upon one operating system. This does simplify the inter-language interface because the operating system presents a consistent interface to all languages on all machines. The interface description language for Matchmaker is Pascal-like. Stubs are generated in the target language using a multi-targeted compiler. Stubs handle all of the marshalling, transport, and unmarshalling of data, and returning it in a form compatible with the application language's calling conventions. Different languages are connected using interprocess communication. Matchmaker supports Perq Pascal, C, Common Lisp and Ada.

Horus: Hewlett-Packard's Horus system [6] divides the problem more vertically. It is built around a core engine that generates all the marshalling, transport and control code in the target language. Horus only supports one RPC mechanism. Differences required to accommodate different languages and machine-specific data representations are embodied in specifications, that together with the interface description, are inputs to the stub generator. The interface description language is Pascal-like. Horus supports Pascal and C, with plans to extend support to Common Lisp.

References

[1] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Trans. on Software Engineering SE-13*,8, pp. 880-894 (Aug. 1987).

[2] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems 2*,1, pp. 39-59 (Feb. 1984).

[3] D. Notkin, A. Black, E.D. Lazowska, H.M. Levy, J. Sanislo, and J. Zahorjan. Interconnecting Heterogeneous Computer Systems. *Communications of the ACM* (February 1988).

[4] S.K. Chung, E.D. Lazowska, D. Notkin, and J. Zahorjan. Performance Implications of Design Alternatives for Remote Procedure Call Stubs. *Proc. IEEE 9th International Conference on Distributed Computer Systems*, pp. 36-41 (Jun. 1989).

[5] B. Einarsson and W.M. Gentleman. Mixed Language Programming. *Software - Practice and Experience 14*,4, pp. 383-395 (Apr. 1984).

[6] P.B. Gibbons. A Stub Generator for Multi-Language RPC in Heterogeneous Environments. *IEEE Trans. on Software Engineering SE-13*,1, pp. 77-87 (Jan. 1987).

[7] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983).

[8] R. Hayes and R.D. Schlicting. Facilitating Mixed Language Programming in Distributed Systems. *IEEE Trans. on Software Engineering SE-13*,12, pp. 1254-1264 (Dec. 1987).

[9] M.B. Jones and R.F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. *Proc. OOPSLA '86*, pp. 67-77 (Sep. 1986).

[10] M.B. Jones, R.F. Rashid, and M.R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 225-235 (Jan. 1985).

[11] M. Schwartz, J. Zahorjan, and D. Notkin. A Name Service for Evolving Heterogeneous Systems. *Proc. 11th ACM Symposium on Operating System Principles* (Nov 1987).

[12] R. Wilensky. *LISPcraft*, W.W. Norton & Co., New York (1984).

Performance Implications of Design Alternatives for Remote Procedure Call Stubs

Sung K. Chung, Edward D. Lazowska, David Notkin, and John Zahorjan

Department of Computer Science
University of Washington
Seattle, WA 98195

October 1988

Abstract

Remote procedure call (RPC) has become widely adopted as a distributed programming paradigm. An RPC facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports communication among application programs while relieving programmers from concern with the mechanisms used, such as data encoding and transport protocol.

Performance is critical in an RPC facility. To date, essentially all examinations of RPC performance have concentrated on kernel-level issues. In the current paper, we take efficient kernel-level support as a "given", and study the performance implications of design alternatives one level up – in the stubs, which insulate the client and server from details about network communication. The design alternatives that we consider were motivated by our three-year experience in the University of Washington's Heterogeneous Computer Systems (HCS) Project. These alternatives represent a collection of well-motivated approaches to achieving "standard" RPC semantics. In particular, we consider the performance implications of compiled vs. interpreted stubs, procedural vs. inline code for moving data to/from packet buffers, block copy vs. individual data item copy for moving data to/from packet buffers, and the presence or absence of byte swapping.

Index Terms – Remote procedure call, distributed system, performance.

This material is based upon work supported by the National Science Foundation (Grants No. DCR-8352098, DCR-8420945, CCR-8611390, and CCR-8858804), U S WEST Advanced Technologies, Digital Equipment Corporation (the External Research Program), Bell Communications Research, Boeing Computer Services, Tektronix, the Xerox Corporation, and the Weyerhauser Company. This work was done while Zahorjan was on sabbatical leave at Laboratoire MASI, University Paris 6.

Authors' addresses: Department of Computer Science FR-35, University of Washington, Seattle WA 98195.

1. Introduction

Remote procedure call (RPC) has become widely adopted as a distributed programming paradigm since its introduction by Birrell and Nelson in the early 1980s [Birrell & Nelson 1984]. An RPC facility provides a user-level mechanism across a communication network that, as much as possible, has the same syntax and semantics as local procedure calls within a high-level language. Hence, RPC supports communication among application programs while relieving programmers from concern with data encoding, transport protocol details, etc.

To a first approximation, an RPC facility works as follows: The client (caller) and server (callee) modules are programmed as if they were intended to be linked together. A description of the server interface – that is, the names of the procedures that the server implements and the types of their arguments – is processed, yielding two *stubs*. The client stub is linked with the client; to the client this stub is indistinguishable from the server. The server stub is linked with the server; to the server this stub is indistinguishable from the client. A subsequent client call to the server is fielded by the client stub, which transmits the arguments of the call to the server stub. The server stub then actually invokes the server, receives any results, and passes them back to the client via the client stub. In this way, the two stubs shield the client and server from the details of network communication.

Performance is critical in an RPC facility. To date, essentially all examinations of RPC performance have concentrated on kernel-level issues. Work in this vein includes the design of transport protocols that are particularly suited to supporting RPC semantics on a reliable local network [Birrell & Nelson 1984], the extension of the transport protocol and of RPC semantics to support the efficient transfer of bulk data [Gifford & Glasser 1988; Liskov et al. 1987], and the detailed optimization of aspects such as thread context switches [Schroeder & Burrows 1988].

In the current paper, we take efficient kernel-level support as a "given", and study the performance implications of design alternatives one level up – in the stubs. This investigation has several key motivations. First, several of the design alternatives make it significantly easier to construct certain kinds of distributed services. Second, improvements in kernel-level performance have made previously insignificant stub costs of greater concern; as very high speed networks become common, the ratio of transport times to stub costs will decrease even further. Third, even for current RPC systems, performance for calls with a large volume of arguments can be bounded by the marshalling, not the transport, costs.

The specific design alternatives that we consider were motivated by our three-year experience in the University of Washington's Heterogeneous Computer Systems (HCS) Project [Notkin et al. 1988]. The HCS Project designed and prototyped techniques for reducing the cost of interconnecting heterogeneous computer systems. The specific areas that we attacked were communication, naming, filing, remote computation, and electronic mail. The demands of the latter four services in a heterogeneous environment highlighted a number of tradeoffs between flexibility and efficiency in the first service – our HCS Remote Procedure Call (HRPC) facility [Bershad et al. 1987]. It is these tradeoffs that we will be examining here.

This paper is *not* of the "Stone Soup" variety frequently encountered these days in the RPC domain: "RPC semantics are nice, but they'd be even nicer if they included { asynchronous communication, multicast, ... }." The design alternatives that we consider represent a collection of well-motivated approaches to achieving "standard" RPC semantics. To be specific, we consider the performance implications of the following stub design alternatives:

compiled vs. interpreted stubs

In a compilation-based approach, each interface has "customized" client and server stubs that are based on the server's interface description; generally, these stubs are produced by a stub generator. In an interpretation-based approach, the system provides a single generic client and a single generic server stub, each of which is parameterized by the server's interface description; individual remote invocations pass actual parameters to these stubs that represent the remote procedure's argument types.

Compiled stubs have the advantage of run-time efficiency. Interpreted stubs have the advantage of flexibility. This flexibility arises in two ways. First consider what happens when a service description is changed. With compiled stubs, the stubs must be regenerated and then the client and server must be recompiled and relinked. With interpreted stubs, this added development cost is eliminated or reduced, depending on the actual change to the service description. Second, and perhaps even more important, is that interpretation makes it easier to construct generic clients and servers that can select the appropriate interface at run time. One example of this situation is the HCS Name Service [Schwartz, Zahorjan & Notkin 1987], in which a client using the name service must first make a call to a central server and then, based on the response of the first call, to an application-specific server. If interpretation had been available, the central server could have made the second call on behalf of the client. Another example of the benefit of interpretation is an HCS "bridge server," which acts as a translator between two incompatible RPC systems. Using compiled stubs requires that a separate bridge be generated for each service interface, while interpretation allows a single system-wide bridge server to be written.

procedural vs. inline code for moving data to/from packet buffers

In the procedural approach, a stub transfers each individual data item to or from a packet using a separate procedure call. For instance, a remote procedure with two integers as input arguments and a third as an output argument would use three procedure calls for data movement. In the inline approach, the code to perform such transfers are in the main body of the stub, eliminating the overhead of a procedure call for each data item.

The inline approach improves performance, especially given the cost of procedure calls. However, the procedural approach has two potential benefits. One benefit is that generating inline code can be difficult, especially in the absence of a compiler with the appropriate support. The other benefit is that calling the procedure indirectly allows great flexibility, especially with respect to handling heterogeneity. For instance, a stub can invoke a procedure to marshall an integer without concern for whether the integer is copied directly or has its bytes swapped. Hence, a single stub can be generated and used for RPC facilities that have significantly different data representations by linking the stub with appropriate marshalling routines.

individual data item copy vs. block copy for moving data to/from packet buffers

Elements in RPC arguments, such as fields or array elements, can be treated as individual items. Alternatively, when the items are to be copied to/from contiguous locations, a smaller number of block copies can be used to copy all the needed elements.

The cost of executing a stub can be reduced significantly using block copy, since it is generally cheaper to move bytes using fewer copy instructions. However, treating each element as an individual item makes it easier to write both a stub generator and a stub interpreter. Additionally, it may be difficult to identify the situations in which block copy can be used, since the layout of the data is compiler-dependent.

presence or absence of byte swapping

When two heterogeneous machines are communicating using RPC, data translation is often necessary. The most common form of translation is byte-swapping, which is needed when transferring integers between VAXes and SUNs, for instance. Most RPC systems define a standard "on-the-wire" representation for integers, letting each stub translate its data as needed. In such systems, if both the client and the server use the same representation, but they differ from the standard, then two unnecessary swaps are done.

In each of these dimensions, we are interested in determining rules of thumb that give some feel for the comparative costs. For instance, does the added flexibility of interpretation cost us a factor of two or a factor of ten relative to compilation? Answers to these questions should give guidance to the designers of future RPC facilities.

The remainder of the paper is organized as follows. Section 2 describes our experimental methodology. Section 3 presents the measurement results and our analysis. Section 4 provides a brief conclusion.

2. The Experiment

The relative performance of the alternatives depends on the characteristics of the particular interface under consideration; for example, the advantage of inline data movement over procedural data movement will increase with the number of arguments in the interface. We were thus confronted with a "workload characterization" problem. Our approach is to consider three different procedures that represent plausible extremes, and thus will indicate how well each stub design alternative does in the face of particular styles of data composition. The first procedure has a single argument: a single (long) integer. The second procedure is passed an array of 256 integers. The third procedure is passed six arguments: the first three are integers, the fourth is an array of 256 integers, the fifth is an array of 86 records each of which contains three integers, and the sixth is a record containing an integer and a string of 1024 characters. In the notation of our RPC interface description language, we denote these three interfaces as:

1. (I)
2. (A[256] I)
3. ((I) (I) (I) (A[256] I) (A[86] (I I I)) (I S<1024>))

Having settled on this workload characterization, we then implemented a collection of stubs that differed from one another only in the alternatives outlined at the end of the previous section. Some combinations of alternatives were omitted because they don't make sense; block copy in concert with byte-swapping is an example. Table 2 in the next section shows the specific combinations of alternatives that we explored.

The primary subdivision in our experiments is that of compiled vs. interpreted stubs. The compiled stubs are all based on the stubs we use in HRPC (which are produced by a stub generator created by modifying Cornell's Courier stub compiler [Johnson 1985]). The HRPC stubs use procedure calls for each data item, but are not bound with respect to byte-order. These stubs were manually modified to cover the sub-cases we wished to explore.

The interpreted stubs were based on a straightforward interpreter adapted for these experiments from one developed by Kimiko Gosney for a Franz Lisp version of HRPC [Gosney 1987]. In addition to the arguments, the interpreter accepts a short-form description of the types of the arguments, the form of which is similar to the interface description given above.

For all our measurements, we eliminated calls made to the transport layer, since the costs of transport do not change in the face of the alternatives. Although the transport costs cannot be ignored when measuring the full cost of RPC facilities, eliminating the transport calls represents the bounding condition of increasing network speeds increase and the associated decreasing transport costs.

Careful repeated measurements were made using the microsecond clock on a VAX/Ultrix system.

3. Measurement Results and Analysis

3.1. A Simple Example

We begin this section with a simple example to illustrate our approach. The example, which appears in Table 1, is a small excerpt of our full measurement study, which appears in Table 2. In particular, the example considers four different stub designs (versus ten in the full study) and one procedure (versus three in the full study). Specifically, Table 1 shows measured marshalling costs for compiled stubs with either single-item copy or byte copy of arguments into packets, and with either inline or procedural code for this data movement. The procedure has a single integer argument: (I) in our notation. Thus, the stub must move four bytes of data and check and update buffer pointers.

	Inline	Procedural	Procedure Call Overhead
Byte Copy	15μs	40μs	25μs
Item Copy	10	35	25
Byte Copy Overhead	5	5	

Table 1: Example Measurement and Analysis Excerpted from Table 2

Table 1 shows that for the system under consideration (a MicroVAX-II running Ultrix 2.0), using compiled stubs for a procedure with a single integer argument:

- Marshalling costs vary by a factor of four, from 10μs to 40μs, depending upon the stub design chosen.
- The cost of procedural stubs over inline stubs is significant (25μs, roughly tripling the inline marshalling cost) and is independent of other choices made.
- The cost of byte copy stubs (used if byte swapping is necessary) over item copy stubs is not nearly so significant (5μs); it too is independent of other choices made.

This simple example was presented mainly to clarify our approach, in particular our focus on the *relative* performance of the design alternatives rather than on the *absolute* performance of our experimental software. It's also important to note that although our measurements are of course for a particular computer system, we will present parameterized equations at the end of this section that allow our results to be extended to other systems. Finally, it is essential to take the measurements for marshalling in the context of the cost of a round-trip RPC call, including transport. Original RPC facilities made invocations in roughly 50 milliseconds; our basic HRPC invocation with no arguments costs roughly 17 milliseconds; the best reported times for such basic calls in any RPC system are in the two millisecond range.

3.2. Principal Performance Comparison

Table 2 contains our principal measurement results. It shows the marshalling costs for each of the three procedures in our "workload", for each of ten different combinations of stub design alternatives. In addition to the measurement results for each of these 30 cases, Table 2 contains (in parentheses) an analytic estimate of the results, calculated from a simple model of RPC stub performance that we have constructed and that we will discuss later. Note that the results in Table 1 were excerpted from rows 6-9 in column 1 of Table 2's results.

We begin by exploring the performance difference between interpreted and compiled stubs. This difference reflects the extra overhead of interpretation. One obvious observation is that the amount of interpreter overhead is highly dependent on the data structure of the argument being marshalled. For a single integer (data type (I)), for instance, the interpreter overhead involves some initial setup and the time to decode a single type specification code, a total of roughly 70μs. This is not a large absolute value (especially in comparison to full round-trip RPC costs, which are measured in milliseconds, not microseconds) but turns out to represent a substantial proportion of the total marshalling time. For an array of 256 integers (data type (A[265] I)), additional time is required for the initial setup, but this time is amortized over all of the array elements, so the total interpreter overhead becomes insignificant. In fact, Table 2 shows that the compiled stubs were actually slightly slower than the interpreted stubs for this case (when procedural rather than inline data movement was employed) – a definite anomaly caused by the difference in the for-loops of the compiled stub and the interpreter, among other minor implementation details and measurement noises.

We can see a significant difference between compiled and interpreted stubs in marshalling the (Structure). But most of the difference is caused by a single portion of the (Structure): more than 90% of the extra interpreter overhead is attributable to the array of records, (A[86] (I I I)). Since the interpreter

Stub Type			Data Type (I)	Data Type (A[256] I)	Data Type (Structure)*
Interpreted	Inline	Byte Copy	0.085 *ms* (0.083)†	4.75 *ms* (4.82)	23.3 *ms* (23.9)
		Item Copy	0.079 (0.078)	3.40 (3.53)	19.0 (21.3)
	Procedural	Byte Copy	0.108 (0.108)	11.2 (11.2)	35.7 (36.9)
		Item Copy	0.101 (0.103)	9.90 (9.96)	31.0 (34.3)
	Block Copy		0.079 (0.079)	0.585 (0.628)	15.7 (16.0)
Compiled	Inline	Byte Copy	0.015 (0.015)	4.60 (4.62)	12.1 (11.9)
		Item Copy	0.010 (0.010)	3.32 (3.34)	9.65 (9.32)
	Procedural	Byte Copy	0.040 (0.040)	11.3 (11.0)	28.9 (27.1)
		Item Copy	0.035 (0.035)	10.0 (9.73)	25.8 (24.5)
	Block Copy		0.035 (0.035)	0.457 (0.432)	4.08 (3.87)

* (Structure) ≡ ((I) (I) (I) (A[256] I) (A[86](I I I)) (I S<1024>))
† () denotes analytic estimate of marshalling time

Table 2: Principal Comparison of Marshalling Costs (MicroVAX-II, Ultrix 2.0)

does not have *a priori* information about nested structures, such structured arguments are handled by recursively calling the interpreter. And each field of a record is individually processed for each recursion. Such a combination of inefficient aspects of the interpreter, which is inherent to all interpreters to some extent, explains the large interpretation overhead shown in the measurement data.

Summarizing these observations concerning interpreted versus compiled stubs, Table 3, which is based on the data in Table 2, shows the percentage of total marshalling time due to interpretation for various interfaces and stub types. The key conclusion is that the relative cost of interpretation is significant for large, irregularly-structured arguments, but insignificant otherwise. (For small arguments, the absolute costs might be high, but these will still be small relative to the overall cost of an RPC invocation.) Interpreter overhead is determined by the target data structure and is almost independent of other design choices.

Stub Type			Data Type (I)	Data Type (A[256] I)	Data Type (Structure)
Interpreted	Inline	Byte Copy	80%	4%	51%
		Item Copy	86	6	63
	Procedural	Byte Copy	63	2	33
		Item Copy	67	2	39
	Block Copy		86	4	76

† $\frac{\text{interpreter overhead}}{\text{marshalling time}}$

Table 3: Percent of Marshalling Time Due to Interpretation

We next consider the performance difference between procedural and inline stubs. Since each elementary data item causes a procedure call in a procedural stub, the degradation due to a procedural stub structure is easily calculated as *number of elementary data items* × *procedure call time*. There are also "combining costs" to handle structured data; for instance, to marshall an array of N records, if a record is handled by a "factored" marshalling procedure, it takes N procedure calls in addition to those required for the marshalling of the record fields.

Table 4 is analogous to Table 3, except that it shows the percentage of total marshalling time due to procedure calls. Our general conclusion is that a procedural organization can be expected to degrade performance by a factor of about two, more-or-less independently of other design choices.

Stub Type			Data Type		
			(I)	(A[256] I)	(Structure)
Interpreted	Procedural	Byte Copy	23%	57%	36%
		Item Copy	25	65	42
Compiled	Procedural	Byte Copy	63	57	52
		Item Copy	83	64	59

† $\frac{\text{sum of procedure call times}}{\text{marshalling time}}$

Table 4: Percent of Marshalling Time Due to Procedure Calls

The performance difference between the byte copy method and the item copy method is not so great as for the other pairs of design alternatives. Of course, data copying time is highly dependent on implementation details and on the data copy instructions of the target machine; also, as was the case for procedure call overhead, data copying overhead is proportional to the data size. Nonetheless, our conclusion is that there is a negligible performance implication to this choice.

The conclusion for block copying of data (that is, the movement of large numbers of data items using a single instruction) is considerably different. Block copy, where feasible, can improve marshalling performance dramatically. There are two main reasons for this. First, with block copy, data copying itself is faster. Secondly, for block copying it is assumed that there is no need for byte swapping or concern about the internal data structure; thus there is no "combining cost" (assuming the size of a data structure is known).*

Block copy causes an interesting interaction between interpreted and compiled stubs. For compiled stubs all the data sizes for non-variable size structures are known at compile time. But for the interpreted case, as we have discussed earlier, such information is generally unavailable for user defined data structures. This lack of information of data sizes makes the interpreter inefficient for marshalling structured data, as shown in Table 2 for the two block copy cases for (Structure).

3.3. An Analytic Model

As noted earlier, we constructed a simple analytic model of stub costs – a linear equation involving several parameters.

The construction of such a model has two objectives. First, if the results of the model match the measurement data, then we have confidence that our abstractions really do capture the essence of the phenomena we are trying to understand. Comparison of the parenthesized and non-parenthesized results in Table 2 shows that the match between measured and predicted results is excellent. Second, the model allows the measured results to be extended to other computer systems, removing the system dependence inherent in any measurement study.

* But sometimes the cost of knowing the size of a variable size data structure is unexpectedly high. For example, in the C language, to obtain the size of a 1024 byte string requires 2.7ms using strlen(), a standard string length function. String representations in other languages store the length explicitly, reducing this cost to a small constant.

The cost model for the data type (A[256] I) (an array of 256 integers) is shown in Table 5. Similar cost models can be generated (automatically, based on the interface description) for other data types. The definitions of the variables in the cost equations, along with the values of these variables for a MicroVAX-II running Ultrix 2.0, appear in Table 6. (The initial term in each expression in Table 5 represents the cost for sending the size of the array.)

Stub Type			Cost Equation
Compiled	Inline	Byte Copy	$L_b+256\times(L_b+f)$
		Item Copy	$L_i+256\times(L_i+f)$
	Procedural	Byte Copy	$L_b+p+256\times(L_b+p+f)$
		Item Copy	$L_i+p+256\times(L_i+p+f)$
	Block Copy		$L_i+b(256\times4)+\alpha$
Interpreted			Add $(B+A+e)$ to each entry above

Table 5: Cost Model for (A[256] I)

Symbol	Meaning	Value
L_b	move a long integer by byte-by-byte copy	15μs
L_i	move a long integer by individual data item copy	10
f	for-loop bookkeeping	3
p	call for a marshalling procedure	25
$b(n)$	move n bytes by block copy(bcopy), b(1024)	382
α	block copy stub initial overhead (per a packet size)	40
B	initial setup for the interpreter	45
A	initial setup for an array in the interpreter	128
e	processing an elementary type specification code in the interpreter	23

Table 6: Parameter Values for a MicroVAX-II Running Ultrix 2.0

We close with two caveats. First, as we have seen, procedure call cost and data copy cost are the major contributors to marshalling time. These costs are machine dependent. For example, a machine may have highly optimized procedure call that is particularly fast relative to data movement. In such a case, the benefits of inline stubs would not be as great as indicated in Table 1.

Second, we note that the generality of our measurements is of course limited. Nonetheless, we feel that we have demonstrated the relative performance of various approaches to stub design, and that the incremental performance differences can be parameterized in terms of a few key characteristics.

4. Conclusions

Our objective in this paper has been to assess the performance implications of certain design alternatives for remote procedure call stubs. The design alternatives that we considered provide "standard" RPC semantics with varying degrees of flexibility. This flexibility, while particularly valuable in a heterogeneous environment, can be important elsewhere as well. We have provided measurements and analysis that we feel will be useful to designers of RPC systems who must consider whether a particular "flexibility enhancement" is worth the performance compromise. Our results can be summarized as follows:

- Compiled stubs have a significant performance advantage over interpreted stubs only in the case of large, irregular parameters. However, the flexibility of interpretation may often lead to other benefits of system structure.
- Inline data movement offers, for the machine we measured, a factor of two improvement over procedural data movement, almost independently of other design decisions. On a machine where

procedure invocation is less expensive relative to basic copy operations, the benefit will still be significant, although perhaps less than a factor of two.

- Byte swapping is not particularly costly relative to single item copying of data to packets.
- Block transfer of data to packets, where possible, offers a very significant performance advantage over single item copying.

Acknowledgments

Henry Levy and Brian Pinkerton assisted with the measurements.

References

[Bershad et al. 1987]
Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering SE-13*,8 (August 1987), pp. 880-894.

[Birrell & Nelson 1984]
Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2,1 (February 1984), pp. 39-59.

[Burrows & Schroeder 1987]
Mike Burrows and Mike Schroeder. Performance of Firefly RPC. DEC Systems Research Center, November 1987.

[Gifford & Glasser 1988]
David K. Gifford and Nathan Glasser. Remote Pipes and Procedures for Efficient Distributed Communication. *ACM Transactions on Computer Systems* 6,3 (August 1988), pp. 258-283.

[Gosney 1987]
Kimiko Gosney. Heterogeneous Remote Procedure Call for Franz Lisp. (M.S. thesis.) Technical Report 87-07-03, Department of Computer Science, University of Washington, July 1987.

[Johnson 1985]
J.Q. Johnson. XNS Courier under UNIX. Cornell University, March 1985.

[Liskov et al. 1987]
Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler, and William Weihl. Communication in the Mercury System. Programming Methodology Group Memo 59, Laboratory for Computer Science, Massachusetts Institute of Technology, October 1987.

[Notkin et al. 1988]
David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting Heterogeneous Computer Systems. *CACM 31*,3 (March 1988), pp. 258-273.

[Schwartz, Zahorjan & Notkin 1987]
Michael Schwartz, John Zahorjan, and David Notkin. A Name Service for Evolving Heterogeneous Systems. *Proc. Eleventh ACM Symposium on Operating Systems Principles* (November 1987), pp. 52-62.

The Virtual System Model for Large Distributed Operating Systems

B. Clifford Neuman

Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

bcn@cs.washington.edu

Technical Report 89-01-07
April, 1989

Abstract

Scale is a critical element in distributed systems. The effect of scale on the user has received little attention. As the number of distinct computers that compose a distributed system grows, it becomes increasingly difficult for users to organize and find objects and services. Today's users are able to cope because they tend to use only a small number of computers, and because only a tiny portion of the objects and services in the world are easily accessible. This observation suggests one way of organizing large systems: let each user see a smaller system containing only those parts that are of interest. The virtual system model provides a framework within which users can build a view of a system in which the parts of interest are logically nearby. This report discusses the virtual system model and describes a global file system being built using the model.

1 Introduction

As the the size of a distributed system grows it becomes increasingly difficult for users to organize and find objects and services. Today's users are able to cope because only a tiny portion of the objects and services in the world are available to them. Users needn't be concerned with objects in which they have little interest. One way to help users cope with very large systems is to let each user see a smaller system containing only those parts that are of interest.

This report describes the *virtual system model*, a new model for organizing large distributed systems. The virtual system model supports the maintenance of customized views of the system. Each view is called a *virtual system*. The virtual system model has several advantages over existing organizations because: (1) as systems become larger and larger, a single view containing all objects in the system becomes difficult to work with and multiple views are needed; (2) the maintenance of multiple views and the ability to include parts from others allows objects to be organized in a manner that

This research was supported in part by the National Science Foundation under Grants No. DCR-8420945 and CCR-8611390.

allows them to be more easily found; (3) the virtual system model allows users to easily control the selection of servers when a particular service is desired; (4) by associating virtual systems with objects to be protected, the mechanisms by which protection is provided can be better tailored to the owner's security requirements; (5) by associating virtual systems with processes and applications to be run, the environment in which they are run can be controlled and access to objects outside that environment can be restricted (if desired); and (6) since virtual systems include the list of the processors which may be used, they provide a mechanism to bind programs to particular processors.

In looking at very large systems, three aspects must be considered. The first is the access mechanism: how an object is accessed once it has been found. The second aspect is object location: given an object's name, how the object is found. The third aspect is organization: how users see the object hierarchy.

In addressing these issues, it was decided that the following characteristics should be supported: (1) users should gain access to local and remote objects in the same way, though the underlying access mechanism might be different; (2) the name of an object should not depend on the location from which is it accessed, and an object's name should not constrain its location; (3) each user should be presented with a view of the system in which those parts that are not of interest are hidden; at the same time, (4) users must be able to access objects outside of their view once learned about through other means; (5) users should be able to customize their views of the system, not just by naming hierarchies that they import into their namespaces, but by customizing the imported hierarchies as well; and finally, (6) any solution must extend beyond the file system to allow the tailoring of users' views of other system components as well.

The virtual system model provides support for the construction of customized views by users and groups of users. The naming and access mechanisms that support these views satisfy the characteristics just mentioned. The virtual system is the collection of system components included in a customized view. Each user might have his or her own virtual system, though it might start out as a copy of some prototype. Individuals with multiple roles can construct overlapping virtual systems, each corresponding to a different role. Projects or other entities that might benefit from a separate view of the world can have associated virtual systems. Thus, a project might have a virtual system to which users interested in the project might connect. The programmers on that project, however, will probably each have their own virtual system, each a superset of that for the project.

The virtual system model applies to many services. Among them are the file system, naming, authentication, authorization, and the use of processors, services and applications. Because of the large number of files on even small collections of systems, the file system provides a natural example and a focus for a prototype implementation to test the ideas of the model. It is described in Section 3. Sections 4 through 7 discuss the other services affected by the model. In Sections 8 and 9 other aspects of the model are discussed and conclusions are drawn.

2 Shortcomings of existing approaches

Many existing systems provide access to files scattered across large collections of computers. These system will be described in the context of the three issues discussed earlier: the access mechanism, object location, and organization.

Early systems, such as the ARPAnet, addressed few of these issues. Access to local objects was simple, but access to remote objects required one to first locate the object, and then to access it using a mechanism different from that used in the local case.

Systems such as NFS[13] address the access mechanism, but little else. In NFS, remote file systems can be mounted locally and the same mechanism can be used for both local and remote file access. It is still necessary, however, to locate the desired files from the universe of accessible files, and to mount the appropriate file systems. This two-step process makes it cumbersome to maintain links between individual files on different file systems, and imposes constraints on the placement of files and the organization of the file system.

Systems such as Andrew[4, 6], Locus[12, 16] and Sprite[11, 17] address the organization problem by attempting to present a single view of a system that extends across the users and sites that compose it. Such an approach follows from the goal of name transparency and source-location-independence, i.e., the name of an object should be independent of the location from which the reference is made. The object location mechanism in these three systems is based on looking in a table for a prefix of the pathname. Although the object location is not specified by the pathname, objects with pathnames sharing a common prefix (as found in the prefix table) must reside on the same file system. This makes it difficult to truly distribute similar information over a number of hosts, especially when a particular file should logically appear at multiple points in the file hierarchy.

The organizational mechanism becomes more important as the number of objects that compose the system grows. Users have a harder time trying to find things. If a system included every processor, file, service and user in the world, users of that system would be overwhelmed by the information that was available. Finding useful information could become impossible. One way to address this problem is to let each user see a smaller system containing only those parts that are of interest.

QuickSilver[3] applies this approach to the file system by supporting user-centered namespaces. For every user, a nameserver stores a list of directory prefixes along with the information needed to locate a user's files. Users can only access files that have been included in their namespaces. Several types of links allow files and directories from one user's namespace to be included in another's. By updating their prefix table and links, users are able to tailor their namespace, making it easier to find and name the objects they frequently use.

QuickSilver does not completely solve the location and organizational problems of large systems. A file's location is still not completely independent of its primary pathname. In fact, the existence of a primary pathname leads to a shortcoming with links: the source and the target of the link are not equivalent [1]; if the target is later deleted, the file is gone. Another shortcoming is that none of the

[1] Except for hard links, the effect of which (in QuickSilver) is more like making a copy of a file.

systems described so far allow the user to tailor the namespace below the level at which a directory has been included by a link or prefix table entry. Finally, QuickSilver provides little support for finding and organizing files that might be of interest, but which have not yet been included in the user's namespace.

Directory mechanisms in capability-based distributed systems such as Eden[1] and Amoeba[7] support the naming of all objects, not just files. Directories are objects that map from names of objects to the capabilities for the named objects. Capabilities correspond to links in other directory systems. Capability-based systems address many of the problems that arise in the other systems discussed in this section. They support user-centered naming in the sense that each user can start with a different set of capabilities, and can organize them as desired. Ignoring access rights, all capabilities for an object are equivalent. Removing a capability from one directory does not leave the others dangling. Finally, a capability-based system does not place constraints on an object's location. Objects whose capabilities appear in a single directory can be scattered across many sites.

Like the other systems described in this section, neither Eden nor Amoeba extend the user's ability to tailor the namespace to lower levels of the hierarchy. Capability-based system also have problems of their own. Capabilities have two purposes: they allow one to find the object, and they provide access rights. While this is not a problem in itself, it does mean that giving away access to a directory also grants access to the objects in that directory. This can discourage users from making their directories available to others.[2] What is needed is a mechanism that allows one to share one's organization of a collection of objects without also sharing one's access rights.

Before proceeding to a more detailed discussion of a file system based on *the virtual system model*, the relative merits of a customized view must be discussed. Does it make life easier or harder for the user? An objection to systems like QuickSilver is that names are not unique. The same name might refer to different objects when specified by different users. Relative names might be easier to remember, but they are harder to share. The virtual system model avoids this problem by allowing one to specify names of the form virtual_system:filename. Such names refer to the same object from all virtual systems and make sharing easier. The virtual system model makes life easier for the user by reducing the amount of clutter that must be searched when looking for objects of interest. The ability for an object to appear at multiple points in multiple virtual systems also makes it less likely that an object of interest will be missed. The next section describes a file system based on the virtual system model and discusses ways that it can be used to organize information.

[2]One way around this problem requires maintaining two separate hierarchies. One contains one's own rights, and the other, the rights to be granted to others. Unfortunately, this approach requires additional maintenance. New objects must be added in both places. A second approach involves giving out a restricted directory capability that causes the directory service to restrict the rights before returning capabilities for additional objects. Since rights are type specific, this doesn't work when a directory contains objects of different types. It also restricts one's ability to grant greater rights for some objects than for others.

3 The Virtual File System

A file system is currently under construction to test the basic ideas of the virtual system model. A prototype is presently running at the University of Washington and on several sites in another part of the Internet. The file system has two parts: a directory mechanism and an access mechanism. The virtual system model primarily affects the directory mechanism. Multiple access mechanisms will ultimately be supported because differing parameters such as latency, bandwidth, and access patterns will place different constraints on the access mechanism at different times. Once a file has been found, an appropriate access mechanism will be chosen.

The *global file system* is a collection of *virtual file systems*, each associated with one or more virtual systems. Each virtual file system has a root and appears hierarchical, but it does not fully satisfy the constraints for a hierarchy (loops are allowed). Items can appear multiple times within a file system, as well as in multiple file systems. Objects will be found, and virtual file systems built up by users as they learn of objects outside their current environment.

The remainder of this section briefly describes the global file system that is being built and discusses ways to organize objects within it. Additional information about this file system can be found in [10].

3.1 Directory Service

A significant component of the global file system is the directory service. Directories map from names to object pointers. In the file system the objects are files and directories. Because the directory service can be used to name objects other than files and directories, and because the objects included in a directory do not need to be located on the same system as the directory, the directory service is actually much closer to directory services in object-based systems such as Amoeba[7] than to directories in traditional file systems.

Figure 1 shows parts of two virtual systems. Directories 1 and 2 are the roots for bcn's and lazowska's systems respectively. Only those files and directories that are useful in the discussion that follows are included in the diagram. The actual virtual systems are really much larger.

Each virtual file system has a root directory, and the files and directories that are descendants of the root directory appear to form a hierarchy. In the global file system, however, files are not organized completely hierarchically. Files may exist multiple times in the hierarchy[3] , and it is quite acceptable to have directory links that form loops. It is expected that different virtual file systems will overlap and that files and directories will appear in multiple virtual file systems. It is likely that some files will appear in almost all systems, though they may be very deep in the hierarchy, if they are not of particular interest to the user. Those files in which the user is interested

[3] In Figure 1 "this paper" is found in bcn's virtual system with the names /bcn/papers/virt-system-model, /bcn/vsm/paper, and in /authors/Neuman/virt-system-model. In lazowska's virtual system, it can be found with the names /authors/Neuman/virt-system-model, /users/bcn/papers/virtual-system-model, /users/bcn/vsm/paper, and /subjects/dist-systems/virtual-system.

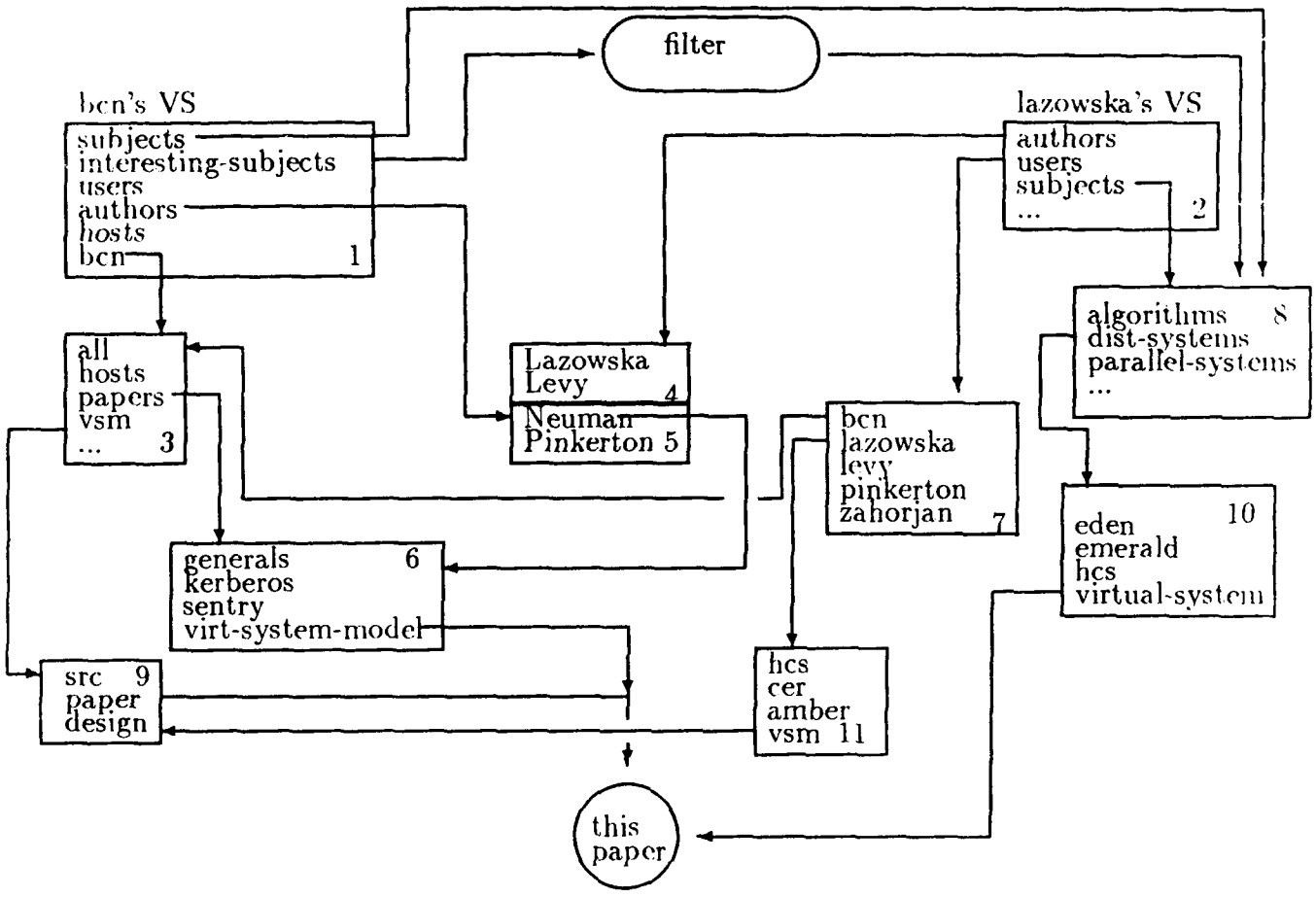


Figure 1: Parts of Two Virtual Systems

will be much closer to the root. It is in this way that the user's virtual file system appears to be customized.

Information about individual files will be stored along with the files. Among this information is a (possibly incomplete) list of back links. These links are pointers to the directories which contain links to the file. This information can be used to find other information which is similar or related to that in a particular file.

Other information that helps users find what they are interested in may be stored along with the directory link. One such piece of information is a filter. A filter is a program and associated data that can be used to decide whether a file should be included when a directory is listed. If a link to a directory has an associated filter, then only those files accepted by the filter will be displayed. A filter can do more than just approve or disapprove directory entries. It can transform names, or perhaps even insert additional entries obtained through some other mechanisms besides simply

reading the linked directory hierarchy.

In addition to traditional links, the global file system supports union links [4]. The files in a directory that is linked by a *union* link appear as if they are part of the directory making the link instead of as a subdirectory. The order in which union links are made is significant. In the case of name conflicts, it is the first link with a given name that is taken. The union link can be used in conjunction with filters to provide a very powerful mechanism for reorganizing file systems for a particular user or purpose.

Links to files are equivalent. There is not a primary or "real" name for a file. If a file is created, and someone else makes a link to it, then the file is deleted, the file still exists with the name of the link. The equivalence of links presents a number of problems in the area of file owership, garbage collection, and what happens when a file with multiple links is modified. These issues are addressed in [10].

Information about files and directories is accessible through directory servers that run on each host that is part of the global file system. All directory queries are directed to these servers. The local format of directories and other information is hidden from the user by the servers. Servers can store directories and other information in the native format for the system on which they reside, or they can use their own format. Information required for the global file system which is not part of the native format can be stored in supplemental (shadow) files.

Files which were not created as part of the global file system are easy to incorporate. Directory links may be made to files created through a system's native file system as long as a directory server runs on the host containing the files. Answers to requests for information about such files will be the value from the native file system where appropriate, a derivative value, where a mapping exists, a default value, or an indication that the information is not available.

3.2 File Storage and Access

Files are stored in the native format for the system on which they physically reside. Because native file systems do not maintain all the information needed by the global file system, the directory server on the custodial host must maintain the additional information itself. Some of this information is used for replication and access control. Other information includes a reference to the owning virtual system and a partial list of back links to the directories that have links to the file.

Files are accessed using existing remote file access protocols. In the initial implementation, Sun's Network File System is used. The directory mechanism maps from names in the global file system to the custodial host and name of the file on that host. That information can be used by existing file access mechanisms to access the desired file.

[4] Union links are based on the idea of a union mount proposed by Rob Pike at the first *Workshop on Workstation Operating Systems* in Cambridge, Massachusetts, November, 1987. A Union mount allows one to mount multiple file systems on a single mountpoint, the result being the union of the files in the multiple file systems.

3.3 Access Control

Control of access to the actual files stored in the file system is handled by the underlying file access mechanism. Many remote file access mechanisms available today do not provide adequate security. In such cases, added security is desirable, but beyond the scope of this discussion. Such security mechanisms may make use of the supplemental information (such as an access control list) stored along with the file.

Access control must also be applied to information in the directory servers. Access control information may optionally be associated with individual directory links. Queries can include an optional authenticator that can be checked by the directory server to decide how to respond to the query.

3.4 How the Global File System Can be Used

This part presents examples of how a global file system based on the virtual system model might be used to organize and find information in a large distributed system. This is not an exclusive list.

3.4.1 Personal Organization

Users will build their own hierarchies of files by creating directories, subdirectories, and files of their own, and adding links to files, directories and subdirectories created by others. Files that are frequently accessed by a user will probably have short names from the root of their virtual system. Since directories of others (and hence, whole hierarchies) can be added to a user's virtual system, the system will probably contain files that a user has never accessed and might not even know about. These files, however, will be much deeper in the hierarchy.

Directory 1 in Figure 1 is the root for bcn's view of the world. His home directory, however, is the subdirectory "bcn". In this organization, subdirectories of "/bcn" contain files and directories of a personal interest, whereas the remainder of the root directory forms his view of the rest of the world. Other users might choose to organize their system differently. For example, one might include those files and directories frequently of personal interest in the root with one's view of the rest of the world relagated to a subdirectory.

When users link other directories to their own hierarchy, it will be possible to add information to the link to specify how much of the merged hierarchy is to be included. It will also be possible to customize parts of the attached hierarchy. An example of this is seen in the subdirectory "interesting-subjects" of directory 1 (Figure 1). This filter uses information on who else has links to files to decide which of the files in subdirectories of directory 8 should be listed.

Each directory and file that a user maintains will be owned by that user. Parts of a user's hierarchy, however, may be owned by other users. Access control information will be maintained along with each file or directory, and with each directory link, and will determine who is allowed to read the

file or search the directory. It is expected that users will make parts of their hierarchies accessible to others, but how much will be decided by the individual.

3.4.2 Project Organization

Just as users will each have their own virtual system, it is expected that projects will have separate virtual systems too. It is likely that everything that is part of a project's virtual system will also be part of the virtual system of at least one member of the project.

Directory 9 in Figure 1 shows the project directory for the virtual system project. In addition to being included in the virtual systems of those involved with the project, it could also be the root of a virtual system of its own. A project virtual system serves several roles. It is a prototype virtual system that can be given to, or merged with that of new users when they become part of the project. It also provides a starting point from which those wanting information about a project can look. Finally, it provides a logically central starting point from which everything related to a project should be accessible.

3.4.3 Index by Author

It is expected that users would maintain a single directory containing papers of theirs which they consider published, and which they want others to be able to access. A service wanting to provide an index of published papers by author would simply set up links from its "author" directory to these directories in each authors hierarchy. By doing this, the indexing service only needs to make updates when a new author needs to be added. Today, library card catalogs provide indices by author. In the future, libraries could be one of the providers of indices by author.

In Figure 1, directories 4 and 5 form an index by author. Both bcn and lazowska maintain their own indices, but by agreement, they decided to distribute the work of maintaining it. Each directory includes the other using a union link. Figures 2a and 2b list the contents of directories 4 and 5 without expanding union links. Anyone with a link to either of these directories will see the contents listed in Figure 2c.

3.4.4 Indices by Topic

Just as users might maintain a directory of their own papers, they might also maintain directories containing links to interesting files or directories on a particular topic. Look at directory 10 in Figure 1 for an example of such a directory. It contains links to papers on distributed systems. Other user might maintain their own list of papers on distributed systems, but with a different emphasis. If a number of people in a field decide that they each trust the views of the others, then they might collaborate to provide a directory whose contents are the union of each of their directories. Users with an interest in a particular topic can use any of the collections that are readable, or the union of several of them.

```
   Neuman,Clifford      JUNE.CS.WASHING /u1/bcn/papers
   Pinkerton,Brian      JUNE.CS.WASHING /u1/bp/Papers
U                       KRAKATOA.CS.WAS /u1/lazowska/authors
```

Figure 2a: Listing of bcn:/authors without expanding union links

```
   Lazowska,Edward      KRAKATOA.CS.WAS /u1/lazowska/papers
   Levy,Henry           WHISTLER.CS.WAS /u1/levy/papers
U                       JUNE.CS.WASHING /u1/bcn/vfs/bcn/authors
```

Figure 2b: Listing of lazowska:/authors without expanding union links

```
   Lazowska,Edward      KRAKATOA.CS.WAS /u1/lazowska/papers
   Levy,Henry           WHISTLER.CS.WAS /u1/levy/papers
   Neuman,Clifford      JUNE.CS.WASHING /u1/bcn/papers
   Pinkerton,Brian      JUNE.CS.WASHING /u1/bp/Papers
```

Figure 2c: Listing of bcn:/authors, union links expanded

As with the author index, it is likely that master indices will be maintained by libraries or other organization. These indices will allow one to find the directories containing information on a particular topic. Directory 8 is an example of such a master index. There are likely to be multiple central indices, each of which might compete for use based on their own strengths such as how complete their index is, how accurate it is, or how much junk you get back when using the index of a competitor.

3.4.5 Browsing

One way that information can be found using a file system organized with the virtual system model is through browsing. If you are interested in a particular topic, you can connect to the virtual system of someone who you know is also interested in that topic, or perhaps to the virtual system for a related project. You could then look through those virtual systems for documents or files of interest. Of course, you would only see those files that the owner of the virtual system has authorized you to see. The inability to see some information, though, hopefully tends to weed out information you would not be interested in anyway.

In order for browsing to work best, virtual systems owned by projects should include a directory containing links to related work. Users would maintain directories containing pointers to files that they consider interesting or relevant. Files in which others would (or should) have little interest should be kept from their view by applying appropriate protections.

Browsing is considerably more likely to be effective under the virtual system model than on more traditional file systems. The virtual system model encourages users to make their own links to the files in which they have an interest. As such, interesting files are likely to appear in the hierarchies of multiple people, thus increasing the likelihood that they will be found by browsers.

3.4.6 Finding Things

In society today, if something is available that is of interest, it is usually found through directories such as the phone book or yellow pages, through reading newspapers and other periodicals, or by word of mouth. In the computer science community, these sources of information are supplemented by technical papers, electronic mail and mailing lists. These methods of discovery are natural, and it is likely that they will continue to find significant use even once other mechanisms are in place.

The virtual system model allows much of this information which might be useful in finding objects, but which to date could only be obtained by external means (such as asking the appropriate person) to be included as part of the file system. This information provides a matrix through which users can navigate to find the desired information. Services might even spring up to help users navigate through this matrix. An example of such a service is described in [14] and makes use of resource discovery agents. These agents accept queries from users and use the information provided by the user to find objects in which the user is interested. The multitude of links in a system based on the virtual system model can provide the information needed to direct such searches.

3.5 Comparison to Existing File Systems

In Section 2, certain shortcomings of existing approaches were discussed. The file systems considered were NFS, Andrew, Sprite, Locus, and QuickSilver. Here these approaches are explicitly contrasted to the global file system built using the virtual system model.

In all five systems, the mechanism used to find files places constraints on their location. This is the case even though the storage site is not part of the pathname. The granularity of distribution is at the level of individual file systems instead of individual files. All files in the same part of the file hierarchy will be located at the same storage site. In the global file system, distribution to storage sites is done on a file by file basis, and this constraint is avoided.

All five systems provide at least a limited level of customization. This customization is primarily the result of using symbolic links relative to one's home directory. With the exception of QuickSilver, links in these systems are not equivalent to the primary pathname for the files. Moving or deleting the file or directory that is the target of the link leaves a link that no longer works. Several types of links are supported by QuickSilver, though none are equivalent to the primary pathname for the file in all cases [5]. Links in QuickSilver do, however, solve some of the reference problems just described. In the global file system, there is no primary pathname, all links to files are equivalent, and there is no problem with dangling references.

[5] Hard links result in a copy of the file being made if the link crosses a machine boundary.

None of the five systems provide the tools necessary to make links as useful as possible. Customization by links is limited to the name of the link, and more detailed customization of the "renamed" hierarchy is not possible. In the global file system, filters and union links allow one to further customize the linked hierarchies.

NFS allows file systems to be mounted at arbitrary points on the local system. This provides a mechanism for customization on a system by system level, since each system might mount remote partitions on different mountpoints. Andrew, Sprite, and Locus each provide a single global namespace within which all files are named. They avoid the possible confusion that arises from having different namespaces on different physical processors. Both QuickSilver, and the global file system support customization on a user by user level. This customization is independent of the physical processor being used.

QuickSilver addresses customization through the user-centered namespace. It enforces use of the customized view since the only way to access the files or directories of others is through the links and updated prefix table entries customizing the namespace. This differs from the global file system which allows the virtual system name to be explicitly specified, thus allowing access to external objects.

Compared to NFS, Andrew, Locus, Sprite, and QuickSilver, the global file system provides tools allowing a significantly greater level of customization of the user's view of the file system. The union link in conjunction with active filters allow lower levels of an included hierarchy to be customized. These tools, together with a directory organization that does not tie files or directories to particular hosts allows users to organize files in a manner that best suits their own needs, and the needs of those who need to find the files.

4 Naming

So far, only the file system has been described. The virtual system model extends to other system components as well. The system component most closely related to the file system is naming. The directory service described in the last section can be used to name more than just files and directories. It can be used to resolve user-assigned names for services, other users, hosts, files, and even other virtual systems. Objects may have multiple names, and across different virtual systems, the same name can refer to different objects. Within a single virtual system, however, a name can only refer to a single object [6]. Thus, there exists a many-to-one mapping from *virtual_system:name_within_that_system* to any object that is part of the global system.

The directory service provides a mechanism to resolve user-level names within a virtual system. User-level names resolve to globally unique system-level names which can be used to identify, and ultimately to locate the object. Services resolve to the host on which the service is provided and the name or number of the port to which one must connect. Filenames resolve to the custodial host, and the name of the file on that host. In both of these cases, the system-level hostname can

[6] If objects are replicated, the set of replicas can be thought of as a single object.

be further resolved using existing hierarchical nameservices such as the Internet Domain Naming System[15], or DEC's global naming system[5].

5 Authentication & Authorization

Authentication is used to make sure that an entity claiming to have a particular name is in fact, the entity to which that name refers. Authorization is used to make sure that a named entity is allowed to perform a particular operation. For both to work, there must be an accepted mapping from a set of names to entities requiring authentication. Although this mapping does not need to be the same in all places, authorization is significantly simplified if it is. Since virtual systems overlap, allowing the use of non-unique user-level names for authentication could create parts of the system where two different sets of names are in use. This would complicate access control for objects in those overlapping regions. For this reason, authentication and authorization will be based on the globally unique names of users. This does not preclude having an additional level of indirection through which users specify names.

It is expected that authorization will be accomplished primarily through access control lists. The concept of groups is necessary for access control lists to provide a flexible interface for access control. Groups can contain users or other groups. As with names, groups should be represented uniquely within access control lists. Users, though, might use different names to identify them. A problem arises in deciding how to display the content of ACLs to the user. With groups, it is perhaps even more important for the user to see a name with which he is familiar. Mapping back from the unique name to a local name might be costly, however.

One solution is to give the user the option of how he wants access control lists displayed. The average user would probably choose the local names. Since average users might only have a small number of users and groups defined within their virtual system, the mapping might not take as long as for a more advanced user. In any case, information is needed to help users figure out what users and groups are if the groups do not map to local names, or for which local names do not exist. Each group can have a comment associated with it. These comments can be displayed next to the local or unique name when the ACL is listed.

So far, the naming side of authentication has been discussed, and how these names are used for access control. Authentication itself has not been talked about. One of the problems that arises as a system spans multiple organizations is lack of trust. Services can't be required to trust every authentication server that exists. It might not even be the case that there is a single authentication server trusted by everyone. A particularly paranoid application might only trust authentication performed solely by its local authentication server.

Another issue concerned with authentication is the distribution of authority. Unless, for every communicating pair of entities, there exists some trusted entity sharing a secret with both parties, then multiple authentication servers might be involved in authenticating a principal.[7] It is important

[7] Even with public key systems, the public key to party A must be known to a key repository whose public key is known by B.

that the end service know which authentication servers were involved when deciding whether to trust the authentication. More on this can be found in [2].

There may be different competing collections of authentication servers. Such services might differ in who they are run by, or perhaps even in the protocol required for initial authentication when the user logs on. It should be possible, from within each virtual system, to decide which authentication services one is willing to trust. By attaching an "owning" virtual system to each object to be protected, this choice of who to trust is passed along to individual objects.

An area related to authentication and access control that has received little attention is accounting. It is related in the sense that once one's quota for a service has been used up, or when one has run out of money to pay for a service, access might be denied. One also doesn't want to get billed for a service that was used by someone else. Accounting will become more important in a system like the one being described because the system provides the connection between service providers and consumers that may not be within the same organization. Services analogous to banks will be required, and a mechanism to allow users to authorize their bank to pay a server will become part of the access protocol for certain services. See [7, 8] for related discussion.

6 Processes and Processors

Each virtual system will be associated with zero or more processors. These will be the processors on which applications will run. Processors within a single virtual system should be able to be of different types, and the processor chosen should be based on the requirements of the application as well as other information available at the time (such as load, etc). Applications should be able to choose processors from within the virtual system on which they are running, within which they were installed, or, alternatively, on other processors that they find dynamically.

When a process or application is attached to a virtual system, a closure is formed. A closure is code to be executed and the set of bindings seen by a process. Just as an attached virtual system can be used to restrict the processors on which an application runs, it can also be used to restrict the objects that can be accessed. This can be accomplished by setting a flag indicating that processes running within a particular virtual system should only be able to access objects that are part of that virtual system, and that the ability to switch to other virtual systems should be disabled. By building a virtual system with limited size, and by protecting the virtual system so that it can't be modified from within, processes can be run in a protected environment. In order for this to work, the system on which the process executes must enforce the restrictions imposed by the virtual system. By suitably restricting what a process can read and write, the virtual system model can provide support for multi-level secure systems.

7 Services and Applications

In existing systems, services and applications differ in the way they are accessed. A service typically provides basic operations and is usually accessed across a network. An application is typically a local program, but it may provide a front end for one or more services. In the virtual system model applications can be associated with set of processors on which they may run. When a user runs an application, it might run remotely. The distinction between an application and a service will be more a matter of whether the interface is intended for humans, or for other programs.

Given the name of a service or application, clients will use the directory service to find the server (or executable). If multiple instances exist, the client will have to make a choice. The choice is made by including the selected server or set of servers in one's virtual system. The use of filters on links in the directory service provides a useful tool for choosing a server. A link can be made to a directory containing pointers to instances of a service. A filter associated with the link can be used to filter out all instances that do not satisfy the client's constraints (e.g. price).

8 Autonomy

The physical systems that are tied together by the virtual system model are autonomous. Each can operate independently from the others. Naturally, one would not be able to access objects or services which reside on a system that is unreachable due to network or hardware problems. Availability can be enhanced by replication, though access to replicated objects can impose its own overhead and constraints.

The user level naming mechanism has no central database, and no critical nodes. The user level naming mechanism maps names to addresses which are really system level names. In some cases, resolving the system level names might require access to critical sites. Attempts will be made to minimize such dependencies.

9 Conclusions

The virtual system model provides a framework within which users can build a customized view of the objects and services available in a large distributed system. The approach described in this report exhibits all of the characteristics mentioned in the introduction: to the user, local and remote objects are accessed the same way; the name of an object depends neither on the objects location, nor the location of the user accessing it; users are able to customize their views of the system; and the approach extends to all components of the system, not just the file system.

Filters and union links are two important features of the directory service on which the the virtual system model is built. These tools allow a user's view to apply to new objects in the system as they are created. Those and other features of the directory service allow objects to be organized

in a manner that will allow them to be found in a very large system.

The virtual system model extends to naming, authentication, and authorization, to the use of network services, and to the running of application programs. This ability to adopt different views in these areas is particularly important in an environment that consists of many organizations where users need control over the services that are trusted, those with which one does business, and the processors on which applications can run.

A file system based on the virtual system model is presently under construction and a prototype is running on several sites scattered across the Internet. As experience is gained, and as other pieces of the system are put in place, changes are expected in the way people think about and use distributed computer systems. The virtual system model can provide the basis for a global operating system that more closely parallels the way people interact in society.

Acknowledgements

Ed Lazowska, Hank Levy, David Notkin, and John Zahorjan helped me to refine the ideas presented in this report. Helpful comments on earlier drafts were also received from Brian Pinkerton, Terry Gray and Kathy Faust. Ed Lazowska deserves special thanks for the numerous discussions out of which many of the ideas emerged. Some of the ideas emerged from discussions with Alfred Spector and other instructors at the Arctic '88 Course on Distributed Systems.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–59, January 1985.

[2] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder. A global authentication service without global trust. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 223–230, 1986.

[3] Luis-Felipe Cabrera and Jim Wyllie. QuickSilver distributed file services: An architecture for horizontal growth. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 23–27, March 1988. Also IBM Research Report RJ 5578, April 1, 1987.

[4] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. In *Proceedings of the Symposium on Operating Systems Principles*, November 1987. Also *Transactions on Computer Systems*. 6(1):51-81, February 1988.

[5] Butler W. Lampson. Designing a global name service. In *Proceeding of the ACM Symposium on Principles of Distributed Computing*, 1985.

[6] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[7] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[8] B. Clifford Neuman. Sentry: A discretionary access contol server. Bachelor's Thesis. Massachusetts Institute of Technology, June 1985.

[9] B. Clifford Neuman. Issues of scale in large distributed operating systems. Generals Report, Department of Computer Science, University of Washington, May 1988.

[10] B. Clifford Neuman. A global file system based on the virtual system model. Technical Report In Preparation, Department of Computer Science, University of Washington, 1989.

[11] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–35, February 1988.

[12] G. Popek and B. Walker, editors. *The Locus Distributed System Architecture*. M.I.T. Press, Cambridge, Massachusetts, 1985.

[13] R. Sandberg et al. Design and implementation of the Sun network file system. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

[14] Michael F. Schwartz. Resource discovery in a massively distributed environment. Department of Computer Science, University of Colorado (Boulder), November 1987.

[15] Douglas B. Terry, Mark Painter, David W. Riggle, and Songnian Zhou. The Berkeley internet domain server. In *Proceedings of the 1984 Usenix Summer Conference*, pages 23–31. June 1984.

[16] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The Locus distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[17] Brent B. Welch and John K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 184–189, May 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1990	Final

4. TITLE AND SUBTITLE
ADVANCED NUMERICAL TECHNIQUES OF PERFORMANCE EVALUATION VOLUME II

5. FUNDING NUMBERS
C: N66001-87-D-0136

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Washington
Department of Computer Sciences
Seattle, WA 98195

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Ocean Systems Center
San Diego, CA 92152-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NOSC TD 1837

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This two volume document provides various advanced numerical techniques used in performance evaluation.

14. SUBJECT TERMS
performance evaluation

15. NUMBER OF PAGES
483

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT