1

Optimistic Parallelization

Greg Morrisett[1] Maurice Herlihy[2]

October 1993

CMU-CS-93-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Supported in part by a National Science Foundation Fellowship. Email address: jgmorris@cs.cmu.edu.

[2]Email address: herlihy@crl.dec.com.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Parallelism, optimistic, transactions.

Abstract

To transform a sequential program into a concurrent program, a compiler typically divides the sequential program into a partially-ordered set of basic blocks, allowing unrelated blocks to execute concurrently. Blocks may execute concurrently only if there are no dependencies among them, and therefore a compiler can introduce concurrency only to the extent that it can guarantee the absence of dependencies. A limitation of this technique is that it is necessarily conservative: it may be difficult or impossible to prove the absence of dependencies even when no dependencies exist.

This paper investigates *optimistic parallelization*, a complementary technique for parallelizing sequential code. Blocks with potential conflicts are allowed to execute in parallel, and conflicts are detected at run-time. When a conflict is detected, the conflicting blocks are rolled back and re-executed in sequential order. Optimistic parallelization can enhance concurrency when the compiler cannot prove the absence of dependence among independent blocks, and when dependencies occur, but are sufficiently rare.

We show how conflict detection and roll-back can be accomplished efficiently through relatively simple changes to the caches and the cache-coherence protocol of a shared-memory multiprocessor. We then show how a compiler might exploit these mechanisms when parallelizing programs. Finally, using simulation results, we show that optimistic parallelization using our mechanisms can give good performance.

1 Introduction

To transform a sequential program into a concurrent program, a compiler typically divides the sequential program into a partially-ordered set of *basic blocks* (single-entry single-exit code blocks), allowing unrelated blocks to execute concurrently. For example, consider the following loop:

```
for (i = 0; i < N; i++)
    A[i] = 2 * B[i];
```

Each iteration of this loop might be considered a basic block, and these blocks might be distributed among the processors on a multiprocessors as follows:

```
for (i = 0 + PROCESSOR; i < N; i += NO_OF_PROCESSORS)
    A[i] = 2 * B[i];
```

This transformation, however, fails to preserve correctness if any of the following data dependencies occur:

- *flow-dependency*: an earlier iteration writes a variable read by a later iteration.
- *anti-dependency*: an earlier iteration reads a variable written by a later iteration.
- *output-dependency*: two iterations write to the same variable.

A compiler can introduce concurrency only to the extent that it can guarantee the absence of such dependencies. The Banerjee test [2] is the basis for most compile-time techniques for proving the absence of data dependencies. (For example, see [4, 14, 15])

A limitation of such techniques is that they are necessarily conservative: it may be difficult or impossible to prove the absence of dependencies even when no dependencies exist. For example, the procedure **permute** of Figure 1 has an output dependency if there are two index values, i_1 and i_2, such that $B[i_1] = B[i_2]$. A compiler may parallelize this loop only if it can establish that for every invocation of **permute**, no such i_1 and i_2 exist. In addition, the compiler must ensure that the arrays A and B do not overlap in memory, nor do A and C. Proving such properties is undecidable in general, and often difficult or impossible in practice.

This paper investigates *optimistic parallelization*, a complementary technique for introducing concurrency into sequential programs. Blocks with potential conflicts are allowed to execute in parallel, and conflicts are detected at run-time. When a conflict is detected, the conflicting blocks are rolled back and re-executed in sequential order. Optimistic parallelization can enhance concurrency in circumstances when the compiler cannot prove the absence of dependence among independent blocks, and when dependencies do occur, but are sufficiently rare. Optimistic parallelization does not exclude the use of conventional methods when absence of conflict is detectable.

The premise of this paper is that simple hardware support can make optimistic parallelization an effective technique for introducing parallelism into certain kinds of sequential programs. Optimistic concurrency control, in one form or another, is an old idea. Our

```
void permute ()
{ int i;

  for (i = 0; i < N; i++)
     A[B[i]] = C[i];
}
```

Figure 1: Problematic Loop

contribution is to explore the feasibility of optimistic methods in a specific context: parallelizing sequential code on shared-memory multiprocessors. Optimistic techniques can provide adequate performance only if:

- Data conflicts are sufficiently rare.
- Conflict detection is sufficiently inexpensive.
- Roll-back is sufficiently fast.

In this paper, we focus on the last two issues. To make conflict detection and roll-back fast, we propose a set of simple modifications to standard caches and cache consistency protocols. To test our approach, we hand-compiled a number of programs found in the literature, and ran them on a simulated multiprocessor incorporating our modifications. Our results are promising: we were able to speed up a number of applications, some substantially. We believe that optimistic techniques merit further study.

2 Architecture

This section describes the basic architectural support needed for optimistic parallelization. The description is given in terms of transactions on the shared memory. In later sections we will introduce additional refinements to the proposed mechanisms.

A *transaction* is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:

- *Serializability*: Transactions appear to execute in a serial, one-at-at-time order.
- *Atomicity*: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either *commits*, making its changes visible to the other processes, or it *aborts*, causing its changes to be discarded.

Whenever the compiler is unable to establish the absence of dependencies among concurrent blocks, those blocks are executed as transactions. The notion of a transaction originated in the database literature (viz. [7]). Unlike database transactions, which may access large amounts of data residing on a disk, our transactions are short-lived activities that access a relatively small number of memory locations in primary memory. Concurrent database transactions may (usually) be serializable in any order, but our transactions must be serializable in the order of their corresponding basic blocks.

In addition to the usual set of instructions, the architecture provides the following *transactional instructions*:

- **Trans_Read** reads the value of a shared variable into a local variable.
- **Trans_Write** tentatively writes the value in a private variable to a shared variable. This new value does not become visible to other processors until the transaction successfully commits (see below).
- **Commit** attempts to make the transaction's tentative changes permanent. It *succeeds* only if no other transaction has updated any location in the transaction's read or write set, and no other transaction has read any location in this transaction's write set. If it succeeds, the transaction's changes to its write set become visible to other processes. If it *fails*, all changes to the write set are discarded. Either way, **Commit** returns an indication of success or failure.
- **Abort** discards all updates to the write set.

This architecture is a simplified version of *transactional memory*, a cache structure proposed by Herlihy and Moss [8]. A complete description of the transactional memory implementation is beyond the scope of this abstract (see [8] for details). For now, we remark that transactional memory is implemented by modifying standard *ownership-based* cache consistency protocols. It requires a small, fully-associative *transactional cache* in addition to the regular cache. Non-transactional operations use the same caches, cache controller logic, and consistency protocols they would have used in the absence of transactional memory. Custom hardware support is restricted to caches and their controllers; transactional memory requires no other changes to standard processor architectures.

3 Using Transactional Memory

In this section we show how to use transactional memory for optimistic parallelization, and we propose some simple extensions for efficiency.

Here is a first attempt at parallelizing the loop from Figure 1:

```
for (i = 0 + PROCESSOR; i < N; i += NO_OF_PROCESSORS) {
    restart:
      t1 = Trans_Read(&C[i]);
      t2 = Trans_Read(&B[i]);
      Trans_Write(&A[t2],t1);
      while (counter != i) /* wait */ ;
      if (!Trans_Commit()) {
        backoff ();
        goto restart;
      }
      Increment (&counter);
}
```

```
for (i = 0 + PROCESSOR; i < N; i += NO_OF_PROCESSORS) {
      Set_Priority_Register(N - i);
    restart:
      t1 = Trans_Read(&C[i]);
      t2 = Trans_Read(&B[i]);
      Trans_Write(&A[t2],t1);
      while (counter != i) /* wait */ ;
      if (!Trans_Commit())
        goto restart;
      Increment (&counter);
}
```

Figure 2: Optimistically Parallelized Loop

If the computation terminates, the proper serialization is observed. Unfortunately, this simple translation can lead to livelock. If a later iteration writes to a location before an earlier iteration accesses it, then the earlier iteration's transaction will continually abort and restart, while the later transaction will wait forever for the counter to be incremented.

3.1 Priority Registers

Effective parallelization requires that later iterations be aborted in preference to earlier iterations. To this end, we augment transactional memory as follows. Each processor is given a *priority register* that holds a fixed-size value. When a conflict occurs, the cache consistency protocol aborts the transaction whose register holds the lesser value. If the values are the same, the original semantics of transactional memory is preserved (either transaction can be aborted).

The example loop might now be parallelized as shown in Figure 2. Earlier iterations have higher priority than later iterations, and if a conflict occurs, the earlier iteration will progress. The transaction with the highest priority will never be aborted by a data conflict.

As an additional optimization, if a lower-priority transaction is about to write to a variable read or written by a higher-priority transaction, it can simply be stalled until the higher-priority transaction commits or aborts.

When a priority register is about to overflow, we can re-normalize the blocks' priorities, giving the "earliest" transaction the highest available priority, or processors can re-normalize priorities on the fly by keeping track of the highest and lowest priority transactions in the system.

3.2 Exceptions

Exceptions such as an address fault or divide by zero can be handled as follows. When the processor receives an exception, it delays the transaction until its priority is greater than or equal to any other transaction's priority. At that point, it attempts to checkpoint the block by committing its partially completed transaction. Because there are no lower-priority transactions, checkpointing the block will not cause any data conflicts. If the commit is successful, then it handles the exception, and then starts a new transaction to execute the remainder of the block. If the commit is unsuccessful, it handles the exception and restarts.

As mentioned above, transactional memory uses a small transactional cache for conflict detection and recovery. Transactional cache overflow might be avoided if the size of the transactional cache is available to the compiler, but it is preferable to compile programs in a configuration-independent way. When the transactional cache is about to overflow, it simply sends an interrupt back to the processor, and the interrupt is handled as described above.

4 General Control Constructs

So far, we have considered only loops. In this section, we sketch a general transformation for other control constructs, with special attention to procedure calls.

At or near the beginning of each basic block, a thread is forked to execute the "next" block. The earlier block is given higher priority than the later block (and any blocks forked by the later block.) Branch prediction techniques [18] may be needed to guess which block will be next. If the guess is wrong, then the speculative block (and the blocks it forked) can be aborted and rolled back.

For certain constructs, additional control information can be used by the compiler to schedule basic blocks earlier or to avoid aborting work that is always necessary. As an example, consider an if-statement:

```
if (E1)
  S1;
else
  S2;
S3;
```

The compiler can arrange to have E1, S1, S2, and S3 evaluated in parallel as transactions. Once the evaluation of E1 is complete and the direction of the branch is determined, either S1 or S2 can be forced to abort. However, since control flow must always go through S3, it does not need to be aborted unless there is a data conflict with one of the earlier statements.

Later blocks may need values computed by earlier blocks (*e.g.*, a loop index). These values can be stored in shared variables. If we can determine statically that a block depends on a value computed by an earlier block, then we can delay forking the later block until the value is computed. If we cannot determine statically whether such a dependency exists, the later block can read the variable transactionally, ensuring that if an earlier block updates the variable, the later block will be aborted and restarted. Values such as loop indices should be calculated as soon as possible.

Calling procedures in parallel requires using cactus stacks for activation frames, or allocating the frames from the heap. For recursive procedures, it is not always possible to assign priorities statically. For example, consider the following binary-tree traversal.

```
void traverse (tree t)
{
   if (t != NULL) {
     traverse (t->left);
     traverse (t->right);
   }
}
```

To execute the traversal in parallel, we must assign priorities so that all nested calls traversing the left-hand subtree have higher priority than any call traversing the right-hand subtree. Such priorities can be assigned dynamically by allocating each invocation a range of priorities, where every priority allocated to a left-hand call is higher than any value in the allocated range of the right-hand call. The depth to which recursive calls can be parallelized is limited by the size of the priority register.

5 Simulation Results

Our basic premise is that support for extended transactional memory can make optimistic parallelization effective. To test this hypothesis, we hand-compiled a number of programs found in the literature, and ran them on a simulated multiprocessor incorporating our modifications. Each program was parallelized using both conventional and optimistic techniques.

We kept our experiments as simple as possible. All processors were started at the beginning of the program, executing the same code. For "pessimistic" parallelization, we parallelized loops for which the absence of dependencies was easy to prove statically. The processors executed the iterations of the loop in round-robin fashion with no synchronization. All other code was executed by a single processor. Minimal barrier synchronizations were inserted between the serial and parallelized code to prevent race conditions. No other parallelization techniques (such as renaming to eliminate dependencies) or combining trees for associative operations were used.

For "optimistic" parallelization, we parallelized loops for which it was not apparent whether data dependencies would exist at runtime or not. The reads and writes were converted to transactional reads and writes and a shared counter was used to force the proper serialization. At the end of the loop, a barrier synchronization was performed and the shared variable was reset.

5.1 Proteus

Our programs were simulated using a version of the Proteus [3] simulator which we modified to support enhanced transactional memory. Proteus is an execution-driven simulator system for multiprocessors developed by Eric Brewer and Chris Dellarocas of MIT. The program to be simulated is written in a superset of C. References to shared memory trap to the simulator, and other instructions are executed directly, augmented by cycle-counting code inserted by a preprocessor. Because most of the program is executed directly by the host processor, large simulations can be run relatively quickly. Proteus does not capture the effects of instruction caches or local caches.

The simulator can be configured to support a variety of multiprocessor architectures. We started with a bus-based, sequentially consistent, cache-coherent architecture using the Goodman Snoopy-cache protocol [6]. We augmented the cache simulation to support transactional memory and most of our enhancements for optimistic parallelization.

The following parameters describe the machine(s) that we simulated: A maximum of 8 processors were used. Each processor had a direct-mapped 64 kilobyte cache with a line size of two 32-bit words. The transactional caches associated with each processor held 64 lines. Each processor had a 32-bit priority register. Cache access latency was 1 cycle while memory access latency was 4 cycles.

5.2 Simulation Results

The programs that we chose to simulate are drawn from a range of applications. They are necessarily small since parallelization had to be done by hand. We chose programs and algorithms which were published as our baseline sequential code and concentrated on programs whose control was loop-based and for which optimistic parallelization looked promising. Data sets were chosen essentially at random. Since Proteus is entirely deterministic, the same data sets were used for all versions of a program.

For each program, we give figures which show the speedup for the optimistically parallelized code and, if applicable, the pessimistically parallelized code. In the following paragraphs, we give a brief description of each of the programs and attempt to explain these performance results.

The **knapsack** program is taken from [17, page 596] and uses dynamic programming. The goal is to maximize the value of the elements that can be placed into a bag of fixed capacity, where each element has an associated size and value. As in most dynamic programming problems, there is a potential flow-dependency between earlier and later iterations. Thus, the core of the program cannot be parallelized using conventional techniques. However, conflicts are dependent on the data, so the core can be optimistically parallelized. Figure 3 shows the speedup for the optimistically parallelized code. As a data set, we used a fixed-capacity knapsack, a fixed number of items, and random sizes and values for each different item. The sizes of the items were restricted to be a small fraction of the capacity of the knapsack.

The **convex** program is taken from [17, page 364] and is a simple $O(n^2)$ program that finds the convex hull of a set of randomly placed points. Essentially, one point known to be on the hull is chosen and points with the minimum angle from the last chosen point are successively added. Neither of the main loops can be parallelized using conventional techniques. We chose to parallelize the inner loon optimistically. Figure 4 shows the speedup for the optimistically parallelized code.

The **radix** program is a radix sort of 1024 32-bit random values, using an 8-bit radix. The program is taken from [17, page 140]. The program has an outer loop that is executed a small number of times (in this case 4) and nested within it are five separate loops. Only two of these loops may be conventionally parallelized given our compilation model. An additional loop can be optimistically parallelized. Figure 5 shows the speedup for both the pessimistically and optimistically parallelized code.

The **solver** program is a simple $O(n^3)$ program which solves a set of n linear equations

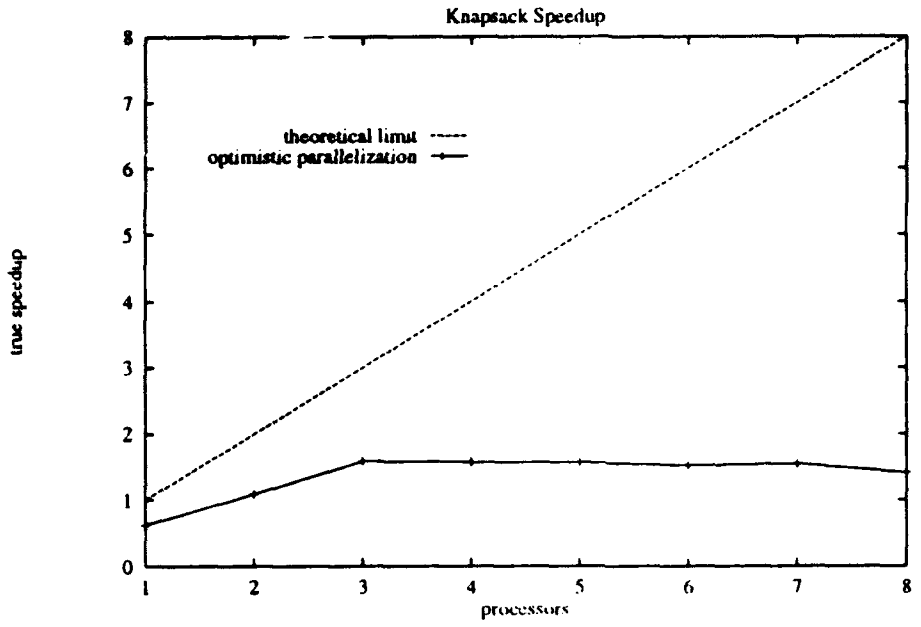

Figure 3: Knapsack

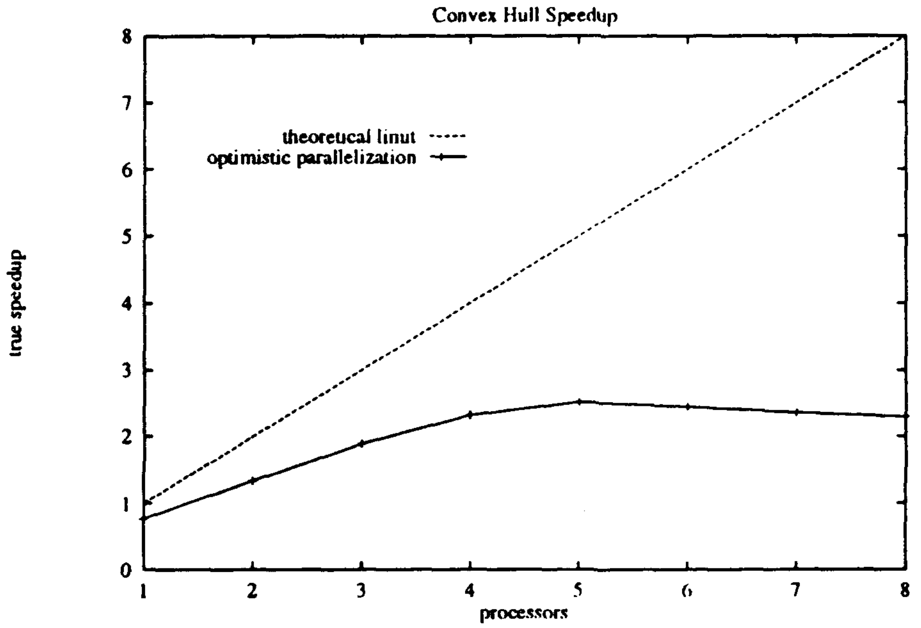

Figure 4: Convex Hull

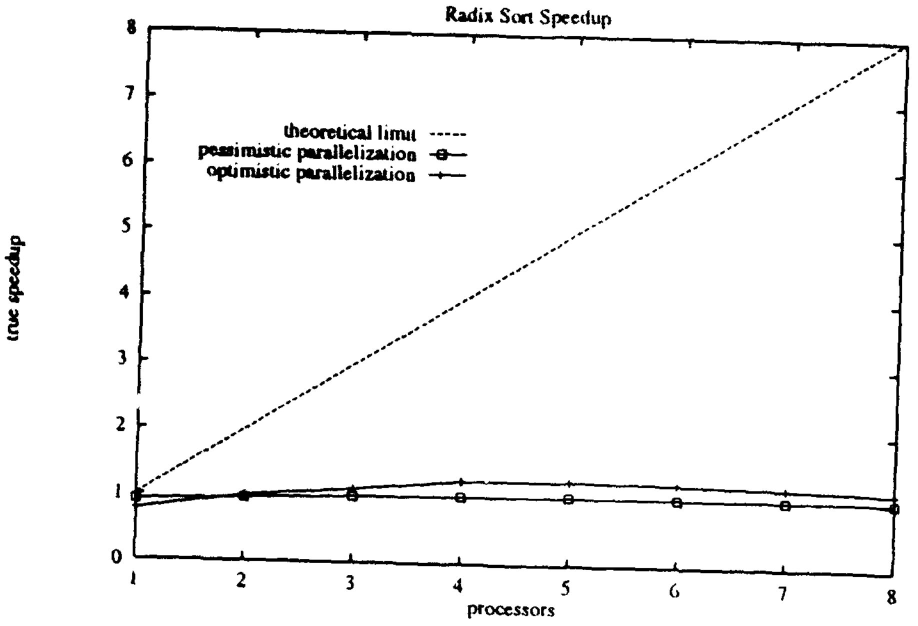

Figure 5: Radix Sort

with n unknowns using Gauss-Elimination. The coefficients of the equations were chosen at random. The program is take from [17, pages 539-540] with some optimizations added as suggested in [16, page 36]. The program consists of two phases, an elimination phase which drives a matrix to an upper-triangular form, and a substitute phase which calculates the unknowns. To make row interchange fast, an auxiliary index array is used to access the rows. The use of this array leads precisely to the potential data dependency problem of Figure 1. As a result, only the innermost loop of the elimination phase can be pessimistically parallelized. We chose to optimistically parallelize the second-most-inner loop. The substitute phase has a definite data dependency across its inner-most loop and cannot be pessimistically parallelized. Even though the dependency is definite, we applied optimistic parallelization since some work can still be done in parallel. Figure 6 shows the speedup for both the pessimistically and optimistically parallelized code. Note that the pessimistic code outperforms the optimistic code for more than 4 processors. We suspect that this is due to the large transaction size of the loops that were chosen to be parallelized (see below).

Some general conclusions can be drawn from these benchmarks. For instance, in all of the benchmarks, the overhead of doing a loop transactionally becomes acceptable as soon as more than one processor is used. However, speedup seems to level off at around 3-5 processors in all cases. One reason speedup levels off is that there is a limited amount of parallelism in the programs and more importantly, only loops were identified for parallelization. Furthermore, the parallelism within these loops is limited by the data dependencies that occur at runtime.

Another reason speedup levels off is due to inefficiencies in the translations. A portion of this inefficiency is due to the naive implementation of the barrier synchronizations and the shared location used to serialize transactions. This can be alleviated by using more sophisticated techniques such as counting networks or sense-reversing barriers. However,

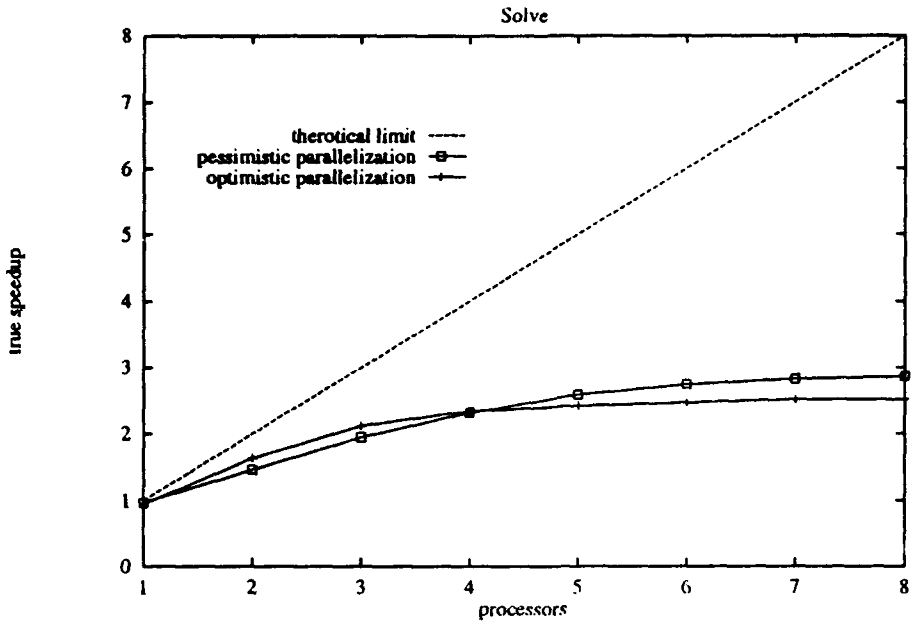

Figure 6: Equation Solver

such techniques can increase the latency of operations, making them unattractive for small numbers of processors.

Some of the inefficiency in the translation is due to the small granularity of the work done by each processor. Small transactions make it difficult to amortize the cost of doing the synchronizations. For instance, the transactions of the knapsack program are only 4 lines of code. The size of transactions can be increased by using techniques such as strip-mining or loop-unrolling. However, making a transaction larger increases the probability that a conflict will occur with some other transaction, especially as the number of processors grows. Larger transactions also require larger transactional caches.

To demonstrate the effect that transaction size can have on execution time, we ran the `radix` program, varying the number of loop-iterations per transaction and the number of processors. Results for these tests are shown in Figure 7. As expected, very small transactions (one iteration) give poor performance with any number of processors. Larger transactions give better performance for small numbers of processors, but performance can degrade for large transactions and large numbers of processors.

6 Related Work

There is a vast literature on optimistic techniques for database synchronization. The two earliest and most influential papers are by Thomas [19] and by Kung and Robinson [12].

Knight [11] proposed an architecture in which basic blocks were scheduled to run in parallel with transactional semantics. He also proposed the use of a shared counter to force the proper serialization. The ParaTran System [10, 20] applied these ideas in an optimistically parallelizing compiler for Scheme. ParaTran used software techniques adapted from the

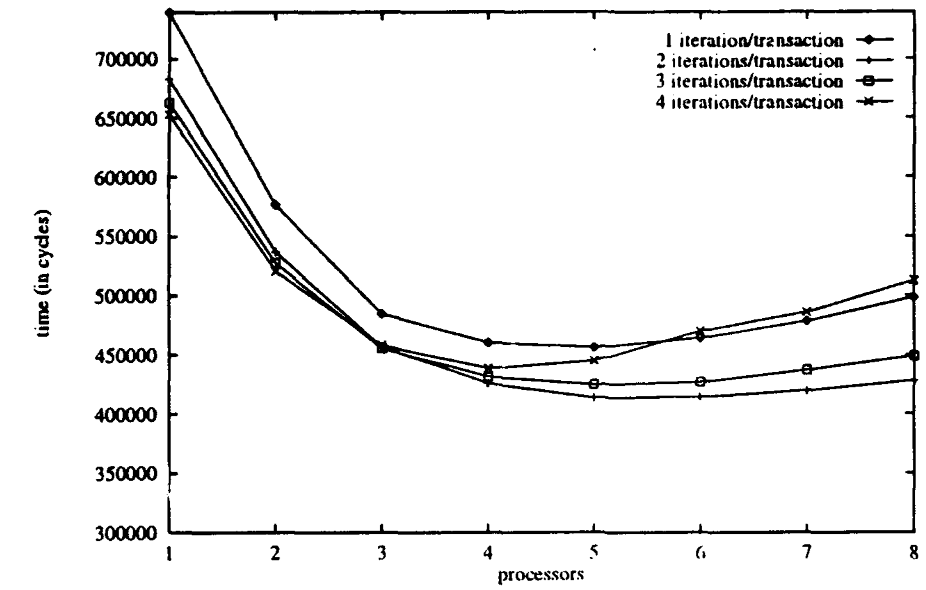

Figure 7: Varying Transaction Sizes in Radix Sort

database literature for conflict detection and recovery.

Franklin and Sohi [5] propose a hardware architecture that optimistically parallelizes code at runtime. Processors execute basic blocks in parallel. Serialization is guaranteed by organizing processors in a queue, and branch prediction is used to determine the next basic block. A "future file" is used to forward register values from one processing element to the next, and an "address resolution buffer" detects conflicts. Franklin and Sohi ran simulations of real programs on this architecture and observed substantial speedups.

Although Franklin and Sohi's architecture resembles ours in several respects, it has two limitations. First, it is a radical departure from traditional architectures, requiring a complete change of the processing elements. Our architecture requires modest changes to caches and their controllers, and support for a few new instructions. Second, in their architecture, processors are forced to execute a serial stream of instructions, while in our scheme, a multiprocessor may still execute independent instruction streams. Recent studies [1, 13, 21] have shown that taking advantage of independent instruction streams can have a significant impact on performance.

In [9], Larus and Huelsbergen propose two techniques which support *dynamic* program parallelization. Dynamic parallelization, like optimistic parallelization, can find more parallelism than static analysis by using runtime information. Unlike optimistic parallelization, dynamic parallelization first tests to see if it is safe to run a parallelized version of code and if not, falls back on a sequential version of the code. Obviously, testing to see if parallelization can be done must be cheap. Thus, these tests are usually conservative approximations.

References

[1] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351. IEEE, May 1992.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[3] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A high-performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Sept. 1991.

[4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 162–175. ACM, 1986.

[5] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67. IEEE, May 1992.

[6] J. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 124–131. IEEE, June 1983.

[7] J. Gray. *Notes on Database Operating Systems*. Springer-Verlag, Berlin, 1978.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for highly concurrent data structures. Technical Report ?, Digital Equipment Corp., Cambridge Research Laboratory, Apr. 1992.

[9] L. Huelsbergen and J. R. Larus. Dynamic program parallelization. In *Proceedings of 1992 ACM Conference on Lisp and Functional Programming*, pages 311–323, June 1992.

[10] M. Katz. ParaTran: A transparent transaction based runtime mechanism for parallel execution of scheme. Technical Report MIT/LCS/TR-454, Massachusetts Institute of Technology, July 1989.

[11] T. Knight. An architecture for mostly functional languages. In *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, pages 105–112, Aug. 1986.

[12] H. T. Kung and J. T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[13] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57. IEEE, May 1992.

[14] Z. Li and P. Yew. Practical methods for exact data dependency analysis. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.

[15] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[16] W. H. Press. *Numerical Recipes in C: The Art of Scientific Programming.* Cambridge Univ. Press, 1988.

[17] R. Sedgewick. *Algorithms in C.* Addison Wesley, Reading, Mass., 1990.

[18] J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[19] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[20] P. Tinker and M. Katz. Parallel execution of sequential Scheme with ParaTran. In *Proceedings of 1988 ACM Conference on Lisp and Functional Programming*, pages 28–39, July 1988.

[21] D. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Apr. 1991.