Speculative Computation in Multilisp

by

Randy Brent Osborne

MIT/LCS/TR-464

November 1989

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014-83-K-0125 and N00014-84-K-0099.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): MIT/LCS/TR-464	5 MONITORING ORGANIZATION REPORT NUMBER(S): N00014-83-K-0125, N00014-84-K-0099
6a NAME OF PERFORMING ORGANIZATION: MIT Lab for Computer Science; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION: Office of Naval Research/Dept. of Navy
6c ADDRESS (City, State, and ZIP Code): 545 Technology Square, Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code): Information Systems Program, Arlington, VA 22217
8a NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA/DOD; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd., Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification): Speculative Computation in Multilisp

12 PERSONAL AUTHOR(S): Osborne, Randy B.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM _____ TO _____	1989 December	263

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	speculative computation, optimistic computation, sponsors, parallel Lisp, Multilisp, scheduling

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis demonstrates by experiments that performing computations in parallel before their results are known to be required can yield performance improvements over conventional approaches to parallel computing. We call such eager computation of expressions *speculative* computation, as opposed to conventional *mandatory* computation that is used in almost all contemporary parallel programming languages and systems. The two major requirements for speculative computation are 1) a means to control computation to favor the most promising computations, and 2) a means to abort computation and reclaim computation resources.

We investigate these requirements in the parallel symbolic language Multilisp, and conclude that we need the following support for speculative computation: for controlling computation we need ordering (ranking of computations by their promise), demand transitivity, (over)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Judy Little	22b TELEPHONE (Include Area Code): (617) 253-5894; 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1986-607-047

Unclassified

19. cont.

and modularity, and for reclaiming computation we need explicit, reversible reclamation with automatic naming of descendants.

The main contribution of this work is a *sponsor model* which provides this support for speculative computation in Multilisp. A *sponsor* is an agent which controls the allocation of resources to computation. This sponsor model handles control and reclamation of computation in a single, elegant framework.

We describe an implementation of this sponsor model and present performance results for several applications of speculative computation. The results 1) demonstrate the importance of aborting useless computation, 2) demonstrate the importance of controlling computation, and 3) provide experimental evidence of the benefit and power of our sponsor model and support for speculative computation. Our support for speculative computation adds expressive power to Multilisp, with the ability to control computation, and also adds computational power, with significant performance improvements — we observed as much as 26-fold speedup.

We also discuss the optimal scheduling of speculative computation and present some new results for optimal scheduling in some simple cases.

Speculative Computation in Multilisp

by

Randy Brent Osborne

Submitted to the Department of Electrical Engineering and Computer Science
on November 6, 1989, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We demonstrate by experiments that performing computations in parallel before their results are known to be required can yield performance improvements over conventional approaches to parallel computing. We call such eager computation of expressions *speculative* computation, as opposed to conventional *mandatory* computation that is used in almost all contemporary parallel programming languages and systems. The two major requirements for speculative computation are: 1) a means to control computation to favor the most promising computations and 2) a means to abort computation and reclaim computation resources.

We investigate these requirements in the parallel symbolic language Multilisp. We conclude that we need the following support for speculative computation: for controlling computation we need ordering (ranking of computations by their promise), demand transitivity, and modularity, and for reclaiming computation we need explicit, reversible reclamation with automatic naming of descendants.

The main contribution of this work is a *sponsor model* which provides this support for speculative computation in Multilisp. A *sponsor* is an agent which controls the allocation of resources to computation. This sponsor model handles control and reclamation of computation in a single, elegant framework.

We describe an implementation of this sponsor model and present performance results for several applications of speculative computation. The results 1) demonstrate the importance of aborting useless computation, 2) demonstrate the importance of controlling computation, and 3) provide experimental evidence of the benefit and power of our sponsor model and support for speculative computation. Our support for speculative computation adds expressive power to Multilisp, with the ability to control computation, and also adds computational power, with significant performance improvements — we observed as much as 26-fold speedup.

We also discuss the optimal scheduling of speculative computation and present some new results for optimal scheduling in some simple cases.

Keywords: speculative computation, optimistic computation, sponsors, parallel Lisp, Multilisp, scheduling

Thesis Supervisor: Robert H. Halstead, Jr.
Title: Research Affiliate, Laboratory for Computer Science

Acknowledgments

I would like to thank the following people:

Bert Halstead — for being such a great thesis supervisor and terrific person. Throughout this thesis research, Bert served as a sounding board for my ideas, contributed wise suggestions and insightful comments, and provided encouragement. His guidance and suggestions have contributed greatly to this thesis.

Dave Gifford and Rishiyur Nikhil — for their contributions, as thesis readers, to this final document.

Laura Bagnall Linden — for ParVis, a program visualization tool which I used very heavily during the experimental part of this thesis work. Laura's ParVis system provided a means to see what was happening (and what was not happening) with the scheduling of speculative tasks and thus I did not have to rely on inaccurate inferences. This information was a tremendous assistance in understanding the reasons for an application's performance and improving its performance.

Hitoshi Takagi — for his patience in helping me learn a little Japanese and his company as my officemate during the year of the most substantial work on my thesis.

Juan Loaiza — for many informative discussions while we shared an office and for his help in understanding the original Multilisp implementation.

Ingmar Vuong — for interesting discussions about scheduling theory.

Dan Nussbaum — for helping to keep the Concert Multiprocessor and its software working (as long as it did). His efforts are greatly appreciated.

My wife Kathrin — for her patience, support, and love during my years as a student at M.I.T.

To my daughter Charlotte Anne

Prologue: A need for speculation

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth

Then took the other, as just as fair,
And having perhaps the better claim

from *The Road Not Taken* by Robert Frost

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The future construct of Multilisp has proven very versatile and successful for achieving parallelism in symbolic computation [Hals86d]. However, experience with parallel symbolic computation has led to the recognition that "speculative" styles of computation may be more effective for certain applications, notably searches, than the "mandatory" style of computation obtained using future [Hals85,Hals86c,Hals86d]. Speculative computation is eager evaluation where the result(s) of the evaluation may be unnecessary. Speculative computation involves a gamble whereby one trades additional, possibly unnecessary, computation for potentially faster execution.

Speculative computation has two requirements. First, because in general computation resources are limited, we would like to reclaim the resources — processor cycles, memory cells, and machine resources — devoted to unnecessary computation. Thus speculative computation requires the ability to "abort"[1] computation and reclaim computation resources.

The second requirement follows naturally from the first, assuming that computation resources are limited: given that some of the computation may be unnecessary, we would like to arrange the use of resources to favor the most promising computations. Thus speculative computation requires the ability to control computation resources.

We investigate these requirements in this thesis and present a model for speculative computation in Multilisp. This model should furnish an archetype for speculative computation in other parallel languages. We describe an implementation of this model and present performance results for several applications of speculative computation.

The remainder of this introduction is divided into sections which give a detailed definition of speculative computation and related terms (Section 1.1), present some examples of speculative computation (1.2), discuss the potential of speculative computation (Section 1.3), introduce Multilisp (Section 1.4), describe our goals and approach in (Section 1.5), and finally present an overview of the remainder of the thesis.

[1] As we discuss later, we do not necessarily have to kill unnecessary computation.

1.1 Definitions

1.1.1 Speculative Computation

Given some definition of computation appropriate to the computational paradigm at hand, we classify computation into three groups:

1. computation known to be required,
2. computation known not to be required, and
3. computation not known to be required or not required.

This classification is with respect to a given state of knowledge at a given time and a given specification of the program in which the computation is embedded. The state incorporates perfect knowledge about the past and whatever information may be known about the future, such as input data and analysis of the program. The knowledge about the future may be less than the total available due to incomplete analysis. (Of course, even the total knowledge available is necessarily incomplete due to external nondeterminism (future inputs may be unknown), internal nondeterminism in the program, and theoretical limitations, such as decidability, on program analysis.) The program specification is a set of conditions on the results and side-effects produced by the program (and perhaps also conditions on the inputs and the execution environment) such that the program is considered correct if every execution of the program (subject to the input and environment conditions) is guaranteed to meet these conditions. We use the program specification to define what it means for a computation to be required or not required. A computation is required (i.e. in group 1) if, with the given state of knowledge, the computation is definitely always necessary to meet the program specification. Likewise, a computation is not required (i.e. in group 2) if, with the given state of knowledge, the computation is definitely not ever necessary to meet the program specification. Finally, a computation is in group 3 if, with a given state of knowledge, the computation could either be necessary or unnecessary. In this case, the given state contains insufficient information to determine if the computation is required or not required.

As a program executes, the state of knowledge increases. Thus during program execution the above classification constitutes a succession of improving approximations with time, starting with some *a priori* designations and becoming further refined, as time goes on and more information becomes available, and ending with all computation divided into groups 1 and 2[2], assuming the computation terminates.[3]

[2]This is not always true: even after all computation terminates, some (past) computation may remain in group 3 because it still may not be clear if the computation was necessary or unnecessary. If the computation is not performed the program may execute in such a way that it could either meet the specification or not depending on the nondeterminism realized.

[3]However, it may only be necessary for some of the computation in a program to terminate in order to classify all the computation in the program.

Computation in group 1, i.e. computation that is known to be required, is relevant computation. Computation in group 2, i.e. computation that is known not to be required, is irrelevant computation. Computation in group 3 is speculative computation. We also refer to relevant computation as mandatory computation. Thus a speculative computation may become mandatory during its course of computation, as more information becomes available to indicate that it is necessary. For instance, a mandatory computation may select a particular speculative computation from among several; that computation becomes necessary and the rest become unnecessary. Or, further input data or control information obtained with time may indicate that a speculative computation is actually necessary.

Lazy computation is computation that is started when the computation is actually required, that is, at the latest possible time. By contrast, eager computation is computation that is started early, before it is required, but with certainty that it will be required.[4] Speculative computation is computation that is started before it is required, like eager computation, but without any assurance that it will be required later, unlike eager computation.

If all computation is functional, the classification of computation is simplified. In this case, a demand-driven interpreter evaluating a program defines the specification of that program. Thus, whether or not a computation is required reduces to whether or not a demand-driven interpreter ever (irrevocably) demands the computation. Also, the set of computations demanded by a demand-driven interpreter up to time t constitute the minimal state of knowledge at time t.

Consider first a deterministic and purely functional program P. In this case a computation C in P is relevant if a demand-driven interpreter evaluating P either will demand C (based on the state of knowledge) or does demand C. Computation C is irrelevant if a demand-driven interpreter will never demand C. Finally, computation C is speculative if we cannot determine, based on the given state of knowledge, whether or not a demand-driven interpreter will ever demand C.

In the case of a nondeterministic and functional program P we have to be more careful since demanding the operand of a nondeterministic operator may not imply that the operand is required. To handle nondeterministic operators like parallel or (described in Section 1.2.1), we introduce the notion of a choice time. First we define a choice computation to be any computation which a given nondeterministic operator may choose. The choice time is then the time at which the nondeterministic operator chooses amongst the choice computations. Prior to the choice time, the demand-driven interpreter provisionally demands all choice computations. (With a demand-driven interpreter, the choice time always occurs after the interpreter demands the result of the nondeterministic operator.) To avoid problems with non-termination, we assume a *fair* demand-driven interpreter that provisionally demands all choice computations equally.[5] At the choice time, this fair demand-driven interpreter irrevocably demands the chosen computation and "undemands" all the non-chosen computations. The chosen computation is, of course, required and the non-chosen

[4] This is not the universal meaning of eager computation — some people use it to mean speculative computation.

[5] Thus the classification of computation is with respect to this fair demand-driven interpreter.

computations, provisionally demanded and then undemanded, are not required.

Thus for a nondeterministic and functional program P we amend our definitions as follows. Computation C in P is relevant if a fair demand-driven interpreter evaluating P either will or does demand C irrevocably (i.e. never undemands C). Computation C is irrelevant if a fair demand-driven interpreter will never demand C irrevocably. Finally, computation C is speculative if we cannot determine, based on the given state of knowledge, whether or not a demand-driven interpreter will ever demand C irrevocably. These definitions reduce to the previous definitions for the deterministic case with the following addition for nondeterministic operators: Prior to the choice time of a nondeterministic operator all the choice computations are speculative and subsequent to the choice time all the non-chosen computations are irrelevant. If the chosen computation is part of another computation, the chosen computation's relevance is determined in the same way. Otherwise, its relevance depends, as in the deterministic case, on whether it is demanded by a demand-driven interpreter.

To help illustrate these definitions, consider the following example. Assume that we have choice(C_1, C_2, C_3), where choice is a nondeterministic operator. Then C_1, C_2, and C_3 are the choice computations. Let R be the current relevance (relevant, irrelevant, or speculative) classification of the choice expression. Then, prior to the choice time, at which time the choice operator selects one of the C_i (we assume only one is selected), the demand-driven interpreter provisionally demands all the C_i and all the C_i are speculative, unless R is irrelevant, in which case all the C_i are irrelevant too. When the choice operator selects, say, C_j, the demand-driven interpreter "undemands" all C_i for $i \neq j$ and irrevocably demands C_j if necessary (i.e. if C_j is not fully computed). Then all C_i for $i \neq j$ are irrelevant, regardless of R, and C_j has relevance R. Or put another way, the selected computation C_j is *conditionally relevant* with respect to the choice expression.

With side-effects, a demand-driven interpreter is not sufficient to define relevance. Computation not explicitly demanded by such an interpreter may be required, and hence relevant, for the side-effects that it may perform, such as writing a shared variable, or releasing a lock or semaphore. Thus with side-effects we must use the general classification given earlier. However, side-effects raise some tricky issues with this classification (such as how do we determine if a given computation with side-effects is necessary to meet the program specification). We will not pursue these issues here since their resolution does not add substantially to the understanding of speculative computation. This does not mean that we avoid side-effects in the sequel: the principles of speculative computation may certainly be exploited without resolving these issues. Furthermore, there are many useful applications involving side-effects for which these issues do not arise.

We have given a temporal classification of computation. Sometimes we want to classify computation after a program terminates, in a *post mortem* fashion. In such cases, we are interested in questions such as was computation C speculative or not? This is an ill-formed question, though, since our classification of C depends on our viewpoint, which may change with time and program history. Initially C may appear to be speculative but later information may reveal it to be relevant or irrelevant. Thus to answer this within our framework, we need to specify a state, since the relevance of a computation is with respect to a given state.

Note that with our definitions, speculative computation may exist in conventional programs and even in sequential programs.

Classifications of Speculative Computation

There are two orthogonal ways of classifying speculative computation. The first classification is based on determinacy. Speculative computation is:

1. deterministic if the semantics of its application is deterministic, i.e. if the result is a function of data and control dependences only, and
2. nondeterministic if the semantics of its application is nondeterministic, i.e. if the result is a function of scheduling behavior in addition to the data and control dependences. An example of nondeterministic speculative computation is the concurrent application of several solution strategies where we are only interested in the first strategy to succeed. We call such a race amongst alternatives in which any alternative will suffice "first-of" speculative computation.

The second classification is based on the way in which resources are used for speculation. We distinguish three flavors:

1. multiple-approach speculative computation

   In this flavor, the speculation is in pursuing multiple approaches simultaneously, as in first-of speculative computation, where not all the approaches are necessary (but at least one is). A dominant characteristic of this flavor is aborting irrelevant computation, i.e. aborting unnecessary approaches. Multiple-approach speculative computation is, perhaps, the most obvious form of speculative computation.

2. order-based speculative computation

   In this flavor, the speculation is in the order in which the computations are performed. Not all the computations are necessary so this order is important. There may be an optimal order but this may not be known *a priori*. Thus the goal is to use an order which has good average behavior (as in minimum average completion time). Order-based speculative computation is invariably resource constrained: the order matters because there are insufficient resources to perform all computations simultaneously. Order-based speculative computation is not restricted to multiprocessors. For example, the execution of a branch and bound algorithm on a sequential computer is an example of order-based speculative computation (cf. the example in Section 8.5). Lastly, order-based speculative computation typically does not involve any aborting of computation.

3. precomputing speculative computation

   In this flavor, the speculation is in precomputing some quantity for possible future use. Unlike in multiple-approach speculative computation, where at least one approach is

necessary, precomputation is not necessarily required. Also, unlike multiple-approach speculative computation, precomputing speculative computation computation often does not involve any aborting of computation (because the precomputation terminates before it is known if the result is required). Precomputing speculative computation is already familiar in sequential computing as caching.

In practice most speculative computation involves a mix of these three flavors, especially multiple-approach and order-based speculative computation. Indeed, ordering becomes important as soon as there are insufficient resources.

1.1.2 Optimistic Computation

Optimistic computation can mean the same thing as speculative computation, though perhaps with a connotation suggesting more "success" (i.e. a higher ratio of necessary to unnecessary computation) than with speculative computation. However, most people (e.g. in concurrency control [Herlihy,Kung], in simulation [Jeffer], in fault tolerance [Strom,Johnson], and in software maintenance [Bubenik]) use optimistic computation to mean a subset of deterministic speculative computation in which there is a particular concern to undo all side-effects ever performed by an aborted computation, so it appears that the computation never occurred. We call this "atomic (or indivisible) semantics" — all or none of the side-effects persist. That is, side-effects obey a transaction or encapsulation [Bubenik] model.

Our notion of speculative computation is broader than the usual notion of optimistic computation. Our notion covers nondeterminism and certain styles of data precomputation (e.g. speculative streams which we mention in Section 1.2.5) which do not fit into the all-or-nothing model. The key distinction of our approach is the inclusion of both data and control dependences to determine if a computation is necessary, rather than just control dependences (as in [Bubenik]). We feel this leads to a finer-grained approach to speculative computation.

1.2 Examples

We present five quite different examples of speculative computation which we use to motivate discussion in the rest of the thesis.

1.2.1 Parallel search

Search is a particularly rich domain for speculative computation. Consider the problem of searching the subspaces $S_1, S_2, \ldots, S_n$ for a target. To perform this search both quickly and efficiently, we want to perform the subspace searches concurrently (subject to availability of machine resources) and terminate all remaining subspace searches when we find the target,

to avoid wasting machine resources. That is, we want both parallelism and control over the parallelism. These subspace searches are examples of speculative computation.

Without language support for speculative computation, as in conventional Multilisp, we must either abandon the goal of terminating useless subsearches or we must have each subsearch explicitly check for termination. The former is inefficient and the latter can be awkward and suffer from lack of expressiveness (as we discuss later). Thus the benefits of support for speculative computation are efficiency and ease of performing parallel search.

These benefits are important because of the importance of parallel search in symbolic computation. The A.I. domain, for example, with its heavy emphasis on search techniques, seems particularly attractive for parallel search with speculative computation.

Probably the simplest examples of parallel search are parallel or and and, which we call por and pand respectively. (por E_1 E_2 ... E_n) returns the value of the E_i that first evaluates to a non-nil value and nil if all the E_i evaluate to nil. In contrast, (pand E_1 E_2 ... E_n) returns nil when any E_i evaluates to nil and true if all the E_i evaluate to non-nil. (We define por and pand more precisely in Appendix B.) In both cases, any remaining E_i evaluations may be aborted after a result is returned. These two nondeterministic operators represent perhaps the most important potential application of speculative computation because of the ubiquity of or and and.

Parallel search, in general, is an example of multiple-approach speculative computation and por and pand, in particular, are examples of nondeterministic speculative computation.

1.2.2 Branch prediction: parallel `if`

A good example of branch prediction is parallel `if`. Suppose *pred* in the expression

(`if` *pred consequent alternate*)

takes a long time to evaluate. Then we might like to evaluate *pred*, *consequent*, and *alternate* concurrently to reduce the total execution time. If *pred* evaluates to true, we accept the result of evaluating *consequent* and abort the evaluation of *alternate* (if it is still in progress). If *pred* evaluates to false, we accept *alternate* and abort *consequent*.[6] Precomputing *consequent* and *alternate* is an example of deterministic, multiple-approach speculative computation.

This branch prediction example demonstrates another benefit of speculative styles of computation: the relaxation of synchronization constraints to reduce the critical path length. By relaxing the synchronization constraints, we mean relaxing the constraints on when computation is actually performed, while still obeying the overall data and control constraints. As with parallel search, the objective is to reduce the critical path length *efficiently*.

[6] And if *consequent* and *alternate* evaluate to the same value, we could abort *pred*.

1.2.3 Producer-consumer parallelism: the Boyer Benchmark

Written as a parallel Multilisp program, the Boyer Benchmark [Gabr85] is an example of producer-consumer parallelism with an interesting twist. Given an input expression, the Boyer Benchmark determines whether the expression is a tautology based on a database of rewrite rules. The producer successively rewrites the input expression according to the rewrite rules to obtain an if-then-else tree. The consumer, which operates concurrently with the producer, traverses this if-then-else tree, checking for consistency between each predicate and its consequent and alternate. The interesting twist is that not all the rewrites are necessarily required by the tautology checker. For example, the rewrite rule for and is (and a b) → (if a (if b #t #f) #f). If a happens to be #f, there is no need to rewrite expression b. However, the producer does not know this until the consumer terminates without demanding this rewrite. (a may not be so simple, or a may be shared by some other expression.)

In an attempt to reduce the execution time we could perform all rewrites eagerly, gambling (in a form of branch prediction) that they will be required. However, this application of speculative computation fails if there are too few processor resources (see Section 8.4) because the machine becomes saturated with speculative rewrites and swamps out the tautology checker computation. That is, unnecessary rewrites use resources that otherwise would be devoted to necessary rewrites and tautology checking and thus lengthen the execution time. To counteract this problem, we could perform all rewrites lazily but then the execution time is long due to insufficient parallelism.

To solve this problem, we need a way to order the allocation of resources to speculative activities according to their relative promise. For Boyer two ordering levels are sufficient: one for the tautology checker (i.e. consumer) and one for the rewrites (i.e producer) whereby the consumer can preempt the producer for processor resources. However, we also need some way to promote a rewrite to the consumer level when we find it necessary. That is, we need what we call *dynamic* ordering. With such static and dynamic ordering, the Boyer Benchmark is an example of order-based speculative computation.

(There is also the issue of aborting all the useless rewrites when the tautology checker terminates, but aborting has already received adequate mention.)

1.2.4 Ordering: the traveling salesman problem

In the previous example we saw that ordering of resources for speculative activities can be very important because the activities have different promises of being required and not all of the activities are necessarily required. This example takes ordering to its extreme.

Consider a branch and bound algorithm to solve the traveling salesman problem. We would like to expand nodes representing partial tours in parallel according to some heuristic so we can focus our machine resources on the most promising partial tours first. The faster we can obtain a complete tour and the better the quality of this tour, the more pruning we can do, thus decreasing the total execution time. One way to achieve our desired ordering is

via an explicit priority queue (programmed in the language). This is an awkward solution. A better solution is to extend the notion of ordering — which we already argued was necessary for cases like the previous example — to an arbitrary infinitum of orderings. In both the Boyer Benchmark and the traveling salesman problem the fundamental problem is the same: ordering the allocation of resources to activities according to their relative promise. Thus we should use the same mechanism to solve this problem in both cases.

The speculation in the traveling salesman problem is in the *order* in which resources are allocated to node expansion, i.e. it is order-based speculative computation. This notion of ordering is the key idea missing in most other approaches to speculative computation (as discussed in Chapter 3). Aborting exists in the traveling salesman problem, but it is implicit aborting: a node checks the cost of the node with respect to the current best cost of a complete tour and simply terminates if the cost exceeds the best cost.

1.2.5 Precomputing streams

A stream is a possibly infinite list of objects [Abelson]. The illusion of infinity is maintained by generating the list lazily: elements are added to the tail of the list incrementally as demanded (thus advancing the tail). The idea in precomputing a stream is to extend the tail of the stream, by computing several elements ahead, before these elements are actually demanded. This idea is a form of branch prediction — we hope to reduce the critical path by doing a certain amount of precomputing. We call a stream with such element precomputing a *speculative* stream to underscore that it is an example of (precomputing-based) speculative computation.

The easiest way to control a speculative stream is to specify the number of elements to precompute. This control is inadequate in general, though, since the amount of computation required per element could be non-uniform — it could increase exponentially for instance. To ensure adequate control over the effort we devote to precomputing stream elements, we need to be able to control the duration for which we precompute stream elements.

1.2.6 Other examples

We list some other applications of speculative computation below.

- Pattern matching — Parallel search could be used to match patterns.
- Rule-based interpreters — A rule-based interpreter could be realized as a parallel search on a database of rules. [Miller] explored such an example.
- Symbolic integration — Alternate methods for integration like a fast heuristic routine for common special cases and the Risch algorithm, a general but slow procedure, could be tried simultaneously. Symbolic algebra in general seems like a rich application area for speculative computation [Watt].

- Hypothesize and test — This popular A.I. paradigm could be parallelized to generate and test many hypotheses simultaneously and terminate upon finding the first successful hypothesis.

- Alpha-beta pruning — Many positions could be explored simultaneously while maintaining the sequential flavor that is so important to minimize the total amount of work.

- Simulated annealing — Several candidates for the next state could be investigated simultaneously, thus reducing the critical path if the chosen candidate is rejected. An alternative idea proposed by [Chamber] is to precompute all possible accept/reject branches simultaneously (to a given depth). This amounts to "pipelining" the sequential simulated annealing algorithm by massive branch prediction.[7]

- Speculative convergence — This is a form of branch prediction suggested by [Soley] in which loop iterates greater than i proceed concurrently with testing loop iterate i for termination. This speculation avoids undue serialization of the iterations with the termination test.

1.3 Potential of Speculative Computation

The central idea behind speculative styles of computation is to use excess resources to reduce the average execution time. As the examples in the previous section illustrate, the key to realizing this idea is to relax synchronization constraints, thereby decreasing the critical path length, by using excess resources and controlling the order of computation. In decreasing the critical path, it is important to use these excess resources efficiently. Inefficient use of excess resources may preclude future opportunities for speculative computation.

Speculative computation offers the potential to turn excess resources into possibly faster execution. In other words, speculative computation can lead to the more efficient use of machine resources. With speculative computation we are trying to exploit the narrow ground between too little mandatory computation and too much mandatory computation. We are trying to fill this gap with speculative computation.

Efficient support for speculative computation will encourage a more aggressive exploitation of parallelism. Such support will make it possible to extract parallelism from problems thought to be either too expensive to parallelize or inherently sequential. Some problems are too expensive to parallelize highly in the conventional framework because of poor control over resource use. For example, searches may be inefficient when highly parallelized because it may be very difficult or impossible to kill unnecessary subsearches and allocate resources to subsearches in relation to their promise. Other problems, such as simulated annealing, are inherently sequential and are difficult to parallelize effectively in a conventional

[7]The Multiflow Trace used this branch prediction form of speculative computation successfully at the architecture level.

framework. However, speculative styles of computation allow one to follow the sequential flavor — but not all the sequential constraints — of such applications while still exploiting parallelism to speed execution.

1.4 Multilisp

Multilisp is a version of the Scheme programming language extended with explicit parallelism constructs. Multilisp is based on a shared memory paradigm and includes side-effects — hence the explicit parallelism constructs [Hals85]. This section summarizes the major differences of Multilisp from Scheme. Further information on Multilisp is available in [Hals85] and [Hals86b] and several papers [Hals86c,Hals87,Hals86d] describe example applications.

1.4.1 Parallelism constructs

The principal parallelism construct is `future`.

(`future` *exp*) creates a new thread of computation to evaluate *exp* and immediately returns a placeholder for the result. This placeholder, or future object, may be manipulated just as if it were the result of evaluating *exp* — that is consed into data structures, passed to and from functions, etc. — unless or until it is an argument to a strict operation, such as plus, which requires the value of each operand. When this happens, the thread attempting the strict operation suspends until the placeholder is *determined* with the result of evaluating *exp*.

We call a thread of computation a *task*. Tasks are conceptual entities below the language level. When a task attempts a strict operation on a placeholder, we say that the task *touches* the placeholder. If the placeholder is undetermined, the task blocks on the placeholder, as described above. Otherwise, the value is automatically (and implicitly) extracted from the placeholder. All strict operations in Multilisp implicitly touch all their placeholder operands to ensure that each such operand is determined. Thus join synchronization occurs implicitly when the placeholder value is actually required. Multilisp also provides a construct to explicitly touch its operand and effect join synchronization.

(`touch` *exp*) is an identity function which is strict in its argument. If *exp* is a placeholder `touch` touches the placeholder and returns the placeholder value (blocking if necessary). Otherwise, `touch` simply returns *exp*.

Figure 1.1 displays an example fragment of Multilisp code. The `future` in line 1 creates a new task to evaluate (`foo 2`) and returns a placeholder for the eventual result of this application, which is bound to `x`. Line 2 performs a non-strict operation on the placeholder, consing a pair containing the placeholder and returning this pair. Lines 1 and 2 demonstrate the power of Multilisp. The producer in line 1 and the rest of the computation, including eventual consumers, can proceed concurrently. When a consumer actually requires the producer's value represented by the placeholder, the necessary producer-consumer synchro-

```
(let ((x (future (foo 2))))  ;1
   (if y
       (cons x 1)            ;2
       (+ x 1)))             ;3
```

Figure 1.1: An example Multilisp fragment

nization occurs implicitly as a part of the strict operation that actually requires the value. In line 3 the consumer is the plus. Plus is a strict operation which requires the actual value represented by the placeholder bound to x. Thus the task performing the plus operation implicitly touches x and suspends until x is determined when the evaluation of (foo 2) completes.

Whereas future is an eager construct that begins evaluation of its argument any time after the task is created, delay is a lazy construct.

(delay *exp*) creates a new task to evaluate *exp* and immediately returns a placeholder for the result, like future. However, unlike future, delay does not begin evaluating *exp*, i.e. executing the task, until the placeholder is touched.

Sometimes it is convenient to have a placeholder without an associated task for write-once synchronization, like the I-structures in the dataflow language Id [Nikhil].

(make-future) creates and returns an empty placeholder, i.e. future object.

(determine-future *fut exp*) explicitly determines the undetermined future object *fut* to *exp* and returns *exp*. Each task created by a future or delay ends with an implicit determine-future. It is permissible, though not encouraged, to explicitly determine any future object — even one with an associated task — with determine-future. It is an error to (explicitly or implicitly) determine a future object more than once.

1.4.2 Scheduling

The task executing a future expression, such as (future *exp*), is called the *parent* task and the task created to evaluate the argument expression *exp* is called the *child* task. A processor which executes a future always pursues the child task after creating the child task and future object, rather than the parent task. This is the unfair scheduling for resource management discussed in [Hals85]. The construct dfuture is exactly like future except for opposite scheduling of parent and child tasks: a processor which executes a dfuture always continues the parent task after creating the child task and future object.

1.4.3 Side-effects

Multilisp includes the usual collection of side-effect operations found in Scheme. To enforce task synchronization (for correct operation in the presence of side-effects) Multilisp provides atomic operations and semaphores.

The following two atomic operations are extensions of the `set-cxr!`, (x= a or d) mutators in Scheme.

(`rplaca-eq` *pair new old*) performs the following eq check and possible swap atomically: If the car of *pair* is eq to *old*, the car of *pair* is replaced by *new* and *pair* is returned. If the car of *pair* is not eq to *old*, nil is returned.

(`rplacd-eq` *cell new old*) performs the same eq check and possible swap as above atomically, but based on the cdr of *pair* rather than the car.

(`make-sema`) makes and returns a free binary semaphore object.

(`wait-sema` *sema*) makes the semaphore *sema* busy if *sema* was free. Otherwise, it suspends and enqueues the executing task on the semaphore *sema*. `wait-sema` is Dijkstra's classical P operation.

(`signal-sema` *sema*) makes the semaphore *sema* free if no tasks are queued on the semaphore. Otherwise, it dequeues and resumes one of the tasks enqueued on *sema*. `signal-sema` is Dijkstra's classical V operation. The system tries to enforce first-come-first-served (FCFS) ordering on semaphore requests, but this ordering is not guaranteed.

1.4.4 Task touching

Touching, as defined above, only occurs between tasks and placeholders, or future objects: a task touches a future object. To ease the presentation in the rest of this thesis, we generalize this notion of touching when the touchee future has an associated task. If a task A touches task B's future object (i.e. the future object which task B determines with its result), we say task A touches task B.

Figure 1.2 provides an example in the context of Figure 1.1. The circles represent tasks and the rectangle represents the future object bound to x. Task B is evaluating (`foo 2`) and will determine future object x with the result. Task A, meanwhile, is attempting to perform the plus operation in line 3 of Figure 1.1. Task A touches future object x as signified by the arrow from task A to the placeholder in Figure 1.2. Generalizing, we say that task A touches task B.

1.4.5 Task terminology

For convenience in describing the implementation, we categorize tasks operationally as mandatory or speculative. For now we loosely define mandatory and speculative tasks

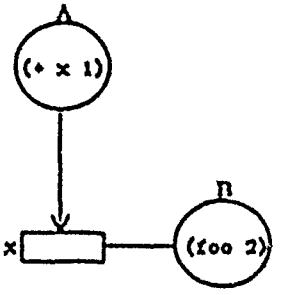


Figure 1.2: Task touching.

as tasks performing mandatory and speculative computation respectively. We will give a precise meaning to these terms in Chapter 6 when we describe the model we implemented.

1.5 Goals

Our goals in this thesis work were as follows:

1. Investigate the requirements of speculative computation in Multilisp applications.
2. Provide a model for speculative computation in Multilisp.
3. Provide support for speculative computation in Multilisp.
   (a) Avoid incompatible changes to the existing Multilisp language definition, and in particular, retain `future`. Since `future` has endured an important test of time at the hands of many programmers [Hals86d], we believe it is a sound base.
   (b) Minimize the impact of support for speculative computation on the performance of traditional, mandatory computation.
4. Show examples illustrating the support for speculative computation and demonstrating the power gained by that support.

To achieve goal 3 we provide special support for speculative computation in Multilisp in the language and in the implementation. This support expands the notion of "touching" `future`s. This support is necessary for sufficient expressiveness and efficiency. For example, as we already mentioned in 1.2.1, we need support for terminating speculative computation to avoid the awkwardness and lack of expressiveness of explicit checking. This topic is a major subject of Chapter 2.

In previous work we constructed a scheduler on top of Multilisp for experimenting with speculative computation. This layered approach allowed more general control of computation than explicit checking, but was too inefficient. Since Multilisp provides no way to control the execution of an arbitrary task, we were forced to install "checkpoint" calls

which interacted with a user level scheduler at regular intervals in each task. Performing the system scheduling in this manner proved to be prohibitively expensive. Furthermore, this layered approach was still not sufficiently general: we were not able to model blocking on undetermined future objects. Such blocking occurs implicitly in Multilisp and is not accessible from the user level. This meant that any desired computation control could be frustrated by unobservable task touching. Hence we need special support.

1.6 Preview

Chapter 2 discusses the four main problems with speculative computation in an imperative environment. Chapter 3 summarizes related work. Chapter 4 describes our approach to the four main problems. Chapter 5 introduces our sponsor model for speculative computation and Chapter 6 describes a particular subset of this model that we implemented, called the "touching model". Chapter 7 considers solutions to the problems posed by side-effects in the context of our touching model. Chapter 8 discusses in detail several applications of speculative computation with performance results. Chapter 9 considers the optimal scheduling problem for some simple cases. Chapter 10 describes implementation details, such as the mechanism for aborting useless computation. Finally, Chapter 11 presents our conclusions and thoughts for future research.

Chapter 2

Main problems

In this chapter we discuss the four main problems involved in supporting speculative computation in Multilisp. We assume that computations potentially compete for some limited computation resources. (If this is not the case, all computation can be treated as mandatory and speculative computation is moot.) Because these resources are limited, it is important to use them efficiently. Therefore, in general we must control speculative computation and reclaim irrelevant computation.

2.1 Controlling Computation

Because not all speculative computation is necessary, the order in which we allocate resources to computation is important, as the Boyer Benchmark demonstrates. Furthermore, not all speculative computation is equal. Some computations may be more promising than others, as the traveling salesman example demonstrates. Thus we want to allocate resources to computation to favor the most promising computations. For instance, we certainly want mandatory computation to preempt the resources utilized by speculative computation. More generally, we want some sort of ordering of computation.

For concreteness in this chapter, we assume priorities for specifying this ordering. The reader should view priorities in this chapter as a representative means of ordering. In particular, any mention of priorities in this chapter applies as well to any other means of specifying ordering.

Issue 1: Determining the relative promise, i.e. priority, of computations

In general, the programmer[1] must decide the relative promise of computations, as controlling computation relies on meta-knowledge about a program's function, inputs, and purpose as

[1] Or some programmer, such as the systems programmer or the library programmer.

well as its operating environment. The optimal assignment of priorities is a scheduling problem which we mention in connection with Issue 4.

Issue 2: Interaction of computation

The promise of computations, i.e. priorities, must be transitive. Without this transitivity, interaction of computation may subvert the desired ordering. For instance, suppose that computation C_1 demands the result of (uncompleted) computation C_2. If C_1's priority is greater than C_2's, then C_1's progress is effectively that of the lower-priority C_2, subverting the desired ordering of C_1 with respect to other computations.

Transitivity (of priorities) provides the dynamic ordering of resources to computation that is so essential in the Boyer Benchmark.

Issue 3: Modularity

We need a way to preserve the functionality of a group of related speculative computations, such as in the Boyer Benchmark and the traveling salesman problem, wherever the group appears. We call this the *modularity principle*: we should be able to embed any speculative computation or group of speculative computations as subcomputation(s) within some larger speculative computation. For example, we should be able to have the traveling salesman problem as a disjunct in por while retaining both the desired ordering of the disjuncts with respect to each other and the desired local ordering of the traveling salesman computations with respect to each other.

To support this modularity, we want each "module" to have a new, local priority space relative to the parent priority. This allows modules to be nested in hierarchical fashion.

This modularity is important. It allows arbitrary nesting of speculative computation with the assurance that local ordering relationships will be retained. This increases the ease of programming and expressiveness. If all priorities occupy the same flat priority space the lack of modularity can be quite troublesome. For example, if the traveling salesman problem is a disjunct in a por, all the priorities in the traveling salesman problem must be carefully adjusted so as not to interfere with the other disjuncts.

Issue 4: More complex control

Sometimes we need complex, dynamic control of computation. Consider por. We may start with some *a priori* idea of the relative priority for the disjuncts but this assignment may change as information is garnered from the disjuncts and some disjuncts complete. Furthermore, the result of the por may be demanded by some other computation. Should this demand be propagated to all disjuncts or just the most promising ones or propagated according to some other policy? The answer is affected by the computational requirements of the disjuncts, other computations in the system competing for the resources, and the

urgency of the demand. This is the sort of scheduling problem that we must analyze to ultimately determine the control we want.

Issue 5: Other types of control

Priorities provide only one dimension of computation control. However there are three primary degrees of freedom in computation control:

1. the amount of resources available to speculative activities

   One problem with ordering of computation is that one activity with the same position in the ordering (i.e. the same priority) as another can effectively starve the other activity of resources. Normally, activities in the system operate at different relative priorities and starvation of resources is exactly what we desire to ensure that the most important activities can proceed. Sometimes, though, there are situations where we would like two or more activities to share the available system resources. For example, we might want all the activities to run simultaneously, each with reduced resources if necessary. This implies that each of these activities operates with the same priority. The problem then is that there is no way to ensure that the resources are allocated equally (or in whatever desired ratio) across the activities. The allocation will in fact tend to be very unfair, favoring the activity with the greatest number of tasks (assuming that all the activities' tasks operate at the same priority). Thus an activity with a large number of tasks can effectively starve another of resources.

2. the rate at which an individual computation may progress

   Sometimes we would like the ability to multiplex the set of tasks comprising some activity, or at the very least, control the rate at which a task executes. In [Korn82], Kornfeld described a problem which can be solved faster, on average, by multiplexing the tasks on the available processors rather than by running some subset of the tasks at maximum rate.[2] Kornfeld has termed parallel algorithms possessing this property *combinatorially implosive*. Such algorithms have the property that intermediate results generated by one approach (say top-down) can prune nodes in another approach (say bottom-up) and vice versa (intermediate results from bottom-up can prune top-down nodes). (See [Korn82] for a good example.) When such approaches run concurrently, the resultant synergy in generation and exchange of intermediate results causes a collapse in the total processing required.

3. the duration for which an individual computation may proceed

   Occasionally we would like to be able to control the duration for which a speculative activity executes. Such control is important for potentially infinite computations. A good example is speculative streams: we would like to control the horizon for which we precompute stream elements as discussed in Section 1.2.5.

[2]Kornfeld actually described this phenomenon in the context of a single processor, but it obviously pertains to multiprocessors as well (if for no reason other than only one processor may be available).

Independent control of the amount and rate of resource use allows various tradeoffs, both in the number of tasks pursued in a speculative activity and in the rate at which they are pursued (like multiplexing tasks). A single dimension of control, such as that given by priorities, cannot provide such tradeoffs. For example, with priorities only, we can get the starvation problem described under point (1) above. In addition, with priorities we cannot express the duration control that is essential for potentially infinite computations, such as speculative streams.

2.2 Reclaiming Computation

To use machine resources efficiently we must reclaim the resources devoted to irrelevant computation. The essential issue is the reclamation of *computation* resources so they may be recycled for other computations. We assume that:

1. processors are expensive and few in number, and
2. memory is cheap and plentiful.

(These assumptions correspond to the basic state of the art for multiprocessor implementations.) Thus by computation resources we really mean *processor* resources. Reclamation of computation state, i.e. the storage used by a computation, is an independent issue. Such reclamation is conventionally performed by a memory garbage collector.

The two important issues for computation reclamation are:

1. Reclamation speed

   If there are no excess resources, irrelevant computation must be reclaimed as quickly as possible. If there are excess resources, the speed of reclamation is not as important. Still, it is desirable to reclaim irrelevant computation under such conditions because such computation requires management overhead and affects the performance of other computation through memory traffic and interconnect contention.

2. Run-away task phenomenon

   This phenomenon occurs when an irrelevant computation spawns descendants faster than they can be reclaimed. Obviously, this phenomenon must be avoided. There must be a guarantee that all irrelevant computation will eventually be reclaimed.

We categorize the various methods for reclaiming computation according to the following criterion. Since a task is the entity that is assigned processor resources (by the scheduler) in Multilisp, we use tasks as the granularity of computation. Thus an irrelevant task in the sequel means a task whose computation is irrelevant. The following discussion remains applicable to other parallel processing paradigms by suitably interpreting "task" with respect to the given paradigm.

1. How does the method determine if a task is irrelevant?

   There are two ways to determine if a task is irrelevant: implicit and explicit detection. In implicit detection, the system deduces that a task is irrelevant. In explicit detection, the user declares a task to be irrelevant.

2. How does the method reclaim a task's computation?[3]

   There are three ways to reclaim computation, i.e. cause a task to release its processor resources:

   (a) kill the task: cause the task to permanently quit executing
   This is, of course, irreversible.

   (b) "short-circuit" the task: cause the task to abandon the remainder of the computation and immediately return a nonsense or "no-value" result
   Again, this is irreversible.

   (c) suspend the task
   This is potentially reversible.

2.2.1 Computation reclamation methods

To illustrate the spectrum of reclamation methods and to advance our discussion by example, we describe four representative methods of reclaiming computation.

Garbage collecting tasks

There is no operational test that a system can perform to determine if a task is irrelevant (short of actually performing its computation or maybe some other computation). There are two ways to get around this problem and perform implicit relevancy detection. The first way is to approximate the relevance of a task by its accessibility, which can be determined operationally. Thus a task (or more properly, the computation state of a task) is considered relevant if and only if it is accessible from the root pointer set, so its result can be accessed. This amounts to the garbage collection of tasks, an idea first suggested by [Baker78b].

Garbage-collecting tasks really amounts to extending normal garbage collection to tasks. To determine the accessible tasks we must determine the set of all accessible objects (since objects can have pointers to tasks).

There are two problems with garbage-collecting tasks, both following from the assumption that a task is relevant if and only if it is accessible:

1. A task may be irrelevant but still accessible.

[3]To reiterate, by computation reclamation we mean reclaiming the computation's processor resources. This may or may not be connected with reclaiming the computation's state, i.e. memory resources.

2. Inaccessibility implies irrelevance only if all the computation is functional. In the presence of side-effects, as in Multilisp, an inaccessible task can still be relevant through the side-effects it may perform.

On the positive side, garbage-collecting tasks prevents run-away tasks: all running tasks are stopped during a garbage collection and not restarted until proven accessible. We discuss this in more detail in Section 4.2.

Context-driven reclamation

The other way to perform implicit relevancy detection is by context. In many cases, we can deduce that a task is irrelevant from the context in which it appears. Consider the following example with parallel `if` (which we call `pif`).

(`pif` *pred conseq alter*)

Assume that `pif` creates tasks to evaluate *pred*, *conseq*, and *alter* respectively. If *pred* evaluates to true (false) we know that *conseq* (*alter*) is relevant and that *alter* (*conseq*) is irrelevant. Similarly, with por (or pand) all remaining disjuncts (conjuncts) are irrelevant once one disjunct (conjunct) returns a true (false) value. In both these examples, we can deduce the relevancy of tasks from the context. Then any of the three methods mentioned earlier can be used to actually reclaim the task's computation.

There are three problems with context-driven reclamation:

1. It can only correctly deduce irrelevancy in the absence of side-effects. Even though a task's result may no longer be required, the task can still be relevant through the side-effects it may perform. Note that context-driven reclamation approximates relevancy by accessibility, so it is really a special form of garbage collecting tasks.

2. It is not general-purpose. It can only deduce the relevancy of tasks in certain well-defined contexts.

3. It cannot deduce the relevancy of descendant tasks (unless a lot is known about the structure of the application) since there need not be any connection between the relevancy of a task and its children. Hence this method by itself is inadequate and cannot prevent the run-away task phenomenon.

Explicit Termination (Explicit Checking)

In this form of reclamation, reclamation is "wired" into the application. The programmer arranges for each computation he/she may ever want aborted to periodically check some termination condition and simply terminate, returning a value if this condition is true. This value is either a special terminal value (perhaps a "no-value" like in [Soley] — see Section

3.7) or a nonsense value. In any event, computation is essentially short-circuited by a true termination condition. Note that the programmer performs both the relevancy detection and actual reclamation explicitly.

There are three problems with explicit checking:

1. inserting all the termination checking

   Every descendant computation and (potentially) every function must have termination checks. With system help this may be manageable, but it certainly can be difficult and awkward manually. Furthermore, some functions might be outside the scope of an application, having been compiled in different modules or previously defined in the environment. How do we perform termination checking in these "unknown" functions which may spawn descendant computations?

2. nested termination

   If several pors are nested, do we have to have an independent termination check for each por?

3. sharing of computation

   All parties which might ever share the result of a computation must understand if that computation could ever terminate and what to do if that computation does terminate. For example, a disjunct E in a por may spawn a descendant computation whose result is shared with some function F external to the por. What happens if some other disjunct returns a true value first and disjunct E is terminated? What value is returned to function F? Unfortunately, in the presence of side-effects it is all too easy for this type of sharing to occur.

Since each task checks for termination, there cannot be any run-away task phenomenon with explicit checking, at least amongst the tasks with checking. If the checking is not thorough, and misses descendant tasks spawned by calls to unknown functions, irrelevant tasks may proliferate.

User-directed reclamation

In this form of reclamation, the programmer declares a task to be irrelevant — through some sort of construct, like "kill-task" — and the system kills the task. Another variation is for the system to merely suspend such tasks.

Since the user manages the irrelevance declaration, the run-away task phenomenon may occur in identifying and dealing with descendant tasks (depending on the implementation details).

The main problem with user-directed reclamation is managing all potentially irrelevant tasks. We discuss this problem in Section 2.2.2.

Method	Irrelevancy detection	Actual reclamation method
Garbage collection	implicit, by accessibility	kill tasks
Context-driven	implicit, by context	kill, short-circuit, or suspend tasks
Explicit checking	explicit, by terminate flag	short-circuiting tasks
User-driven	explicit, by annotation	kill, short-circuit, or suspend tasks

Table 2.1: Characteristics of reclamation methods

Context-directed and user-directed reclamation are actually closely related. Context-directed reclamation relies on implicit relevancy detection whereas user-directed reclamation relies on explicit relevancy declaration. Both may use the same back-end methods to actually reclaim task computation. The distinction between context-directed and user-directed reclamation is fuzzy and depends strongly on the user's perspective. To the end-user, a construct like `pif` or por may appear to employ purely implicit reclamation: the user needs to declare nothing. To the systems programmer writing library routines or macros to implement `pif` and por, these constructs may be implemented using purely explicit reclamation.

Summary

Table 2.1 summarizes the characteristics of each of these methods.

2.2.2 Problems with computation reclamation

Problems with relevancy detection

There are problems with both the implicit and explicit methods of relevancy detection. With implicit relevancy detection, the problem is side-effects. As we mentioned earlier, the presence of side-effects means that an inaccessible task can still be relevant.

There are two problems with explicit relevancy detection. The first problem is that a programmer may declare a relevant task irrelevant. Although a programmer (or programming system) can potentially understand an entire program, the programmer may choose not to, may not be able to, or may simply make an error. Thus a relevant task may demand the result/effects of a task declared irrelevant, by touching the task or by requiring the side-effect(s) the task will perform (e.g. release a lock or semaphore).

If a task is killed when it is declared irrelevant, then *lockout* deadlock will result if some other task should later touch it (since the placeholder of the irrelevant task will never be determined). One solution to this problem is to signal an error when a dead task is touched, in effect declaring it an error to mistakenly declare a task irrelevant.

This solution has two problems. First, a dead task may be touched at a point in time

and space far from the point at which it was mistakenly declared irrelevant. Thus the error signalled may be far removed from the actual error, making it difficult to understand and repair. A single mistake could result in multiple errors, as different tasks touch the same dead task. Second, the relevance of a task may change with time. Initially, the best information available may suggest that a task is irrelevant. However, information may later be produced (e.g. from new input data) that suggests that the task is in fact relevant. There is no way within this first solution to express this "imperfect" relevance knowledge.

The second problem with explicit irrelevancy detection is managing the irrelevant tasks. It is very awkward to manage the names of all the irrelevant tasks at the user level. For example, with explicit checking the user must manage all potentially irrelevant tasks by inserting termination checks and with user-directed reclamation the user must maintain lists of all potentially irrelevant tasks. A more fundamental problem is getting the names of irrelevant tasks in the first place. This problem arises with unknown function calls. The user has no way of tracking the descendants that such calls may create.

Problems with actual reclamation methods

All three methods for actually reclaiming computation — killing, short-circuiting, and suspending — have problems. The most serious problem with killing is that it is irreversible. This means that lockout deadlock is always a possibility in the presence of side-effects. As we argued in the previous section, declaring it an error to demand the result (value or effect) of a killed task is inadequate.

Short-circuiting is a particular form of killing (the short-circuited computation is never completed) and even without side-effects has problems with unknown function calls, nesting of computation, and sharing of computation.

Suspending is attractive because it is potentially reversible and thus can circumvent lockout deadlock. Side-effects still pose difficulties for the "safety net" of suspending, because we have to be able to propagate the demand for a side-effect to the task(s) responsible for performing it. (Or, prevent such task(s) from being suspended in the first place).

2.2.3 Summary

Table 2.2 summarizes the potential problems of each method of irrelevancy detection and actual computation reclamation.

In addition to the speed of reclamation and possible run-away task phenomena, the essential issues are:

1. How much of the reclamation must the user manage?

   Does the user or the system manage:

   (a) irrelevancy detection

Irrelevancy detection method	Potential problems
Implicit	Side-effects
Explicit	Mistakes due to user or side-effects Managing tasks Unknown function calls

Actual reclamation method	Potential problems
Killing	Demanding killed tasks, for result (by touching), or for side-effect
Short-circuiting	Awkward to insert checks Unknown function calls Nested computation Shared computation, i.e. demanding killed tasks
Suspending	Side-effects

Table 2.2: Problems with reclamation methods

(b) listing all irrelevant tasks and descendants

(c) the actual reclamation — checking or interrupting named tasks

2. Is the reclamation reversible?

The essential support required for computation reclamation is:

1. some way to recover from incorrect irrelevancy detection

   The sources of incorrect irrevelancy detection are too numerous and perverse — side-effects, sharing of computation, and user mistakes — to realistically prevent incorrect decisions.

2. some way to record the name of descendants

   This addresses the problem of naming the descendant tasks of unknown function calls.

2.3 Side-effects

With speculative computation, as with conventional parallel computation, the main issue with side-effects is proper synchronization. The emphasis on relaxing synchronization constraints and the ability to abort computation give rise to three additional issues.

1. Persistence

When a task is aborted what happens to the effects it has performed? If such effects persist after the task is aborted the semantics of the application may be violated. For example the task may still hold some lock or semaphore, possibly leading to deadlock. Or more generally, it may be possible to deduce that a task executed (and even for how long) by examining its effects, when in fact none of the task was to be visible once aborted. Other times, persistence of effects may be desirable: we may be depending on persistence or the effects may benign, such as if nobody observes them.

One possibility is to completely undo all side-effects performed by an aborted computation, yielding "all-or-nothing" semantics for side-effects, as in atomic transactions.

2. Interference

With speculative styles of computation we might temporarily violate precedence constraints — to speed execution — and repair the execution order later. This could lead to interference, i.e. name conflicts, that would otherwise not occur. For example, in branch prediction both branches could side-effect the variable x. We must ensure that x has the proper value afterwards. This entails both a synchronization issue (ensuring the proper final value for x) and a persistence issue (what if one branch side-effected y and the other branch did not — which effect persists?). Both of these problems can be addressed (at considerable potential expense) by encapsulation methods in which each speculative thread has a copy of the store and the copies are merged afterwards.

3. Relevance

Certain side-effects that would be performed by an aborted task may be relevant. This is the case with the lock/semaphore example given previously. There can also be more subtle interactions: a task may later write a shared object for which other tasks are waiting. This implicit communication introduces two types of problems. First, it is not possible, in general, to determine which tasks "view" the side-effects a task may perform. This makes implicit identification of irrelevant tasks impossible. Second, it is not possible, in general, to determine which tasks generate the side-effects a task may require.[4]

In many instances the only reason that the relevance issue surfaces in the lock or semaphore example is because of the persistence in the first place of the side-effect to acquire the lock or semaphore. This demonstrates the close coupling between relevance and persistence. In other instances, relevance is an issue even without persistence. In the lock/semaphore example some other task T_2 could have acquired the lock/semaphore on behalf of task T_1 with T_1 still responsible for its release. Thus the persistence of task T_1's side-effects has nothing to do with the deadlock and T_1's relevance.

[4]Complicating these two problems is the fact that it is not even possible, in general, to determine *a priori* if a task will perform a side-effect.

2.4 Errors and Exception Handling

Speculative computation raises the following two new issues in error and exception handling:

1. Control-related errors

   A task controlling the resource use of others may encounter an error or exception condition. How does this affect the controllee(s)?

2. Irrelevant errors

   Errors may be or become irrelevant by virtue of occurring in an irrelevant task or in a task which later becomes irrelevant. Recognizing this, how are errors and exception conditions treated?

These issues and the whole problem of errors and exception handling in a parallel computing environment need to be investigated further.

2.5 Summary

The four main problems with supporting speculative computation in Multilisp are controlling computation, reclaiming computation, side-effects, and errors and exception handling. The key issues in controlling computation are the interaction of computation, the modularity of computation control, and the desired control policy. The important issues in reclaiming computation are the speed of reclamation and the degree of management required by the user. Reclamation should be reversible and there should be support for naming descendant tasks. Carefully distinguishing between the reclamation of computation resources (what we mean by computation reclamation) and computation state makes reversible reclamation possible.

Chapter 3

Related Work

Many researchers have noted the potential of speculative computation and proposed various constructs for introducing speculative computation (although few seem to have been concerned about controlling it). In this chapter we survey the most substantial efforts to date (that we know of) concerning speculative computation, giving extensive coverage of other Lisp-based approaches. We criticize these other efforts with respect to the problems discussed in Chapter 2.

3.1 Sponsors

Kornfeld and Hewitt defined a sponsor as "an agent that provides computational effort" [Korn81b]. A succession of Actor-based [Agha] languages, starting with Ether [Korn79], and continuing with Act2 [Theriault] and Acore [Manning], have incorporated sponsors. In Ether, for example, each activity has a sponsor which controls the amount of processing power that the activity receives. A sponsor is created with an initial amount of processing power which may be distributed as the sponsor directs, such as to any sponsors it may create. A sponsor may redistribute any processing power that it in turn may receive. A computation may be aborted by sending a "stifle" message to its sponsor.

Sponsors provide a framework to control computation resources. Certainly, various forms of speculative computation can be expressed within this framework, but this potential of sponsors has not been developed very much. Kornfeld investigated some ideas in this direction, notably his discovery of combinatorial *implosion* [Korn82] and some work with cryptarithmetic puzzles [Korn81a].

3.2 Burton's Work

In [Burt85a] and [Burt85b] Burton considered support for speculative computation in a simple functional programming language. To control computation, he proposed a single

primitive function, which he termed `priority`. This function takes two arguments: an expression E to evaluate and a real-valued priority r. It operates essentially as `future` in Multilisp with the addition of a priority: `priority`(E,r) creates a *speculative* task[1] to evaluate expression E at priority r and immediately returns an empty placeholder object which the task later fills with the result. A task accessing an empty placeholder is blocked until it is filled. Computation resources are allocated to favor high-priority tasks over lower ones Only the ordering implied by the relative values of the priorities is important in this determination. Mandatory tasks (those not created with `priority`) are always favored over speculative tasks. Although not mentioned by Burton, it is apparent that any child task of a speculative task (whether or not the child is created with `priority`) is also speculative.

If a mandatory task demands the result of a speculative task, the speculative task is upgraded to mandatory. Nothing analogous occurs if one speculative task demands the result of another, i.e. priorities do not obey transitivity [Burt89]. Thus Burton's work suffers from the priority subversion described in Section 2.1.

Because Burton considers only a functional language, a speculative task is irrelevant when its placeholder is inaccessible. Thus one could use implicit reclamation — garbage collecting tasks — for reclaiming irrelevant computation. Burton suggests the explicit reclamation approach of Grit and Page [Grit]. This approach consists of spawning "killer" processes to traverse the descendant tree. Burton also suggests using this same approach, with "promoter" processes replacing "killer" processes, to upgrade speculative computation to mandatory.

Problems

The major weakness with Burton's approach centers on his system of priorities which suffers from the following three problems:

1. Lack of expressiveness

   As discussed in Section 2.1, priorities provide only one dimension of computation control.

2. Lack of modularity

   The priorities in Burton's work occupy a global space: there is no hierarchical structure. Thus a programmer must manage a global space of priorities, making it difficult to develop and understand pieces of a program independent of the whole.

3. Lack of flexibility

   The priorities in Burton's work are fixed. Thus the computation resources allocated to speculative activities cannot be reassigned in the event that better information

[1] Burton used the term computation instead of task. We use task in this description to avoid confusion with the types of computation defined in Section 1.1.1.

becomes available. Furthermore, the priorities lack transitivity, leading to priority subversion.

3.3 DAPS

Another functional language effort which addresses speculative computation is Hudak's work on Distributed Applicative Processing Systems (DAPS) [Hud84]. Hudak presents a model for distributed graph reduction which incorporates *vital* (i.e. mandatory) tasks, *eager* (i.e. speculative) tasks, and *reserve* tasks [Hud83]. In our terminology his reserve task is a speculative task that is no longer required by the activity which created it but still is (or may be) required by some other activity. Hudak describes an implicit reclamation scheme to collect garbage graph nodes, delete irrelevant tasks, and detect dormant (i.e. deadlocked) graph nodes in this model [Hud82]. This scheme uses a distributed mark-sweep algorithm which runs concurrently with the graph reduction process. The mark phase consists of two processes. The first marking process traces from the task roots on each processing element to find all deadlocked nodes. After this process completes, a second marking process traces from the graph root to find all vital, eager, and reserve tasks. After the mark phase completes, the sweep phase deletes all irrelevant tasks in the system task queues and performs a conventional memory sweep.

Problems

1. There is no mechanism in Hudak's model to control resources allocated to speculative computation.
2. The garbage collection scheme has problems, as discussed in Chapter 2, when extended to a computation model such as Multilisp's which includes side-effects.
3. The garbage collection scheme may be too slow to discover that a task is irrelevant. Irrelevant tasks can run until the next sweep phase completes, which may be a long time (see discussion in Section 4.2.1).

3.4 MultiScheme

MultiScheme [Miller] is a version of Scheme [Abelson,Rees] extended with parallel constructs. The chief such construct is `future`, which MultiScheme inherited from Multilisp. MultiScheme is very similar to Multilisp although their implementations differ greatly. MultiScheme is based on M.I.T. Scheme (an extension of Revised^3 Scheme [Rees]) and includes special hooks into the implementation for experimenting with different scheduling strategies. Multilisp, in contrast, takes a closed view of the implementation.

MultiScheme provides two forms of support for speculative computation. The first is the procedure `disjoin`. This procedure takes an arbitrary number of undetermined

placeholders as arguments and returns a placeholder for the first argument to be determined. Thus `disjoin` can be used to introduce "first-of" styles of speculative computation where the first alternative to return success renders all other alternatives irrelevant. The second form of support provided by MultiScheme is a stop-and-copy version of the Baker and Hewitt algorithm [Baker78b] to garbage collect irrelevant tasks. MultiScheme provides for the "finalization" of objects. One garbage collection cycle before an object disappears from the system (as a result of being inaccessible from the main root), user supplied code can be invoked to "finalize" the object, releasing locks and cleaning up. See [Miller] for the details of this mechanism.

More recently, Epstein [Epstein] has added priority-based scheduling to MultiScheme using the scheduling hooks mentioned above. Each task has a priority and the scheduler endeavors to run only the highest-priority tasks during each time slice. The priority of a task is at least as great as that of any tasks waiting for its value: when a task touches another task, the touchee's priority is promoted to the toucher's priority. Any change in the touchee's priority is propagated in the same way to the task it may be touching, and so on, down the *touch chain*. This priority system is also used in reverse to reclaim computation: when a task is no longer required (e.g. if it is a remaining alternative after a `disjoin` returns) the task priority is downgraded to a very small value. If this task is touching another, the touchee task is also downgraded (provided the touchee has no higher-priority toucher), and so on down the touch chain. Epstein's work is based on ideas outlined in our proposal [Osborne].

At this time, no one, including Epstein, has run any applications with Epstein's additions for speculative computation. Thus we cannot compare MultiScheme's support for speculative computation with ours on the basis of performance.

Problems

1. Although Epstein's work has provided MultiScheme with an ability to control computation resources, the control lacks modularity.

2. There is a lack of expressiveness in the types of speculative computation that may be introduced. For instance, there is no way to express speculative streams.

3. Downgrading task priorities lacks expressiveness for reclaiming computation since untouched descendant tasks are not downgraded. There is no mechanism to automatically downgrade all the children of a downgraded task and no mechanism to automatically downgrade a collection of tasks. Thus the user must explicitly track all tasks and their decendants that he or she might want to reclaim.

   The burden of explicitly tracking all tasks is so onerous in MultiScheme (consider, for example, descendant tasks generated by "unknown" function calls) that, in fact, only the top level tasks in a `disjoin` or `pif` are reclaimed by downgrading. All descendants are reclaimed implicitly via garbage collection. (This means that there is no run-away task problem). This leads, though, to the next problem.

4. Garbage-collecting tasks suffers from two problems. First, it can be slow to discover that a task is irrelevant. Thus it detracts from the efficiency with which the resources may be used. Second, it reclaims all inaccessible tasks, even those that are mandatory by virtue of the side effects they are performing.

   On the positive side, the Baker and Hewitt garbage collection algorithm used in MultiScheme is guaranteed to collect all run-away tasks.

5. Finalization, which is intended to deal with the problem of exclusive resources (e.g. locks and semaphores) held by aborted tasks, is slow to occur. Not only must one wait until a garbage collection flip occurs, but one must also wait until the object becomes inaccessible before it can be finalized.

3.5 Qlisp

Qlisp [Gabr84,Gabr88,Gold88,Gold89] is a version of Common Lisp [Steele] extended with explicit parallelism constructs. Qlisp and Multilisp are quite similar in their approach to exploiting parallelism. Qlisp even has a construct similar to `future`. In fact, the main "q" constructs in Qlisp — the parallel constructs — have been implemented as macros in MultiScheme.

Qlisp has two forms of support for speculative computation: heavyweight futures (to be implemented soon) and four ways to reclaim computation.

Heavyweight futures are designed for Or-style parallelism (where only some subset of a set of tasks may be required). A heavyweight future is a single future object and a set of tasks.[2] A combining algorithm accumulates the task results to generate the future value. The spawn construct which creates a heavyweight future takes a set of keyword parameters for this combining algorithm. These parameters specify:

1. an initial value, *init*, for the accumulator,
2. a filter function, *Filter*, which determines if a task result should contribute to the future value,
3. a combining function, *Combine*, for updating the accumulator with a task result,
4. a terminate function, *Terminate*, for determining when to return the accumulator value, and
5. a count, *count*, which specifies the number of times the terminate function must be satisfied.

The combining algorithm is as follows. *result* denotes the accumulator value.

[2] Not all the tasks need be known *a priori*; they can also be contributed dynamically.

1. *result* ← *init*

2. when task *i* returns val_i:

```
if count > 0
  if Filter(val_i) {
    result ← Combine(val_i, result)
    if Terminate(result) {
      count ← count − 1
      if count = 0
        determine future to result and kill all remaining tasks
    }
  }
```

The killing of tasks with heavyweight futures is an example of the context-driven reclamation discussed in Section 2.2.1. This is the first way to implicitly kill tasks in Qlisp. In this case, only the tasks associated with the heavyweight future are killed; any descendants of these tasks are not killed.

Constructs such as qor and qand (Qlisp's por and pand) can be built on top of heavyweight futures.

The second way to implicitly kill tasks in Qlisp is with garbage collection. Unlike in MultiScheme, garbage collection in Qlisp is not the primary mechanism for reclaiming computation; it is intended as a backup instead. Thus the speed of garbage-collecting tasks is not a critical issue in Qlisp. The second problem we noted earlier with garbage-collecting tasks is that inaccessible, but still mandatory (via side-effect) tasks may be incorrectly killed. Qlisp allows tasks to be declared "for effect" when created. Such tasks are never killed by the garbage collector.

Qlisp also includes two ways to explicitly kill tasks. The primitive `kill-process`[3] kills the tasks(s) associated with a given future. It is an error to touch a future whose task(s) have been killed.

The other way to explicitly kill tasks is with the `catch` and `throw` constructs.

(`catch` *tag form*) evaluates *form*. If this evaluation produces a value, the value is returned as the value of the `catch` expression.

(`throw` *tag value*) causes *value* to be "thrown" to the nearest `catch` in the dynamic task creation chain which matches *tag*. Upon receiving a thrown value, `catch` immediately returns that value and kills all tasks spawned while evaluating its *form*. (Qlisp also has a `return-from` construct which is equivalent to `throw` except the tag is lexically scoped.) Thus `catch` and `throw` can be used for "one of" styles of speculative computation (e.g. OR parallelism) where one alternative is selected from several explored concurrently.

[3]Tasks are called processes in Qlisp.

Qlisp provides a solution to the problem of killing tasks possessing an exclusive resource.

(unwind-protect *form cleanup*) evaluates *form* and always evaluates *cleanup* before

1. the unwind-protect returns, even if *form* causes a throw to be evaluated, or
2. the task evaluating unwind-protect is actually killed.

Problems

The major problem with exploiting speculative computation in Qlisp is the lack of any way to control computation resources. We list four other problems below.

1. Lack of expressiveness

   As with MultiScheme, there is no way to express certain types of speculative computation. For instance, there is no way to express speculative streams.

2. Task touching

   With explicit task killing, a task can touch a killed task. In the case of catch/throw this can happen even without side-effects — the value returned by a throw could be a pointer to a task spawned within its matching catch. Touching a killed task is handled by signalling an error. As discussed in Section 2.2.2, this can lead to bizarre and undesirable error behavior.

3. Task killing lacks sufficient power

   Except for garbage collection, which intended as a backup, each way of killing tasks in Qlisp lacks sufficient power. The simplest way to kill tasks is with kill-process. However, kill-process does not kill child tasks and thus the user must explicitly track all tasks and their descendants that he or she might want to kill. This need to explicitly track all tasks makes it especially difficult to deal with "unknown" function calls which may create child tasks.

   The next way of killing tasks is implicitly in conjunction with heavyweight futures. The supposed advantage of heavyweight futures over more primitive implementations using the catch and throw mechanism (besides ease of expression) is that a heavyweight future clearly defines the lifetime of the future's associated tasks and thus these tasks can be killed implicitly. However, the children of these tasks may have a lifetime exceeding that of the heavyweight future. (The future object of a child task could "escape" the heavyweight future by side-effect or by the return value.) Thus only the tasks immediately associated with a heavyweight future (and not their descendants) may be killed implicitly.[4] This leaves the user responsible for explicitly killing all the

[4]Even implicitly killing these immediate tasks is problematic: some of them could be required for their side-effects, such as releasing a lock or semaphore. The unwind-protect construct provides protection against this problem.

descendant tasks. We have already mentioned the problems of explicitly killing with `kill-process`.

The alternative, and last way to kill tasks, is to use `catch` and `throw`. This method kills all descendant tasks, unlike the previous two methods, but now runs into problems with side-effects. Because computation is killed when it is thought to be irrelevant, there is no way to restart it if proves to be necessary. Side-effects can make it very difficult to determine when a task is irrelevant, especially in large systems.

4. Run-away tasks

   Descendants of a `catch` may be spawned faster than they can be killed. One expensive way to prevent this is for a task to check up its chain of `catches` before spawning.

3.6 PaiLisp

PaiLisp [Ito] is another parallel Lisp based on Scheme. PaiLisp is built on top of PaiLisp-Kernel, which is Scheme plus four parallel support constructs. Only two of these constructs are relevant here: `spawn` and `call/cc`. The familiar `future` and `delay` constructs are built on top of PaiLisp-Kernel. PaiLisp also contains a number of constructs utilizing speculative computation: `pcond`, which is a speculative version of `cond`, and `par-and`, and `par-or` which are speculative versions of `and` and `or` respectively (like our `pand` and `por`). These speculative constructs are also built on top of PaiLisp-Kernel.

The key to speculative computation in PaiLisp is `call/cc`. PaiLisp-Kernel extends the semantics of Scheme's `call/cc` (see [Rees]). As in Scheme, `call/cc` creates a continuation and applies the argument of `call/cc`, which must be a procedure of one argument, to this continuation. However, each such continuation retains the name of the task that created it. When a continuation is invoked (with some argument *arg*), the caller's task is compared with the continuation's creator task. There are two cases.

1. If the caller's task is the same as the creator's task, then the call causes a "goto" within that task, just like in Scheme. The `call/cc` which created the continuation returns with *arg*.

2. If the caller's task differs from the creator's task, then the call causes a "goto" within the creator task just as if the continuation were invoked by the creator task. Thus the `call/cc` in the creator task which created the continuation returns with *arg*. The caller task proceeds concurrently with the goto in the creator task and receives `undefined` as the return value of the call.

Thus a continuation is a goto strictly within its task of creation. A continuation invoked by a task other than its creator amounts to remotely forcing a goto in the creator task. This provides a means for one task to terminate another (as well as other inter-task control). To illustrate this we describe the spawn construct.

(spawn *e*) creates a task to evaluate *e* and returns undefined. A task in PaiLisp-Kernel is considered to have a definite start and end. Thus to terminate a task we can just force it to goto the end of the task. The following code captures the "ending" continuation and saves it in the variable p.

```
(spawn (call/cc (lambda (cont) (set! p cont) e)))
```

To terminate *e* anytime during its execution we merely have to invoke the ending continuation saved in p, as in

```
(p 'dummy)
```

This causes the task evaluating *e* to jump to the end and terminate. Thus PaiLisp's extension to `call/cc` allows the ability to kill tasks.[5] [Ito] describes how to use this ability to construct pcond, par-and, and par-or and, of course, this ability could also be used in other applications of speculative computation.

Problems

1. There is no way to control computation resources.
2. Task killing via task ending continuations is inadequate.

   "Killed" tasks cannot be restarted. Thus this form of computation reclamation suffers from the problems with sharing and nesting of computation described in Section 4.2. Also, in Pailisp-Kernel we have to explicitly manage each task that we might ever want to kill. For instance, we have to explicitly retain the task ending continuation of each such task. However, we can avoid the manual killing of descendant tasks that this implies by an appropriate macro interface which saves the ending continuation of a task's children in some place that the task's ending continuation can find and invoke them. This trick would give automatic naming of descendants and thus avoid the awkwardness and the difficulties with unknown function calls described in Section 4.2.
3. Task touching

   As in Qlisp, a task can touch a killed task.
4. Side-effects

   There is no provision for dealing with tasks that may possess exclusive resources (e.g. locks and semaphores) when killed.
5. The continuation-based termination mechanism seems very expensive.

[5] There seems to be at least one major problem with this extension, however: what happens if a continuation is invoked after its creator task has returned a result and terminated?

3.7 Speculative Computation in Dataflow

Soley [Soley] investigated speculative computation in the dataflow language Id [Nikhil]. His interest was primarily in supporting the multiple-approach speculative computation prevalent in A.I.

In the first part of his thesis, Soley introduces nondeterminism into the determinate language Id to give the ability to control speculation. He suggests extending Id with a single nondeterministic primitive: a multiple assignment cell (in contrast to the write-once cells (I-structures) already in Id). As Soley demonstrates, this addition is sufficient to allow computation reclamation by explicit checking. Thus his multiple assignment cell brings Id up to the latent ability of imperative languages with respect to speculation control. Soley points out that this explicit checking approach is awkward and cannot deal with unknown function calls (as we argued in Section 2.2.1).

Soley addresses these problems in the second part of his thesis where he describes language and system support for speculation. In the Id system, requests to the system for memory allocation and code block execution (i.e. function calling) are managed by nondeterministic entities called *managers*. Soley's idea for controlling computation is to place an agent between the application code and the system manager(s) to "filter" these manager requests. Since managers are part of the Id language, Soley describes this agent itself as a manager. This elegant idea allows Soley to express the desired computation control completely within the Id language (i.e. by writing appropriate Id code for the agent-managers).[6] For example, an agent-manager could allow computation to proceed based on some priority of the requests.

Soley performs computation reclamation in the same manner: he filters requests through agent-managers that check some flag for reclamation. There are two strategies when the flag indicates reclamation. The agent-manager can either suspend computation by failing to pass requests on to the system manager, or terminate computation by returning a special "service denied; please terminate" result to requests. Soley chose the latter strategy so that the underlying dataflow graphs would be self-cleaning. To indicate termination, an agent-manager returns the special value "no-value". Soley modified all major language schemas, such as function-calling, to take appropriate actions when an input is "no-value". (See the Appendix of [Soley] for details.) In general, if any input to a dataflow operator is "no-value", the result will be "no-value" as well. The predicate no-value? "captures" such propagated "no-values".

This method of computation reclamation amounts to explicit checking at the granularity of manager requests (i.e. allocation and function calls). Reclamation is achieved by "short-circuiting" computation, forcing it to terminate prematurely by returning a special nonsense result, as described in Section 2.2.1, rather than suspending computation. Unknown function calls pose no problems with this computation reclamation since managers are inherited dynamically by called code. Thus any called code, even if it is an "unknown"

[6] Though he had to extend the language and system architecture a bit.

function, always has a well-known manager to perform termination checks on allocation and function calls.

Soley abstracted and encapsulated his support for speculative computation in three constructs: `speculate`, `terminate`, and `priority`.

`speculate` *f lst*

calls *f* on each element of *lst* in parallel. Let us call the application of *f* to an element of *lst* a branch. (Soley calls this a task, but we prefer branch to avoid confusion.) `speculate` returns the result of any one branch (presumably the first branch to yield a result). In this sense, `speculate` is like McCarthy's amb operator [McCarthy]. Unfinished branches are terminated when `speculate` returns. Thus `speculate` is basically parallel or, i.e. por (assuming that not all branches return nil).

`terminate` *dummy*

injects "no-value" into the executing computation, causing that branch to be terminated.

`priority` *priority*

sets the priority of the executing branch to *priority*. Each branch can have a different priority.

`terminate` and `priority` must be executed in the scope of a `speculate`.

`speculate` creates an agent-manager for each branch to control computation and uses an array to communicate the relative success/failure of branches. It is certainly possible to build other constructs for speculative computation using the same agent-manager approach.

Using a dataflow simulator, Soley investigated different approaches to solving the Eight Puzzle (see Section 8.6 or [Nilsson]) using his support for speculative computation. His results were mixed. While he found that speculation greatly reduced the critical path, this only happened if the search depth was sufficiently large. In other words, he found the overhead for speculative computation to be rather large. This is not surprising given the expense of managers and his rather coarse-grain checking. When he experimented with priorities, he found that the overhead of priority management completely overwhelmed any benefit.

Problems

1. Lack of expressiveness

   As mentioned earlier, `speculate` is basically por and thus is only applicable to or-like parallel search with a static number of branches. Not all speculative computation fits this mold. Two non-conforming examples are or-like search with a dynamic number

of branches and speculative if. These two examples, and many similar examples, can be accommodated by implementing new constructs in terms of underlying agent-managers.

However, a more fundamental problem with Soley's approach is his exclusive reliance on the use of priorities for computation control. Thus his approach suffers the same lack of expressiveness as described in Section 2.1. For example, Soley cannot express speculative streams.

2. Priority subversion

   Priorities are not transitive, leading to the effective subversion of priorities discussed in Section 2.1 when one computation blocks on another computation of lower priority. This is difficult problem to deal with because it touches on a fundamental problem with controlling computation in dataflow.

   Each instruction in dataflow is an independently scheduled thread (i.e. task). This ultra-fine-grained parallelism makes it too expensive to control computation at the thread level (by attaching a priority with each instruction, for example). The alternative is to organize instructions into collections and attempt to control computation by controlling the collection. This is the approach that Soley took: he organized computations into "branches" and assigned priorities to these branches.[7] This alternative implies a restriction in the granularity of control which forces a certain lack of control: individual computations comprising a collection cannot be controlled. Thus there are only two choices available when a computation in some collection A demands the result of a computation in some collection B: either transfer A's priority to all the members of B or do not transfer any priority. Both alternatives are inadequate because the independent computations of B must be treated collectively.

3. Reclamation of shared computation

   Soley's method of computation reclamation solves the nested computation and unknown function call problems with explicit checking, but still suffers from the shared computation problem. This is only a latent problem because there is no way to express shared computation in Id with Soley's extensions. Since Id evaluates all expressions eagerly, there is no way for one expression to be evaluated simultaneously in the scope of two non-nested speculates. (There is, of course, the possibility of branches communicating via I-structures.)

   This changes, however, with the addition of laziness to Id as proposed by [Heller]. A lazy thunk could be simultaneously in the scope of two non-nested speculates and thus represent shared computation if demanded by both speculates. What happens if one speculate terminates its branch(es) which demanded the lazy thunk while the other speculate still demands the thunk's value?

   In Soley's scheme, each branch of a speculate is assigned an agent-manager which all computations in that branch inherit via the dynamic scoping of managers. The

[7] The only such "branches" in Soley's work are the well-formed branches of his speculate construct and he only assigned priorities to branches relative to other branches of the same speculate. We generalize both these notions here.

first branch to demand the lazy thunk will start the thunk in the branch's scope, and thus the thunk computation will inherit that branch's agent-manager. Subsequent demands for that thunk by other thunks will block awaiting the value (we are assuming memoization of lazy thunks). If this first branch is now terminated, the thunk may (depending on timing) evaluate to "no-value" instead of its "proper" result.

3.8 Parallel Logic Languages

Researchers in parallel logic languages have long recognized the potential of a form of multiple-approach speculative computation which they call Or-parallelism. The idea of Or-parallelism is to evaluate all the potentially unifiable clauses (i.e. "matching") clauses for a given goal in parallel. (See [Shapl] for example.) There are two types of Or-parallelism: one-solution Or-parallelism and all-solution Or-parallelism. One-solution Or-parallelism is essentially like por: when one evaluation succeeds the remaining evaluations may be aborted. All-solution Or-parallelism involves no speculation *per se* since all evaluations are required. However, in those parallel logic languages retaining the cut operator (from sequential Prolog), there can still be speculative computation in connection with cuts. See [Hausman] for example. Or-parallelism is a simple idea, but its exploitation is difficult, in general, due to the interference problem described in Section 2.3: the variable bindings for the parallel clauses may clash. The general solution to this problem requires multiple environments, with one environment per clause. Much of the work to date in Or-parallelism has been concerned with reducing the copying overhead involved with such multiple environments. (See, for instance, the introduction of [Warren].)

With respect to Or-parallelism, work on parallel languages has split into two camps: those committed to Or-parallelism as their primary means of concurrency and those committed to And-parallelism (a type of mandatory parallelism in logic languages), with limited Or-parallelism. To distinguish these approaches, we call the former "primary" Or-parallelism and the latter "secondary" Or-parallelism.

The most well-developed work in primary Or-parallelism seems to be that of the Aurora System [Lusk], resulting from the collaboration of researchers from the Argonne National Laboratory, the University of Manchester, and the Swedish Institute of Computer Science. The Aurora System is an all-solution, Or-parallelism extension of Prolog with cut (so there is still speculative computation). Although these researchers have recognized the potential of speculative computation, they are still at an early stage. They have no means for controlling Or-parallelism and their scheduler treats speculative work the same as mandatory work.

The main work in secondary Or-parallelism is in "committed choice languages". Such languages nondeterministically choose a single unifiable clause to reduce. (This choice is irreversible, so there is no backtracking like in sequential Prolog.) The only Or-parallelism, then, in such languages is the one-solution Or-parallelism in choosing the unifiable clause to reduce. This Or-parallelism can be non-trivial, however, because the *guard* part of clauses must be evaluated before clause selection. There are three main committed choice languages with such Or-parallelism. PARLOG [Clark] allows Or-parallelism as long as

compile-time analysis can determine that multiple environments are not required (i.e. there is no interference). Guarded Horn Clauses (GHC) [Ueda] takes a run-time approach. GHC allows Or-parallelism with the proviso that if execution of a clause could lead to interference (by binding a variable non-local to that clause) the execution of the clause suspends until either some other process binds the variable or the clause is selected for commitment. Flat GHC (FGHC) is a subset of GHC that does not require suspension because of restrictions placed on the *guard* part of a clause. Flat Concurrent Prolog (FCP) [Shap2] (an evolution of Concurrent Prolog which did allow all-solution Or-parallelism) allows one-solution Or-parallelism because, like GHC, it places restrictions on the clauses so that suspension is never required. None of PARLOG, GHC, and FCP have any means for controlling Or-parallelism.

The work of Chikayama et al [Chika] offers a higher-level approach to speculative computation in logic languages. They describe an operating system for a parallel inference machine (PIM). This operating system, called PIMOS, provides features which, although not designed for speculative computation *per se*, may be used for supporting speculative computation. PIMOS is implemented in an extension of the KL1 language, which is a variant of FGHC. The main extensions useful for speculative computation are priorities and *shōens*, managers for controlling computation which have much in common with the groups we present in Chapter 5. Both goals and clauses may have priority annotations attached for directing operating system scheduling. All priorities apply only locally within a processor cluster, though, not globally, since PIM is a loosely-coupled multiprocessor. A shōen controls the execution of a specified goal in two ways. First, each shōen has a priority range, specified at creation time, in which all the computation comprising this goal executes. That is, all the computation within a shōen executes with priorities relative to the priority range of the shōen. In the current implementation the priority range is fixed once a shōen is created. It is not clear if the goal and clause priorities are also fixed. Second, a shōen controls the resources — such as time and space — allocated to the shōen's computation according to the meta-program "script" of the shōen. This resource control feature may be used to abort computation. Presently, execution time is the only resource that a shōen can control.

Shōens can be nested; thus shōens provide modularity.

Problems

The main problem with KL1 is lack of demand transitivity. Three factors contribute to this. First, the priorities — at least the shōen priorities — are (currently) fixed. Second, there is no priority interaction across processor clusters. Third, there is no explicit relationship between the demander of a logic variable's value and the computation which will produce the variable's value. Thus there is no general way to determine which computation to demand Therefore logic variables are fundamentally different from the futures and tasks in Multilisp in a way that poses difficulty for speculative computation.

3.9 Summary

Aside from Epstein's work (based on our work), Burton's work (not implemented), and Chikayama's shōens, these approaches to speculative computation address only half the issues: they all address reclamation, but fail to address controlling computation. Furthermore, the computation reclamation in all these approaches is inadequate in one way or another: either it is too slow (garbage collecting tasks), irreversible, or has no support for naming descendants (explicit methods).

We want to put these criticisms in perspective though. Our implementation has problems too — notably the same lack of modularity and lack of speculative streams for which we have criticized other work. See Section 6.3 for a discussion of deficiencies. However, our implementation is a subset of a model for speculative computation, which we present in Chapter 5, which does address these problems. The applications we describe in Chapter 8 support our statement about computation reclamation. In the Emycin example described in Section 8.3, aborting useless computation is important but garbage collecting tasks is totally inadequate: no garbage collection occurs during execution.

Chapter 4

Approach

In this chapter we discuss our approach to the four main problems described in Chapter 2.

4.1 Controlling Computation

Computation control is a scheduling problem. Optimal solution of scheduling problems is often difficult (both in determining the solution and implementing it) even if sufficient information is available. They have an annoying tendency, for example, to be NP-hard. In most cases in practice we do not have sufficient information available and we must either resort to simplifying assumptions to make the optimal scheduling problem tractable or abandon the optimal scheduling problem and pursue heuristic/*ad hoc* scheduling policies. We pursue the former in Chapter 2 and the latter here.

We believe the most important control is some way to order the allocation of resources[1] to speculative computations in accordance to their relative "promise". This dimension of control specifies the order in which tasks are allocated available resources, especially the order in which tasks are selected for running if more resources become available or selected for suspending if fewer resources become available. However, ordering is not sufficient. We believe there are three other important dimensions of control over resource use:

1. amount of resource use (i.e. total resource use over time)
2. rate of resource use (i.e. instantaneous resource use)
3. duration of resource use

Given the singular importance of ordering we believe that ordering and the last three

[1] In accordance with our assumption in Section 2.2 we concentrate exclusively on processor resources. These remarks apply equally, however, to other resources, such as memory.

dimensions should be independent. The last three dimensions are related, and hence dependent, as follows:

$$a_{A_j} = \sum_{i \in A_j} \int_{t=0}^{d_i} r_i(t) dt$$

where a_{A_j} is the amount of resources devoted to activity j, comprised of tasks in the set A_j, $r_i(t)$ is the rate of task i as a function of time, and d_i is the duration of task i.

As discussed in Section 2.1, control must include demand transitivity in the ordering and some mechanism for modularity.

Our ideas for controlling computation are still *ad hoc* and heuristic. To move our ideas past this point we need to do two things. First, we need to study scheduling problems to determine optimal scheduling policies and the required control. We hope to either apply these results directly or extrapolate them to practical situations. In Chapter 9 we start by analyzing optimal scheduling of simple pors and pands. Second, we need to study many more applications to determine the desired control in different situations. In the rest of this thesis (excepting Chapter 9) we focus on basic control mechanisms.

4.2 Reclaiming Computation

From the discussion in Section 2.2, it should be obvious that we need support for reclaiming computation. Reclamation without support, as exemplified by explicit checking, is too awkward and suffers from difficulties such as unknown function calls, nesting, and sharing discussed in Section 2.2.1. While reclamation without support may be useful in simple cases, it lacks expressive power in general.

Ideally, we would like both irrelevancy detection and reclamation to be completely implicit at the end-user level. The problems with side-effects discussed in Section 2.2 interfere with this utopia. A more pragmatic goal is the reduction of non-essential explicitness at the end-user level. There are two different ways of achieving this goal. The first way is by making the irrelevancy detection and reclamation implicit at the system level. Garbage-collecting tasks is an example of this approach. The second way is by hiding system-level explicitness with suitable user interfaces, such as libraries and macros. Context-driven reclamation is an example of this approach. We call these two approaches implicit reclamation at the system level and explicit reclamation at the system level.

4.2.1 Implicit reclamation at the system level

The only way to perform implicit reclamation at the system level is by garbage-collecting tasks. We reject this method because we believe it is too inefficient. In implicit reclamation, irrelevant computation is reclaimed when a garbage collection occurs, which is too infrequently and is too costly, we argue.

The exact point in the garbage collection cycle at which irrelevant computation is reclaimed depends on the garbage collection mechanism. There are three main garbage collection methods: mark and sweep, reference counting, and scavenging. Reference counting cannot reclaim cyclic structures so we do not consider it further.

Mark and sweep collectors can employ either a "conservative" strategy or a "lenient" strategy. These strategies are described in [Baker78b] (though the terminology is ours). In the conservative strategy, all running tasks are stopped when marking begins and no task may begin running until it has been marked as accessible from the root pointer set (and therefore relevant). Thus irrelevant computation is reclaimed at the initiation of a garbage collection cycle. (The associated computation state is not reclaimed until the end of the garbage collection cycle.) In the lenient strategy, a task may continue to run after the initiation of the garbage collection cycle until either the task attempts to mutate a marked object or the marking phase of the collector reaches the task. In either case, the task must suspend until the collector marks the task as accessible from the root pointer. If this never occurs, the task's storage is recycled by the sweep phase — i.e. the task is irrelevant. Thus in the lenient strategy irrelevant computation is reclaimed at any point up until the end of the sweep phase.

Scavenging (i.e. copying) collectors can only use the conservative strategy of stopping all tasks prior to a garbage collection flip. A task may resume running, though, when it is copied into newspace (indicating that it is accessible from the root pointer set and hence relevant).[2] Thus irrelevant computation is reclaimed when a garbage collection flip occurs.

Therefore the frequency of implicit computation reclamation depends on the the period between garbage collection cycles. Normally garbage collection is invoked when free storage is nearly exhausted, which typically takes a fairly long time. Irrelevant tasks might well run to completion in this time. The exact time depends on the size of the heap, the amount of non-garbage surviving the last garbage collection, and the rate of storage allocation, i.e. consing. Thus the frequency of implicit computation reclamation is tied to factors that do not necessarily have anything to do with the creation, rate of creation, or even presence of irrelevant computation! An application which does little consing, for instance, will have an inordinately long inter-garbage collection period and hence very infrequent computation reclamation.

There are three ways we could address the problem with the infrequency of implicit reclamation:

1. We could invoke garbage collection more frequently (than is required by storage concerns).

   This is unattractive because it incurs the cost of garbage collection, more frequently. Generally, garbage collection is performed as infrequently as possible for good reason:

[2] A lenient strategy whereby a task in oldspace continues to run will not work in general since there is no room left in oldspace for the task to allocate storage. (Presumably that is the reason a garbage collection was initiated).

to amortize its large cost over as much time as possible. The cost consists mainly of two components:

(a) the synchronization cost

This is the cost, in wasted processor cycles, of getting all the processors to agree to begin the garbage collection cycle. This cost depends on the synchronization mechanism and the number of processors, so we consider it fixed.

(b) the tracing cost

This is the cost, in time, to trace all pointers from the root pointer set[3] and thus determine the transitive closure of accessible, i.e. non-garbage, objects. For scavenging collectors, the tracing cost includes copying accessible objects to newspace.

Thus the tracing cost, which dominates the cost of garbage collection, depends on the amount of non-garbage, i.e. number of objects retained after garbage collection. This means the tracing cost is independent of the amount of storage allocated (and independent of the number of irrelevant tasks)! Thus the cost per object allocated is minimized by garbage-collecting as infrequently as possible. By collecting more frequently than this, we pay a greater cost per object allocated.

In addition, many objects survive for a relatively long time. Tracing these objects thus represents a fixed cost incurred at every garbage collection.

(A mark and sweep collector has a third component: the sweep cost. This is the time to sweep all of memory, return all unmarked objects to the free list, and clear the marks.)

Therefore garbage-collecting more frequently both incurs the fixed cost of garbage collection more frequently and increases the cost per object allocated. This increased cost may possibly overshadow the benefit of reclaiming irrelevant tasks more quickly. Even worse, we pay this cost independent of the number and existence of irrelevant tasks.

2. We could invoke the garbage collector explicitly whenever we believe there are enough irrelevant tasks to justify the cost.

Even if we can determine when we have irrelevant tasks (this may not be too hard in certain situations), this approach has two problems:

(a) We have to trace all accessible objects whether tasks or not.

The cost depends on the number of non-garbage objects and is independent of the number of irrelevant tasks. There is no reason why we should have to perform the expensive tracing of all objects just to isolate irrelevant tasks. At the very least we should only have to search through task objects.

[3] With incremental garbage collectors tracing may be interleaved with processing.

(b) It is unattractive if we are continually generating irrelevant tasks.
In this case, this approach reduces to the previous one where we invoked the garbage collector more frequently.

3. We could use generation garbage collection [Lieberman].

Generation garbage collection can significantly reduce the cost of garbage collection by reducing the average number of accessible objects that must be traced. This reduces the tracing cost. The synchronization cost, however, is unchanged.[4] This reduced cost does not entirely solve the problems with implicit reclamation. Four major problems still remain:

(a) The reclamation time is still coupled to the consing rate.
Thus the reclamation time can still be inordinately long if there is little or no consing. The reduced cost of generation garbage collection makes it reasonable to garbage collect more frequently than dictated by storage concerns, but the garbage collection cost is still non-zero (see next point), thus limiting reclamation frequency. Also, as noted previously, this cost is independent of the number and existence of irrelevant tasks.

(b) We still must trace all accessible objects in a generation, whether tasks or not.
We still have to examine a large number of task objects in a generation which are not irrelevant tasks or even tasks. Thus the cost per irrelevant task collected is still high, especially if there are few irrelevant tasks. This still makes it expensive to explicitly invoke the garbage collector and to run the garbage collector more frequently than necessary for storage concerns.

(c) An irrelevant computation might still take a long time to be reclaimed if it or an inaccessible object that points to it (directly or indirectly) gets promoted to older generations.

(d) A task may still be irrelevant but accessible.

Thus generation garbage collection merely scales down the costs, but the costs are still significant.

Note that we cannot allow tasks to be promoted to older generations like other objects otherwise we will get irrelevant tasks in older generations. Such tasks will take a very long time to be reclaimed unless we garbage collect all generations frequently, destroying the advantage of generation garbage collection. This has the consequence of increasing the number of objects we must trace in the youngest generation to reclaim irrelevant tasks. However, the additional garbage collection cost incurred is probably small.

Implicit computation reclamation tries to satisfy the different goals of computation reclamation and storage reclamation with the same mechanism, garbage collection. Computation reclamation must have a small response time to minimize resource wastage and a

[4] Although one might be able to combine generation garbage collection with area garbage collection [Bishop].

small running time to avoid discouraging its invocation. This implies the garbage collector must run frequently. (The only thing we can do for the the running time within the implicit framework is employ generation garbage collection.) By contrast, the chief goal of storage reclamation is a small cost. This implies the garbage collector must run as infrequently as possible. The conflict in these goals means a compromise in which computation reclamation is either infrequent and hence costly in resource wastage or frequent and costly in overhead (or a muddle of both).

We escape this dilemma by employing mechanisms appropriate to the different reclamations. For computation reclamation we employ explicit reclamation and for storage reclamation we employ conventional garbage collection: thus, orthogonal mechanisms for orthogonal purposes.

Garbage-collecting tasks is still attractive as a backup reclamation mechanism, as proposed in Qlisp. We need some way, however, to declare a task relevant for the effects it may perform, such the "for effect" declaration in Qlisp. Alternatively, we could list such tasks with a "registry" so that they remain accessible.

4.2.2 Explicit reclamation at the system level

As mentioned in the previous section, we employ explicit reclamation. The essential support required for explicit computation reclamation is:

1. Some way to *reversibly* reclaim a task's computation resources, i.e. allow the task to resume if necessary.

   A task's computation resources can be irreversibly reclaimed, i.e., the task terminated, only if it can be proven that neither the task's result nor any effects it might perform are required by any other task. Side-effects make this determination very difficult, if not impossible in some situations. In large systems, for example, it can be very difficult to understand all the ways in which computations can interact. The ability to resume a task after its computation resources have been reclaimed provides an important safety net.

2. Some way to automatically name all the descendants of a task declared irrelevant.

   The possibility of unknown function calls makes it impossibly difficult for a user to track all the descendants of an irrelevant task. Ideally, we would like the system to manage all the descendants of an irrelevant task, automatically declaring them all irrelevant when their parent is declared irrelevant.

To achieve the first support, we stun[5] a task when it is declared irrelevant; that is, suspend execution of the task as discussed in Section 2.2.[6], thereby reclaiming its computation resources (which we argued earlier is the important issue), while still maintaining

[5]Terminology introduced by Goldman and Gabriel in [Gold88].

[6]By suspend here, we mean a pause in a task's execution, not the state of suspension caused by the Multilisp suspend primitive.

its computation state. Thus the task may be restarted later if it is touched by a relevant task (or if it must perform a relevant side-effect). Instead of stunned, we say that a task is *stayed* when its computation resources have been reclaimed. To declare a task irrelevant is to *stay* the task, and the act of declaring a task irrelevant is called *staying* the task.

The second support is a compromise between explicit and implicit irrelevancy detection: we have explicit naming for "root" irrelevant tasks and implicit naming for all their descendants.

In later chapters we present details on our explicit computation reclamation mechanism and describe how we prevent run-away tasks and how we deal with the problems posed by side-effects.

4.3 Side-effects

Of the three issues described in Section 2.3, we explicitly address only the third issue, the relevance of side-effects. We do not address the issues with persistence and interference, i.e. name conflicts. We provide no special support for non-persistence, such as rolling-back tasks and undoing their effects, nor do we provide special support for non-interference, such as "encapsulating" tasks.

We do not provide these mechanisms for two reasons. First, we do not believe that persistence and interference issues are important in our intended application domain. We are primarily interested in functional speculative computation and nondeterministic speculative computation. We are not focusing on optimistic computation and we are not pursuing applications such as databases involving significant atomic transaction-like activity. Rather, we are interested in applications with few side-effects -- we believe that side-effects should be used sparingly. Furthermore, of those applications with side-effects, we believe few actually require roll-back or encapsulation. In many situations we believe that roll-forward strategies, such as we discuss in Chapter 7, would be just as effective as roll-back strategies.

Second, we believe that these mechanisms are too expensive in a medium grain parallelism model such as Multilisp. If side-effects requiring non-trivial non-persistence and non-interference are a frequent and important part of an application, we believe that other coarser-grained paradigms are more appropriate and efficient than that provided by Multilisp.

Users which require non-persistence and/or non-interference must provide the functionality themselves. Non-persistence and non-interference are basically just special forms of synchronization and thus they can be built on top of the low level synchronization primitives we do provide. This could be a very painful approach in some cases, such as for checkpointing and committing (especially with nested activities), so we do not recommend it.

We address the relevance of side-effects in Chapter 7.

4.4 Errors and Exception Handling

We ignore in this work the issues associated with errors and exception handling introduced by speculative computation. We do so mainly to reduce the scope of the work to manageable proportions. There still remain questions on how to best treat errors and exception handling in conventional (i.e. mandatory) parallel computation.

To handle errors and exception conditions, we believe that related computations should be organized into units (such as the groups we mention later). Then when an error or exception occurs all the tasks in the unit should be halted and a debugger entered. This is the model of error handling in Mul-T [Kranz].[7] It has two desirable features. First, all the tasks in the affected unit are halted. This allows the user to investigate the state of related tasks, any of which might have actually caused the error. It also prevents related tasks from generating a cascade of errors as a result of the first error or running indefinitely because an errant task failed to perform some action. This last part is important for curtailing speculative computation if a "control" task encounters an error. Second, only the tasks in the affected unit are halted. Thus, unrelated tasks may continue running.

We believe that errors and exception conditions in irrelevant tasks should be handled just as in relevant tasks, the rationale being that a user would (or perhaps should) always want to know when an error occurs. We can imagine instances where this might not be the case but we regard such instances as pathological.

The two main issues in extending the Mul-T error handling model for speculative computation are adding nested units and from a user's point of view, ensuring that speculative task controllers and controllees are in the same unit. Mul-T currently supports only one level of units, at the read-eval-print-loop level.

4.5 Summary

Optimal control of speculative computation is a difficult scheduling problem, so we focus on heuristic control. Two essential components of control are demand transitivity and modularity. In the next chapter we present a model for controlling computation which includes these two components.

We employ explicit computation reclamation at the system level. Implicit computation reclamation, i.e. garbage collection, is too slow. We achieve explicit reclamation by causing tasks to be *stayed*, which suspends task execution thereby reclaiming processor resources. A stayed task may be restarted, so this reclamation is reversible. This reclamation is accessible to the end-user as either context-driven reclamation or user-directed reclamation, but with automatic naming of all descendants to avoid the problem of managing descendants of unknown function calls.

[7] Our "units" are called groups in the Mul-T model. Mul-T's groups should not be confused with the groups we introduce later, although they are similar in design and purpose.

Of the issues with side-effects, we only address relevance (in Chapter 7). We ignore errors and exception handling.

Chapter 5

A Model for Speculative Computation

Speculative computation involves both eager and demand-driven components. The eager component is obvious: we start computation before it is required. As discussed in Section 2.1 two assumptions make this eager component non-trivial. First, not all the computation need be required. Second, there are insufficient resources available to perform all the pending computation simultaneously. The issue, then, is how to make the most efficient use of the limited resources. We want to perform as much of the computation as we can before the computation is actually demanded. Also, we must have some way to add and jettison speculative computation as the ebb and flow of mandatory computation leaves more and less resources, respectively, available for speculative computation. Hence, the importance of the ordering that we talked about earlier.

The demand-driven component is less obvious but just as important. This component arises because a speculative computation may be demanded by other computation. For example, a speculative computation may be demanded by a mandatory computation. More important, though, is the interaction of speculative computation, i.e. one speculative computation demanding another. The issue in this case is how to incorporate such interactions into the eager-based scheduling of computation. The specific issues, mentioned in Chapter 2, are demand transitivity and modularity.

A model for speculative computation must incorporate both components. The model must include some way to express the scheduling of computation and some way to express the interaction of computation while maintaining demand transitivity and modularity. We present a model for speculative computation in Multilisp that meets these requirements. This model is also suitable for other task-based (i.e. multiple independent threads of computation) parallel languages.

5.1 The General Sponsor Model

Our model for speculative computation in a task-based language like Multilisp is based on the concept of sponsors described in Section 3.1. To run, a task needs certain computation resources, like processing power (the use of a processor) and memory. The allocation of resources to tasks in our model is controlled by the *attributes* that a task possesses. Sponsors supply these attributes. Each task may have zero or more sponsors which contribute attributes to that task. Thus, the sponsors of a task collectively determine the computation resources allocated to that task. A task without a sponsor does not run. Later we will describe the attributes in our model but for now they remain abstract.

We call the attributes that a task possesses *effective attributes*. The effective attributes of a task are determined by some function combining the attributes of each of the task's sponsors. In symbolic form, the relationship is

$$effattrib(T) = F(S_1(T), S_2(T), ..., S_n(T)) \tag{5.1}$$

where $effattrib(T)$ denotes the effective attributes of task T and $S_i(T)$ denotes the i^{th} sponsor of task T. We call equation 5.1 a *combining-rule*.

5.1.1 Sponsor types

There are four types of sponsors in our model:

1. external sponsors

   These sponsors supply absolute attributes. We discuss later why we call these sponsors external.

2. toucher sponsors

   When one task touches another, the toucher task sponsors the touchee task with the effective attributes of the toucher task. This sponsorship is removed when the touchee task determines its future object (and hence no longer needs sponsorship).

3. ask sponsors

   A task may be sponsored by any other task. The sponsor attributes in this case are the effective attributes of the sponsoring task. Toucher sponsors are merely a special case of task sponsors, except that the addition and removal of toucher sponsors occurs automatically with touch and determine respectively, whereas the addition and removal of task sponsors is always explicit. Task sponsors offer a way, for example, for a parent task to sponsor its children, which is something we expect will be useful.

4. controller sponsors

   The three previous types of sponsors are all passive — they merely act as fixed attribute sources or pass on attributes from other sources. Controller sponsors, by

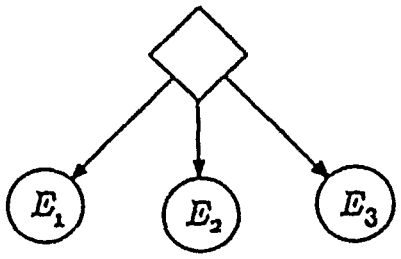


Figure 5.1: por with a controller sponsor

contrast, are active. As their name suggests, they are really controllers or managers. They receive sponsorship and distribute it dynamically among the tasks in their control domain according to some built-in control strategy. The control may reflect changes in sponsorship, changes in the domain tasks, and changes in other resource demands in the system. Controller sponsors implement the complex control that we sometimes want, as perhaps with por, as discussed in Issue 4 in Section 2.1. Figure 5.1 depicts the disjunct tasks of a por with a controller sponsor. The circles are the tasks and the diamond is the controller sponsor. The arrows denote sponsorship. The controller sponsor sponsors each disjunct task according to the sponsor's control strategy. This control strategy decides how to distribute attributes from tasks sponsoring the por to the por disjuncts, such as when a task demands the por result (and hence sponsors the por) and when a disjunct completes or a disjunct spawns another descendant task.

5.1.2 Sponsor networks

The collection of sponsors and tasks for some computation form a *sponsor network*. A sponsor network is a directed graph representing the sponsor and task relationships for some computation. There are two types of nodes in such a network: tasks and controller sponsors. For controller sponsors the node represents the dynamic control embodied by such sponsors. The directed arcs represent sponsorship; they have weights representing sponsor attributes. These arcs emanate from a sponsor source (any of the four types of sponsors) and impinge on a task or controller sponsor. The weights on all the arcs incident on a task node are combined according to the combining-rule to yield the effective attributes of the node. The weights of all the arcs incident on a controller node are combined according to the node's controller policy.

Figure 5.2 shows an example sponsor network. As before, the circles are tasks and the diamond is a controller sponsor. The short horizontal arrows represent external sponsors; the solid arrows represent touch sponsors; and the dotted arrows represent task sponsors. External sponsors are the source of all attributes. These attributes are distributed in a dynamic fashion by passive toucher sponsors according to the dynamic interconnection of

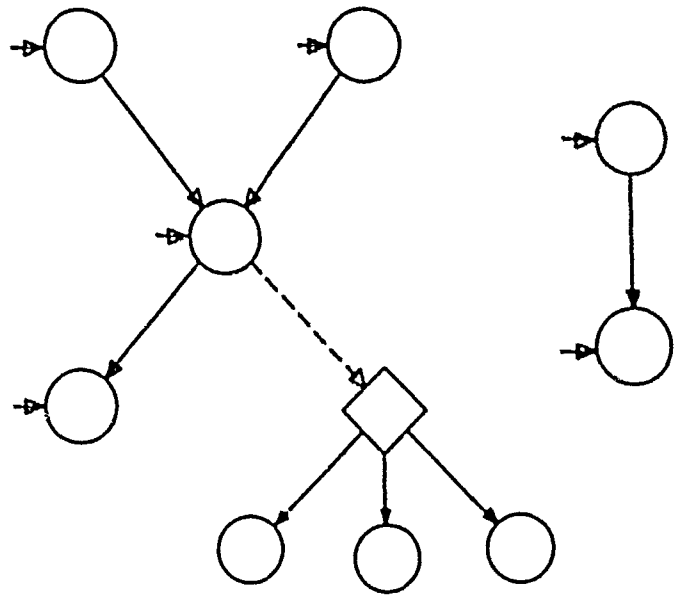


Figure 5.2: A sponsor network

sponsors by touching, by passive task sponsors according to explicit connection, and by active controller sponsors according to their control strategy. The external sponsors can be thought of as external sources of attributes for this sponsor network; hence, the name "external".

A sponsor network is dynamic: during the course of a computation, nodes come and go corresponding to the lifetime of tasks, arcs come and go corresponding to sponsor relationships, and arc weights change as sponsors change their sponsorship. The network may be disconnected, as depicted in Figure 5.2. The network may also be cyclic.

Cycles can arise in two ways. The first way is by a *touch cycle*, in which a chain of tasks all touch each other, forming a cycle of touch sponsors. All the tasks in a touch cycle are deadlocked: each is blocked waiting for the next to produce a result. Deadlocked tasks are always a possibility (unfortunately) in Multilisp — especially in the presence of side-effects[1] — and the user must take care to avoid deadlocks.[2] We consider these deadlocks as pathological, i.e. as errors to be avoided.

The second way that a cycle can arise is by a non-touch cycle of sponsors. The ability to explicitly describe arbitrary sponsor relationships with task and controller sponsors, unlike with toucher sponsors which implicitly follow touching relationships, provides the freedom to directly connect sponsors in cycles. A cycle in a sponsor network is semantically

[1] In the absence of side-effects, deadlocked tasks can only occur through the use of `letrec`.

[2] If the deadlock concerns only irrelevant tasks, the user can simply ignore the deadlock.

meaningless, but nevertheless errors may occur occasionally as the consequence of allowing arbitrary sponsor relationships. We purposely do not want to curtail expressive power by restricting the possible sponsor relationships.

Thus in both cases, a cycle in a sponsor network is an error condition, but we do not want to restrict sponsors to make its occurrence impossible.

5.1.3 Attribute propagation

Whenever a change occurs in a sponsor network, such as a change in connectivity — a sponsor added or removed from a task — or the attributes of a sponsor change, the effective attributes of a task may change (depending on the combining-rule employed). Any change in the effective attributes of a task may in turn lead to changes in the tasks sponsored by that task, and so on. We call this recursive process of updating attributes *attribute propagation.*

The state of a sponsor network is given by its connectivity and the effective attributes at each node. For a fixed connectivity, attribute propagation moves the network state towards equilibrium, where equilibrium is defined as the fixpoint of the equations determined by the connectivity, the combining-rule, and the external sources. (We assume that equilibria exist and are all stable. This depends on the combining-rule and the controller sponsor strategies. We will not address this issue, except in the choice of combining-rule in Section 5.2.1.) That is, attribute propagation solves the network equations in a distributed manner, analogously to a neural network.

For now, we idealize the sponsor network by assuming instantaneous attribute propagation. Thus the network responds immediately to any changes in external sponsors, controller sponsors, or connectivity. Under this view, the network moves from one stable equilibrium state to another and during its sojourn in an equilibrium state the network is static.

With instantaneous attribute propagation, the only divergence, i.e. failure to reach a fixpoint, that can arise is due to connectivity cycles like those discussed earlier.[3] We call this *static* divergence. If attribute propagation is not instantaneous, we can also get *dynamic* divergence, which may occur with or without static divergence. We discuss dynamic divergence in the next chapter in the context of an implementation.

Finally we can see, with the aid of Figure 5.2, what our sponsor model is all about. On one hand we have a collection of external sources supplying attributes which are redistributed by the task and controller sponsors to achieve some desired scheduling. This is the eager component. On the other hand, the top-level task places an external demand on the network for a result which is propagated by the toucher (and also controller) sponsors. This is the demand-driven component.

[3] Since instantaneous attribute propagation already presupposes termination of attribute propagation.

5.2 The Special Sponsor Model

We now consider a specialization of the sponsor model with two different types of attributes. Each sponsor in this model contributes:

1. the desired ordering of the sponsored task with respect to other tasks, and

2. resource limits, such as the rate at which the task may run, the duration the task may run, and the amount of memory the task may use.

Tasks are scheduled for execution according to their ordering and the resource limits apply once the task is executing. That is, a sponsor provides a "ticket" which specifies a place in line and a "permit" which specifies the amount of service that the ticket holder may receive. A task presents its tickets to the scheduler in a bid for service, and if accepted, the task receives resources not exceeding that mentioned in the permits. (We discuss later how a task's possibly multiple tickets and permits from different sponsors are combined.) Note that a sponsor in our model does not actually provide computational effort like the sponsors in Hewitt's model (see Section 3.1). In our model, it is the scheduler which finally provides computational resources, and in the order specified by the tickets. This difference reflects our belief that ordering is the most important control for speculative computation.

For concreteness and without loss of generality, we describe the ordering of tasks by numerical priorities. Thus the "ticket" that a sponsor provides is a priority, which may change with time. For the scope of this chapter, priorities are merely a convenient way of describing the ordering of tasks; they are not necessarily the means by which the user specifies this ordering.

The effective attributes for each task are comprised of an effective priority and effective resource limits A task is scheduled for execution according to its effective priority. A task can preempt all lesser-priority tasks for resources. The effective resource limits are the limits in effect when the task executes.

5.2.1 Combining-rules

Each attribute conceptually has its own combining-rule. We focus on the combining-rule for priority in this section since priority is the most important attribute in our model. We treat priority as orthogonal to the other attributes.

The combining-rule for priority combines the sponsor priorities of a task to yield the effective priority of that task. Let $F_p()$ denote the combining-rule for priority. We want this combining-rule to have three properties.

Demand Transitivity

The first property is demand transitivity. When one task touches another, we want the priority of the touchee to be at least that of all its touchers. If $effpri(i)$ is the effective priority of task i and $touch(i)$ is the set of tasks touching task i, then this formalizes as the statement

$$\forall j \in touch(i) \Rightarrow effpri(i) \geq effpri(j) \tag{5.2}$$

for all tasks i. Demand transitivity is very important. It prevents the desired ordering from being subverted when a high priority task blocks on a low priority task. This property is essential for toucher sponsors. In a sense, all sponsors indicate a "demand" for a task and thus we would like the combining-rule for all sponsor priorities to obey the demand transitivity property expressed in equation 5.2 (where $touch(i)$ becomes $sponsors(i)$, the set of tasks sponsoring task i).

Combining-rule tolerance

The sensitivity of a combining-rule to changes in the sponsor priorities affects the frequency and number of combining-rule updates. Ideally we would like a combining-rule that is somewhat "tolerant" to changes in the sponsor priorities while still having sufficient expressive power, such as obeying Demand Transitivity. The notion of combining-rule sensitivity is best understood by example.

Consider the simple additive combining-rule:

$$effpri(i) = \sum_{j \in sponsors(i)} p(j)$$

where $p(j)$ is the priority contributed by sponsor j. Any change in any sponsor priority causes a change in the effective priority. This change must propagate through the touch network until reaching a terminal node (a task that is not touching another). (We call this *priority propagation*.) This combining-rule is maximally sensitive.

By contrast, with the *max combining-rule*

$$effpri(i) = \max_{j \in sponsors(i)} p(j) \tag{5.3}$$

the effective priority is tolerant of changes in the non-maximal sponsor priorities. Thus many changes in the sponsor priorities can conceivably occur without causing a change in the effective priority and further propagation. The max combining-rule filters out "irrelevant" changes in sponsor priorities.

Convergence in presence of cycles

As stated before, we do not want to outlaw cycles in a sponsor network. The issue then, is the behavior of attribute propagation in the presence of such cycles in the sponsor network. Specifically, attribute propagation should not cycle endlessly and fail to converge.

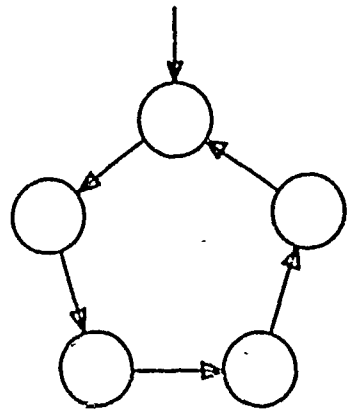


Figure 5.3: Touch cycle with addition combining-rule

This implies that the combining-rule must not be strictly increasing in its arguments. The simple addition combining-rule, for example, is non-terminating. Consider the example in Figure 5.3. Because the addition combining-rule is strictly increasing in its arguments, equilibrium is never attained; the effective priority of each task in the cycle increases without bound. When augmented with some sort of cycle detection, like a counter, this combining-rule is terminating, but then termination does not represent a fixpoint (equilibrium) of the network equations.

The max combining-rule, by contrast, is terminating, assuming instantaneous updates, since it is not strictly increasing in its arguments. Furthermore, termination implies equilibrium (at least locally within the cycle). (See Section 6.1.1 for an example of why the max combining-rule requires instantaneous updates for termination in the presence of cycles.)

5.2.2 Max combining-rule

The max combining-rule, given by equation 5.3, has the three properties given above and two other properties. First, the max combining-rule is the least upper bound combining-rule to satisfy demand transitivity. By least upper bound, we mean that for any combining-rule function, $F_p()$, for priority which satisfies demand transitivity we have

$$F_p(x_1, x_2, \ldots, x_i, \ldots, x_n) \geq max(x_1, x_2, \ldots, x_i, \ldots, x_n)$$

for all priorities x_i (which is just demand transitivity). Thus the max combining-rule never yields a priority higher than its input priorities. Second, the max combining-rule is time-invariant so it is simple to implement. Because of these five properties, we choose the max combining-rule as the combining-rule.

Note that this choice of max combining-rule does not apply to controller nodes since they can have arbitrary combining-rules depending on the node controller policy.

5.2.3 Combining-rules for other attributes

Unlike with the priority attributes, the desired properties for the resource attributes are open to debate, in part because the choice of resources is open to debate. We intend the sponsor model as a general model for speculative computation so we do not want to lock any particular choice of resources and their combining-rules into the model. Indeed, we view the sponsor model as really a spectrum of models with different resources and combining-rules. We feel that rate and duration are important control parameters for any model so we use them as example resources.

We advance the following combining-rule as one possible combining-rule for resource attributes.

Let $S_i(T)$ denote the i^{th} sponsor of a task T. Let $p(i)$ be the priority contributed by $S_i(T)$ and $r(i)_j$ be the limit in resource j contributed by $S_i(T)$ (j = rate, duration, etc.). Finally, let $effpri(T)$ be the effective priority of task T and $effr_j(T)$ be the effective resource limit for resource j of task T. Then the combining-rule is

$$\forall j \; effr_j(T) = r(i)_j$$

where i is the sponsor for which $p(i)$ is maximal. That is, the maximum priority sponsor is essentially the "owner" of the task and supplies the resource limits. This sponsorship is removed when any of the resource limits are attained. For example, sponsor $S_i(T)$ is removed when the duration of task T while "owned" by $S_i(T)$ exceeds $r(i)_{duration}$. The highest priority remaining sponsor then becomes the owner, adjusting $effpri(T)$ as necessary and providing new resource limits.

With this combining-rule, sponsors are really governors, which meter out resources until any of their resources is exhausted. (Some resources, like rates, are not really resources themselves, but a specification on resource use, so they cannot be exhausted like a duration. The metering analogy still applies to these "resources" though.) In this respect, our sponsors are like Hewitt's sponsors, but unlike Hewitt's sponsors, only the top-priority sponsor (of a task) performs this metering. Thus our sponsors have a two-layer hierarchy of ordering and resource metering.

The above combining-rule for resources has the property that the resource consumption of a task is always subservient to the task ordering; a task never receives more resources than dictated by its ordering. For instance, the ordering of tasks to resources is never subverted by the number of sponsors a tasks has. However, we are not ready to cast this combining-rule in stone. One anomaly with this combining-rule is that the effective resource limit can decrease if the effective priority increases: the maximum-priority sponsor could change to one with a smaller resource limit. This anomaly generally poses no problems except for some resources, like rates. It is not clear it makes sense to have a task's effective rate decrease when the task is touched by a higher priority task. One alternative to deal with this anomaly is to sum the resources contributed by each sponsor to determine the effective resource limits, i.e.

$$\forall j \; effr_j(T) = \sum_i r(i)_j$$

5.3 Computation Reclamation and Side-effects

The max combining-rule allows us to handle explicit computation reclamation in a very simple and uniform fashion within our sponsor model. To declare a task irrelevant we merely have to remove its sponsors. If we mistakenly declare a task irrelevant then we automatically re-sponsor it when we later touch it, so there is no problem with deadlock as suggested in Section 2.2.2.

We can even handle reclamation in the presence of side-effects. The problem posed by side-effects is deadlock: a task which may perform a relevant side-effect may be mistakenly declared irrelevant. To solve this problem, we simply have to extend our idea to ensure that our demand for a side-effect ensures sponsorship of the computation that will perform the side-effect. We follow up on this idea in Chapter 7.

5.4 Groups

Controller sponsors are the cornerstone of modularity in our sponsor model. We call a controller sponsor and its collection of sponsored tasks a group. The controller sponsor introduces and manages a new control space for tasks in that group. The local control provided by the controller sponsor insulates the group from outside, and thus provides modularity. The controller sponsor manages:

1. the ordering of tasks within the group (ordering management)

   The controller sponsor defines a new ordering space for the group and manages the ordering within the group with respect to the ordering outside the group. Thus the priority of tasks in a group may be local to that group. Groups may be nested with all local priorities relative to the immediate parent group or to other groups, depending on the group controller sponsor. This gives us the modularity to solve the problems with the traveling salesman problem and the Boyer Benchmark: we can simply create a group for each of these applications with local ordering. These groups can then be embedded in any fashion desired in other speculative computation.

2. the allocation of resources within the group and the distribution of resources from sponsorship outside the group (sponsorship management)

   The controller sponsor receives sponsorship supplying ordering and resource limits (attributes) from outside the group and controls the ordering and the allocation of resources (such as amount, rate, and duration of computation) through the distribution of attributes. Sponsorship management includes the control of the ordering within the group. (The interaction of ordering inside the group with the ordering outside the group falls under ordering management.)

3 the interaction of the group with computation outside the group (interaction management).

Through its ordering of group computation, distribution of resources, and interaction with extra-group computation, a controller sponsor effects a control policy. We let this control policy be arbitrary and depend on any system state. For concreteness, let us assume we can specify the control policy by a program.

5.4.1 Ordering space management

Ordering space management involves translating the group ordering space to fit appropriately in the external ordering space.

For ordering-based speculative computation, such as in the Boyer Benchmark and the traveling salesman problem, the desired ordering management often amounts to uniform translation, i.e. uniformly translating the entire ordering space of the group. In such cases, we want to treat the group as a single point in the ordering space external to the group while still retaining the ordering space within the group. Thus the controller sponsor must merge the group's internal ordering space into the external ordering space and when a group is demanded by a task or another group, the group's ordering space must be translated to fit within the appropriate place in the demander's ordering space. The diagram in Figure 5.4 makes this clear (assuming the ordering is expressed by priorities). To preserve the desired ordering expressed by the priorities, all the priorities in group G_1 must be translated to fit between the priorities of tasks T_{n-1} and T_{n+1} in group G_2. Mere scaling, as would result if the task priorities were actually implemented as relative priorities, is insufficient because there is no way to restrict the range of priorities to be between the priorities of T_{n-1} and T_{n+1} in G_2. Note also that the priority space of G_2 must be managed to allow for future descendants to have priorities between that of T_{n-1} and G_1 and between that of G_1 and T_{n+1}.

Uniform translation, as depicted in Figure 5.4, is not always sufficient for ordering management, even in ordering-based speculative computation. For example, we may not always want the lowest priority task in a group to take precedence over the highest priority task in the next lowest priority group. In this case, ordering management becomes considerably more complicated since the group must be treated as multiple points in the external ordering space. Hence, different parts of the ordering space may be translated differently and interaction with other controller sponsors may be necessary to coordinate the overall desired ordering. See Section 5.4.3 on interaction management.

5.4.2 Sponsorship management

For multiple-approach and precomputing speculative computation we require active control of ordering and allocation of resources. For multiple-approach speculative computation, we must be able to allocate resources amongst the multiple approaches, controlling the relative ordering, rate, and duration of approaches. For precomputing-based speculative computation we must be able to control the resources we expend on precomputing versus the resources to perform the main-stream computation (and versus the resources to precompute

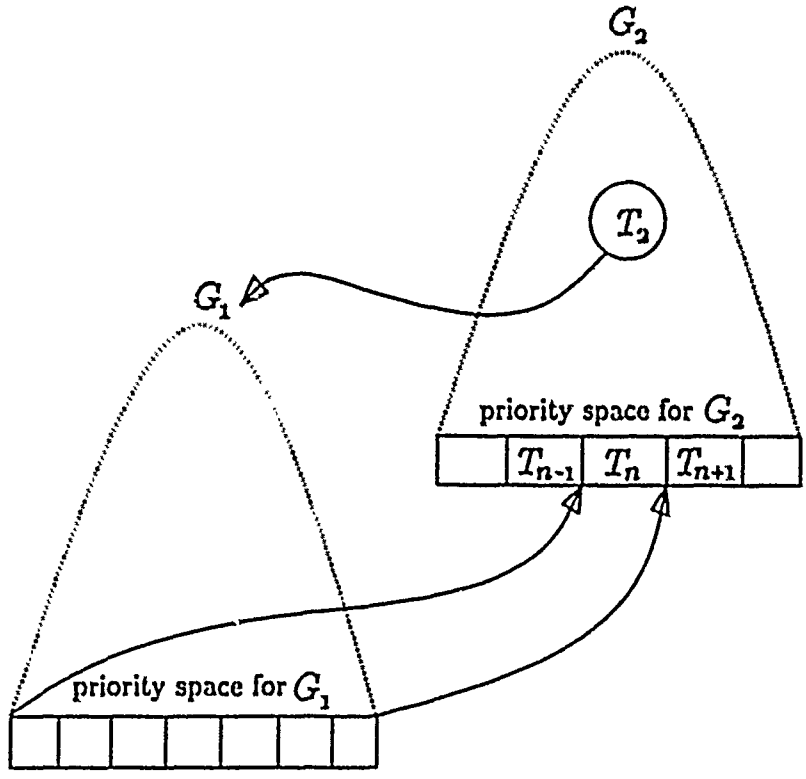


Figure 5.4: Uniform translation of group ordering

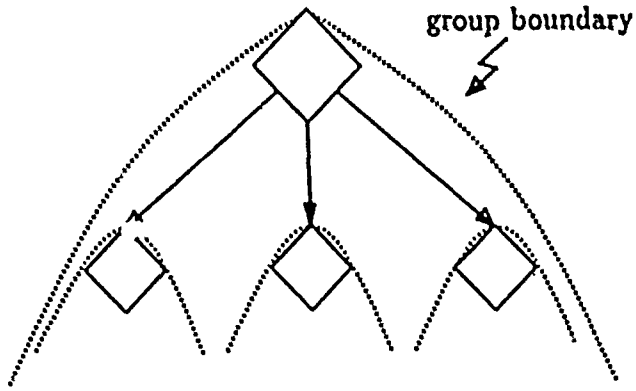


Figure 5.5: An example of por with groups

different quantities.)

As an example, consider (por E_1 E_2 E_3). We want this por contained in a group for two reasons. First, we want modularity — we want to be able to use this por anywhere without concern for its internal structure. Second, we want the dynamic control provided by a controller sponsor to distribute any changes in sponsorship for the por amongst the disjuncts and to possibly reallocate sponsorship amongst the disjuncts as disjuncts complete. Let us assume that each disjunct, E_i, comprises several tasks. Then we might also like modularity for each of the disjuncts so we can control each disjunct independent of the others. Figure 5.5 illustrates por with a group for the overall por and a group for each disjunct.

5.4.3 Interaction management

The exchange of information between controller sponsors is sometimes necessary for proper allocation of resources. Consider the following two examples.

Multiple controllers

Suppose a group may be sponsored simultaneously by multiple controllers, i.e. the group may be in the control domain of multiple controllers. Then problems can arise if the the allocation of resources amongst the activities sponsored by a group is critical and that group's controller is unaware of the influence of another controller within its control domain. In this case, the other controller may disrupt the intended resource allocation.

As a specific example, consider two por groups sharing a disjunct, as illustrated in Figure 5.6. Disjunct E_2 is a group, with its own controller, which is in the control domain of both por controller sponsors. If both por controllers are unaware of their joint sponsorship of E_2, they both may decide that it is best to allocate all their resources to evaluating E_2.

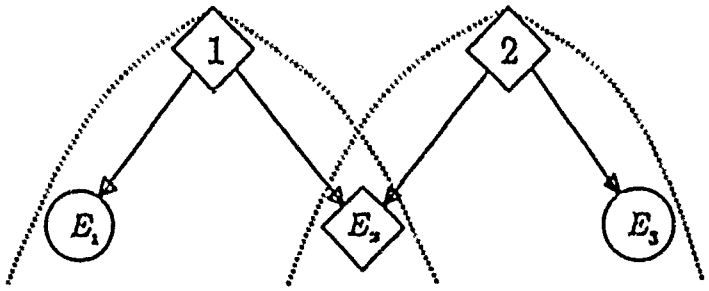


Figure 5.6: Two pors sharing a disjunct

However, if we use the combining-rule described earlier, one of these controllers becomes the "owner" and contributes all the resources, while the other one sits idle. With coordination, controllers 1 and 2 may agree to sponsor say E_2 and E_3, respectively.

Thus sometimes feedback is necessary between the controllers allocating resources and the activities consuming resources so the controllers can compensate for multiple sources.

Nesting

Suppose we have (por E_1 E_2) and E_2 turns out to be another por (say (por E_3 E_4)). We would like the controller sponsors of these two pors to coordinate their allocation of resources so the net effect is no different than that in (por E_1 E_3 E_4). Similarly, if we have nested pors and pands, we would like the respective controller sponsors to communicate. Strict modularity — i.e. modularity without interaction of controller sponsors — does not yield optimal performance in general. See, for instance, Example 4 in Section 9.4.

These examples show that sometimes we need communication and cooperation between controller sponsors. To support such communication and cooperation, a controller sponsor must be able to supply information to external agents about its resource requirements as well as consume resources. This information might take the form of the expected time until completion given a certain amount of input resources. See the scheduling problem formulation in Section 9.1.2 for example.

5.4.4 Groups as black boxes

As Figures 5.4 and 5.5 hint, a group is intended to encapsulate a single activity — a collection of related tasks working towards a common goal — which produces a single result. The controller sponsor is the gatekeeper by which the activity is demanded by other activities; it is the interface between the ordering space of the group and other ordering

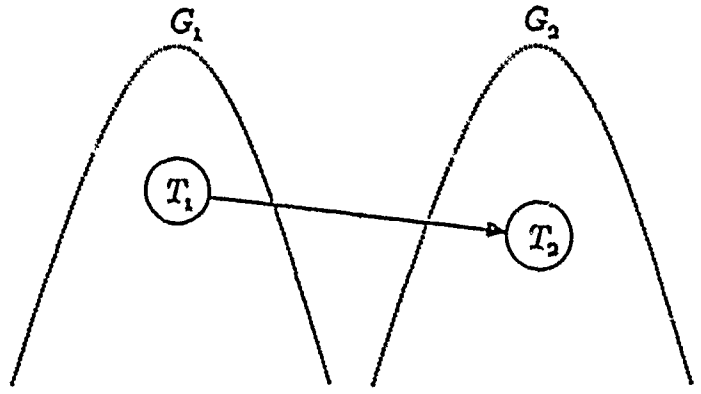


Figure 5.7: Non-root touch interaction

spaces. Or put differently, the controller sponsor makes an activity appear as a black box to activities outside that group, thus enforcing modularity. While this is the way in which groups are intended, it is not entirely realistic due to two problems: non-root group interaction and partial results.

5.4.5 Non-root group interaction

We refer to the controller sponsor as the "root" of a group. If all interaction with a group is through the root, everything is fine: the controller sponsor can propagate sponsorship in some appropriate manner to the members of the group.[4] This interface interaction is enforced only by programmer discipline. We do not force all interactions to occur via the root since it may be useful occasionally to break this abstraction barrier. This leads to possibility of non-root interaction with a group.

We distinguish two types of non-root interaction: single control domain interaction and multiple control domain interaction.

Single control domain interaction

In this type of interaction a task within a group (and thus within the control domain of that group) receives sponsorship from an external sponsor, or a toucher or task sponsor outside the group. Figure 5.7 illustrates an example. Task T_1 in group G_1 is touching task T_2 within group G_2. Control domain interaction can destroy modularity. The ordering of a task is no longer entirely local within a group. In Figure 5.7, the local ordering established within

[4]Or rather, it is the responsibility of the controller sponsor to distribute sponsorship in some appropriate manner within the group.

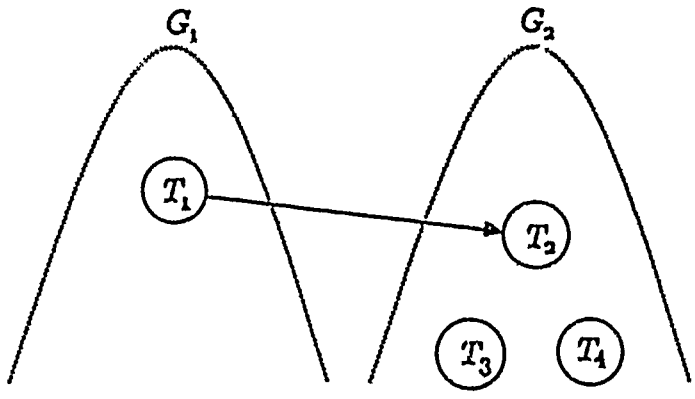


Figure 5.8: Non-root touch interaction

group G_2 can be subverted by the toucher priority contributed to T_2 from T_1. One can argue that this is the natural and expected consequence of non-root interaction and thus such interaction should be avoided if one wants to maintain modularity. At least demand transitivity is maintained by the max combining-rule.

A serious problem arises, however, if a task receiving sponsorship from outside its group spawns descendants. The problem is how these descendants receive sponsorship and how they are controlled. Normally when a task is first spawned within a group it receives sponsorship from the group controller sponsor. Later, the task may also be sponsored by touch and task sponsors, but the task must be spawned first. (The task may also be sponsored by an external sponsor but that reverts the situation to that discussed in the previous paragraph.) However, with non-root interaction, we could end up with a situation where the group has no sponsorship with which to sponsor a task spawned within the group. Consider, for example, the following scenario involving the non-root interaction in Figure 5.7.

Suppose that group G_2 is, one of the disjuncts E_i in the por in Figure 5.5 and suppose that this disjunct is r ired for the por (some other disjunct returns first). The por controller sponsor thus es sponsorship for E_i and thus all the tasks in group G_2 stop executing. Now suppose t task T_1 in group G_1 touches task T_2 in group G_2, thus sponsoring T_2 and causing it to restart. Further suppose that task T_2 spawns several tasks, say T_3 and T_4, to help compute its result. Figure 5.8 shows this situation.

Who sponsors tasks T_3 and T_4? It cannot be their group, G_2, because it has no sponsorship to offer. They could remain unsponsored until they are required by T_2, at which time they will be touched and thus sponsored. However, this is not a good solution since it fails to exploit available parallelism. Another possibility is for T_2 to sponsor T_3 and T_4 via task sponsors. However, this only works if the non-root interaction involving T_2 was anticipated. (Ordinarily, the group sponsor might sponsor T_3 and T_4 for T_2.) The obvious solution is to for group G_1 to sponsor these tasks. (We reject external sponsors for the reasons given

previously.)

In fact, we could put these tasks in group G_1. Even though these tasks may be related to the other tasks in G_2, we could argue that there is no harm in putting them in group G_1 since the abstraction barrier of G_2 has already been blurred by the first interaction in touching task T_2. However, this idea is stymied by two possibilities. First, group G_1 could lose its sponsorship and G_2 could regain its sponsorship. Then we would have the original problem again: who sponsors the descendants of T_3 and T_4? Second, there may be many tasks, of many different groups, touching task T_2. In which of these groups do we put T_3 and T_4?

To address these problems, we make two observations:

1. Tasks T_3 and T_4 must remain sponsored by group G_2.

   Even though G_2 may have no sponsorship to offer at present, G_2 could receive sponsorship at some point in the future when there may be no other sponsorship for T_3 and T_4. (Group G_1 could have lost its sponsorship.) G_2's sponsorship of T_3 and T_4 does not preclude sponsorship by others.

2. Tasks T_3 and T_4 must be sponsored by every other group touching T_2 (for the same reason as in point 1).

Thus every group G_i which sponsors a task like T_2 in a group must sponsor all the descendants of that task.

This means that one task, like T_3, could be in multiple control domains, which brings us to the next type of non-root interaction. We could arrange, via the max combining-rule, for exactly one of the groups G_i sponsoring T_2 to be the "owner" of all the descendants of T_2 at any point in time. Thus all descendant tasks would effectively be within only one group and thus one control domain at any point in time. This group would always be the maximum-priority sponsoring group. This arrangement is too complicated, however, since still other groups can touch the descendant tasks.

Multiple control domain interaction

In this type of interaction a task within the control domain of one group receives sponsorship from a controller sponsor outside the group. This puts the task within the control domain of multiple controllers. Although this interaction has the same flavor as with the multiple controllers described in Section 5.4.3, it is different in a crucial regard. The interaction here involves tasks in multiple control domains, not groups in multiple control domains, as before. The interaction here is beneath the granularity of groups and thus the tasks have no controller sponsors to sort out the multiple control sources.

Multiple control domain interaction at the task level does not necessarily cause any problems because the sponsorship of the multiple controllers is duly combined according to the combining-rule. However, as with groups, problems can conceivably arise if the

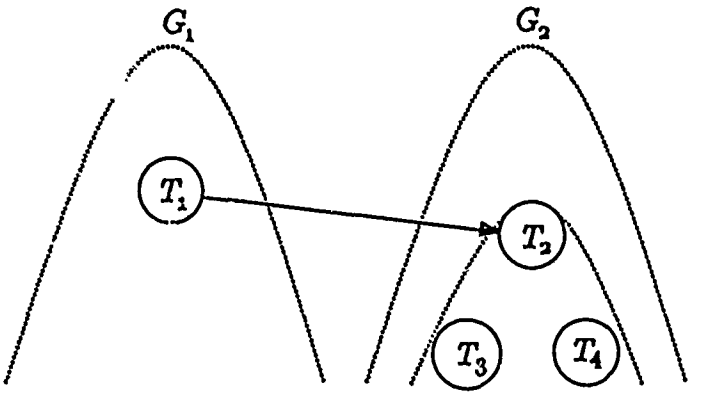


Figure 5.9: Group splicing

allocation of resources amongst tasks in a group is critical since another controller may disrupt the intended resource allocation.

A solution

The best solution, of course, is to arrange for non-root interaction ahead of time and make it root interaction by providing controllers at the interface points, i.e. subdividing the groups. Obviously this takes some foresight and thus is not always practical, but we cannot expect arbitrary control with arbitrary group interaction.

The next best solution, for otherwise impractical cases (and as a "safety net"), is to create a new group dynamically when a non-root interaction first occurs. For example, when task T_1 in Figure 5.7 touches task T_2 in group G_2, we create a new group rooted at T_2 and move T_2 and all its descendants from G_2 to this new group. Figure 5.9 illustrates this procedure, which we call *group splicing*.

Group splicing attacks the problem of non-root interaction by converting non-root interaction into root interaction. By automatically providing a group for non-root interaction, this solution provides:

1. A group for all the descendants of a non-root sponsored task.

   We argued earlier that this is essential. Group splicing is less complicated than the group owner solution described earlier.

2. A controller for interfacing non-root tasks to other control domains.

   This takes care of the problems with non-root multiple control domain interaction.

There are still problems with group splicing, however. One problem is specifying the controller sponsor for the new group created by group splicing. Another problem is that

all the descendants of a non-root interface task are moved into a new group. Sometimes a different splicing strategy may be more appropriate. We are not very concerned by these problems, though, since non-root interactions should not be encouraged in the first place; we are merely trying to provide some reasonable action when they do occur.

Non-root interaction occurs because a task escapes the control domain represented by its group. This containment escape occurs in two ways: by not returning results through the root (as may occur, for example, with side-effects) and by returning partial results.

5.4.6 Partial results

The result that a group returns may be just a partial result, a skeleton to be filled in later. For example, a por group might return a placeholder (i.e. future object) for the winning disjunct or a group to sort a list may return the head of the list and a placeholder for the rest of the list. How can we return a partial result while still sponsoring any subresults?

One way to handle partial results is to simply unsponsor all the tasks in the group when the group returns a result, even a partial result. Thus all computation on any subresults stops until the subresult is actually demanded by some task external to the group. This method, which we call lazy result return, circumvents the issue of maintaining sponsorship of subresults after the group returns a result. The obvious lack of parallelism with this method makes it unacceptable. Furthermore, in turning eager computation into lazy computation at result return time, this method operates counter to conventional, mandatory-style Multilisp.

Thus we must actually continue to sponsor subresults once a group returns a result. This eager result return raises the issue of non-root interaction. A task outside the group could touch one of the subresult placeholders, causing non-root interaction. This poses all the problems discussed in the previous section, like loss of modularity within the group. To prevent these problems, we must somehow convert possible non-root interaction to root interaction.

There a number of different approaches that we could take.

One approach is to design group interfaces so that groups have no subresults, i.e. so there is no export of unfinished computation from a groups. For instance, one could argue that in the list sort example both the list sort and list consumer should be in the same group. This approach is not always possible, such as if many different activities consume the sorted list.

Another approach is the group splicing discussed earlier. Whenever a subresult is demanded, its task and all its descendants are moved into a new group. This would take care of the list sort example — when first touched, each list element would get a new group by group splicing. Performing this group splicing could be quite expensive.

To do better we need to impose more structure on the group result. Instead of returning an arbitrary result, a group could return a group object consisting of the result and a controller sponsor for the subresults. Any sponsorship for one of the subresults, by derefer-

encing the group object to get a subresult placeholder, would be directed to the controller, rather than to the subresult. For instance, in the por example given above, por would return both a placeholder and controller sponsor for the disjunct. Then the sponsorship represented by any touch of the placeholder would be directed to the disjunct group.

This approach packages the information needed for continuing to sponsor the subresults in the result itself so that only the group needs to know the structure of its result. We cannot expect the result consumer to know the structure of the result beforehand, except in special circumstances.

5.5 Summary

In this chapter we proposed sponsors as a means of controlling and allocating resources for speculative computation. Sponsor networks provide both eager and demand-driven components to scheduling. A key idea of the sponsor model is groups for organizing related activities for naming and control purposes. This idea of groups is similar to the groups in Mul-T [Kranz] (which we called units in Section 4.4), but our groups incorporate active control agents and can be nested.

The sponsor model provides a way to view speculative computation and a framework for the control of speculative computation. The actual control that a sponsor network should provide remains an area for future work.

Chapter 6

The Touching Model

In this chapter we describe a subset of the special sponsor model, which we implemented, called the touching model. Touch sponsors play the central role in this model, hence the name. The touching model also has external sponsors and a very primitive type of controller sponsors, but no task sponsors. The only attributes in this model are priorities, which we use to effect the desired ordering of tasks: there are no resources or resource limits in this model. This touching model is purposely very simple to serve as a quick prototype in exploring and evaluating the sponsor model idea for speculative computation.

6.1 The Touching Model

Each task in the touching model has

1. a single external sponsor which provides what we call the *source* priority of the task, and

2. an effective priority.

The external sponsor, toucher sponsors, and controller sponsors of a task T combine to determine the task's effective priority according to the max combining-rule, as described by the following formula:

$$effpri(T) = \max(sourcepri(T), \max_{j \in touch(T)}(effpri(j)), \max_{k \in controllers(T)}(cntlpri(k,T)))$$

where $effpri(i)$ is the effective priority of task i, $sourcepri(i)$ is the source priority of task i, $cntlpri(i,j)$ is the priority that controller i contributes to task j, $touch(i)$ is the set of tasks touching task i, and $controllers(i)$ is the set of controller sponsors for task i. We describe a primitive form of controller sponsors in Section 6.2.

All priorities are in the interval $[0, MAX]$.[1] Tasks are scheduled for execution according to their effective priority: only runnable tasks of the highest effective priority run at any point in time. Tasks with effective priority 0 are not runnable; but their computational state is retained (until it becomes inaccessible and is garbage-collected) and thus the task may be restarted if its effective priority becomes non-zero.

6.1.1 Priority propagation

The sponsorship of a task may change due to a change in existing sponsors, the addition of new sponsors (by touch or the addition of a controller sponsor), and the removal of existing sponsors (controller sponsors only). Whenever such a change occurs, the effective priority of the task is updated if necessary according to the max combining-rule. Any change in the effective priority of this task may require a change in the effective priority of any tasks sponsored by this task, and so on, down the *sponsor tree*. (Chains of toucher sponsors — which we call *touch chains* — form the branches of a tree and controller sponsors serve as the forks in the tree.) We call this recursive process of updating priority *priority propagation*.

Priority propagation occurs implicitly in response to a change in sponsorship. For example, whenever a task touches another, the toucher implicitly sponsors the touchee and implicitly propagates any change in priority. Unlike in the idealized model in Chapter 5, priority propagation in the touching model is not instantaneous: determining if a change in sponsor priority changes the effective priority and updating the effective priority, if necessary, takes nonzero time. We perform priority propagation eagerly, propagating any changes immediately until termination, and in a depth-first manner. This eager priority propagation is blocking: a computation which (implicitly) initiates priority propagation cannot continue until the priority propagation terminates.

With non-instantaneous priority propagation we get two sources of (dynamic) divergence, i.e. failure to reach a fixpoint of the priority equations:

1. New changes in priority may occur faster than they can be propagated so the network priority equations are always unsatisfied. By new changes, we mean the origination of a change in priority, not the propagation of some past change. A change in priority originates at a node n due to a change in source priority for node n, a change (due to some change in sponsor policy) originating in a controller sponsor sponsoring node n, or a change in connectivity affecting node n — either some task touching node n or the addition (removal) of a controller sponsor to (from) node n.

2. Priority propagation may not terminate.

The first type of divergence results in a certain lack of control: the constant disequilibrium means that control decisions may be based on misinformation and the inability to track new changes means that control decisions may be obsolete (and incorrect) by the time

[1] Conceptually, speaking. See Section 10.1.

they propagate to the appropriate parts of the network. Eager priority propagation limits this damage by blocking the progress of an activity originating a priority change until that change is fully propagated. Thus the rate of priority propagation bounds the rate of new changes. This advantage is the reason we chose eager priority propagation.

The second type of divergence is especially a problem with eager priority propagation, although it can also happen with "lazy" priority propagation. (With lazy priority propagation we might only perform a little propagation at regular intervals of time and still never terminate the propagation.) Non-termination of eager priority propagation causes two problems: the activity originating a priority change is blocked forever and resources are committed forever to propagating the change. We call this type of divergence *priority propagation divergence.*

Priority propagation divergence

In the touching model, priority propagation divergence occurs due to cycles and "run-away create and touch".

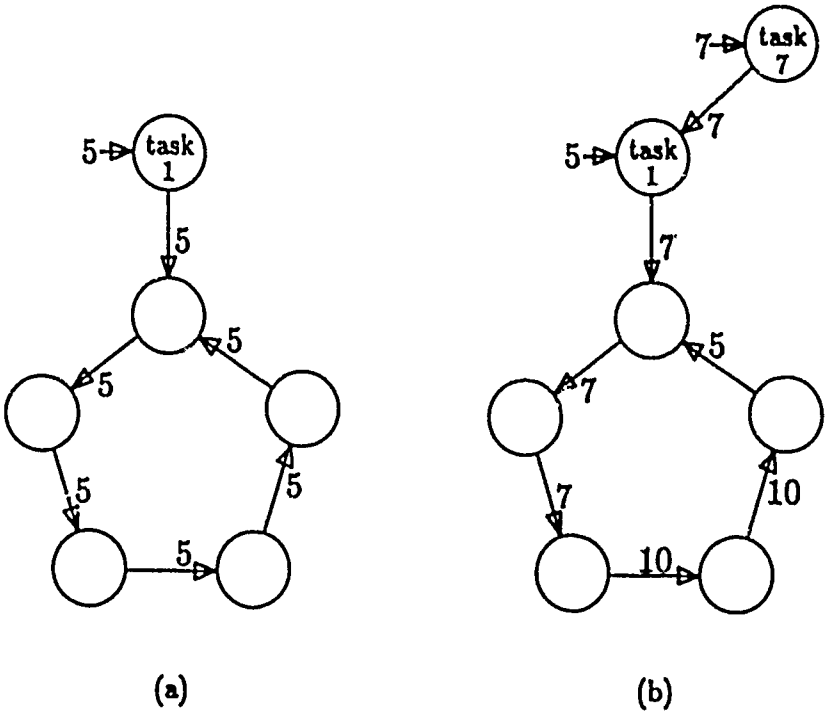
Cycles

For the touching model, the max combining-rule does not ensure termination of priority propagation in the presence of cycles due to the non-instantaneous update of priorities in the touching model. If perturbations in priority occur before full adjustment to an initial perturbation, we can get non-termination with the max combining-rule.

Consider the scenario in Figure 6.1. First the network as shown in Figure 6.1(a) is in equilibrium. Then the effective priority of task 1 changes from 5 to 10, say due to task 7 touching task 1. This then causes a change in priority to begin propagating around the cycle. Suppose, though, that before this priority propagates entirely around the cycle the effective priority of task 1 changes again, decreasing to 7 due to a decrease in the source priority of task 7. This yields the situation depicted in Figure 6.1(b) with two waves of priority changes propagating through the cycle. If both waves propagate at the same speed, they continue to chase each other forever and never reach equilibrium.

Note that such non-terminating behavior with the max combining-rule requires:

- A directed cycle in the touching network
- Two or more propagations active simultaneously within the same cycle. At least one of these propagations has to increase the effective priority and at least one has to decrease the effective priority.
- Each of the propagations to occur with the same speed. In fact, there might be sizable pressures for the propagations to synchronize and move in lock-step due to locks that may be required to change the effective priority of tasks.



Source priority of all tasks is 3 unless otherwise indicated.

Figure 6.1: Non-termination with max combining-rule

Presently we do not worry about cycles in the touching model. As mentioned in Section 5.1.2, a cycle is an error condition in the first place. However, there should be a reasonable policy for handling such errors to avoid infinite cycling of priority propagation. One way to handle cycle errors is to record the number of nodes encountered in a sponsor chain during priority propagation. Then if the number of nodes exceeds some threshold, terminate the priority propagation. This "bounded eager" priority propagation prevents infinite cycling of priority propagation in the presence of cycles and can even lead to equilibrium in cycles, as well as termination. Equilibrium arises when the first $n-1$ propagators in a cycle terminate: the n^{th} propagator will restore equilibrium (assuming it is not terminated prematurely by the exceeding the node threshold) and terminate.

Run-away create and touch

The other way that priority propagation can fail to terminate in the touching model is for a touch chain to grow faster (or as fast as) the rate of priority propagation down the chain. For this to happen we have to continually create and touch new tasks. For example, task i creates and then touches task $i+1$, and task $i+1$ creates and then touches task $i+2$, and so on, forever. We call this run-away create and touch. We regard it as a pathological condition with low probability since it requires a non-terminating computation. In fact, we ignore it.

6.1.2 Computation reclamation

The sponsor model provides an elegant framework for explicit computation reclamation. To declare a task irrelevant we just have to remove its sponsors; an unsponsored task does not run. Moreover, this reclamation is reversible since the task may resume whenever responsored.

We realize this method for explicit reclamation in the touching model as follows. To declare a task irrelevant we set the source priority of the task to 0 and propagate any resultant change in the task's effective priority. We call this process *staying*. If the effective priority of a task is 0, the task is not runnable (as described earlier); we say such a task is *stayed* — its computational resources have been reclaimed. Note that *staying* a task ("declaring the task irrelevant") does not imply that the task will be *stayed* ("considered irrelevant") since the task could still have toucher or controller sponsors, indicating that the task is relevant from the viewpoint of other tasks demanding its value. A stayed task is unsponsored (setting a sponsor priority to 0 accomplishes the same thing as removing the sponsor). Such a task may be restarted by responsoring it, i.e. by making its source priority non-zero, by touching it with another (running) task, or by giving it a non-zero controller sponsor — anything which makes its effective priority non-zero. Table 6.1 summarizes these terms and conditions.

This method of explicit reclamation fits very nicely in the touching model; no new mechanisms are required. One very important feature which falls out of the touching model

Informal term	Technical term	Action or condition
declare irrelevant	to stay	set source priority to 0 and propagate
declaring irrelevant	staying	process of setting source priority to 0 and propagating
considered irrelevant	stayed	effective priority = 0

Table 6.1: Reclamation terminology

framework is that reclamation is reversible.

Finally, we see that the run-away task phenomenon is particular form of dynamic divergence. We mention this phenomenon again in Section 6.2.

6.1.3 Priorities

The choice of priority to effect ordering was driven by pragmatics. Ideally we would like a more direct (and declarative) means of specifying the desired ordering information. Unfortunately, it is not yet clear, except in special cases — such as first-in-first-out (FIFO) ordering (or similar orderings based on task creation time) — how to concisely describe the desired ordering.

Priorities are very simple and allow a great deal of flexibility in expressing desired orderings. For example, if tasks are allocated sparsely enough in the priority space, a priority range can be associated with each task rather than a single priority. This range may then be subdivided amongst the children of the task to obtain nested orderings. We follow up on this idea in Section 6.4.3.

There are two problems with introducing priorities at the user level.[2] The first problem, as alluded to above, is that priorities at the user level increase the imperativeness that the user/programmer has to deal with. The second problem is that priorities themselves lack meaning; they are too abstract. They can only be understood operationally by changing them and observing the effect(s) on program execution. In some instances it may be possible to hide these problems from users with suitable user interfaces, effectively pushing the imperativeness and lack of meaning into the domain of the library or interface author.

6.2 Language Features

Our aim was to provide basic primitives for speculative computation in the spirit of the Multilisp model. The result is a relatively low level substrate that users can build on to provide whatever interface they wish and deem appropriate for their application.

[2]There is no objection to using priorities strictly within the implementation, perhaps to implement some ordering mechanism.

As discussed in Section 1.5, we had two important goals in this implementation. The first goal was to retain the `future` construct and thus retain the spirit of the Multilisp model. The second goal was to minimize the impact of our support for speculative computation on the performance of conventional styles of computation. To accomplish this second goal we distinguished *mandatory tasks* and *speculative tasks*. Mandatory tasks always have fixed effective priority MAX (conceptually speaking — see Section 10.1) and cannot be stayed. Speculative tasks can have any effective priority $\in [0, MAX]$ and may be stayed. Thus mandatory tasks correspond to the tasks in conventional styles of computation and speculative tasks correspond to the tasks in speculative styles of computation. However, these are merely operational terms and do preempt our definitions in Section 1.1: a mandatory task may perform speculative computation and a speculative task may perform mandatory computation. Because mandatory tasks cannot be stayed and cannot be preempted, they can be implemented more efficiently than speculative tasks.

The top level, read-eval-print-loop, task is a mandatory task. All the children of a mandatory task are also mandatory tasks, provided that the children are created with `future`.

(`spec-future` *exp pri*) creates a speculative task with source priority *pri* to evaluate *exp* and immediately returns a placeholder for the result, just as `future` does. All the children of a speculative task are also speculative. If a speculative task executes `future`, the child task is a speculative task with its source priority inherited from its parent.

Orthogonal to this notion of inheritance is the notion of contagion. If a mandatory task touches a speculative task, the speculative task becomes mandatory.

(`make-group` *exp pri*) is the same as `spec-future` except it returns a group object. A group in this implementation is merely the name for a `make-group` task and all its descendant tasks, i.e. the tree of tasks rooted by a `make-group` task. There is no controller sponsor associated with these groups and thus these groups lack modularity. (The addition of full controller sponsors is a project for future work.)

(`stay-group` *group*) performs the staying operation on all members of the group *group*, i.e. it stays all group members. It returns an unspecified value. Since a group includes the `make-group` task and all its descendants, `stay-group` automatically names and stays all descendants. This dynamic naming of of descendants eliminates the difficulty of explicitly tracking descendants and solves the problem with tracking descendants spawned by unknown function calls. `stay-group` essentially integrates the "killer tasks" idea of Grit and Page [Grit], mentioned in Section 3.2, into the touching model.

We mentioned earlier how the run-away task phenomenon is a particular form of dynamic divergence. We prevent the run-away task phenomenon by suspending the creation of any new tasks in a group while any staying is in progress in the group. We discuss the details in Section 10.4.

(`my-group-obj`) returns the group object representing the executing task's group. A task's group is always the task's newest ancestral group.

(make-class *class-type*) creates and returns a class object. A class is collection of tasks and a sponsor in the fashion of the groups mentioned in Chapter 5. Unlike with the group created with the make-group construct, the members of a class are arbitrary and not necessarily all descendants of a common parent. We have implemented three types of classes, each of which corresponds to a different type of primitive controller sponsor:

1. class-all, in which the class sponsor sponsors all the members of the class,
2. class-any, in which the class sponsor sponsors an arbitrary member of the class, and
3. class-pqueue, in which the class sponsor sponsors only the top-priority task in the class.

The class sponsor receives its sponsorship from one or more sponsors (see make-future below) combined according to the max combining-rule.

(add-to-class *obj class*): if *obj* is an undetermined future object f or a class object c, then add-to-class adds either the task associated with f or the class object c to the class *class*. Otherwise, add-to-class does nothing. It returns an unspecified value.

(remove-from-class *obj class*) functions like add-to-class but instead removes *obj* from the class *class*. In addition, when a task terminates, it is automatically removed from any classes to which it belongs.

We implemented these two primitives by "touching" and "untouching," respectively, the given task (or class) with the class sponsor of the given class. Thus classes are a relatively straightforward extension of the touching mechanism. Section 10.3 describes details.

We added an optional argument to the make-future construct described in in Section 1.4: (make-future &optional *class*) creates and returns a placeholder and sponsors the class *class* (if specified) with the maximum-priority task blocked on the placeholder. This sponsorship is removed when the placeholder is determined.

Chapters 7 and 8 provide examples demonstrating and discussing the use of these constructs. Appendix A presents further details on these constructs.

6.3 Deficiencies

All the deficiencies of the touching model result in one way or another from the lack of full controller sponsors. We did not implement full controller sponsors mainly because we were not sure what form the controller sponsors should take. We also had reservations about their run-time cost and implementation difficulty. We wanted to keep the first implementation simple so we could implement it quickly and spend ample time experimenting to assess its merits and deficiencies. We originally planned a second implementation (see [Osborne]) to incorporate our findings but the first implementation and experimentation took much

longer than we expected. In the remainder of this section we concentrate on the major deficiencies we found.

The biggest deficiency in our touching model is lack of modularity: neither our groups nor our classes with primitive controller sponsors provide modularity. Class sponsors merely distribute sponsorship to a collection of tasks without maintaining any local ordering amongst the tasks. We can still do a lot without modularity, as the examples in Chapter 8 demonstrate, provided that there is only one speculative activity in the system. This may be acceptable for a prototype, but is certainly unacceptable for a production system. What we said above for controller sponsors applies here too. Plus, we did not appreciate the importance of modularity when we began this work.

The next biggest deficiency is inadequate treatment of group interaction. Because of the primitive controller sponsors and lack of modularity, this is only a problem with staying. Let us present this argument before discussing the problems with staying.

We have two types of "groups" in the touching model. The first type is that created with `make-group`. These groups lack controller sponsors of any kind. The root of such a group is the `make-group` task. Thus root interaction, such as touching or class-sponsoring the group, amounts to just touching or class-sponsoring the `make-group` task, which is no different than any other task. The interaction is not distributed to any other tasks in the group because the group has no controller sponsor and thus lacks modularity. If the `make-group` task requires the results of other tasks in the group to produce the group result, then the task must specifically demand the results by touching or class-sponsoring the other tasks — there is no eagerness component aside from the source priorities. Non-root interaction is no different from intra-group interaction. Since these groups have no controller sponsors and hence no control domain or modularity, inter-group sponsoring of non-root tasks does not cause control domain interference and does not destroy modularity.

The second type of "groups" in the touching model are classes. Classes are basically stripped-down versions of the full-fledged groups discussed in Chapter 5. The root of a class is the class sponsor. The primitive controller sponsors available as class sponsors preclude modularity but do serve as "group" interface points, allowing root "group" interaction: class sponsorship is distributed to the class members according to the class type. As with `make-group` groups, non-root interaction with classes is no different than interaction with any other task and thus presents no difficulties.

Thus root and non-root interaction present no problems with either type of "group" since controller sponsors are either non-existent or too primitive to provide modularity or real control domains — no problems, that is, except for the interaction of `make-group` groups while staying. Intergroup touching and class-sponsoring creates confusion over who is responsible for staying tasks.

Consider the following example. Suppose we stay group G_1, thereby setting the source priorities of all the tasks in G_1 to 0, and as a result all the tasks in G_1 are stayed. Now suppose task T_2 in group G_2 touches task T_1 in G_1, thus sponsoring T_1 and causing it to become unstayed. Who is responsible for staying T_1 should G_2 be stayed? Obviously T_2

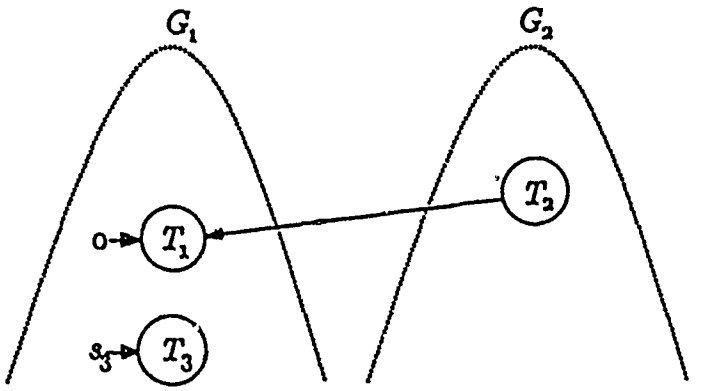


Figure 6.2: Intergroup touching in presence of staying

since it is the one sponsoring T_1. Indeed, when T_2 is stayed, its effective priority of 0 will be propagated to T_1 and T_1 will be stayed too. But now suppose after being touched by T_2, T_1 spawns a task T_3 with a non-zero source priority s_3, as illustrated in Figure 6.2. Who is responsible for staying T_3 should G_2 be stayed? Now we can argue that either T_1 or T_2 should be responsible since T_1, directly, and T_2, indirectly, caused the creation of T_3. However, neither of these tasks are sponsoring T_3: T_3 is sponsored entirely by its external sponsor providing its source priority. Thus T_3 will not be stayed as a consequence of staying T_2 or T_1. (T_3 is a descendant of T_1 and hence a member of G_1 and not G_2.)

The problem is there is no mechanism to remove the source priority of T_3. The obvious solution is to eliminate this source priority and sponsor T_3 by T_1 or T_2. However, this is unsatisfactory. T_3 must be able to run even if T_2 never touches T_3. Thus at the least, T_1 must sponsor T_3. Now how does T_1 sponsor T_3? Unless T_1's code calls for touching T_3, the only way to sponsor T_3 in the touching model is by a class sponsor. However, T_1 has no way to know, in general, that it will be touched and thus has no incentive to put T_3 in a class sponsored by T_1.

(The ideal way to solve this problem is to sponsor T_3 by G_1's controller sponsor and then splice a new group out of G_1 rooted by T_1 when T_2 touches T_1. However, we do not have full controller sponsors.)

Another solution is to have all the descendants of a task like T_1 be a member of all the groups touching T_1. Thus staying any of the groups will surely stay descendants like T_3. This is overkill, however. If all the descendants of T_1 are required, then staying G_2 would needlessly destroy any ordering established by the source priorities of T_1's descendants. This is really example of the multiple control domain interaction problem.

The solution we adopted is to have each task "owned" by the group supplying the maximum priority sponsor. The effective owner of a task, until sponsored by a toucher or class sponsor, is the effective owner of its parent at the time of the task's creation.

Staying a group now involves staying all that group's effectively owned tasks as well as direct descendants. This solves the "unstayable" descendants problem, illustrated in Figure 6.2, and the overkill problem. However, making the effective owner that of the parent at creation time is an arbitrary choice that leaves further problems. We leave these problems for the reader to ponder. There seems to be no good solution to these problems other than group splicing, which we did not implement in our prototype.

Another deficiency is that class sponsors lack an eager component. They have no internal source of sponsorship, like from a source priority, and thus must await sponsorship from another source, such as when the class result is demanded. This reflects the fact that a class sponsor is passive and merely distributes received sponsorship rather than taking an active role and negotiating with other sponsors for sponsorship. The touching model suffers from lack of active agents for implementing dynamic control, such as discussed in Section 2.1 for por.

As a final deficiency, the touching model does not address the issues with partial results (see Section 5.4.6) at all.

6.4 Extensions

In this section we describe various ways in which we believe the touching model should be extended.

6.4.1 Controller sponsors

The most important addition is more advanced controller sponsors. At a minimum we need some way to enforce modularity.

6.4.2 Resource attributes

We need to add resource attributes to achieve other dimensions of control like duration, especially.

6.4.3 Priority ranges

For modularity, we would like to have hierarchies of priority spaces, in which each priority space has its own local ordering relative to the priority of its parent. We call this nested priorities. However, we also want to implement nested priorities efficiency. We can already implement a flat priority space, as in our touching model, efficiently. One way to achieve this objective is to embed nested priorities within a suitably large flat priority space. This amounts to allocating task priorities within a virtual priority space.

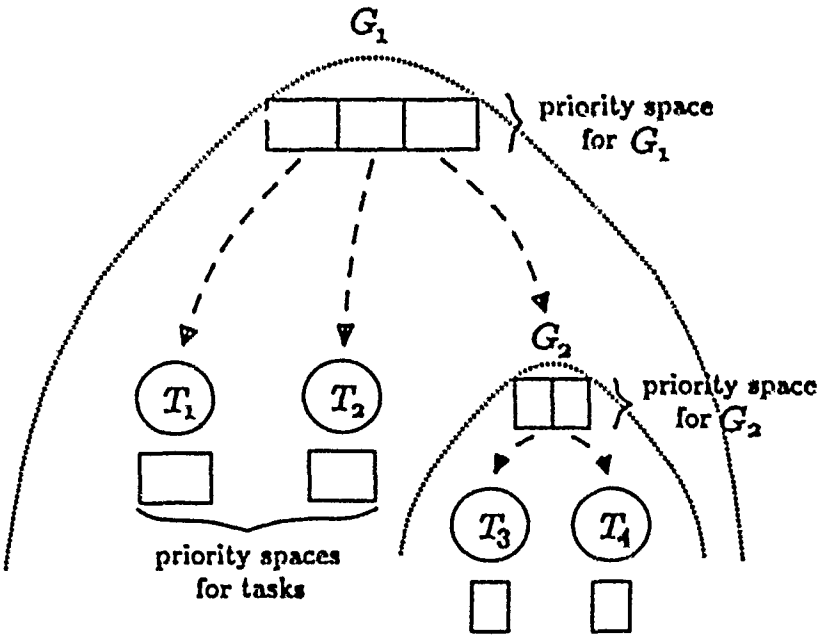


Figure 6.3: Ordering with priority ranges

The idea is to assign a priority *range* to each group and task. Each group and task is then responsible for allocating its priority range amongst its children to achieve nested orderings, as mentioned in Section 6.1.3. Figure 6.3 illustrates an example. Group G_1 has allocated its priority range amongst descendant tasks T_1, T_2, and group G_2 such that $G_2 \succ T_2 \succ T_1$ where $\succ$ indicates ordering. Group G_2 has further allocated its priority range to give a final ordering of $T_4 \succ T_3 \succ T_2 \succ T_1$.

To give the illusion of infinite resolution in the priority space, we can "rebalance" the priority space when one or more priority ranges are smaller than some some threshold. Rebalancing amounts to garbage-collecting the sparsely populated regions of the priority space to redistribute to the more densely populated regions.

The combining-rule can be stated in terms of ranges by taking the maximum and minimum of all the input range endpoints.

The cost of priority ranges comes in migrating the tasks in a priority range due to a change in ordering by the combining-rule or by a controller sponsor. Whole priority ranges must be migrated to new positions as illustrated in Figure 5.4 in Section 5.4.1.

6.4.4 Lazy priority propagation

There are two extremes in priority propagation: eager, in which we immediately propagate as much as possible, and lazy, in which we propagate incrementally, over regular intervals. The hope with lazy priority propagation is that immediate propagation may not be essential and indeed might be wasteful if the priority changes effected would immediately be obsolete. Lazy priority propagation may be beneficial in reducing the number of priority updates, i.e. eliminating useless updates, and in avoiding problems with cycles. However, these benefits must be balanced against the costs of operating with the sponsor network in non-equilibrium for a greater fraction of the time. We should investigate lazy priority propagation.

6.5 Summary

The touching model is a subset of the special sponsor model, based on priorities and lacking any resource attributes. Changes in priority are updated using eager priority propagation which tends to minimize periods of divergence at the cost of non-termination in pathological situations. Explicit reclamation fits within the touching model very elegantly by exploiting the combining-rule and priority propagation mechanisms. The touching model has only primitive controller sponsors and thus lacks modularity. The goal of the touching model was to provide a simple model to experiment with and "sample the waters" of speculative computation.

Chapter 7

Side-effects

As mentioned in Section 2.3, the main issue with side-effects is synchronization: side-effects provide a means for tasks to interact and communicate outside the data dependency (and implicit synchronization) mechanism represented by touching. Thus side-effects provide a means for (explicit) intertask synchronization. The new issue raised by such intertask synchronization in speculative computation is relevance: tasks may be relevant for the potential synchronization represented by their side-effects.

In this chapter we examine the problems posed by side-effect synchronization in speculative computation and solve these problems in the context of our touching model.

7.1 The Synchronization Problem

The synchronization problem posed by side-effects in speculative computation is that one task or a collection of tasks may be waiting for some event — for some side-effect to be performed — by some other task without any way to demand that the event occur. For example, when a task A stalls (blocks or busy-waits) waiting for some other task B to perform a certain side-effect event (such as releasing a lock or a semaphore, or explicitly determining a future, or writing some shared variable), Multilisp offers no means for A to contact B and "demand" that the event be performed. Consequently, if B has a lower priority than A, A could experience substantial delays due to preemption of B by tasks with priority less than A. Far worse is the possibility that B is stayed. Then A is definitely deadlocked. Thus poor performance and deadlock can result from having no way to demand a side-effect event.

7.2 Solution Outline

Task synchronization must obey the following property.

Property 1: No speculative deadlock Suppose that a program fragment which solves a given synchronization problem involves one or more speculative tasks (tasks subject to preemption and staying). If this program fragment is deadlock-free when all speculative tasks are replaced by mandatory tasks, then the original fragment involving speculative tasks should also be deadlock-free, provided the synchronization is relevant. We define the synchronization to be relevant if at least one task waiting for the synchronization to occur is relevant. (Recall that in our touching model we consider a task to be relevant if its effective priority is greater than 0.)

This property is essential. To address the performance issue, we desire that task synchronization also possess the following two properties.

Property 2: Demand transitivity (Priority inheritance) A task or tasks performing some event for which other tasks are waiting should run at the priority of at least the maximum priority waiter. That is, demand transitivity should be extended to all situations in which a task or tasks is/are waiting for another.

Property 3: Priority access Access to mutual exclusion regions should be in priority order.

Priority inheritance attempts to reduce the synchronization delay by giving synchronizer tasks at least as much importance as the tasks waiting for them. Note that Property 2 implies Property 1. (We assume the scheduler cannot starve a high priority task in favor of low priority tasks.) Priority access attempts to serve tasks in the order of their importance to reduce average synchronization delay (weighted by priority) and reduce the need for priority inheritance. Although these two properties are reasonable goals, their implementation costs must be balanced against their benefits. In several examples in this chapter we abandon our pursuit of one or both of these properties.

7.2.1 Philosophy of solution

Solutions to the problems of side-effects have two flavors according to the manner in which Property 1 is maintained (assuming that the tasks producing the synchronizing event can be identified):

1. Roll-back

   Tasks in the process of producing a synchronizing event are allowed to be stayed but only after they undo any state changes they made that might otherwise lead to deadlock.

2. Roll-forward

   Tasks may not be stayed while they are in the process of producing a synchronizing event for which other relevant tasks are waiting.

Roll-back schemes have a number of problems. First, they have limited applicability. They only work if some other task(s) will produce the synchronizing event after the original synchronizing task(s) is/are stayed and rolled back. Thus roll-back schemes will work when tasks compete for access to a mutual exclusion region but will not work for tasks blocked waiting for some task to determine a placeholder or write some shared variable. Second, roll-back schemes lack completeness. They address Property 1 and neglect Property 2. (They may or may not address Property 3.) Without Property 2, a synchronizing task can be preempted while other higher priority tasks wait on it, thus allowing unnecessary synchronization delay. Third, roll-back schemes can be complicated to implement. If the synchronizing task(s) involve(s) significant state changes, such as spawning a non-functional task, restoring the state is non-trivial. Finally, roll-back schemes do not fit very well in our touching model.

For these reasons, we do not consider any roll-back schemes here. Instead, we favor roll-forward schemes which are more in spirit of our touching model, as should be obvious shortly.

MultiScheme and Qlisp both support variations of the roll-back strategy (not necessarily linked to computation reclamation though). MultiScheme uses object finalization, as described in Section 3.4 and in [Miller], and Qlisp uses the `unwind-protect` form described in Section 3.5 and in [Gabr88].

7.2.2 Roll-forward solutions

To achieve Property 1 in the roll-forward paradigm (assuming the synchronization is solved in a way that would be correct if all the tasks were mandatory) we need:

1. some way to recognize when the progress of a task is stalled waiting for some synchronizing event,
2. some way to determine which task or tasks is/are responsible for performing the synchronizing event (the synchronizer task(s)), and
3. some way — either *a priori* or dynamically upon recognizing 1 — to prevent tasks identified in 2 from being stayed.

The first requirement is trivial when a task blocks on a placeholder or a semaphore, but can be more difficult with non-blocking types of synchronization, such as spin-locks. The second requirement is a key problem. Often the task(s) responsible for performing the synchronizing event can be identified implicitly, such as in simple mutual exclusion problems, but this is not always the case. Our approach in such cases is to explicitly identify the responsible tasks.

There are a number of ways to meet the third requirement. The simplest is to make a task "non-stayable" while it may be performing a synchronizing event. While very simple,

this *a priori* approach has two difficulties. First, it prevents a task from ever being stayed while it is in a "critical region", even if no other task is waiting for it to exit the region. Thus it may be too conservative. Second, and most important, the "non-stayable" solution does not prevent a task from being preempted while other higher priority tasks wait on it. In other words, it fails to achieve Property 2.

Another *a priori* approach is to make the task "non-preemptable" while it may be performing a synchronizing event. This condition may be stronger than required or desired — we may want high priority tasks, such as mandatory tasks, to preempt low priority synchronizing tasks. Thus this approach over-achieves Property 2. This drawback may be quite acceptable, given the simplicity of this approach, if tasks are "non-preemptable" for only a short time.

A third approach is the dynamic approach. It avoids under- or over-achieving Property 2 by parameterizing the preemption level of the synchronizing task(s) by the priority of the tasks waiting for the synchronizing event. This approach is, of course, more complicated because all the tasks involved in a synchronization event must communicate to find a compromise preemption level for the synchronizing task(s). Furthermore, this must happen dynamically in response to changes in task priorities. Nevertheless, in many situations the dynamic approach is potentially more efficient than *a priori* approaches.

Most of the remainder of this chapter involves the extension of our touching model to implement this dynamic approach. The basic idea is to sponsor the task(s) producing some event by the tasks waiting for the event. In this case we propagate the priority of the maximum priority task waiting for a synchronizing event to the task or collection of tasks that will perform that event.

To complete our discussion on implementing Properties 1 through 3, we note that Property 3 requires some way to control which waiting tasks are resumed after a synchronization event occurs. This fits into our extended touching model fairly well.

7.3 Examples

To motivate further discussion, we look at concrete examples of common synchronization mechanisms and their problems.

7.3.1 Locks

A lock is a shared variable which is tested and updated indivisibly to synchronize tasks without blocking; the waiting, if any, is busy waiting.

```
;; initialize lock
(define a-lock (cons nil nil))

;; evaluate thunk while holding a-lock
(define (with-lock thunk)
   (if (not (rplaca-eq a-lock '#t nil)) ; if car of a-lock is nil,
                                        ; atomically replace it by '#t
       (begin (thunk)
              (rplaca a-lock nil))      ; clear lock
       (with-lock thunk)))              ; spin until get lock
```

Figure 7.1: A simple spin-lock

Mutual exclusion — spin-locks

Figure 7.1 shows an example of mutual exclusion via spin-locks. Once a-lock is initialized, any task calling with-lock will evaluate thunk in a mutual exclusion region.

The most obvious problem here is that a task may be stayed while evaluating thunk and holding the lock. Thus other tasks may spin forever waiting for the lock. We could solve this problem by either the *a priori* approach, in which a task is non-stayable or non-preemptable while it holds the lock, or by the dynamic approach, in which the waiters propagate their attributes to the task with the lock.

The *a priori* approach requires some means to indivisibly grab the lock and enter the non-stayable/non-preemptable region. Otherwise, if the lock is grabbed first, the task may be stayed before entering the non-stayable/non-preemptable region, or if the non-stayable/non-preemptable region is entered first, the task may continue to spin once it has been stayed. Of course, the task could spin in a loop where it enters the non-stayable/non-preemptable region, attempts to grab the lock, and immediately exits the region if unsuccessful. However, entry and exit of non-stayable/non-preemptable regions could be expensive so we want to avoid the unnecessary entries and exits with this approach.

The dynamic approach requires some way to identify the task holding the lock, some way for a task to determine when it is waiting for a lock, and some way to propagate the attributes of a waiting task without blocking. These first two items may involve non-trivial overhead, and these three items only guarantee Properties 1 and 2.

Typically, the mutual exclusion region guarded by a spin-lock is short, and thus the presence or absence of Properties 2 and 3 have minimal effect. In this case, the *a priori* approach is the most cost-effective. If this is not the case, busy-waiting is probably not efficient anyway and other synchronization methods, such as semaphores, should be considered.

In the case of spin-locks (and a few other synchronization problems — see Section 8.3

```
;; initialize lock
(define a-lock (cons nil nil))

;; evaluate thunk while holding a-lock
(define (with-lock thunk)
   (let ((action (delay (begin (thunk) ; evaluate the thunk when touched
                               nil)))) ; return nil to clear lock
      (rplaca-eq a-lock action nil)    ; if car of a-lock is nil,
                                       ; atomically replace it by action
                                       ; otherwise block on delay in car
      (touch action)))                 ; start evaluating thunk
```

Figure 7.2: A simple spin-lock with a "delay device"

on Emycin), we can provide a solution guaranteeing Properties 1 and 2 (and not over-achieve Property 2) without any new constructs. Figure 7.2 shows such a solution. Each task entering with-lock competes to indivisibly test the car of the lock cell for nil and if so, replace it by a delay to evaluate thunk. The winning task touches the delay to force evaluation of thunk. Losing tasks block on the delay in the car of the lock cell. (rplaca-eq touches its arguments.) When thunk is evaluated, the delay is determined to nil, thus clearing the lock cell, and any tasks blocked on the delay resume execution of rplaca-eq and compete again for the lock.

Since all the tasks waiting for the lock touch the task responsible for releasing the lock, this task has at least the priority of the maximum priority waiter, thus guaranteeing Property 2 and hence Property 1.

This solution converts the spin-lock problem into the framework of our touching model. In so doing, we have changed the semantics of this solution slightly from a true spin-lock. Tasks failing to get the lock block rather than spin. If thunk takes more than a short time to evaluate this may be an improvement over spinning. However, if thunk does take only a short time, the queueing and restarting associated with blocking may add significant overhead. Ideally, we would like to propagate attributes to the task evaluating thunk without blocking.

This solution lacks Property 3, but this should not be an important problem. If it were, the circumstances are probably such that semaphores would be a better choice than spin-locks.

7.3.2 Placeholders

Placeholders are a very useful synchronization mechanism. We categorize their use by the number of potential determiners for a given placeholder.

Initialize:

```
(define first (make-future)) ; make a placeholder
```

Master task:	Slave tasks:
...	
(master-before)	...
(determine-future first nil) ; continue slaves	(touch first) ; await synch condition
(master-after)	...
...	

Figure 7.3: A placeholder example with one determiner

One determiner

Figure 7.3 shows a typical application of a placeholder with one determiner.

The two problems here are the master task may be stayed before determining the placeholder and the master task may run slowly, perhaps being subject to unnecessary preemption, while higher priority tasks are blocked on the placeholder. In short, the problem is that Property 2 is not guaranteed. Property 3 is irrelevant here.

We could solve this problem by making the master task non-preemptable until it determines the placeholder, but this is too strong a solution. Instead, we would like the tasks blocked on the placeholder to sponsor the task that will determine the placeholder. In some cases, such as is suggested in Figure 7.3, the determiner may be known *a priori*. In such cases we might be able to use a variation of the trick in the previous section, as illustrated in Figure 7.4.

If the determiner is not known *a priori*, we have the problem discussed in the next section.

Multiple Potential Determiners

A prime example of this use of placeholders is multiple-approach speculative computation. Figure 7.5 shows such an example where we are interested only in the first solution to a set

Master task:	Slave tasks:
...	
(set! first (future	...
(begin (master-before) nil)))	(touch first)
(touch first)	...
(master-after)	
...	

Figure 7.4: Placeholder example with "delay device" variation

```
(define first (cons 0 nil)) ; initialize lock

(define first-solution (make-future)) ; create synchronization placeholder

;; attempt to solve the given problem
(define (solve problem)
   (let ((a-solution (work-on-problem problem)))
      (if (solution? a-solution)      ; was a solution found?
          (if (rplaca-eq first 1 0)   ; if so, test lock to see
                                      ; if first solution
              ;; if first, determine placeholder to solution
              (determine-future first-solution a-solution)))))

;; attempt to solve all problems simultaneously
(define (find-first-solution problems)
   (mapcar (lambda (prob) (future (solve prob))) problems) ; fork solvers
   first-solution)                    ; return first solution
```

Figure 7.5: A placeholder example with multiple potential determiners

of problems. For simplicity, we assume that a solution will be found for at least one of the problems.

As before, a task may be stayed or preempted before determining the placeholder. The problem here is that we do not know which task will determine the placeholder. Thus we must ensure that none of the potential determiners is stayed or preempted. One solution is to make all the potential determiners non-preemptable until the placeholder is determined. Not only is this solution too strong, as before, but it also is complicated by the need to re-enable preemption in all the "failed" determiners. A better solution is to sponsor all the potential determiners of the placeholder until the placeholder is determined. In fact, noting that Figure 7.5 is really por, we might like a controller sponsor to distribute sponsorship to the potential determiners. (This is precisely the approach we take in Section 8.1).

7.3.3 Semaphores

Semaphores may be used in many different ways and as the basis for other synchronization mechanisms like monitors [Hoare] and (conditional) critical regions [Brinch]. (Thus for theoretical reasons we need examine only semaphores, although there may be efficiency reasons to favor these other mechanisms. We leave the study of these other synchronization mechanisms for future work.) We present a few examples of various ways in which semaphores can be used and the problems encountered in these cases.

Simple Mutual Exclusion — serializers

Semaphore serializers have the following simple format:

```
(wait-sema sema)
some action
(signal-sema sema)
```

Some action is guarded for mutual exclusion by a single binary semaphore, `sema`, which is initially free. (As mentioned in Section 1.4, `wait-sema` and `signal-sema` are our names for Dijkstra's P and V operations respectively.)

One way to meet Properties 1 and 2 for serializers is to make tasks non-preemptable while in the mutual exclusion region. However, given the overhead already involved in blocking on a semaphore, the benefits of the dynamic approach based on our touching model come for little additional cost. Thus we consider only this approach in this subsection.

For serializers it is fairly clear what is required to meet Properties 1 through 3. We would like to:

1. record the task that is in the serializer mutual exclusion region,
2. sponsor the task in the mutual exclusion region with the priority of the maximum-priority waiter, and
3. admit waiting tasks to the mutual exclusion region in priority order. We assume that ties in priority ordering are broken in first-come-first-served (FCFS) order.

In this situation we could add the necessary mechanism to our model so that these actions will occur implicitly, just like touch sponsorship and priority propagation occurs implicitly when a task blocks on a future. Unfortunately, the following examples indicate that such an implicit mechanism will not suffice in general.

Example: Readers and writers problem

In this problem, which is a variation of the simple mutual exclusion problem, there are two types of tasks accessing a shared object. The first type (the "readers") may access the object concurrently so long as all tasks of the second type are excluded. The second type (the "writers") require exclusive access to the shared object.

Figure 7.6 shows a solution to a variant of the readers and writers problem in which a writer must wait until there are no pending readers (thus writers may starve).[1] readcount

[1] We assume that waiters for a semaphore are granted the semaphore in first-come-first-served (FCFS) order, thus readers do not starve.

Reader

```
(wait-sema mutex)          ; (1)
(incr1 readcount)          ; (2)
(if (= readcount 1)
    (wait-sema wrt))       ; (3) first reader must wait for writer to finish
(signal-sema mutex)        ; (4)
read operation
(wait-sema mutex)          ; (5)
(decr1 readcount)          ; (6)
(if (= readcount 0)
    (signal-sema wrt))     ; (7) let a writer proceed
(signal-sema mutex)        ; (8)
```

Writer

```
(wait-sema wrt)            ; (9)
write operation
(signal-sema wrt)          ; (10)
```

(incr1 x) and (decr1 x) are macros which expand to (set! x (+ x 1)) and (set! x (- x 1)) respectively.

Figure 7.6: A solution to the readers and writers problem

indicates the number of current readers and mutex is a binary semaphore for updating readcount atomically. wrt is a binary semaphore for the mutual exclusion of readers and writers.

The solution in Figure 7.6 contains three serializers: lines 1 through 4 and lines 5 through 8 for updating readcount and lines 9 through 10 for writing. Thus this solution suffers from the same two problems as simple serializers:

1. A low priority task (reader/writer) in a mutual exclusion region may block the entry of higher priority tasks waiting for access to that region. Deadlock (to readers/writers) may result if the low priority task cannot make progress, such as if it is stayed.
2. Access to the mutual exclusion regions — for readcount update and writing, via (wait-sema mutex) and (wait-sema wrt) respectively — is not necessarily via priority order.

The solution in Figure 7.6 contains another critical region: from the (wait-sema wrt) in line 3 through (signal-sema wrt) in line 7. This critical region is fundamentally different from the critical regions in the simple serializers so far: there may be more than one task in the critical region simultaneously. Furthermore, only the first and last tasks to enter and exit this critical region in a read "epoch" do so via (wait-sema wrt) and (signal-sema wrt) respectively. No one task necessarily holds the wrt semaphore for the entire duration of a read epoch.

Coupling between the simple serializers and this new critical region introduces a number of new problems.

First, readers are blocked (on wrt for the first reader and on mutex for the rest) until the present writer exits the write serializer. If a writer is stayed in the write serializer, all readers (and writers) will be deadlocked.

Second, writers are blocked until the last reader in the current read epoch completes line 7. If any reader is stayed between lines 3 and 7, writers will be deadlocked. Note that only writers are deadlocked if readers are stayed between lines 4 and 5.

Third, readers are blocked by a reader in either of the two readcount serializers.

Fourth, readers and writers do not enter the critical region in priority order with respect to each other. We would like a blocked writer of priority p to prevent any new readers of priority less than p from entering the critical region.

To deal with these first three problems, we would like to extend our solution for the first two problems. We would like:

1. the readers blocked on mutex to sponsor the reader in the first or second readcount serializers,
2. the readers blocked on wrt in line 3 to sponsor the writer in the write serializer, and
3. the writers blocked on wrt in line 9 to sponsor
   (a) the writer, if one is present, in the write serializer, and
   (b) otherwise the readers in the critical region between lines 3 and 7.

3(b) raises the following issue: which readers between lines 4 and 5 do the writers blocked on wrt sponsor? This sponsorship can affect the order in which these readers gain access to the second readcount serializer and thus this order may not correspond to the order expressed by their "original" priorities. For example, if the tasks blocked on wrt sponsor all the readers between lines 4 and 5 with the same priority, these readers will gain access to the second readcount serializer in FCFS order rather by the order of their original priorities. We will examine this issue later.

Example: Producer-consumer problem

The producer-consumer problem consists of some number of tasks synchronizing the production and consumption of values via a finite buffer. A producer task computes a value and inserts it in the buffer where a consumer task later retrieves it. Figure 7.7 shows the code for a producer-consumer problem involving a buffer of size N. mutex is a binary semaphore for atomic insertion and deletion to/from the buffer. empty is a general semaphore, with initial value N, which counts the number of empty slots in the buffer. Complementing empty is the general semaphore full. Its initial value is 0 and it counts the number of full slots in the buffer.

Producer		Consumer	
`(wait-sema empty)`	; (1) wait until at least one empty slot	`(wait-sema full)`	; (5) wait until at least one full slot
`(wait-sema mutex)`	; (2) indivisibly add an item	`(wait-sema mutex)`	; (6) indivisibly delete an item
insert in buffer		delete from buffer	
`(signal-sema mutex)`	; (3)	`(signal-sema mutex)`	; (7)
`(signal-sema full)`	; (4) indicate a full slot	`(signal-sema empty)`	; (8) indicate an empty slot

Figure 7.7: A solution to a producer-consumer problem

This formulation[2] has the familiar two problems associated with a simple serializer like mutex. It also has the additional problem that a consumer could be deadlocked waiting on `full` if every producer is in one of the following three states:

1. stayed before line 1

   We call producers outside lines 1 through 4 *external* producers.

2. stayed between lines 1 and 4

3. blocked on `empty`

Likewise, a producer could be deadlocked waiting on `empty` if every consumer is in one of the following three states:

1. stayed before line 5

   We call producers outside lines 5 through 8 *external* consumers.

2. stayed between lines 5 and 8

3. blocked on `full`

These problems are unique among the semaphore examples presented so far: they involve tasks (the external producers and consumers) outside the semaphore regions. In this respect the producer-consumer problem is similar to the placeholder example presented in Section 7.3.2.

To solve these new deadlock problems we would like:

1. The producers blocked on `full` to sponsor

   (a) any producers between lines 1 and 4, and

   (b) if none, the external producers (which includes the tasks blocked on empty).

[2]Streams [Abelson] provide a more elegant way to achieve producer-consumer synchronization. However, buffer-based formulations offer better control over storage use.

2. The producers blocked on empty to sponsor

   (a) any consumers between lines 5 and 8, and

   (b) if none, the external consumers (which includes the tasks blocked on `full`).

As in the readers/writers problem, there is the question of exactly which external producer and external consumer we should sponsor.

In addition, we would like tasks blocked on empty and `full` to enter their respective critical regions in correspondence with their priority order.

Example: Simulation of monitors

A monitor consists of data, procedures, a body, and a mutual exclusion region. The body is executed once when the monitor is initialized and thereafter all interaction with the data is via the procedures defined in the monitor. The body of each such procedure executes in the mutual exclusion region, thus at most one procedure invocation may execute at any time. Since monitors are intended for the coordination of concurrent processes, monitors provide a way for procedure bodies to perform synchronization via condition variables. If c is a condition variable, then (`cwait c`) causes the calling process to block on condition c, and (`csignal c`) awakens a process[3] (if any) blocked on condition c. (`csignal c`) has no effect if there are no processes blocked on c. Note that these synchronization constructs are not semaphores, but merely block-wakeup constructs (atomicity is already ensured by the mutual exclusion region). When a process executes (`csignal c`) there may be a choice between continuing the signalling process in the mutual exclusion region or stopping it and awakening a process blocked on condition c to continue in the mutual exclusion region instead. For reasons stated in [Hoare], it is customary to obey the Immediate Resource Requirement [Ben-Ari] in such instances, which gives priority to resuming processes blocked on the signalled condition over continuing the signalling process.

We consider here a restricted form of the monitors presented above: each procedure must have at most one `csignal` and such a `csignal` must occur as the last statement of the procedure. We make this restriction to simplify the example; it is not necessary in practice or in the simulation of monitors by semaphores. With this restriction, the Immediate Resource Requirement gives priority to resuming processes blocked on the signalled condition over admitting new processes to the mutual exclusion region.

To simulate a monitor M obeying the above restrictions, we associate with M a binary semaphore s, initially free, to serialize access to the mutual exclusion region of M. The entry point(s) of each monitor procedure begin(s) with (`wait-sema s`) and the exit point(s) end(s) with either (`signal-sema s`) or a `csignal` (as per the restriction above).

Every condition variable c has an associated integer variable `ccount`, initially 0, to count the number of processes waiting for c and a binary semaphore `csem`, initially locked,

[3]Monitors customarily utilize FCFS ordering for awakening processes.

Monitor	Simulation	
procedure entry	`(wait-sema s)`	; enter mutual exclusion region
procedure exit (w/o csignal)	`(signal-sema s)` procedure return	
	For every condition c:	
`(cwait c)`	`(incr1 ccount)`	; (1) increment # processes waiting for condition c
	`(signal-sema s)`	; (2) exit mutual exclusion region (so no deadlock)
	`(wait-sema csem)`	; (3) wait for signalling of condition c
	`(decr1 ccount)`	; (4) decrement # processes waiting for condition c
`(csignal c)`	`(if (> ccount 0)`	; (5) check if any processes blocked on condition c
	`(signal-sema csem)`	; (6) if so, start one up
	`(signal-sema s))`	; (7) otherwise exit the mutual exclusion region
	procedure return	

Figure 7.8: Simulating monitor M with binary semaphores

to actually block the processes.

Figure 7.8 shows how M may be simulated using binary semaphores. Figure 7.9 illustrates a kind of state diagram for M (with only one condition c — the extension for multiple conditions is straightforward). The circles indicate states and the rectangles indicate semaphores. The arc labels indicate the condition causing the state transition.

There are four problems with the code in Figure 7.8 in a speculative computation environment. The first two are the familiar problems with serializers.

First, a low priority task in the mutual exclusion region may block the entry of higher priority tasks waiting for semaphore `s` or `csem`. Deadlock may result if the low priority task cannot make progress, such as if it is stayed.

Second, access to the mutual exclusion region, via `(wait-sema s)` or `(wait-sema csem)`, is not necessarily via priority order.

Third, in more complex uses of semaphores, such as in this example, deadlock may occur even if a task is stayed in certain regions unguarded by semaphores. In Figure 7.9, the waiters for condition c (i.e. all the tasks between lines 2 and 3) are unguarded by any semaphores, yet deadlock will result if all these tasks are stayed. The reason is that tasks in this "critical region" are included in the monitor state by ccount. When a task executes `(signal-sema csem)` in line 6, it in effect "passes" the semaphore s which it currently possesses to one of the tasks waiting on condition c and hence waiting for re-entry to the mutual exclusion region. Thus the task that will signal s will not be the same as the one that waited for it. The fact that ccount was > 0 indicates that there should be tasks waiting on condition c, but they all might have been stayed before making it to that point. In any event, if none of the tasks that incremented ccount can ever re-enter the mutual exclusion region (and release semaphore s or pass it to yet another task), then no task can ever enter the mutual exclusion region.

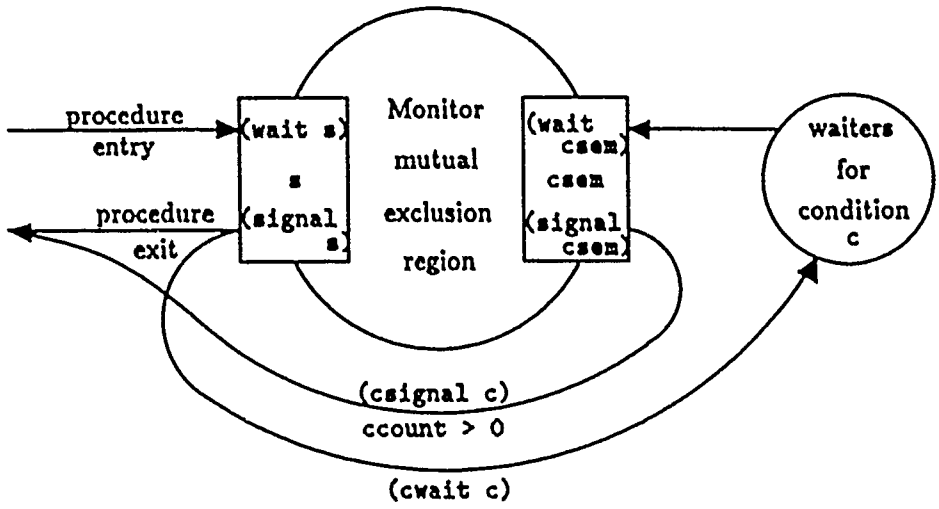


Figure 7.9: State diagram of monitor M simulation

(One can also view semaphore wrt in Figure 7.6 as being "passed" from reader to reader. However, this view is less natural for the readers and writers problem.)

The last problem is similar to the fourth problem with the producers/consumers example. Certain tasks may be blocked on a condition semaphore waiting for other task(s) to enter the monitor and signal the condition. For example, we could use a monitor to perform producer/consumer synchronization. A consumer task could block on a condition indicating the buffer was empty until some producer task entered the monitor and signalled a full slot in the buffer.

To deal with these two new problems, we would like to extend our solution for the first two problems. We would like:

1. The tasks blocked on s to sponsor
   (a) the task, if any, in the mutual exclusion region, and
   (b) otherwise the waiters for condition c in the "critical region" between lines 2 and 3 where c is the condition signalled by the most recent task to exit the mutual exclusion region. (If there are tasks blocked on s and there is no task in the mutual exclusion region, the most recent task to exit the mutual exclusion region must have done so via csignal.)
2. The tasks blocked on csem (for every condition c) to sponsor
   (a) the task, if any, in the mutual exclusion region, and
   (b) otherwise the task(s) responsible for eventually signalling csem.

7.3.4 Other types of side-effects

Side-effects can certainly be used in other ways to achieve intertask synchronization and thus lead to possible problems. The most general way is by busy-waiting on some change in state effected by another task, such as changing a binding (`set!`) and mutating a structure (`set-car!`, `vector-set!`, `string-set!`, etc.). The issues here are exactly the same as with locks, the only difference being the non-indivisibility of the state changes. In the interest of program clarity and understandability, we believe that programmers should be encouraged to use a small set of powerful primitives. Thus we eschew direct treatment of general busy-waiting synchronization in favor of locks, which are strictly more powerful. Our techniques for dealing with the problems of locks should also apply to the problems with general busy-waiting mechanisms (though additional constructs may be necessary).

A different type of side-effect is input/output (other than by the read-eval-print-loop). Input, by definition, is always demanded and thus presents no problems in speculative computation. Output, by contrast, is not always demanded and thus has the same relevance problem as task synchronizing side-effects: a task could be stayed before printing some output. There is no direct way to demand output, just like there is no direct way to demand other types of side-effects. We can either ensure that the task performing the output is not preempted or we can arrange to sponsor the task(s) responsible for performing the output, just like we did with side-effects. Thus output really presents the same intertask synchronization problem as with other side-effects.

7.4 Solutions

In this section we consider solutions to the problems exemplified by the synchronization mechanisms in the previous section. First we briefly discuss non-preemptable regions for simple mutual exclusion synchronization. Then we discuss at some length a sponsor-based, dynamic approach in the context of our touching model for precedence constraint and more complex mutual exclusion synchronization.

7.4.1 Non-preemptable regions

We obtain non-preemption by promoting a task to mandatory status temporarily. The primitives `promote-task` and `demote-task` cause the executing task task to enter and exit a non-preemption region respectively. While these primitives suffice for many applications — such as guaranteeing non-preemptable output, they are insufficient for serializers. For spin-locks and semaphores, we only want to promote a task once it has entered the critical region. This requires new constructs. For spin-locks we introduce a pair of new primitives: `rplac`x`-eq-mand` (x = a or d). These primitives are styled after the `rplac`x`-eq` primitives of Multilisp. Like (`rplac`x`-eq a b c`), (`rplac`x`-eq-mand a b c`) indivisibly tests if the `c`x`r` of a is eq to c and if so, replaces the `c`x`r` of a by b and returns a, and otherwise returns nil. However, these new primitives also indivisibly promote the current task to mandatory

```
;; initialize lock
(define a-lock (cons nil nil))

;; evaluate thunk while holding a-lock
(define (with-lock thunk)
   (if (rplaca-eq-mand a-lock '#t nil)
       ;; if car of a-lock is nil, atomically replace it by '#t
       ;; and promote the task to mand
       (begin
          (thunk)
          (rplaca a-lock nil)       ; clear lock
          (demote-task))
       (with-lock thunk)))
```

Figure 7.10: Non-preemptable spin-lock

status if the eq test is non-nil. Thus the problem with spin-locks in Section 7.3.1 may now be solved as shown in Figure 7.10.

Semaphores — at least semaphores obeying just Properties 1 and 2 — can be built from rplacx-eq-mand and other primitives.

7.4.2 Sponsors

To solve the relevance problem of side-effect synchronization efficiently — i.e. obeying, but not over-achieving, Properties 1 and 2 — we need to be able to transmit the demand of a task awaiting some event to the task(s) responsible for further progress (or lack thereof). Sometimes this involves demanding a single task, but many times the context does not uniquely identify a single task — there may be a collection of tasks, any one of which may be the one that actually does the synchronization. Since we do not know which task in such instances, we have to conceptually demand all the tasks in the collection.

We use class sponsors to solve the relevance problem. Class sponsors allow us to temporarily sponsor a task or a collection of tasks, thus transmitting demand, without touching — i.e. blocking on — the tasks. The following sections demonstrate how to solve the relevance problems in the previous synchronization examples with sponsors in the context of our touching model.

7.4.3 Placeholders

To solve the problems with placeholders, we need the tasks blocked on a placeholder to sponsor a defined class of potential determiner tasks. This is the reason for the optional

```
(define determiners (make-class 'all)) ; or (make-class 'pqueue)     ; 1

(define first (cons 0 nil)) ; initialize lock

;; create synchronization placeholder
(define first-solution (make-future determiners))                    ; 2

;; attempt to solve the given problem
(define (solve problem)
   (let ((a-solution (work-on-problem problem)))
      (if (solution? a-solution)  ; was a solution found?
          (if (rplaca-eq-mand first 1 0)                             ; 3
              ;; if first, determine placeholder to solution
              (begin
                (determine-future first-solution a-solution)
                (demote-task))))))                                   ; 4

;; attempt to solve all problems simultaneously
(define (find-first-solution problems)
   (mapcar (lambda (prob)
             (add-to-class (future (solve prob)) determiners))       ; 5
             problems) ; fork solvers

   first-solution)  ; return first solution
```

Figure 7.11: Solution to multiple potential determiners problem

class argument to make-future in Section 6.2. Figure 7.11 shows how to solve the multiple potential determiners problem in Section 7.3.2 with classes.

The line numbers indicate lines with changes from Figure 7.5. Line 1 creates a class for all the potential determiners. Line 2 creates a placeholder which sponsors these potential determiners. Thus any task blocked on this placeholder sponsors the potential determiners and thereby propagates the demand for the placeholder result to the potential determiners. Line 5 creates and adds each problem solver to the potential determiners class. The atomically entered non-preemptable region defined by lines 3 and 4 prevents the "winning" task from being stayed in the exclusive region before it determines the placeholder. This non-preemptable region is not necessary if the class sponsors every member of the potential determiners class (e.g. if the class type is all). In this case, the winning task — whichever one it is — will always be sponsored by the tasks waiting on the placeholder and thus can not be stayed. However, this non-preemptable region is necessary if the class sponsors only some of the potential determiners (such as e.g. with class type pqueue). In this case, the winning task may be one of the tasks not sponsored by the class and would cause deadlock

if stayed in the exclusive region.

The single determiner problem in Figure 7.3 is trivial to solve in a likewise manner.

7.4.4 Semaphores

To solve the problems with semaphores, we need to be able to define classes of potential signallers, like the potential determiners with placeholders, and sponsor these classes by the tasks waiting for semaphores. Unlike with placeholders though, we also need some way for a task to transit through different classes as it enters and exits critical regions. We also want to admit waiting tasks to critical regions in priority order of the tasks. We introduce the following constructs. (Some of these are modifications of the constructs in Section 1.4.)

(`make-sema` &optional *class* (*count* 1)) creates and returns a semaphore object which consists of a count of the tasks which may still enter the critical region, a priority queue for tasks waiting to enter the critical region, and a class for waiting tasks to sponsor, which we call the *sema class*. The count field is initialized to *count*, or 1 if *count* is omitted. If count is initialized to 1, the semaphore is a binary semaphore; otherwise it is a general semaphore. The maximum priority task in the priority queue sponsors the sema class. The sema class is initialized to *class* (or nil if *class* is omitted) and is accessible via the construct

(`get-sema-class` *sema*) and may be set via

(`set-sema-class` *sema class*).

Thus, the class that waiting tasks sponsor may change dynamically. We show the advantages of this feature later. A waiting task may sponsor several classes by defining the sema class to be a class of classes.

(`wait-sema` *sema* &optional *cr-thunk*) is a standard semaphore wait operation augmented with a "critical region thunk". If present, the optional argument *cr-thunk* should be a procedure of zero arguments (otherwise an error will occur). The task executing `wait-sema` tests the count associated with *sema* and, if nonzero, decrements the count and promotes itself to mandatory status. Otherwise, the task enqueues itself in the priority queue of waiters for *sema* and suspends. The test and these subsequent actions (either decrement and promote or enqueue) occur indivisibly. If the count was nonzero, the task applies *cr-thunk* (if present), demotes itself to its proper status, and finally returns from `wait-sema`. Thus *cr-thunk* executes as a mandatory task. This enables *cr-thunk* to perform critical operations, such as adding the task to the sema class of *sema*, without danger of being stayed.

(`signal-sema` *sema* &optional *cr-thunk*) is a standard semaphore signal operation augmented with a "critical region thunk" like in `wait-sema`. If *cr-thunk* is present, the task executing `signal-sema` promotes itself to mandatory status, applies the zero argument procedure *cr-thunk*, signals the semaphore *sema* (as described shortly), and then demotes itself. Thus *cr-thunk* can perform critical operations without danger of being stayed or preempted until the semaphore is released. With the optional argument *cr-thunk*, `signal-sema` is

syntactic sugar for

```
(promote-task)
(cr-thunk)
(signal-sema sema)
(demote-task)
```

If *cr-thunk* is omitted, the task executing `signal-sema` signals *sema* as follows. If the count associated with *sema* is zero, the task dequeues the maximum priority (suspended) task from the priority queue of waiters for *sema*, promotes this task to mandatory status, and resumes it. Otherwise, the executing task increments count. The test and subsequent action in either case occur indivisibly.

Finally, we introduce two convenient macros for entering and exiting classes.

(`enter-class` *class*) adds the evaluating task to the given class.

(`exit-class` *class*) removes the evaluating task from the given class.

These two macros expand to (`add-to-class` (`my-future`) *class*) and (`remove-from-class` (`my-future`) *class*) respectively. (`my-future`) returns the future object of the executing task.

The basic idea with these constructs is two-fold. The first part is to always ensure that any task in a "critical region" is a member of some class. The second part is to ensure that the tasks blocked on a semaphore sponsor the appropriate class of tasks responsible for releasing the semaphore.

The first part involves, in general, defining a set of classes for tasks and the transitions between these classes. That is, a task's trajectory through a semaphore system (such as in the examples given in Section 7.3.3) transits between various classes. The classes can be defined using the `make-class` construct and the class transitions can be defined using the `add-to-class`/`enter-class`, and `remove-from-class`/`exit-class` constructs.

The second part involves defining at each point in time the appropriate class that the tasks blocked on a semaphore should sponsor. This "semaphore sponsor class" (a.k.a. sema class) is defined initially by the argument to the `make-sema` construct and thereafter may be changed using the `set-sema-class` construct. Thus, in a sense, a semaphore may also transit between classes. A more correct view is that a sponsor "indirection cell" transits between classes (with `set-sema-class` defining the transitions).

These two principles allow us to achieve Property 2 and hence also Property 1, as we will demonstrate shortly for our examples of Section 7.3.

A second basic idea with these new constructs is the priority access order to critical regions, implemented by the priority queue mechanism for tasks blocked on a semaphore. This priority queue mechanism allows us to finally achieve Property 3.

We now demonstrate solutions, using our new constructs, to the problems with the

examples in Section 7.3. These examples should help the reader understand the principles described above.

Simple Mutual Exclusion — serializers

The problems with simple semaphore serializers are solved straightforwardly:

```
(wait-sema sema (lambda () (enter-class cr-class)))
some action
(signal-sema sema)
(exit-class cr-class)
```

where `cr-class` is initialized by `(make-class 'all)` and `sema` is initialized by `(make-sema cr-class n)` (n= 1 for a binary semaphore).

Each task enters and exits `cr-class` as it enters and exits the mutual exclusion region respectively. The maximum priority task waiting to enter the mutual exclusion region sponsors the task(s) in the mutual exclusion region via its membership in `cr-class`. This guarantees Property 1 and 2. `wait-sema` maintains a priority queue of tasks waiting to enter the mutual exclusion region and admits them in priority order, thus guaranteeing Property 3.

Obviously, we can easily define an interface that hides the notion of classes from the user in this case. The following is one possibility.

```
(define (make-serializer)
  (let* ((mutex-class (make-class 'all))
         (sema (make-sema mutex-class)))
    (lambda (thunk)
      (wait-sema sema (lambda () (enter-class mutex-class)))
      (thunk)
      (signal-sema sema)
      (exit-class mutex-class))))
```

This makes and returns a "serializer" procedure which takes an argument thunk and evaluates the thunk in a mutual exclusion region as shown below.

```
(define serialize (make-serializer))
...
(serialize (lambda ()                    ; evaluate serialized-code in
             serialized-code))           ; mutual exclusion region
```

The readers and writers problem

This problem may now be solved as shown in Figure 7.12. The main idea is to have two classes: one for the readers or writer in the read/write critical region (i.e. with access to the shared object) — we call this the `accessor-class` — and one for the mutual exclusion region of the `readcount` serializer — we call this the `mutex-class`. Any tasks blocked awaiting access to the critical region sponsor the readers or writer in the critical region. Any readers blocked awaiting entry to the `readcount` mutual exclusion region sponsor the task in that region. These sponsorships guarantee Properties 1 and 2. The semaphores admit tasks to the critical and mutual exclusion regions in priority order and thus guarantee Property 3. This solution does not, however, guarantee this priority access order to the critical region across readers and writers.

We now give a line by line description of Figure 7.12. Readers blocked on the `mutex` semaphore region in line 1 sponsor `mutex-class`.[4] As a reader enters this `mutex` mutual exclusion region in line 1, line 2 adds the reader to `mutex-class`. If this reader is the first in a read epoch, it tests the `wrt` semaphore in line 4 for entry to the read/write critical region. If successful, line 5 adds the reader to `accessor-class`. If unsuccessful, the reader blocks on the `wrt` semaphore and sponsors `accessor-class` via the sema class of `wrt`. In this case, note that any readers blocked on `mutex` (in lines 1 or 9) sponsor this reader, which in turn sponsors `accessor-class`. This transitivity ensures that the maximum-priority reader always sponsors `accessor-class`. If the reader is not the first in a read epoch, line 6 simply adds it to `accessor-class`. Finally, the reader exits the `mutex` mutual exclusion region and `mutex-class` in lines 7 and 8 respectively. The reader, now in the read/write critical region, remains in `accessor-class`.

When we sponsor tasks in `accessor-class`, we are careful to only sponsor the maximum-priority task in this class (by virtue of the pqueue class type). This ensures that the relative order of readers established by their priorities in line 1 is not subverted when `accessor-class` is sponsored. (Note that there is never more than one writer in `accessor-class`, except possibly momentarily after a writer exits the critical region in line 18 but before it exits `accessor-class` in line 19.) For example, if `accessor-class` had class type `all`, all the readers between lines 8 and 9 could have the same priority (from a high priority writer blocked on `wrt`) and thus readers would gain access to the second `mutex` mutual exclusion region in FCFS order rather than in the order of their original priorities.

The exit of readers from the read/write critical region is straightforward. Readers blocked on `mutex` in line 9 again sponsor `mutex-class`. Readers finally exit `accessor-class` in line 14. This exit must follow line 12 to ensure that a waiting writer sponsors the last reader in a read epoch until the reader releases `wrt`.

Writers blocked on `wrt` in line 16 sponsor `accessor-class`. Lines 18 and 19 are straightforward.

[4] When we say that the waiting tasks blocked on a semaphore sponsor a class, we mean that the maximum-priority waiter task sponsors the class.

Initialization

```
(define accessor-class (make-class 'pqueue))
(define wrt (make-sema accessor-class))
(define mutex-class (make-class 'all))
(define mutex (make-sema mutex-class))
(define readcount 0)
```

Reader

```
(wait-sema mutex                                          ; (1)
           (lambda () (enter-class mutex-class)))         ; (2)
(incr1 readcount)                                         ; (3)
(if (= readcount 1)                                       ; (4)
    (wait-sema wrt
               (lambda ()
                  (enter-class accessor-class)))          ; (5)
    (enter-class accessor-class))                         ; (6)
(signal-sema mutex)                                       ; (7)
(exit-class mutex-class)                                  ; (8)
read operation
(wait-sema mutex                                          ; (9)
           (lambda () (enter-class mutex-class)))         ; (10)
(decr1 readcount)                                         ; (11)
(if (= readcount 0)
    (signal-sema wrt))                                    ; (12)
(signal-sema mutex)                                       ; (13)
(exit-class accessor-class)                               ; (14)
(exit-class mutex-class)                                  ; (15)
```

Writer

```
(wait-sema wrt                                            ; (16)
           (lambda ()
              (enter-class accessor-class)))              ; (17)
write operation
(signal-sema wrt)                                         ; (18)
(exit-class accessor-class)                               ; (19)
```

Figure 7.12: A better solution to the readers and writers problem

```
(define (make-rw-serializer)
  (let* ((accessor-class (make-class 'pqueue))
         (wrt (make-sema accessor-class))
         (serialize (make-serializer))
         (readcount 0))
    (cons
      (lambda (read-thunk)
        (serialize
          (lambda ()
            (incr! readcount)
            (if (= readcount 1)
                (wait-sema wrt
                           (lambda ()
                             (enter-class accessor-class)))
                (enter-class accessor-class))))
        (read-thunk)
        (serialize
          (lambda ()
            (decr! readcount)
            (if (= readcount 0)
                (signal-sema wrt))
            (exit-class accessor-class))))
      (lambda (wrt-thunk)
        (wait-sema wrt
                   (lambda ()
                     (enter-class accessor-class)))
        (wrt-thunk)
        (signal-sema wrt)
        (exit-class accessor-class)))))
```

Figure 7.13: A user interface for the readers and writers problem

Note the two parts of this solution as described earlier. We defined a set of classes — `accessor-class` and `mutex-class` — so that each task in a critical/exclusion region is in one or more classes and we defined transitions between these classes to match the trajectory of tasks through the semaphore system. Then we ensured that the tasks blocked on a semaphore always sponsor the class of tasks responsible for releasing the semaphore.

The use of `mutex` and `mutex-class` in Figure 7.12 mirrors in every way the previous serializer example, and thus we could use the `make-serializer` abstraction here.

As before, we can easily define an interface that hides the notion of classes from the user. Figure 7.13 shows one possibility which incorporates our earlier `make-serializer` abstraction. `make-rw-serializer` makes and returns a pair consisting of a reader serializer and a writer serializer. Each of these serializers takes an argument thunk to evaluate in the read/write critical region. The following example illustrates their use.

Initialize:

```
(define rw-serializer (make-rw-serializer))
(define reader (car rw-serializer))
(define writer (cdr rw-serializer))
```

Readers:	Writers:
...	...
; perform a read:	; perform a write:
(reader (lambda () *read-operation*))	(writer (lambda () *write-operation*))
...	...

The producer-consumer problem

This problem may now be solved as shown in Figure 7.14. This solution consists of four classes: `empty-class` for the producers between `(wait-sema empty)` and `(signal-sema full)`; `full-class` for the consumers between `(wait-sema full)` and `(signal-sema empty)`; `ext-producer-class` for all the potential producers external to the critical regions; and `ext-consumer-class` for all the potential consumers external to the critical regions. We use the `make-serializer` abstraction defined earlier to ensure proper sponsorship of tasks in the mutual exclusion region and priority access to this region. Producers originate in `ext-producer-class`, transit to `empty-class` (lines 2 and 3), pass through the serializer while still in `empty-class`, and exit `empty-class` (line 12). Similarly, consumers originate in `ext-consumer-class`, transit to `full-class` (lines 14 and 15), pass through the serializer while still in `full-class`, and exit `full-class` (line 24). If producers and/or consumers continually cycle producing and consuming, respectively, `(enter-class ext-producer-class)` and `(enter-class ext-consumer-class)` should follow lines 12 and 24 respectively.

The idea, sponsorship-wise, is as suggested in Section 7.3.3. Producers blocked on `empty` sponsor `full-class` if there are one or more consumers in that class and `ext-consumer-class` otherwise. We implement this by updating the sema class of `empty` in lines 17 and 23. The counter `num-full-cl` tracks the number of consumers in `full-class`. The binary semaphore `full-cl-sema` ensures atomic updating and testing of this counter. Note that we do not need classes for this semaphore since it always executes as a mandatory task in the *cr-thunk* region. Likewise, consumers blocked on `full` sponsor `empty-class` if there are one or more producers in that class and `ext-producer-class` otherwise. We implement this by updating the sema class of `full` in lines 5 and 11 under the control of the counter `emp-num-cl`, which is guarded by the binary semaphore `emp-cl-sema`. The ability to modify the sema class of a semaphore is essential here since the class responsible for releasing a semaphore may change with time. In essence, the sema class is an indirection cell for sponsorship.

`empty-class` and `full-class` have class type `pqueue` so that we do not disrupt the priority ordering of tasks in these classes. To make progress we only need to sponsor a task in each of these classes, not all the tasks. `ext-producer-class` and `ext-consumer-class` have class type `all` because we neither know nor can predict (in general) which task in these

respective classes will produce or consume. As with the multiple potential determiners problem in Section 7.3.2, we could also use a controller sponsor with some other other sponsorship policy. With class type `all` for these classes, the order in which producers, for instance, blocked on `empty` enter `empty-class` may not be in accordance with their original priorities. In this case, this deficiency is not very important since there can be multiple tasks in the `empty-class` anyway.

The above solution, with its classes and two additional semaphores (we could also use locks), is rather expensive. A tempting cheaper solution is to promote producer and consumer tasks to mandatory status for the duration of the general semaphore critical regions. However, this cheaper solution is deficient: deadlock can occur if all the external producers (or external consumers) are stayed. Thus we need the external producer and consumer classes and some way to sponsor them, and thus we also need the counter variables and their guards. (However, we could use something like a fetch-and-add primitive to test and update the counter variables, thereby avoiding semaphores.) Finally, if tasks sponsoring the external classes have mandatory status, all the tasks in these external classes will be sponsored, unnecessarily, at the maximum priority.

The simulation of monitors

This problem may now be solved as shown in Figure 7.15. This solution has a class `monitor-class` for the task in the monitor mutual exclusion region and two classes for every condition c: `c-class` for all the tasks outside the mutual exclusion region waiting for condition c and `c-producer` for all the potential signallers of condition c outside the mutual exclusion region. The solution closely follows the diagram in Figure 7.9. On entry to the monitor via a procedure call, a task transits from all possible `c-producer` classes (line 2) to `monitor-class` (line 5). On performing a `(cwait c)`, a task transits from `monitor-class` (line 13) to `c-class` (line 10) to await the appropriate `csignal`. (The overlap of `monitor-class` and `c-class` in lines 10 and 13 ensures that a task does not "drop" through a crack between these classes and fail to be sponsored.) When a task per-

Initialization

```
(define ext-producer-class (make-class 'all))     ; external producer class
(define ext-consumer-class (make-class 'all))     ; external consumer class
(define empty-class (make-class 'pqueue))
(define full-class (make-class 'pqueue))
(define empty (make-sema ext-consumer-class N))   ; buffer size is N
(define full (make-sema ext-producer-class 0))
(define serialize (make-serializer))
(define num-emp-cl 0)                             ; # in empty class
(define emp-cl-sema (make-sema))                  ; semaphore for num-emp-cl
(define num-full-cl 0)                            ; # in full class
(define full-cl-sema (make-sema))                 ; semaphore for num-full-cl
```

Figure 7.14: A better solution to the producer-consumer problem (continues on next page)

All potential producers

```
(enter-class ext-producer-class)                    ; all potential producers
```

All potential consumers

```
(enter-class ext-consumer-class)                    ; all potential consumers
```

Producer

```
(wait-sema empty                                    ; (1) wait for an empty slot
           (lambda ()
              (exit-class ext-producer-class)       ; (2)
              (enter-class empty-class)             ; (3)
              (wait-sema emp-cl-sema)               ;
              (if (= num-emp-cl 0)                  ; (4) empty-class was empty
                  (set-sema-class full empty-class)) ; (5)
              (incri num-emp-cl)                    ; (6)
              (signal-sema emp-cl-sema)))
(serialize (lambda () add to buffer ))              ; (7) indivisibly add an item
(signal-sema
    full                                            ; (8) indicate a full slot
    (lambda ()
       (wait-sema emp-cl-sema)
       (decri num-emp-cl)                           ; (9)
       (if (= num-emp-cl 0)                         ; (10) empty-class now empty
           (set-sema-class full ext-producer-class)) ; (11)
       (signal-sema emp-cl-sema)))
(exit-class empty-class)                            ; (12)
```

Consumer

```
(wait-sema full                                     ; (13) wait for a full slot
           (lambda ()
              (exit-class ext-consumer-class)       ; (14)
              (enter-class full-class)              ; (15)
              (wait-sema full-cl-sema)
              (if (= num-full-cl 0)                 ; (16) full-class was empty
                  (set-sema-class empty full-class)) ; (17)
              (incri num-full-cl)                   ; (18)
              (signal-sema full-cl-sema)))
(serialize (lambda () delete from buffer ))         ; (19) indivisibly delete an item
(signal-sema
    empty                                           ; (20) indicate an empty slot
    (lambda ()
       (wait-sema full-cl-sema)
       (decri num-full-cl)                          ; (21)
       (if (= num-full-cl 0)                        ; (22) full-class now empty
           (set-sema-class empty ext-consumer-class)) ; (23)
       (signal-sema-class full-cl-sema)))
(exit-class full-class)                             ; (24)
```

Figure 7.14 continued

forms a csignal, it exits monitor-class (line 24) after first allowing a task waiting for this signal, if any, to enter the mutual exclusion region (lines 21 and 14) and monitor-class (line 18). Finally, on exit from the monitor via procedure return, a task exits the monitor-class (line 8).

Sponsorship works as described in Section 7.3.3. Tasks blocked on s sponsor monitor-class if it contains a task and the most recently signalled c-class otherwise. We implement this by updating the sema class indirection cell for s in lines 3 and 16 when a task enters monitor-class, in line 11 when a task transits from monitor-class to c-class (for a specific condition c), and in line 22 when a task exits monitor-class (for a specific condition c). Tasks blocked on csem sponsor monitor-class if it contains a task and c-producer otherwise (to get a task to do (csignal c)). We implement this by updating the sema class for csem in (1) line 4 (for every condition c) when a task enters monitor-class, (2) line 11 when a task exits monitor-class, (3) line 17 when a task enters monitor-class due to signalling condition c, and (4) line 6 for every condition c with non-empty c-class when a task exits monitor-class. It is not necessary to update the sema class for csem in the *cr-thunk* of line 22 since the signalling of csem in line 21 will awaken another task which will immediately update the sema class in line 17. Note that lines 2, 4, and 6 are duplicated appropriately for each condition c. Thus this solution is awkward for a large number of conditions c.

Finally, c-class and c-producer (for every condition c) should have class type pqueue so that we do not disrupt the priority ordering of tasks in these classes. The class type of monitor-class is unimportant since it contains at most one task.

Semaphore Wrap-up

These last three examples clearly demonstrate the need for semaphore operations like the version of wait-sema and signal-sema and like set-sema-class that we suggested. (We have not implemented these operations.) We need the flexibility to have tasks transit between classes and we need the ability to modify the class a semaphore's waiters sponsor since the class responsible for releasing the semaphore may change with time. This last point is illustrated dramatically by the monitor example wherein one such class is outside any region guarded by semaphores.

The solutions we presented with these operations have two drawbacks. First, these solutions are complicated and expensive. The need for proper class membership and sponsor distribution adds much overhead. Furthermore, many unnecessary priority changes can result from overlapping classes, leading to inefficiency. Second, these solutions lack modularity. They cannot be nested. Class membership of any descendants must be handled explicitly. Nevertheless, our solutions illustrate the expressive power of our sponsor approach.

Monitor	Simulation	
procedure entry	`(wait-sema s`	; (1)
	`  (lambda) ()`	
	`  (exit-class c-producer)`	; (2)
	`  (set-sema-class s monitor-class)`	; (3)
	`  (set-sema-class csem monitor-class)`	; (4)
	`  (enter-class monitor-class)))`	; (5)
procedure exit	`(if (> ccount 0) (set-sema-class csem c-producer))`	; (6)
	`(signal-sema s)`	; (7)
	`(exit-class monitor-class)`	; (8)
	procedure return	
	For every condition c:	
`(cwait c)`	`(incri ccount)`	; (9)
	`(enter-class c-class)`	; (10)
	`(set-sema-class csem c-producer)`	; (11)
	`(signal-sema s)`	; (12)
	`(exit-class monitor-class)`	; (13)
	`(wait-sema csem`	; (14)
	`  (lambda) ()`	; (15)
	`  (set-sema-class s monitor-class)`	; (16)
	`  (set-sema-class csem monitor-class)`	; (17)
	`  (enter-class monitor-class)))`	; (18)
	`(exit-class c-class)`	; (19)
	`(decri ccount)`	; (20)
`(csignal c)`	`(if (> ccount 0)`	
	`  (signal-sema csem`	; (21)
	`  (lambda ()`	
	`  (set-sema-class s c-class)))`	; (22)
	`  (signal-sema s))`	; (23)
	`(exit-class monitor-class)`	; (24)
	procedure return	

Figure 7.15: Better solution for simulating a monitor with binary semaphores

7.5 Summary

Side-effects provide a means for intertask synchronization. With speculative computation this means that tasks may be relevant for the side-effects that they may perform. The problem is to ensure this relevance (Property 1) and, more generally, ensure demand transitivity (Property 2). We also desire priority access to critical regions (Property 3).

We presented solutions to these problems for three common synchronization mechanisms with side-effects: spin-locks, placeholders, and semaphores. For concreteness we presented these solutions in the context of our touching model. One can easily generalize these solutions to the general sponsor model.

These solutions yield a spectrum of synchronization techniques. The appropriate synchronization technique depends on the complexity of the required synchronization, the cost of the synchronization mechanism, the duration a task spends in critical regions, and the degree with which we desire to meet Properties 2 and 3. For very cheap synchronization, we can use simple spin-locks and non-preemptable regions. In doing so, we over-achieve Property 2 and abandon Property 3. For intermediate cost, we can use the delay device which has the overhead of blocking to satisfy Property 2 but no priority queue to satisfy Property 3. The delay device is, of course, limited in its applicability. To meet both Property 2 and Property 3 we need both blocking and priority queues, which are expensive.

Chapter 8

Applications

In this chapter we consider several applications which exploit speculative computation in various ways. These applications illustrate the issues with speculative computation, demonstrate our approach to speculative computation with our touching model, and provide examples of the language features of our touching model. All the execution times listed in this chapter were obtained running the specified application with our modified version of the Multilisp byte-code interpreter (see Chapter 10) on either the Concert Multiprocessor, a 32 processor Motorola 68000 based machine [Hals86a], or the Encore Multimax, a 16 processor National 32332 based machine. Each processing element of Concert is about 0.5 to 1 MIPS and each processing element of the Multimax is about 1.5 to 2 MIPS, or approximately two to three times as fast as that on Concert (depending on the application). Concert failed during the final stages of data collection so the Concert data is not as complete as we desired. (Six measurements are missing in Table 8.12.) Since the Multimax was more convenient to use than Concert, we used the Multimax whenever the number of processors was not important to the point we were making. In fact, in the two applications for which we used the Multimax, we only used 8 of its 16 processors.

8.1 Por and Pand

In this section we consider parallel or and and, which we call por and pand respectively. These two nondeterministic operators represent perhaps the most important potential application of speculative computation because of the ubiquity of or and and. Their implementation raises a number of issues central to nondeterministic and multi-approach speculative computation, both in general and in the touching model.

We gave informal semantics for por and pand in Section 1.2.1. See Appendix B for precise semantics. For now we assume por and pand to be syntactic constructs, i.e. macros. Since por is strictly more powerful than pand with our definitions, we concentrate on por in the following sections.

8.1.1 Requirements

Our objective is a version of por which returns the result in minimum time. We assume that sufficient computational resources are available so that we can potentially trade these resources for reduced average execution time. That is, we assume an environment conducive to speculative use of resources. Since there may be other activities competing for these resources, a second objective is to avoid wasting resources. In this context, an implementation of por has five requirements:

1. Initialization – create a task to evaluate each disjunct E_i[1]
2. Race officiating – return the first true value
3. Termination detection – return nil if all E_i evaluate to nil
4. Computation reclamation – abort any remaining (useless) tasks after the first true value is returned
5. Task scheduling – schedule the allocation of the available resources to the disjunct tasks and their descendants to minimize the expected time for por to return a result

This fifth requirement is very general — and difficult. We study this requirement in Chapter 9. For this chapter we satisfy ourselves with a simpler and more practical version of this requirement:

5. Evaluate the disjuncts E_i as concurrently as possible, but in the specified order if processing resources are limited. Unless given explicitly by priorities, the specified order of evaluation is the left-to-right order of disjuncts in por (i.e. FCFS order with respect to the indices i).[2]

Thus, as available resources decrease, por should reduce gracefully to the semantics of sequential or. This allows users to arrange the order of the E_i for minimal execution time with limited resources. Of course, we have no way to enforce this order once we spawn the disjunct tasks since the tasks could block. Even task priorities do not guarantee such an ordering, since a task could still block; priorities only make the ordering more likely. Therefore, we will consider requirement 5 as pertaining to the order in which we spawn disjunct tasks.

8.1.2 Mandatory por

If we ignore requirement 4, we can implement por using conventional mandatory tasks as shown in Figure 8.1. (Note that this implementation is an extension of the multiple

[1] This may not always be the optimum policy: creating a task for each disjunct may increase the execution time if there are insufficient processors available. We ignore this problem for now.

[2] The evaluation order of disjuncts with the same explicit priorities is also left-to-right.

potential determiners example of Figure 7.5.) For simplicity we show por implemented as a function. (We rewrite (por E_1 E_2 ... E_n) to

(por-function (list (lambda () E_1) (lambda () E_2) ... (lambda () E_n)))

by macro-expansion.[3] For simplicity we refer to por-function as por. The ambiguity between por as a macro and a function will be resolved by context.)

Before explaining the implementation in detail we give a quick overview. The implementation creates a placeholder for the result and a task to evaluate each thunk in the argument list. Each task performs race officiating and termination detection. If the thunk evaluates to a true value and it is the first thunk to do so, the task determines the result placeholder to true. If the thunk evaluates to nil and all the other thunks have already done so, the task determines the result placeholder to nil. We perform this distributed termination detection by counting the total number of thunks and the number of thunks that have evaluated to nil so far. For n thunks we have $n+1$ tasks: a parent task spawns n child tasks, one for each thunk, and then awaits the result. This allows the parent to continue as soon as a result is found, regardless of which task finds it.

Now we explain the implementation in detail. Line 1 initializes a lock cell. Its car is used for race officiating synchronization. Initially the car is the symbol *no-result* to signify that no thunk has yet evaluated to true. The cdr of the lock cell is used for termination detection. Initially it is a placeholder for the number of thunks. If we knew this number, as we would with the macro version of por, we would not need this placeholder. We will know the number of thunks after we create a task for each one. Until then, the placeholder serves as a convenient synchronization mechanism which allows tasks to proceed until they actually need the total number of thunks. Line 2 initializes the result placeholder. Line 13 creates a task to spawn the thunk evaluators so that the result may be returned in line 14 without waiting for any of the spawned tasks.

The procedure spawn-tasks creates a task for each thunk and maintains a count of the number of thunks. Line 3 determines the thunk number placeholder to the total number of thunks. Line 6 evaluates a thunk. If the thunk evaluates to true, we perform the race officiating in line 7. The rplaca-eq in this line indivisibly compares the car of the lock cell with the symbol *no-result* and if eq, replaces the car with the symbol *result* and returns true. Otherwise, the rplaca-eq returns nil. That is, we atomically check if this is the first task to get to this point with a true thunk and update the car of the lock cell if so. If the rplaca-eq returns true, indicating the task was indeed first, we determine the result placeholder to the thunk result in line 8. If on the other hand, the thunk evaluates to nil, we perform the termination detection on line 10. First, though, we perform the optimization on line 9: we only proceed to termination detection if no task has yet returned true.

The procedure term-detect performs termination detection. Line 11 attempts to indivisibly decrement the thunk count in the cdr of the lock cell. If the count is decremented

[3] In practice we would implement the macro version directly rather than call a helper function.

```
(define (por thunk-list)
  (let ((lock (cons '*no-result* (make-future)))          ; 1
        (result (make-future)))                           ; 2

    (define (spawn-tasks thunk-lst n)
      (if (null thunk-lst)
          (determine-future (cdr lock) n)                 ; 3
          (begin
            (future (eval-thunk (car thunk-lst)))         ; 4
            (spawn-tasks (cdr thunk-lst) (+ n 1)))))      ; 5

    (define (eval-thunk thunk)
      (let ((v (thunk)))                                  ; 6
        (if v
            (if (rplaca-eq lock '*result* '*no-result*)   ; 7
                (determine-future result v))              ; 8
            (if (eq (car lock) '*no-result*)              ; 9
                (term-detect (cdr lock))))))              ;10

    (define (term-detect n)
      (if (rplacd-eq lock (- n 1) n)                      ;11
          (if (= n 1)
              (determine-future result nil))              ;12
          (term-detect (cdr lock))))

    (future (spawn-tasks thunk-list 0))                   ;13
    result))                                              ;14
```

Figure 8.1: por implemented with mandatory tasks

For simplicity, we assume the argument thunk-list is non-null.

and the count was one — thus all thunks have evaluated to nil — line 12 determines the result placeholder to nil. If the count was not decremented, we call `term-detect` to try once again.

The futures in Figure 8.1 require an explanation. The `dfuture` (see Section 1.4) in line 13 removes task creation and thunk evaluation from the critical path of returning a result, thus meeting the overall objective (return the result in the minimum time). It must be `dfuture` rather than `future` to avoid queueing the parent continuation (line 14) until the call to `spawn-tasks` returns if there are insufficient processors available. The `future` in line 4 creates a task to evaluate each thunk until either exhausting `thunk-lst` or running out of processors. In the latter case, the parent continuation (line 5) is queued (see Section 1.4.2 on scheduling `future` and `dfuture`) and no more tasks are created until a free processor grabs this continuation from the queue. This ensures the left-to-right spawn order of disjunct tasks for requirement 5.

The overhead in this implementation of por is fairly high, but then the requirements on por are fairly complex. Any significant reduction in overhead (other than optimizing for a one-element argument list) requires implementation support from the Lisp system.[4]

8.1.3 Speculative por: version 1

In this section we address requirement 4: we consider a version of por which includes computation reclamation, as well as meeting the other four requirements. Thus this version employs speculative tasks so they may be stayed. Figure 8.2 shows this version. The line numbers indicate the lines which differ from the mandatory por version.

`make-group` in line 3 creates a speculative task, with priority `*pri*`, to spawn the thunk evaluators. Line 1 creates a speculative task to evaluate each thunk with priority `*pri*`. Unless it is desired to evaluate a thunk with a priority other than `*pri*`, the `spec-future` in this line could just as well be a `future` since the children of speculative tasks inherit the priority of their parent unless otherwise indicated. `make-group` returns a group object which uniquely names the tree of speculative tasks consisting of the `make-group` task, the disjunct tasks, and all their descendants (unless they are touched from outside por by mandatory tasks). However, this group object is ignored here;[5] instead, we call `make-group` for its effect.

Line 2 stays the group after we have detected the first true-valued thunk and returned the result. This stays all the tasks in that group and any descendant groups created as the result of disjunct evaluation, i.e. stays all the activity that por created (except if that activity is presently sponsored by some activity outside the por). This is exactly the behavior we desire from por in most cases.

Sometimes, though, we might like to allow the children of the first true thunk to continue

[4]Note, for instance, that we do not need future objects in lines 4 and 13, only the ability to create a task.

[5]We cannot rely on the group object returned here since a group descendant may require the group object (in line 2) before the `make-group` continuation receives and binds the group object.

```
(define (por thunk-list)
  (let ((lock (cons '*no-result* (make-future)))
        (result (make-future)))

    (define (spawn-tasks thunk-lst n)
      (if (null thunk-lst)
          (determine-future (cdr lock) n)
          (begin
            (spec-future (eval-thunk (car thunk-lst)) *pri*)      ; 1
            (spawn-tasks (cdr thunk-lst) (+ n 1)))))

    (define (eval-thunk thunk)
      (let ((v (thunk)))
        (if v
            (if (rplaca-eq lock '*result* '*no-result*)
                (begin
                  (determine-future result v)
                  (stay-group (group-id (my-group-obj))))         ; 2
            (if (eq (car lock) '*no-result*)
                (term-detect (cdr lock))))))

    (define (term-detect n)
      (if (rplacd-eq lock (- n 1) n)
          (if (= n 1)
              (determine-future result nil))
          (term-detect (cdr lock))))

    (make-group (spawn-tasks thunk-list 0) *pri*)                 ; 3
    result))
```

Figure 8.2: por implemented with speculative tasks; version 1

For simplicity, we assume the argument thunk-list is non-null.

after por returns, i.e. we might want to return a partial result as the value of a thunk. We can solve the partial result problem in this case by evaluating each thunk within a separate group and staying all the groups other than the one corresponding to the first true thunk. This is not a very satisfactory solution, though, for two reasons. First, the maintenance of the group list can be awkward. Second, we cannot continue just some of the descendants of the first true thunk and not others. Recall that our touching model currently has no specific mechanism for partial results (see Section 6.3).

The scheduling of tasks works as we desire provided the task invoking por has priority greater than or equal to *pri*. In this case the task created by make-group in line 3 defers to the parent task if there are not enough processors since this task's priority is not greater than the par[illegible]'s priority. The speculative tasks created by spawn-tasks each have the same pri[illegible] *pri* and thus they are processed in FIFO order behind the parent and make-group tasks if there are insufficient processors. (As described in Section 10.2 speculative tasks with the same priority are queued in FIFO order.)

This version of por has two major problems:

1. Deadlock

   If the por is stayed — as a result of being embedded in a larger speculative computation which is stayed — and then restarted when another task touches it, the thunk evaluations will not be restarted, leading to deadlock. The result placeholder breaks the sponsor chain; it fails to pass the touch sponsorship on to the thunk evaluator tasks.

2. What should the task priorities be?

   Fixing all the thunk evaluator priorities at *pri* ignores the relative promise of the thunks (to return a true value) and the relative promise of the por with respect to other speculative activities. Consequently, the thunk tasks could run at a priority less than the maximum priority task touching the por, which could subvert the desired ordering and lead to poor performance: tasks of priority less than the maximum priority por toucher could continually preempt the thunk tasks. The deadlock problem mentioned above is really the extreme of this problem. Another problem is that the task invoking por could have a priority less than *pri*. In this case, the invoking task could be blocked by the make-group task if there are insufficient processors available. To ensure that the make-group task is not blocked by a thunk task if there are insufficient processors available, the thunk tasks must be spawned with a priority less than or equal to the priority of the make-group task. (This ensures the proper spawn order of the thunk tasks if there are insufficient processors.)

8.1.4 Issues with por in a speculative environment

Generalizing from the previous section, there are the following issues involved with por in a speculative environment:

1. Scheduling of disjunct tasks

   Task scheduling divides into two components:

   (a) pre-demand scheduling – How should the disjunct tasks be scheduled before por is demanded by a touch?

   This amounts to the initial priority to give the disjunct tasks and how these priorities should be managed in the face of changes within the por, e.g. children of disjunct tasks created and terminated, and outside the por. This is the eager component of por scheduling.

   (b) post-demand scheduling – How should the disjunct tasks be scheduled after por is demanded by a touch?

   How should the touch sponsorship be distributed? To ensure demand transitivity, at least one of the disjunct tasks must have the priority of the maximum priority waiter. The priority of disjuncts must be dynamic to ensure such demand transitivity in the face of priority changes in por touchers. This is the demand-driven component of por scheduling.

   In a more general view, these two components are really the same: how should the por sponsorship be distributed across the disjunct tasks? If we had full controller sponsors we could address this question in this more general manner, but we do not, so we address these components separately.

2. Modularity

   We need modularity to maintain the desired relative priorities — both the eager and demand-driven components — in the face of changes in whatever computation may contain the por. Such modularity would prevent deadlock if the por was stayed and restarted when touched.

   However, we do not have modularity in our touching model, so to prevent deadlock we must at least have group coherency: when por is restarted after being stayed, all the relevant tasks comprising por must be restarted. To do this we need "demand" continuity from the por toucher to all the relevant tasks.

These issues lead to the following four requirements, which we think of as additions to our simplified requirement 5 in Section 8.1.1:

1. The user must be able to specify initial priorities for the disjuncts

2. At least one disjunct must obey demand transitivity

3. There must be demand continuity to ensure group coherence

4. por must spawn the disjunct tasks as quickly as possible without being preempted and blocking the task invoking the por.

8.1.5 Speculative por: version 2

In this section we present a version of por which meets these additional four requirements. This version adopts very simple (and inflexible) methods to address the underlying issues but it is a start.

We let the user list an initial priority with each disjunct in the argument list. We assume that the user arranges the disjuncts so that they are in non-increasing priority order from left to right. Thus a static left-to-right spawning order encourages the evaluation of the highest priority disjuncts first if there are insufficient processors available while conforming to our earlier requirement 5. The initial priorities let the user convey the relative promise of the thunks. The management of these priorities in face of changes inside and outside the por is left to the user. Thus the deeper issues of initial scheduling have been pushed to the user level.

We combine groups and classes to obtain a group with a primitive controller sponsor. This allows us to easily name all the por tasks for staying and provide group coherency. Task source priorities, specified by the initial disjunct priorities, provide the eager scheduling component. The class sponsor distributes touch sponsorship, thereby ensuring demand transitivity and providing the demand-driven scheduling component. To provide group coherency we insist on demand "continuity": the sponsor chain must be unbroken from the result placeholder to the disjunct tasks.

In this version, shown in Figure 8.3, we use a type `all` class to sponsor all the disjunct tasks. We call this the *touch all* policy because the class effectively distributes the touch to all the disjunct tasks. The marked lines in Figure 8.3 indicate the major changes from Figure 8.2. Line 2 creates the type `all` class for all the disjunct tasks. Line 3 installs this class as the sponsor class of the result placeholder. Thus the maximum-priority task touching the result placeholder sponsors all the class members. Line 7 adds the `make-group` task created in line 9 to the class so that disjunct spawning may be restarted if the por is restarted after being stayed. (The `group-future` in line 8 extracts the future object from the group object created in line 9.) Note that in order for the result placeholder to become available, the por invoker must execute lines 7 to 9 and therefore the `make-group` task must be a member of the class before any tasks demand the por result. Thus there is no lapse in demand continuity here that could lead to deadlock. Finally, line 5 adds the disjunct tasks to the class so that they will be sponsored by the tasks blocked on the result placeholder and thus may be restarted. Since any tasks demanding the por result sponsor the `make-group` task, this task must execute line 5 (provided some disjunct does not first determine the result) and thus there is no lapse in demand continuity here either that could lead to deadlock.

The `make-group` task created in line 9 has maximum priority (`*max-pri*` is bound to *MAX*) so that it quickly spawns all the thunk evaluator tasks. Since this is all that this task does, which should not take very long, we do not worry about this task possibly blocking the parent task if there are insufficient processors.

Once the procedure `eval-thunk` determines the result placeholder, the class is no longer

```
(define (por thunk-pri-list)                                    ; 1
  (let* ((lock (cons '*no-result* (make-future)))
         (class (make-class *class-all*))                       ; 2
         (result (make-future class)))                          ; 3

    (define (spawn-tasks thunk-pri n)
      (if (null thunk-pri)
          (determine-future (cdr lock) n)
          (let ((thunk (get-thunk (car thunk-pri)))
                (priority (get-priority (car thunk-pri))))
            (add-to-class                                       ; 4
               (spec-future (eval-thunk thunk) priority)        ; 5
               class)
            (spawn-tasks (cdr thunk-pri) (+ n 1)))))

    (define (eval-thunk thunk)
      (let ((v (thunk)))
        (if v
            (if (rplaca-eq-mand lock '*result* '*no-result*)    ; 6
                (begin
                  (determine-future result v)
                  (stay-group (group-id (my-group-obj))))
                (if (eq (car lock) '*no-result*)
                    (term-detect (cdr lock))))))

    (define (term-detect n)
      (if (rplacd-eq lock (- n 1) n)
          (if (= n 1)
              (determine-future result nil))
          (term-detect (cdr lock))))

    (add-to-class                                               ; 7
      (group-future                                             ; 8
        (make-group (spawn-tasks thunk-pri-list 0) *max-pri*))  ; 9
      class)
    result))
```

Figure 8.3: por implemented with speculative tasks; version 2

thunk-pri-list is a list of thunk, priority pairs.

sponsored. This would stay the disjunct tasks if they had no other support. However, they still might be sponsored by their source priority settings and descendants of the disjunct tasks would not necessarily be stayed. Thus we still need to stay the group (and thus we still need the make-group construct in line 9). This poses a problem. We want to stay the group and we want to do it after we determine the result placeholder so staying is not in the critical path of returning the result. However, the determining task may be stayed as a result of determining the result placeholder and never get to stay the group. To solve this problem we promote the determining task to mandatory status with the rplaca-eq-mand in line 6.[6]

Unlike the previous version of por, this version actually uses the future objects returned by make-group and spec-future for sponsor propagation.

One problem remaining with this version is that the initial priorities — which represent the eager component — are lost if we stay the por since there is no modularity. However, at least there is no deadlock since the class provides "demand" continuity.

8.1.6 Speculative por: version 3

This version of por is the same as the previous version except for the touch policy. This version sponsors the disjunct tasks (by actually touching them) in the left to right order of their appearance in the argument list. It touches each task until either the task terminates (returning true or false) or a true result is found elsewhere and all por activity is stayed. We call this the *touch order* policy. It only touches tasks when necessary and thus is the opposite of the *touch all* policy.

Figure 8.4 shows this version. Once again we use a class to ensure demand continuity to the spawner task. The spawner task spawns n thunk evaluator tasks, and then touches each task in turn in line 8 until it finds a true result or is stayed. If spawner finds a true result, the result placeholder must have already been determined so spawner just terminates. If the spawner does not find a true result, it determines the result placeholder to nil. Thus spawner performs centralized termination detection in this version of por. To facilitate this centralized termination detection, each of the disjunct tasks returns its value (line 5).

In line 7 we reduce the priority of the spawner task to the minimum running priority (*min-pri* is bound to 1) so the parent task will not be blocked by the spawner task if there are insufficient processors and so that termination detection does not disturb the initial priority of the disjunct tasks (until the por is touched).

Centralized and distributed termination detection each have their advantages and disadvantages. The centralized termination detection here requires an extra task (for termination detection) whereas the distributed termination detection in previous versions spreads this overhead amongst all the disjunct tasks. It is not clear which is more efficient. (In fact,

[6]The promotion to mandatory status does not have to be atomic with the lock here. Instead, we could insert promote-task just before determine-future.

```
(define (por thunk-pri-list)
  (let* ((lock (cons '*no-result* nil))                          ; 1
         (class (make-class *class-all*))                        ; 2
         (result (make-future class)))

    (define (spawn-tasks thunk-pri)
      (if (not (null thunk-pri))
          (let ((thunk (get-thunk (car thunk-pri)))
                (priority (get-priority (car thunk-pri))))
            (cons (spec-future (eval-thunk thunk) priority)      ; 3
                  (spawn-tasks (cdr thunk-pri))))))

    (define (eval-thunk thunk)
      (let ((v (thunk)))
        (if v
            (if (rplaca-eq-mand lock '*result* '*no-result*)     ; 4
                (begin
                  (determine-future result v)
                  (stay-group (group-id (my-group-obj))))))
        v))                                                      ; 5

     (define (spawner thunk-pri-lst)                             ; 6
       (let ((thunk-values (spawn-tasks thunk-pri-lst)))
         (change-priority (my-task) *min-pri*)                   ; 7
         (if (not (true-value thunk-values))                     ; 8
             (determine-future result nil))))

     (define (true-value value-list)
       (if (not (null value-list))
           (if (car value-list)
               '#t
               (true-value (cdr value-list)))))

     (add-to-class                                               ; 9
       (group-future                                             ;10
         (make-group (spawner thunk-pri-list) *max-pri*)
       class)
     result))
```

Figure 8.4: por implemented with speculative tasks; version 3

thunk-pri-list is a list of thunk, priority pairs.

Version	Avg. time to return placeholder	Avg. time to determine result			
		Identical disjuncts	Number of disjuncts		
			1	2	4
Mandatory por	6.9	(lambda () '#t)	15.9	16.5	15.7
		(lambda () nil)	20.1	25.9	36.5
por version 1	8.2	(lambda () '#t)	19.0	19.6	18.6
		(lambda () nil)	21.2	26.5	42.2
por version 2	12.1	(lambda () '#t)	22.2	23.1	21.6
		(lambda () nil)	26.9	38.5	55.1
por version 3	12.1	(lambda () '#t)	21.5	21.9	20.1
		(lambda () nil)	25.2	32.1	47.4

All times in milliseconds, on Concert Multiprocessor with 32 processors. Standard deviation in all results was about 0.5 msec.

Table 8.1: por operation times

the data in Table 8.1 suggests that the overheads are about the same.) In any case, the centralized termination detection is combined with a sponsor distribution policy that must be centralized for efficiency anyway.

8.1.7 Measurements

To gauge the overhead contributed by our support for speculative computation, we measured the performance of the four por versions we presented. For each version, we measured the time required on the Concert Multiprocessor (using 32 processors) to:

1. return the result placeholder once por is called. We call this time the invocation overhead — it is the minimum time for which the progress of a task is impeded after invoking por.

2. return the result (i.e. determine the result placeholder) for argument lists of 1, 2, and 4 thunks. The thunks were either all `(lambda () '#t)` — to evaluate the time to return the first true value — or `(lambda () nil)` — to evaluate the time to spawn all the tasks and return false.

Table 8.1 shows the results. With each version, the return time for `(lambda () '#t)` thunks first increased slightly as the number of thunks went from 1 to 2 and then decreased as the number of thunks further increased to 4. The reason for this phenomenon is not clear; the phenomenon did not occur on the Encore Multimax. `(lambda () '#t)` takes about 0.8 msec to evaluate, which is about a third of the time that it takes to spawn a task. Thus the first task spawned always determines the result placeholder. Therefore the

Version	Invocation overhead	Setup overhead	Spawn overhead
Mandatory por	7	16	5
por version 1	8	19	7
por version 2	12	22	9
por version 3	12	21	7

All times are averages in milliseconds.

Table 8.2: por overhead on Concert Multiprocessor

determine result time for this type of thunk really indicates the time until the first task is spawned. We call this time the setup overhead. Likewise, (lambda () nil) takes much less time to evaluate than the time to spawn a task. Thus the determine result time for this type of thunk really indicates the time to spawn all the tasks. This is the reason the determine result time with (lambda () nil) thunks increases with the number of thunks. The differential in determine result times with n and $n+1$ thunks is the por's overhead of spawning a task (calling spawn-task, actually creating the task, and perhaps adding it to a class or whatever). We call this time the spawn overhead.

Table 8.2 indicates the overhead times. With the mandatory version as a base case, we see that group and speculative task creation in version 1 impose an additional 1, 3, and 2 msec. in the invocation, setup, and spawn overheads respectively. Class creation and manipulation in version 2 add a further 4, 3, and 2 msec. to these overheads respectively. Version 3 has almost the same overhead as version 2 except for the spawn overhead, which reflects the lack of class manipulation time in other than the make-group task. Versions 2 and 3 are about the simplest versions that meet the minimal requirements for por and thus their rather large overheads are not encouraging. However, there are many optimizations remaining for task creation and class manipulation that should substantially reduce these overheads. Certainly, the overhead with more complex versions, such as with full controller sponsors, will be much greater.

8.1.8 Generalizations

There are three major generalizations to the versions of por presented here:

1. Arbitrary policies for the scheduling and management of disjunct tasks.

   Since we do not have full controller sponsors, this generalization is currently beyond our scope.

2. Dynamic number of disjuncts

```
(define (tree-equal? tree1 tree2)
  (if (pair? tree1)
      (if (pair? tree2)
          (and (tree-equal? (car tree1) (car tree2))
               (tree-equal? (cdr tree1) (cdr tree2)))
          nil)
      (equal? tree1 tree2)))
```

Figure 8.5: Sequential version of tree equal

Allow the ability to create disjuncts for the por calculation other than those initially specified as arguments. This takes por more into the realm of tree search. It is quite straightforward to extend our versions of por in this direction.

3. Result streams

   Return successive results in a stream. Again, it is quite straightforward to extend our versions of por in this direction.

Using the basic primitives that we have provided, the user can build his own "libraries" of different constructs like por to suit the applications at hand. Unfortunately, doing so is trickier than we would like due to concerns about demand continuity to prevent deadlock.

8.2 Tree Equal

This application is essentially tree structured pand, the second generalization of pand mentioned in Section 8.1.8. This application clearly demonstrates the importance of aborting useless computation, though it is a bit contrived. However, the benefit of artificiality in this case is a clear and simple presentation of the issues. We use this application to demonstrate why we want special support for aborting useless computation and not explicit termination checking.

The tree equal problem is to determine if two trees are equal in the usual Lisp sense (see p. 14 of [Rees]), that is, if the two trees have the same structure and the fringe (leaf) elements are respectively equal. Figure 8.5 shows a sequential solution to the tree equal problem for binary trees composed of conses.

In the remainder of this section, we explore parallel versions of this solution. Figure 8.6 shows a version which we call the eager version. This eager version creates $2^{\texttt{*fork-level*}}$ tasks which each perform `tree-equal?` sequentially on subtrees of the original trees. The first task to discover unequal subtrees determines the result placeholder to nil (by calling `fail`), causing `etree-equal?` to return nil. If all the tasks find that their subtrees are equal, the line labeled 1 determines the result placeholder to `'#t`. Thus this version amounts to

```
(define (etree-equal? tree1 tree2)
  (let ((lock (cons '*undetermined* nil))
        (result (make-future)))

    (define (int-equal t1 t2 level)
      (if (pair? t1)
          (if (pair? t2)
              (if (< level *fork-level*)
                  (let ((right-branch
                         (future
                           (int-equal (cdr t1) (cdr t2) (+ level 1)))))
                    (and (int-tree-equal (car t1) (car t2) (+ level 1))
                         right-branch))
                  (if (and (tree-equal? (car t1) (car t2))
                           (tree-equal? (cdr t1) (cdr t2)))
                      '#t
                      (fail))
          (if (equal? t1 t2)
              '#t
              (fail))))

    (define (fail)
      (if (rplaca-eq lock '*determined* '*undetermined*)
          (determine-future result nil))
      nil)

    (future (if (int-equal tree1 tree2 0)
                (determine-future result '#t)))                ; 1
    result))
```

Figure 8.6: Eager version of tree equal

Tree compared with tree0	Number of errors	Version		Speedup of Eager
		Sequential	Eager	
tree0	0	13.5	1.78	7.6
tree1	3	1.61	0.67	2.4
tree2	1	3.10	1.39	2.2
tree3	1	6.38	1.31	4.9
tree4	1	12.1	0.36	34
tree5	1	4.27	0.98	4.4
tree6	2	10.1	0.53	19
tree7	1	2.79	1.04	2.7
tree8	2	0.70	0.71	1.0
tree9	3	0.92	0.90	1.0

All times are in seconds.

Table 8.3: Execution time of tree equal versions on various trees

the (mandatory) pand of `tree-equal?` on the subtrees. (Compare with the mandatory por in Section 8.1.2). We could also have implemented this eager version as nested pands. We chose to "flatten" the tree into a single pand for efficiency reasons. This eager version adds conjuncts dynamically once it has started; thus it is an example of the second generalization of pand discussed in Section 8.1.8. This version incorporates static scheduling controlled through the the global variable `*fork-level*` because we wanted to avoid the issues with dynamic scheduling[7] for this application and focus on aborting useless computation (which we will do shortly). All the results reported in this section are for `*fork-level*` set to 3 and 8 processors, so that $2^{\texttt{*fork-level*}} \leq$ the number of processors.

To compare the performance of different versions, we created a "base tree" of depth 13 (8192 leaf elements) and from this base tree generated nine other binary trees with the same number of elements by randomly miscopying elements of the base tree. Each element of the base tree was miscopied with probability 0.0001. We ran both the sequential and eager versions comparing each of these nine trees with the base tree, tree0. Table 8.3 shows the results for 8 processors on the Encore Multimax. The speedup of the eager version over the sequential version depends greatly on the location of the error(s) in the trees. The appreciable, and even superlinear, speedup that we get demonstrates that speculative computation is useful in this application. Another attribute of speculative computation in this application is a reduction in the variance of the execution time. The execution time of the sequential version varies from 0.70 sec. to 10.1 sec., whereas the execution time of the eager version varies from 0.36 sec. to 1.78 sec., a fivefold reduction in the dynamic range.

[7]The issues surround how to "impedance match" the application parallelism to machine resources. Or, more simply, how to keep an unknown number of processors busy without creating and buffering an excessive number of tasks. Impedance mismatch is a major weakness in Multilisp. Methods to deal with such mismatch are the subject of some promising current research. See the remarks on "lazy" futures in [Kranz].

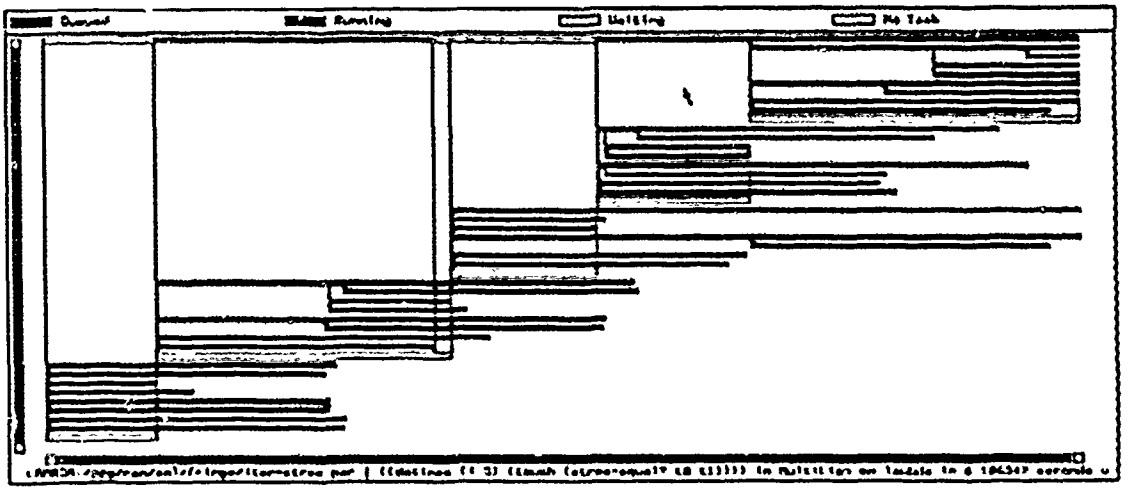


Figure 8.7: Consecutive invocations of `etree-equal?`

Such robustness in execution time is a general property of speculative computation which arises from trying several alternatives concurrently.

Although the eager version achieves the aim we want most with speculative computation — reduced execution time — it has a problem: it does not abort useless computation. The effect of useless computation does not show up in Table 8.3 because there was no other computation competing for machine resources. This would not normally be the case in practice: we would expect the tree equal problem to be embedded in some larger computation. To demonstrate the effect of useless computation, we ran five consecutive invocations of the tree equal problem with the eager version. The average execution time per invocation comparing tree0 and tree1 was 1.22 sec., almost double the execution time (0.67 sec.) of a single invocation.

The ParVis display (See Appendix C for an explanation of ParVis displays) in Figure 8.7 clearly shows what happened. Each invocation is demarcated by the heavy vertical line denoting task creation. The first invocation, for instance, is in the lower left corner. This invocation creates eight tasks, the dark horizontal bars. The hollow horizontal bar is the result placeholder; its length denotes the execution time of the invocation. This first invocation returns a result when the third task (from the top) discovers a mismatch and determines the result placeholder to nil. The seven other tasks, now useless, remain running and interfere with the second invocation. Likewise, useless tasks from each invocation interfere with the following invocation.

Thus this example, while a bit artificial with its large trees and consecutive invocations, clearly demonstrates the importance of aborting useless computation. The only issue is how: should the user cause every task to explicitly check for termination or should there be special support in the language for terminating tasks?

We introduce a version of tree equal with explicit termination detection by first presenting a naive parallel version in Figure 8.8. This naive version merely performs `tree-equal?`

```
(define (ptree-equal tree1 tree2)

  (define (int-equal t1 t2 level)
    (if (pair? t1)
        (if (pair? t2)
            (if (< level *fork-level*)
                (let ((right-branch
                        (future
                          (int-equal (cdr t1) (cdr t2) (+ level 1)))))
                  (and (int-equal (car t1) (car t2) (+ level 1))
                       right-branch))
                (and (tree-equal? (car t1) (car t2))
                     (tree-equal? (cdr t1) (cdr t2))))
            nil)
        (if (equal? t1 t2)
            '#t
            nil)))

  (int-equal tree1 tree2 0))
```

Figure 8.8: Naive parallel version of tree equal

on all the subtrees in parallel and ands all the results. Not only does this version possibly generate useless computation, but it also incorporates no indeterminacy in which subtree returns a false result. If a mismatch is detected in the rightmost subtree, this result cannot be returned until all the subtree matches to its left have completed. That is, the naive version performs termination detection of nondeterministic computations in a fixed, deterministic order.

Figure 8.9 shows the naive version modified to perform explicit termination detection. We call this new version the checking version since every task periodically checks for termination. The first task to discover unequal subtrees sets the `failed-yet?` flag (by calling `fail`) causing the other tasks to terminate, returning a nonsense value, when they call `tree-equal?`. Since terminated tasks return a value we can still perform the termination detection in a fixed, deterministic order, like in the naive version, but now return the first false result. Note, however, that all code called by the checking version — just `tree-equal?` in this case — had be modified to perform termination checking.

By contrast, the speculative version shown in Figure 8.10 required no changes in called code; it is basically just pand modeled on version 3 of por in Section 8.1 — each task touches its children to ensure their completion if a result has not already been returned. The global variable `*fork-priority*` specifies the initial priority of the tasks.

Table 8.4 shows the execution time of the eager, checking, and speculative versions for

```
(define (ctree-equal? tree1 tree2)
  (let ((failed-yet? nil))

  (define (fail)
    (setq failed-yet? '#t)
    nil)

  (define (int-equal t1 t2 level)
    (if (pair? t1)
        (if (pair? t2)
            (if (< level *fork-level*)
                (let ((right-branch
                        (future
                          (int-equal (cdr t1) (cdr t2) (+ level 1)))))
                  (and (int-equal (car t1) (car t2) (+ level 1))
                       right-branch))
                (if (and (tree-equal? (car t1) (car t2))
                         (tree-equal? (cdr t1) (cdr t2)))
                    '#t
                    (fail)))
            (fail))
        (if (equal? t1 t2)
            '#t
            (fail))))

  (define (tree-equal? t1 t2)
    (if failed-yet?
        nil
        (if (pair? t1)
            (if (pair? t2)
                (and (tree-equal? (car t1) (car t2))
                     (tree-equal? (cdr t1) (cdr t2))
                   nil)
             (if (equal? t1 t2)
                 '#t
                 nil))))

  (int-equal tree1 tree2 0)))
```

Figure 8.9: Checking version of tree equal

```
(define (stree-equal? tree1 tree2)
  (let* ((lock (cons '*undetermined* nil))
         (class (make-class *class-any*))
         (result (make-future class)))

    (define (int-equal t1 t2 level)
      (if (pair? t1)
          (if (pair? t2)
              (if (< level *fork-level*)
                  (let ((right-branch
                          (spec-future
                            (int-equal (cdr t1) (cdr t2) (+ level 1))
                            *fork-priority*)))
                    (and (int-equal (car t1) (car t2) (+ level 1))
                         right-branch))
                  (if (and (tree-equal? (car t1) (car t2))
                           (tree-equal? (cdr t1) (cdr t2)))
                      '#t
                      (fail)))
              (fail))
          (if (equal? t1 t2)
              '#t
              (fail))))

    (define (fail)
      (if (rplaca-eq-mand lock '*determined* '*undetermined*)
          (begin
            (determine-future result nil)
            (stay-group (group-id (my-group-obj))))))
      nil)

    (add-to-class
      (group-future
        (make-group
          (if (int-equal (car tree1) (car tree2) 0)
              (determine-future result '#t))
          *fork-priority*))
      class)

    result))
```

Figure 8.10: Speculative version of tree equal

Tree compared with tree0	Version		
	Eager	Checking	Speculative
tree0	1.78	1.96	1.86
tree1	0.67	0.72	0.72
tree2	1.39	1.54	1.49
tree3	1.31	1.50	1.44
tree4	0.36	0.40	0.37
tree5	0.98	1.07	1.02
tree6	0.53	0.59	0.54
tree7	1.04	1.10	1.10
tree8	0.71	0.83	0.74
tree9	0.90	1.02	0.93

All times are in seconds.

Table 8.4: Execution time of tree equal versions on various trees

Version	Total time
Eager	6.10
Checking	3.85
Speculative	3.67

All times are in seconds.

Table 8.5: Time to perform five consecutive comparisons of tree0 and tree1

the same ten comparisons as in Table 8.4. In every case, the speculative version is as fast or slightly faster than the checking version. Thus the speculative version is slightly more efficient. Both versions are slightly slower than the eager version: this is the overhead of termination detection.

Table 8.5 shows the total time for five consecutive invocations of each of tree equal comparing tree0 and tree1. Again, the speculative version is slightly faster than the checking version.

While explicit termination detection may be acceptable in this application, even though it required changes to called code, it does not generalize well. It suffers from the three problems described in Section 2.2.1: inserting the termination checking, termination detection of nested computation, and termination detection of shared computation.

The special language support we have added to Multilisp for aborting useless computation avoids the problems with explicit checking. This support does not burden the

programmer like explicit checking does and hence adds expressive power, making it easier to exploit the computation power of speculative computation.

8.3 Emycin

This application showcases pand used on a large scale, both as independent and nested invocations, and features the interaction of side-effects and speculative computation. The application is based on a kernel of Emycin written in Multilisp by Krall and McGehearty and described in [Krall].[8] Emycin is a complete rule-based expert system with a backward-chaining rule inference engine, a human interface, and a rules development system [Melle]. Krall and McGehearty's kernel includes only the rule inference engine.

Inferencing in Emycin begins with a list of hypotheses and works backwards using a given set of inference rules to determine the truth of each hypothesis. Each hypothesis is a premise, postulating the value of some parameter. The certainty of each such premise (and all unknowns in Emycin) is measured by a numerical value in the range [0,1000] where 200 is the threshold for true. The certainty of each premise is determined by *tracing* all applicable inference rules for the parameter in the premise. Each inference rule has one of the two following forms

$$P_1 \text{ or } P_2 \ldots \text{ or } P_n \rightarrow A_1, A_2, \ldots, A_m$$
$$P_1 \text{ and } P_2 \ldots \text{ and } P_n \rightarrow A_1, A_2, \ldots, A_m$$

where the P_i are premises and the A_j are actions to be performed if the rule is deemed true. These premises are either postulations about parameter values, like the premises in the hypotheses, or and/or trees of such postulations. The actions describe conclusions, where each such conclusion is the (conditional) certainty to conclude for a specific parameter value. A rule is applicable for a given parameter if any action A_j of that rule pertains to that parameter.

Tracing a rule means computing the *predicate certainty* of the rule and if this certainty exceeds 200, deeming the rule true and performing the rule's actions. Denoting the premise tree of the rule by P (so the rule is $P \rightarrow A_1, A_2, \ldots, A_m$), the predicate certainty of a rule is precisely the certainty of premise P which is computed recursively as follows:

$$\text{certainty}(P_i) = p_i$$
$$\text{certainty}(P_1 \text{ or } P_2 \text{ or } P_3 \ldots \text{ or } P_n) = max(p_1, p_2, \ldots, p_n), \text{ and}$$
$$\text{certainty}(P_1 \text{ and } P_2 \text{ and } P_3 \ldots \text{ and } P_n) = min(p_1, p_2, \ldots, p_n)$$

where p_i is the certainty of premise P_i. Performing the rule's actions involves updating the certainties of the parameter values specified in the rule's actions. The details of this updating are not relevant here: it suffices to say that the final certainty of a parameter value

[8] We thank the authors, Ed Krall and Pat McGehearty, and MCC for permission to use this kernel.

is a function of the predicate certainty and conditional certainty specified in the action of each applicable rule for that parameter.

Tracing is itself recursive. the certainty of a simple premise (which is not a combination of other premises) is determined by tracing all applicable rules. If there are no applicable rules for a simple premise, the certainty is determined by consulting a database and if that fails, querying the user. Thus rule tracing always terminates with the input premises.

Krall and McGehearty's kernel is straightforward. As mentioned above, it includes only the rule inferencing engine: it has no user interface. Thus all input premises and their certainties are contained in the database. All parameters and rules are maintained on property lists. There are three types of property lists (plists): parameter plists, rule plists, and an answer plist. Each parameter has a property list giving all the applicable rules for that parameter. Since the rules are fixed during inferencing, the rules are pre-searched to determine the applicable rules for each parameter. Each rule has a property list containing the premises and actions for that rule. (Recall that a premise is an and/or tree of postulations about parameter values.) The answer property list holds all parameter values and their certainties. This answer property list functions as the database. The answer database is initialized to the parameter values and certainties specified by the input premises. Thereafter, the answer property list is updated by side-effect: new parameters are added as inferencing proceeds and the certainties of existing ones are updated by rule actions.

Two forms of synchronization must be obeyed:

1. All tracing of the applicable rules for a parameter must complete before that parameter's certainty is used in other rules. This restriction is necessary to guarantee that the certainty is correct before other rules start using its certainty in their calculations.

2. Each rule must not be invoked more than once. This restriction is necessary to prevent erroneous certainties by performing a rule's actions multiple times.

This synchronization is trivial in Krall and McGehearty's sequential kernel.

In their paper, Krall and McGehearty describe several modifications to parallelize their kernel. Five modifications they describe are:

1. tracing all hypotheses in parallel (one task per hypothesis),

2. applying all the applicable rules for each parameter in parallel (one task per rule),

3. associating a flag which each rule so that no rule is invoked more than once,

4. associating a semaphore with each parameter to prevent the use of the parameter until all applicable rules for that parameter have been traced, and

5. tracing the disjunct premises comprising an or in parallel (one task per disjunct).

```
(define (alltrue premises)
   (if (null (cdr premises))
       (testpremise (car premises))
       (min (future (testpremise (car premises)))
            (alltrue (cdr premises)))))

(define (min a b)
   (if (< a b)
       a
       b))
```

Figure 8.11: Naive parallel and

We started with Krall and McGehearty's sequential kernel with these five modifications and a number of optimizations. The most significant of these optimizations was replacing the semaphores in the fourth modification with locks.[9] We describe these locks shortly. The rest of the optimizations were minor.[10] We call the resulting kernel the basic parallel kernel.

In the rest of this section we investigate parallelizing this basic kernel further using speculative computation. The most obvious remaining source of parallelism is parallel and, i.e. tracing the conjunct premises comprising an and in parallel. Krall and McGehearty investigated a naive parallel and: they simply parallelized the minimum operator. (Recall that the and of premises actually means the minimum of the premise certainties.) Figure 8.11 shows this naive parallel and. Each conjunct of the and (if indeed we can still call it that) is computed in parallel and all conjuncts are required to form the result, even if a false result is already guaranteed by a conjunct found with a certainty less than 200. Thus this version suffers from both useless computation and determinacy in returning the result. Nevertheless, Krall and McGehearty found that this parallel and significantly improved performance in their simulations [Krall] (for the gems data set described later). However, they noted this improvement was accompanied by a major increase in the inefficiency of resource use by useless computation, which could degrade overall performance if insufficient processors are available. They suggested using a speculative and but had no way to follow this up. The idea is to compute the analog of logical pand on the premise certainties: compute the premise certainties in parallel and if any have certainties less than the true threshold 200, immediately return 0, i.e. false, and abort any remaining computations. Otherwise, return the smallest certainty. The fact that we return the minimum in this later case, rather than true (i.e. 1000) is a slight departure from logical pand, but we will

[9] We could have used `delays` instead of locks (we will not go into the details here though). We decided to use locks to minimize changes to the code and to demonstrate a practical application of our methods presented in Chapter 7 for handling locks in speculative computation.

[10] We did not include any of the language optimizations that Krall and McGehearty discussed, namely, adding language primitives for parallel mapcar (`pmapcar`), for touching the elements of a list for synchronization (`touchlist`), and for property list operations (putprop and getprop).

```
;; Trace applicable rules for parameter parm
(define (traceparm parm)
  (let ((rules (get parm 'rules))
        (parm-lock (get parm 'lock))) ; lock cell initialized to (nil nil)
    (if (rplaca-eq
           parm-lock
           (delay (progn (traceparm2 rules) ; trace rules
                         '#t))              ; reset lock
           nil)
        (touch (car parm-lock)))
    ...))
```

Figure 8.12: "Delay-device" solution to locking problem in Emycin

continue to refer to it as an analog of pand, or just pand.

(Note that there is nothing to be gained by using an analogous por since the certainty of or'd premises is the maximum of the disjunct premise certainties. Hence all the certainties must be calculated.)

There is a problem with the naive use of pand in this application: a conjunct or one of its descendants may be in possession of one of the parameter locks when the conjunct is stayed. This could lead to the speculative deadlock discussed in Section 7.3.1 if another non-stayed task is waiting for the lock. We circumvented this problem using the "delay-device" trick presented in Figure 7.2 to solve the same problem with general spin-locks. However, the problem here is a little simpler than with general spin-locks since the critical region is only entered once. Thus our solution in Figure 8.12 is slightly different than that in Figure 7.2. The first task to grab the lock traces the rules and the remaining tasks block on the `delay` to ensure that rule tracing completes even if the parent task of the `delay` is stayed. The lock is reset to `'#t` here to avoid tracing the rules more than once.

We investigated five versions of the Emycin kernel. The five versions were:

1. Sequential and — (Base version)

   This version consisted of the basic parallel kernel described above, sequential and, and the locking described above. (The solution in Figure 8.12 turns out (in Multilisp) to be a very cost effective alternative to other locking methods, such as semaphores, so we used it even when speculative deadlock was not a concern.)

2. Naive parallel and

   Same as the base version, but with the naive parallel version of and in Figure 8.11 used by Krall and McGehearty.

3. Mandatory pand

Version	Inferencing time			
	Almandine		Peridot	
	Average	Std. Dev.	Average	Std. Dev.
Sequential and	2.98	0.05	3.84	0.06
Naive pand	2.31	0.07	2.63	0.06
Mandatory pand	2.39	0.07	2.64	0.07
pand version 2	2.28	0.13	2.53	0.09
pand version 3	2.66	0.34	2.93	0.17

All times in seconds. Statistics for 20 runs.

Table 3.6: Emycin inferencing times for almandine and peridot inputs

Same as the base version, but with the analog of mandatory por in Figure 8.1.

4. pand version 2

   Same as the base version, but with the analog of por version 2 in Figure 8.3.

5. pand version 3

   Same as the base version, but with the analog of por version 3 in Figure 8.4.

We used the same gems data set as Krall and McGehearty. This data set consists of 33 hypotheses about the type of gem, 29 gem parameters, such as color, hardness, and mineral species, and 99 rules. The object is to identify the type of gem from the input parameters. We used two sets of input parameters, one for which Emycin infers the gem almandine, and the other for which Emycin infers the gem peridot. Hereafter we call these two sets of input parameters "almandine inputs" and "peridot inputs" respectively. Both of these input sets were produced by by Krall and McGehearty. Their paper only reported results with the almandine inputs.

The gems data set is a good candidate for pand since the rules use and almost exclusively and many rules invoke other rules recursively. Thus Emycin with the gems data set is a good example of both pand used on a large scale and nested pand invocations.

Table 8.6 shows the results for the almandine and peridot inputs on the Concert Multiprocessor with 32 processors. On the basis of 20 runs for each set of inputs, the execution times of the naive and mandatory pand versions are statistically indistinguishable. This means either that all the conjuncts of the ands must be true, thus eliminating the mandatory version's advantage of nondeterminacy in returning the result, or that the extra overhead of the mandatory version must essentially counteract this advantage. The data in Table 8.7 indicates that many premises are false so the latter must be the reason. The execution time of pand version 2 is only slightly less than the execution times of the naive and mandatory versions. (This difference is barely significant since the means are only one standard deviation apart.) Thus there must either be little opportunity for speculative

Input	Number conjuncts false from ands with > 1 conjunct	Number ands false that have > 1 conjunct
Almandine	189	77
Peridot	180	76

The gems database has 81 ands with more than one conjunct. These 81 ands have a total of 257 conjuncts.

Table 8.7: Characteristics of Emycin with almandine and peridot inputs

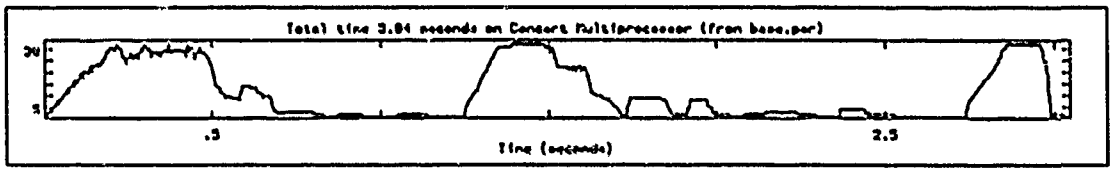


Figure 8.13: Parallelism profile with sequential and

computation with this data set, contrary to our expectations, or the overhead of the pand must be defeating its advantage. The execution time of pand version 3 is worse than the execution time of the naive and mandatory versions. There could be two reasons for this degradation with respect to version 2. First, termination detection in version 3 is centralized, requiring a termination detection task in addition to the conjunct tasks, whereas the termination detection in version 2 is distributed. Second, version 3 propagates demand for the pand result on to the conjunct tasks only when the termination detection task touches each of these tasks in turn. In version 2, demand for the pand result is propagated to all the conjunct tasks via the type `all` class of which all the conjunct tasks are members. To separate these effects, we tried a modification of pand version 3 in which all the conjunct tasks and the termination detection tasks were members of a type `all` class like in version 2. The results for this modified version 3 were essentially the same as for version 2. Thus the class sponsor policy is the reason for the difference between pand versions 2 and 3.

Figures 8.13 through 8.16 shows parallelism profiles (extracted from Parvis displays) for the sequential, naive pand, mandatory pand, and pand 2 versions respectively for the almandine inputs. The parallelism profiles for the peridot inputs are very similar to these.

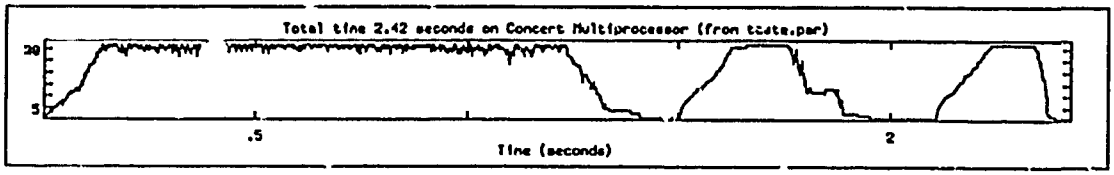


Figure 8.14: Parallelism profile with naive pand

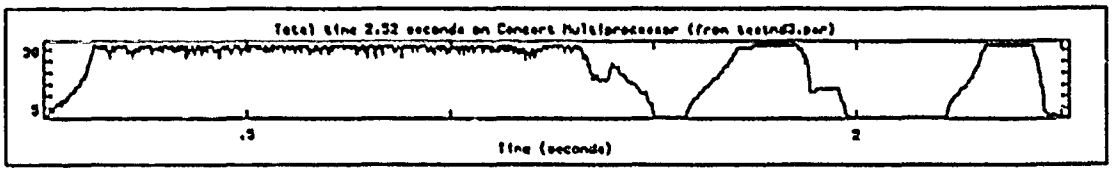


Figure 8.15: Parallelism profile with mandatory pand

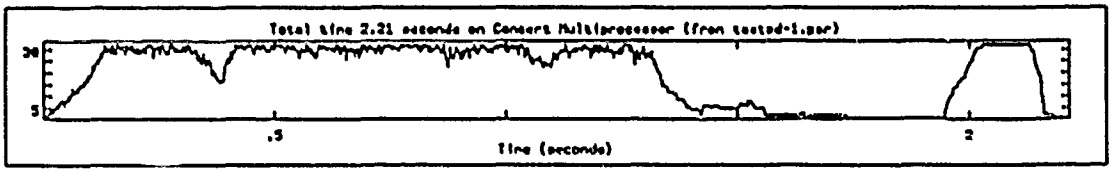


Figure 8.16: Parallelism profile with pand version 2

The parallelism profile for the sequential version in Figure 8.13 features three peaks of parallelism, each which almost saturates the machine, separated by deep valleys of very little parallelism. The first peak is due to the explosion of parallelism from tracing all 33 hypotheses in parallel. At about 0.5 seconds the parallelism falls off rapidly because almost all the rules are blocked waiting for the certainty of the key mineral species parameter. Only a few rules pertain to this parameter, so parallelism is poor until this parameter is fully traced at about 1.3 seconds. At this point, all the rules waiting for the mineral species parameter resume, resulting in the second peak of parallelism. After these rules evaluate their predicate certainties and take their actions, few applicable rules remain, accounting for the decrease in parallelism. At about 2.7 seconds the certainty of the gems parameter is finally computed. Each hypothesis task then evaluates the certainty of its hypothesis, resulting in the final peak of parallelism.

The other pand versions trace all the premises of each and rule in parallel. This means if one premise blocks on the evaluation of a parameter like mineral species, other premises may still continue. And, in fact, if one of these other premises evaluates to false, the rule may conclude without waiting for the blocked premise. This increased parallelism is evident in the remaining parallelism profiles.

The parallelism profiles for the naive and mandatory pand versions in Figures 8.14 and 8.15 are almost identical. Both feature three peaks of parallelism as in the sequential version. The first peak is much longer, though, due to all the rules that can fire in parallel. This additional parallelism mostly fills in the first valley. However, the increased computation (some of it unnecessary) also delays the second peak slightly. Some of this increased computation is merely computation that would have been performed anyway, but after the second peak. This shift in the timing of rule tracing accounts for the decrease in the width of the second valley Thus mandatory parallelism does what we would expect: it has filled in some of the valleys of the sequential version and decreased the total execution time.

The parallelism profile for the pand version 2 in Figure 8.16 is very interesting. The first valley is now completely filled in (though there is a slight dip where it used to be) and

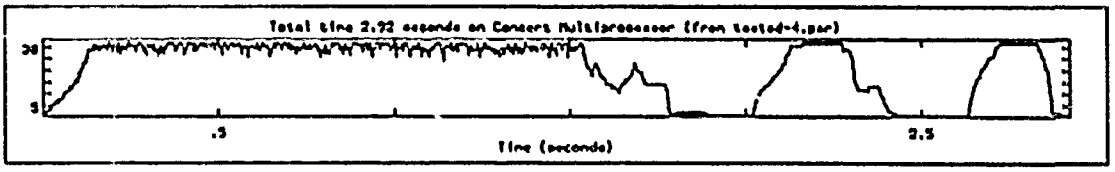


Figure 8.17: Parallelism profile for pand version 2 without staying

almost all the work is performed before 1.5 seconds. If it were not for the long portion after 1.5 seconds with only one or two rules active, this version with speculative parallelism would be significantly faster than the previous versions rather than just barely faster. Figure 8.17 shows the parallelism profile of pand version 2 with staying disabled. Note that the first valley is no longer completely filled in. Thus the complete fill-in of this first valley is due to aborting useless computation. The parallelism in the naive and mandatory versions only partly overlaps this valley because useless computation lengthens the critical path. However, the gain from aborting useless computation is still slight.

It is quite plausible that the relative overhead of speculative computation in pand is too large here since each conjunct task is quite short if the certainty of the associated premise is already known (because the premise is one of the inputs of some other task that has already traced all its applicable rules). To separate the intrinsic merit of speculative computation from the overhead artifacts, we artificially reduced the relative overhead of pand by adding delay loops to the four main procedures of the inferencing engine. Table 8.8 shows the results, again on the Concert Multiprocessor with 32 processors, for different loop counts, i.e. amount of artificial delay.

The results in Table 8.8 clearly show that the benefit of speculative computation increases as the relative overhead of the pand decreases. This confirms our earlier hypothesis about overhead defeating the advantage of speculative computation in this application (at least with the gems dataset). The benefit of speculative computation is still not great, as the figures in Table 8.9 indicate.

The speedup with speculative pand over mandatory pand seems asymptotic to about 15%, compared to speedups of 25% and 50% of mandatory pand over the sequential and. Thus, raw parallelism, not the ordering and aborting of speculative computation is the dominant performance factor here. After further investigation, we found that one reason for this result is that many premises in the gems dataset share the same parameter values. Because each parameter is only traced once (by the first task to get there), this means there is a great deal of sharing of computation between the conjunct tasks of different pands. In other words, there is a great deal of dependency between most pand invocations.

To investigate the influence of this factor, we created an artificial dataset without any sharing between the premises. This dataset consisted of three hypotheses, each with a uniform tree of anded premises — branch factor 3 — to a depth of 3 (for a total of 27 leaf parameters). Each non-leaf premise had exactly one applicable rule. We created the answer database randomly: we chose each non-leaf parameter from a distribution with probability

Loop Count	Version	Inferencing time			
		Almandine		Peridot	
		Average	Std. Dev.	Average	Std. Dev.
20	Sequential and	4.89	0.04	6.11	0.07
	Naive pand	3.86	0.12	4.02	0.13
	Mandatory pand	3.85	0.14	4.01	0.13
	pand version 2	3.56	0.15	3.70	0.17
	pand version 3	4.40	0.50	4.21	0.58
50	Sequential and	7.32	0.09	9.15	0.14
	Naive pand	5.90	0.20	6.04	0.16
	Mandatory pand	5.85	0.14	6.08	0.14
	pand version 2	5.13	0.20	5.43	0.23
	pand version 3	6.35	0.91	6.44	0.88
100	Sequential and	11.46	0.15	14.12	0.16
	Naive pand	9.24	0.29	9.41	0.25
	Mandatory pand	9.13	0.24	9.42	0.53
	pand version 2	7.97	0.33	8.28	0.41
	pand version 3	10.1	2.6	9.74	1.17

All times in seconds. Statistics for 20 runs.

Table 8.8: Emycin inferencing times with artificial delay

Inputs	Versions	Speedup with loop count			
		0	20	50	100
Almandine	Mandatory pand/Sequential and	1.25	1.27	1.25	1.26
	pand version 2/Mandatory pand	1.05	1.08	1.14	1.15
Peridot	Mandatory pand/Sequential and	1.45	1.52	1.50	1.50
	pand version 2/Mandatory pand	1.04	1.08	1.12	1.14

Table 8.9: Speedup of Emycin versions

Version	Avg. Inferencing time
Sequential and	3.3
Naive pand	2.7
Mandatory pand	2.2
pand version 2	1.5

All times in seconds. Statistics for 5 runs.

Table 8.10: Emycin inferencing times with artificial dataset

0.1 for a false-valued parameter. Table 8.10 shows the average execution time on the Encore Multimax with 8 processors. Speculative pand (version 2) leads to about a 50% speedup over mandatory pand here, without any artificial delays. This suggests that the sharing of computation is indeed a significant factor in the performance of Emycin with speculative computation.

Despite the poor absolute performance with speculative computation, Emycin demonstrates a number of important application features and a number of important issues. The important features are:

1. Nested pands

   Emycin is too complex to flatten the pands like we did with the pors in the tree-equal application. This forced nesting of pands is completely transparent with our support for speculative computation. There is no concern, as with explicit termination checking, about which parent pand a child checks for termination.

2. Side-effects

   Emycin is a real example of the interaction of side-effects and speculative computation. Our roll-forward approach to side-effects prevents speculative deadlock in a simple, straightforward extension of our touching model.

3. Importance of aborting useless computation

   The observed performance gains with speculative computation in Emycin, while small, are entirely due to aborting useless computation.

4. Sharing of computation

   With the gems dataset many conjunct tasks of different pands share the same computation. This feature really demonstrates the power of our touching model approach: we can stay all the descendant tasks of a pand with the assurance that any shared computation will not be irrevocably aborted. As long as the shared computation has a sponsor it will not be stayed, and if it is stayed, it can be restarted simply by touching it or giving it a sponsor. Outright aborting of all descendant tasks simply will not work here. Explicit termination checking is also too difficult because of all

the pands that may be sharing a computation: somehow the computation must know all the other computations which share it. Our support for speculative computation automatically manages these concerns.

The important issues raised by Emycin are:

1. Overhead

   The overhead introduced by the various management requirements of speculative computation can defeat its advantage. Thus it is important, obviously, to reduce the overhead. Less obviously, it is important to understand the limitations of speculative computation.

2. Inter-group touching

   The sharing of computation between conjunct tasks of different pands represents inter-group touching: a conjunct task in one pand group touches a similar task in another group.

8.4 Boyer Benchmark

This application has been popularized as a Lisp system benchmark (see [Gabr85]). Given an input expression, the Boyer Benchmark determines whether the expression is a tautology with respect to a database of rewrite rules. The Benchmark successively reduces the expression according to the rewrite rules to obtain an if-then-else tree. For example, the expression `(and (f x) (g x))` matches the rewrite rule

```
(and a b) → (if a (if b #t #f) #f)
```

and hence rewrites to `(if (f x) (if (g x) #t #f) #f)`. If rewriting does not produce an if-then-else tree, the input expression is not a tautology. The Benchmark then walks this if-then-else tree checking for consistency. For each if-then-else expression it checks if the predicate is known to be true or false. If so, it checks either the alternate or consequent respectively. Otherwise, it first assumes that the predicate is true and recursively checks that the consequent is true with respect to this assumption. Then it assumes that the predicate is false and recursively checks that the alternate is true with respect to this assumption.

There are three main opportunities for parallelism in the Boyer Benchmark:

1. applying the rewrite rules concurrently to separate subexpressions,

2. performing the tautology checking in parallel, concurrently checking the consequent and alternate of each `if` expression whose predicate value is unknown (and so on recursively), and

3. performing the rewrite and tautology checking stages in parallel, exploiting their producer-consumer relationship.

Exploiting these opportunities leads to two uses for speculative computation:

1. tautology failure

   If one tautology checker task discovers a contradiction then the input expression is false and all the remaining tautology checker and rewriter tasks may be aborted. Thus we are essentially talking about the pand of all the tautology checker tasks.

2. not all rewrites necessary

   Some rewrites may not be required to determine a tautology. For example, consider the rewrite rule for and given above. If a happens to be `#f`, there is no need to rewrite expression b.

   The possibility of unnecessary rewrites means that:

   (a) ordering is important

   Unnecessary rewrites can use resources that might otherwise be devoted to necessary rewrites and tautology checking and thus lengthen the execution time. We would like to order the allocation of resources to tasks to always favor necessary rewrites over unnecessary ones.

   (b) useless rewrites can continue once tautology checking completes

   We would like to abort all remaining tasks when tautology checking completes. This is really an extension of the tautology failure case to tautology completion.

We focus on the ordering issue in this section. Ordering (at least with respect to necessary versus unnecessary rewrites) is a key factor in the execution time of the Boyer Benchmark.[11] By contrast, aborting all remaining tasks on tautology completion only affects the computation (if any) in which the Benchmark is embedded. We have already seen (in the section on por for example) how we can abort the remaining tasks: enclose the Benchmark in a group and stay the group when tautology checking completes. Thus nothing new is added by considering the remaining task issue.

We consider three versions of the Boyer benchmark in [Gabr85] (all written in Multilisp). The first version employs only conventional, mandatory tasks introduced with three `future`s. One of these `future`s creates a task to perform the tautology checking of the consequent and alternate in parallel. The code containing this `future` is called recursively to initiate tautology checking on the entire if-then-else tree in parallel. The other two `future`s create tasks to perform the rewriting — i.e. generation of the if-then-else tree — in parallel with the tautology checking. The first of these two `future`s is called recursively to create tasks to rewrite each subexpression. These tasks match the subexpressions with the rewrite rules in the database. The second of these two `future`s is called recursively to perform

[11] If there are insufficient processors. This is usually a safe assumption for any non-trivial input expression.

the substitution indicated by the rewrite rule of any match. Thus both the producer (the rewriting) and the consumer (the tautology checking) are parallelized internally as well as with respect to each other.

The first version rewrites all expressions eagerly, ignoring the fact that some rewrites may be unnecessary. Thus we call it the eager version.

The second version is exactly the same as the eager version except the two `futures` in the rewriting code are replaced with `delays`. The eager version gambles that most rewrites are necessary and does them before the checker actually needs them. This version instead gambles that most rewrites are unnecessary and delays doing them at all until the checker actually needs them. We call this the lazy version.

The eager and lazy versions represent two extremes in eagerness and ordering. The eager version performs all rewrites eagerly in some arbitrary (implementation-dependent) order. The lazy version performs dynamic ordering: it allocates resources only to those rewrites which are demanded by the consumer, i.e. checker, and hence necessary. If the fraction of unnecessary rewrites is small or there are abundant resources free (to be wasted) relative to the number of rewrites, then the eager version is better. If on the other hand, the fraction of unnecessary rewrites is large or there are few resources available relative to the number of rewrites, then the lazy version is better.

The third version combines the best of the eager and lazy versions: eager evaluation and dynamic ordering. This third version is exactly the same as the other two except the `futures/delays` in the rewriting code are replaced with `spec-futures`. All these `spec-futures` have the same (arbitrary) priority of 100. It is important only that the priority be between 0 and *MAX* exclusive. Thus the speculative tasks created with these `spec-futures` have priority intermediate between the tasks created with `delay`, which have priority 0 (until touched), and the mandatory tasks created with `future`, which have priority equivalent to *MAX*. This means that rewriting proceeds eagerly unless there are no available resources, in which case the dynamic ordering kicks in and only the necessary rewrites are performed. Thus we expect better performance from this version than the previous two in intermediate operating regions. We call this the speculative version.

We ran the three versions with the three variations of *modus ponens* in Table 8.11 (which are all tautologies) as inputs.

The applicable rewrite rules for these inputs are:

1. `(and a b)` → `(if a (if b #t #f) #f)`
2. `(implies a b)` → `(if a (if b #t #f) #t)`, and
3. `(if (if a b c) d e)` → `(if a (if b d e) (if c d e))`

Table 8.12 shows the results for different number of processors on the Concert Multiprocessor. (The execution time is the time to determine if the input expression is a tautology, i.e. the time to return.) In every case, except with one processor, the speculative version

Name	Input Expression
tc2	(implies (and (implies a b) (implies b c)) (implies a c))
test-case	(implies (and (and (implies (f x) (g x)) (implies (g x) (h x))) (implies (h x) (i x))) (implies (f x) (i x)))
test-case2	(implies (and (and (and (implies (f x) (g x)) (implies (g x) (h x))) (implies (h x) (i x))) (implies (i x) (j x))) (implies (f x) (j x)))

Table 8.11: Boyer test cases

Input	Version	Number of Processors						
		1	2	4	8	16	24	32
tc2	eager	40.0	19.1	10.3	5.6	3.4	3.0	3.0
	lazy	19.9	10.9	8.8	8.0	7.9	8.0	8.2
	speculative	21.5	?	?	5.1	3.3	3.0	3.0
test-case	eager	334	158	80.0	39.8	22.1	16.3	15.1
	lazy	58.8	31.2	21.0	18.5	18.2	18.7	19.4
	speculative	63.8	?	?	14.8	10.6	10.3	10.3
test-case2	eager	2780	1380	674	335	185	170	211
	lazy	135	74.3	44.4	34.8	34.6	35.2	36.8
	speculative	147	?	?	30.2	28.1	27.8	30.6

All times in seconds. ? denotes times unavailable due to Concert failure.

Table 8.12: Execution time of Boyer Benchmark

is faster (and sometimes much faster) than the eager and lazy versions. The reason is that speculative version performs a large fraction of its unnecessary rewrites after it returns whereas the eager version performs a large fraction of its unnecessary rewrites before it returns.[12] This difference is reflected in the number of garbage collection flips each version performs before returning. With 32 processors the eager version had 0, 1, and 11 flips respectively with tc2, test-case, and test-case2 whereas the speculative version had 0, 1, and 2 flips, with the same respective inputs. Due to these flips, the execution time of the eager version for 32 processors is highly variable. The lazy version performs no unnecessary rewrites and had no flips with every combination of inputs and processors.

The execution time of the speculative version with test-case2 on 32 processors is noticeably anomalous with respect to the time with this input on fewer processors. The anomaly is real: we repeated runs several times and obtained the same difference in execution times. The difference is not due to garbage collection since for each combination of versions and inputs the number of garbage collection flips was the same with 24 and 32 processors. Thus the anomaly seems to be due to increased bus contention.

The one-processor case is interesting. With only one processor, the scheduling algorithm always continues the mandatory child task created by a future and queues the parent task (since the machine is always saturated — see Section 10.2). Consequently the eager version completely rewrites the input expression before performing any tautology checking and thus is "maximally" eager with one processor. In contrast, the rewriting in both the lazy and speculative versions is "maximally" lazy with one processor: no child task is pursued until demanded by the consumer. This accounts for the difference between the execution time of the eager and lazy/speculative versions. The difference in execution time of the lazy and speculative versions with one processor is due to speculative task overhead.

With more than one processor, the eager version has an additional tendency to depth-first rewriting like with one processor. In this case, the reason has to do with the scheduling concerned with task touching rather than the scheduling concerned with task creation. When a consumer task touches and blocks on a producer task, the scheduler continues the next arbitrary task on task queue, not necessarily the producer task touched, if it was queued. Consequently, consumer tasks can be blocked on producer tasks while producer tasks continue to proliferate. Mohr reported that this phenomenon is strong enough to ensure mostly depth-first rewriting even when task creation scheduling favors the parent task rather than the child task [Mohr]. Mohr changed the scheduling concerned with task touching, so the scheduler continues the touchee task, if it was queued. With this new blocking algorithm (which is undesirable in general), he found the execution time of the eager version was virtually the same as that of the lazy version [Mohr]. Our speculative version approximates this new blocking algorithm. when a consumer task touches a producer task, the producer becomes a mandatory task and continues, even if it was queued before.

[12] In fact, the total computation time is about the same in both cases. However, we actually stayed these remaining tasks in the speculative version to reduce the measurement cycle time. This staying took little additional time.

Input	Version	Number of Processors				
		1	8	16	24	32
tc2	eager	1.9	1.1	1.0	1.0	1.0
	lazy	0.9	1.6	2.4	2.7	2.7
test-case	eager	5.2	2.7	2.1	1.6	1.5
	lazy	0.9	1.3	1.7	1.8	1.9
test-case2	eager	19	11	6.6	6.1	6.9
	lazy	0.9	1.2	1.2	1.2	1.2

Table 8.13: Ratio of eager and lazy execution times to speculative version execution time

In our own experiment, we found that the queue discipline matters little in the eager version. We changed the queue ordering for mandatory tasks from LIFO to FIFO and noticed no significant difference in execution times. Thus it is ability to dynamically reorder tasks — queued or not — that is important to minimal execution times with the Boyer Benchmark.

Table 8.13 makes the relative performance of the three versions clear. The speedup of the speculative version with respect to the eager and lazy versions displays two trends (though in opposite directions for each version). For the eager version, the speedup decreases as the number of processors increases and increases as the input size increases. For the lazy version, the speedup increases as the number of processors increases and decreases as the input size increases. These trends confirm our expectations. For the eager version, as the number of processors increases there are more "surplus" processors at every point in time to perform unnecessary rewrites, thus reducing the relative effect of unnecessary rewrites. As the number of processors increases without bound, the execution time for the eager version should approach that for the speculative version. As the input size increases the number of unnecessary rewrites increases, thus increasing the relative execution time. For the lazy version, as the number of processors increases there is insufficient parallelism — due to the lazy evaluation of all rewrites — to utilize all processors, thus increasing the relative execution time. As the input size increases, there is more necessary work available to utilize the processors. For large inputs, the execution time for the lazy version should approach that for the speculative version.

Table 8.14 shows exactly how the number of rewrites varies with the inputs. The total number of rewrites in this table is the total number of rewrites performed by the eager version on a single processor. Thus it is the maximum number of rewrites. The number of necessary rewrites in this table is the total number of rewrites performed by the lazy version (on any number of processors). Thus it is the minimum number of rewrites. The ratio of the total number of rewrites to the number of necessary rewrites thus yields an upper bound on the speedup of the lazy version over the eager version. The actual speedup with one processor is about 0.7 of that predicted by this ratio.

Data Set	Total Rewrites	Number Necessary	Percent Necessary
tc2	1213	397	32.7
test-case	11280	1405	12.5
test-case2	91868	3371	3.7

Table 8.14: Rewrite statistics for the three input test cases

Another way of looking at the results in Table 8.12 is as follows:

- The eager version is better than the lazy version if the processors are not saturated with mandatory computation, as with many processors or a small input. Then the eager version exploits the extra processors to perform the rewrites speculatively and reduce the execution time.
- The lazy version is better than the eager version if the processors are saturated with mandatory computation, as with few processors or a large input. Then the lazy version prevents mandatory computation from being crowded out by speculative computation.
- The speculative version is the same or better than the eager and lazy versions for all conditions (except on one processor). It prevents mandatory computation from being crowded out by speculative computation while still utilizing the extra concurrency of the speculative computation. That is, it dynamically adjusts the tradeoff between concurrency and crowd-out of mandatory computation.

These results indicate the importance of ordering speculative computation. (To reiterate, this application has only 2 spec-futures, and performs no aborting.) The key feature of spec-future here is that it creates a second-class type of task that only runs if there are free processors. That is, spec-future is neither fully lazy nor fully eager; it adapts dynamically to the load. This frees the user from scheduling concerns as the machine and input size vary.

Ideally, we would like to use the relative promise of rewrites better in this application. The above three versions assume that all rewrites have equal promise: the eager version assumes all are required, the lazy version assumes none are required, and the speculative version assumes all are required with the same likelihood. We would like to identify necessary rewrites *a priori* so we can avoid doing any unnecessary rewrites. Unfortunately, this is hard to do. Instead, we experimented with imposing an order on all rewrites. We chose an ordering that gave rewrites a priority proportional to their distance from the left-most spine of the if-then-else tree (measured by the depth of recursion). This seemed a reasonable *a priori* ordering. However, the results were no different than with our original speculative version. The impact of this ordering was totally overwhelmed by the impact of the mandatory versus speculative classification. This result underscores that the demand-

driven, dynamic ordering by the consumer, i.e. tautology checker, is the important feature in this application.

The speculative version that we presented is not fully general: it may not be embedded within a large computation without risking two problems. The first problem is possible priority inversion. The most important aspect of the speculative version is that consumers, i.e. tautology checkers, run at a higher priority than producers, i.e. rewriters. However, because of our choice of an absolute priority p (which happens to be 100 here) for the producers, this desired priority relationship will be inverted if the speculative version is invoked by a speculative task with priority less than p. The consumer tasks will all be speculative tasks with priority less than p inherited from their parent, rather than greater than p as we desire. We could choose a different value of p, but what value? The invoker could have any priority, which we should not have to know for abstraction reasons (or be able to know). We could choose $p = 1$ (so almost all invokers have priority $> p$) but this is not a general solution. we could have some local ordering on producers that we want to maintain. To solve this problem we need a local ordering space for the Benchmark, i.e. we need modularity.

The second problem is that eager tasks — tasks not yet demanded — are not restarted if the speculative version is stayed and then restarted by touching the root consumer task. This problem is not fatal, as such tasks will be re-started when they are demanded, but it could impact performance since the speculative version essentially changes into the lazy version. To mitigate this problem, we need group coherency. Ideally, we want to associate a controller sponsor with the Benchmark so we could sponsor all the Benchmark tasks with the demand for the root consumer task.

Comment

The Boyer Benchmark in [Gabr85] is a stupid program (like most benchmark codes). Its performance can be improved dramatically (from a factor of 2 for tc2 to a factor of 50 for test-case2) by making the following two optimizations:

1. Add two rewrite rules to the database:

   `(if #t a b)` → `a` and `(if #f a b)` → `b`

   These two rules primarily reduce the size of the if-then-else tree, speeding both rewriting (since it does not have to construct such a large tree) and tautology checking (since it has a smaller tree to check). These rules also eliminate useless rewriting of b and a respectively. (A further improvement, which we did not try, is to change the rewrite rule for and to `(and a b)` → `(if a b #f)`.)

2. Eliminate the multiple rewriting of common subexpressions.

   In the case of the rewrite rule `(if (if a b c) d e)` → `(if a (if b d e) (if c d e))` the benchmark brainlessly rewrites subexpressions d and e twice: once for each occurrence in the rewritten expression. Eliminating multiple rewriting reduces the number

of rewrites in such cases by a factor of 2. It also reduces the number of unnecessary rewrites.

Together, these optimizations reduce the total number of rewrites for te ..case2 from 91868 to 622 and the number of necessary rewrites from 3371 to 554. .1 percent of the rewrites are unnecessary so there is still potential for the ordering introduced with spec-futures to reduce the execution time. However, since the execution time is so small the input size must be increased to notice a significant impact.

8.5 The Traveling Salesman Problem

This application is the solution of the traveling salesman problem by straightforward branch and bound algorithm. It is a clear-cut example of ordering-based speculative computation.

We describe two different versions: qtrav and strav. Both versions are substantially the same in concept. However, qtrav uses only the conventional (i.e. mandatory) constructs of Multilisp. We describe qtrav here and describe later how strav differs.

The input is a list of cities described by two-dimensional coordinate pairs. The result is the cost and itinerary of the best (shortest) tour of all cities, where a tour is a directed cycle of cities containing no city more than once. qtrav finds the best solution by successively expanding nodes in a branch-and-bound manner. Each node represents a partial solution consisting of a tour, the tour cost, and a list of cities excluded from the tour. The initial node represents an initial tour of three cities. qtrav expands each eligible node into new (child) nodes by choosing the next city from the node's excluded list and inserting it in all the possible places in the tour. Only nodes with a tour cost less than the cost of the best complete tour so far are eligible. Ineligible nodes are pruned, i.e. discarded.

Figure 8.18 shows a program fragment from qtrav for expanding a candidate node.

The variable `next-city` (line 4), which has already been set by the original caller of `expand-candidate`, gives the current city to try from the excluded list. The argument `before` is a list of the cities in the new tour before `next-city` and the argument `after` is a list of cities in the new tour after `next-city`. Line 1 calculates the cost of the new tour and line 2 checks that the new tour is eligible. If so, line 3 spawns a task to try other permutations of cities before and after `next-city` and the present task continues expanding the new tour into a node. Line 4 forms a list representing the new tour. If no excluded cities remain (line 5), the new tour is a complete tour and line 6 atomically updates the cost of the best solution if the solution is cheaper. Finally, line 7 creates the new node. If the new tour was ineligible in line 2, the present task goes on to try other permutations in line 8.

qtrav chooses nodes to expand in minimum heuristic cost order. This heuristic cost is the average cost per city, i.e. the tour cost divided by the number of cities in the tour. The intention is to bias node expansion towards the most promising tours (as defined by this

```
(define (expand-candidate before after)
  (if (null after)
      nil
      (let ((new-cost <calculate cost of new tour>))                    ; 1
        (if (< new-cost (caar best-soln))                               ; 2
            (let ((rest-candidates
                   (future (expand-candidate                            ; 3
                             (cons (car after) before)
                             (cdr after))))
                  (new-tour
                   (future (append (reverse before)                     ; 4
                                   (cons next-city after)))))
              (if (null (cdr excluded-cities))                          ; 5
                  (begin
                    (update-best-soln (cons new-cost new-tour))         ; 6
                    rest-candidates)
                  (cons (list (/ new-cost num-cities-used)              ; 7
                              num-cities-used
                              new-cost
                              new-tour
                              (cdr excluded-cities))
                        rest-candidates)))
            (expand-candidate (cons (car after) before)                 ; 8
                              (cdr after))))))
```

Figure 8.18: Expanding a candidate node in qtrav

heuristic cost) in an attempt to get good solutions rapidly. The faster we can find a tight bound on the best solution, the more we can prune the search space and thus the faster we can complete the search.

To enforce the heuristic cost order, qtrav maintains the nodes in a central priority queue according to their average cost per city. This priority queue is implemented by Multilisp code. (There is no support for priority queues built into conventional Multilisp.)

qtrav employs a number of worker tasks to perform this overall strategy of greedy, minimum (heuristic) cost expansion. Each worker performs a cycle of the following two steps while nodes are available:

1. Pull the minimum (heuristic) cost node off the priority queue
2. Expand the node, pruning out inferior child nodes, and enqueue the remaining child nodes on the priority queue.

To avoid creating an excessive number of nodes, qtrav uses only one worker until it finds the first complete tour. Then, once it has a bound to prune inferior nodes, it starts the other workers. Each worker is itself a parallel task — the children of a node can be expanded in parallel as in Figure 8.18 — and thus qtrav exploits parallelism even during this first stage with a single worker. (The priority queue insertion and deletion routines also exploit parallelism.) However, this parallelism within a worker task presents a problem: how do we determine the number of worker tasks for the optimum tradeoff between inter-worker parallelism and intra-worker parallelism?

strav performs the same branch-and-bound algorithm with the same greedy, minimum (heuristic) cost node expansion strategy as qtrav. However, strav does not use an explicit priority queue to order node expansion. Instead, strav relies on the ordering mechanism for speculative tasks. strav creates a speculative task for each node expansion, with the priority negatively proportional to the heuristic cost. The underlying support for ordering speculative tasks (i.e. the distributed priority task queues described in Chapter 10) thus ensures that nodes representing good tours are favored for expansion over nodes representing bad tours. The actual expansion of a node occurs in the same way as for qtrav. Figure 8.19 shows the version of expand-candidates for strav. This code is executed by a speculative task. Before getting to this code, the speculative task checks if the parent node should be pruned. If so, the task simply terminates. Once in this code, the speculative task checks each candidate child for pruning (in line 1) before representing the child by a node (line 4) and spawning a speculative task (in line 5) to expand the node. The `cost->priority` function in line 6 produces a priority for this speculative task that is negatively proportional (i.e. linearly decreasing) to the child node's heuristic cost. Note that strav uses parallelism in node expansion (lines 2 and 3) just like qtrav.

To get a good first solution as rapidly as possible, strav expands the minimum cost child node as a mandatory task until it obtains a complete tour. To experiment with the expansion of other nodes (as speculative tasks) during this period, we added a parameter `first` to strav. With `first` set to non-nil, the expansion of all nodes other than the

```
(define (expand-candidate before after)
  (if (null after)
      nil
      (let ((new-cost <calculate cost of new tour>))
         (if (< new-cost (caar best-soln))                          ; 1
             (let* ((rest-candidates
                     (future (expand-candidate                      ; 2
                               (cons (car after) before)
                               (cdr after))))
                    (new-tour
                      (future (append (reverse before)              ; 3
                                      (cons next-city after))))

                    (new-cand (list (/ new-cost num-cities-used)    ; 4
                                    num-cities-used
                                    new-cost
                                    new-tour
                                    (cdr excluded-cities))))

               (cons (spec-future (search new-cand)                 ; 5
                                  (cost->priority (car new-cand))) ; 6
                     rest-candidates))
             (expand-candidate (cons (car after) before)
                               (cdr after))))))
```

Figure 8.19: Expanding a candidate node in strav

There are a number of optimizations we could make to this code, such as replacing cons in line 5 by and (we just care about termination). We chose the present version since it clearly illustrates the essential differences between qtrav and strav.

minimum cost node is inhibited until the first complete tour is found. With `first` set to nil, the expansion of these nodes proceeds in parallel with the minimum cost expansion, according to the heuristic cost ordering.

An interesting problem with strav is termination: how do we know we have expanded all eligible nodes and thus can return? With qtrav, termination is trivial: return when the global priority queue of eligible nodes empties. We cannot do the same with strav since the priority task queues may contain speculative tasks from activities in addition to those from strav. One solution is to maintain a count of the number of eligible nodes remaining unexpanded and terminate when this count becomes zero. Another solution is to touch each node expansion task generated and terminate when all these tasks have returned a value. We chose this second solution since it fits the Multilisp paradigm better. With tree structured task creation, like in strav, termination detection by touching is natural and simple: each parent task touches all its child tasks. However, this solution means that the desired node expansion ordering, which is expressed by the source priority of the tasks, may be distorted: the touchee's priority is the maximum of the toucher's priority and the source priority. We want to prevent this distortion. We can do so, while still retaining termination detection by touching, by either disabling the max combining-rule or disabling touch propagation. We have no direct way to disable the max combining-rule, though we could effectively disable it by ensuring that the toucher priority is always less than the touchee priority. We save this alternative for the Section 8.6. Instead we disable touch propagation by breaking the touch propagation path with placeholders. We replaced the `spec-future` in line 5 of Figure 8.19 by the syntactic form `const-spec-future`. `(const-spec-future exp priority)` expands to

```
(let ((α (make-future)))
   (spec-future (determine-future α exp) priority)
   α)
```

where α is an unique identifier. Now instead of touching the `spec-future`, we touch the placeholder which does not propagate the touch on to the `spec-future`. (However, this solution leads to another problem which we discuss later.)

Table 8.15 shows the execution time of qtrav and strav on the Concert Multiprocessor for three city lists, a16, a18, and a20, of 16, 18, and 20 cities respectively. The number of workers in the second column pertains only to qtrav. As indicated at the foot of this table, the letter "f" in the columns for strav denotes the parameter `first`. The graph in Figure 8.20 captures the general trends in these results. For each combination of city list and number of processors in this graph, the qtrav results shown are for the optimum number of workers for that combination of parameters.

From Figure 8.20 we observe that

1. strav is always faster than qtrav (except for a20 with 8 processors: qtrav with 8 workers is slightly faster than strav with `first`= nil). The difference in execution times increases as the number of cities and number of processes increases. In fact,

Number of processors	Number of workers (for qtrav)	City Lists								
		a16			a18			a20		
		qtrav	strav		qtrav	strav		qtrav	strav	
			f=nil	f=#t		f=nil	f=#t		f=nil	f=#t
8	2	30.1	13.0	10.6	145.8	52.0	47.5	244.8	105.9	93.4
	4	17.8			81.0			135.4		
	6	15.2			66.0			108.3		
	8	14.6			63.8			104.4		
16	4	29.9	8.2	6.5	78.0	28.8	26.5	130.9	55.3	49.2
	8	12.1			47.5			77.8		
	12	11.3			42.8			67.9		
	16	11.5			42.9			68.3		
24	8	12.3	7.3	5.6	47.9	22.4	21.0	78.1	40.9	36.3
	12	11.3			41.4			66.0		
	16	11.6			40.5			63.8		
	20	11.7			41.2			64.6		
	24	11.8			41.8			66.0		
32	8	12.8	7.0	5.4	50.0	19.8	19.0	81.5	36.8	33.3
	12	11.8			43.3			68.4		
	16	12.0			42.3			66.4		
	20	12.1			43.0			66.7		
	24	12.2			43.9			68.7		
	32	12.6			44.6			71.1		

Table 8.15: Execution time of Traveling Salesman Problem on the Concert Multiprocessor

All times are averages in seconds. f denotes the parameter `first` in strav.

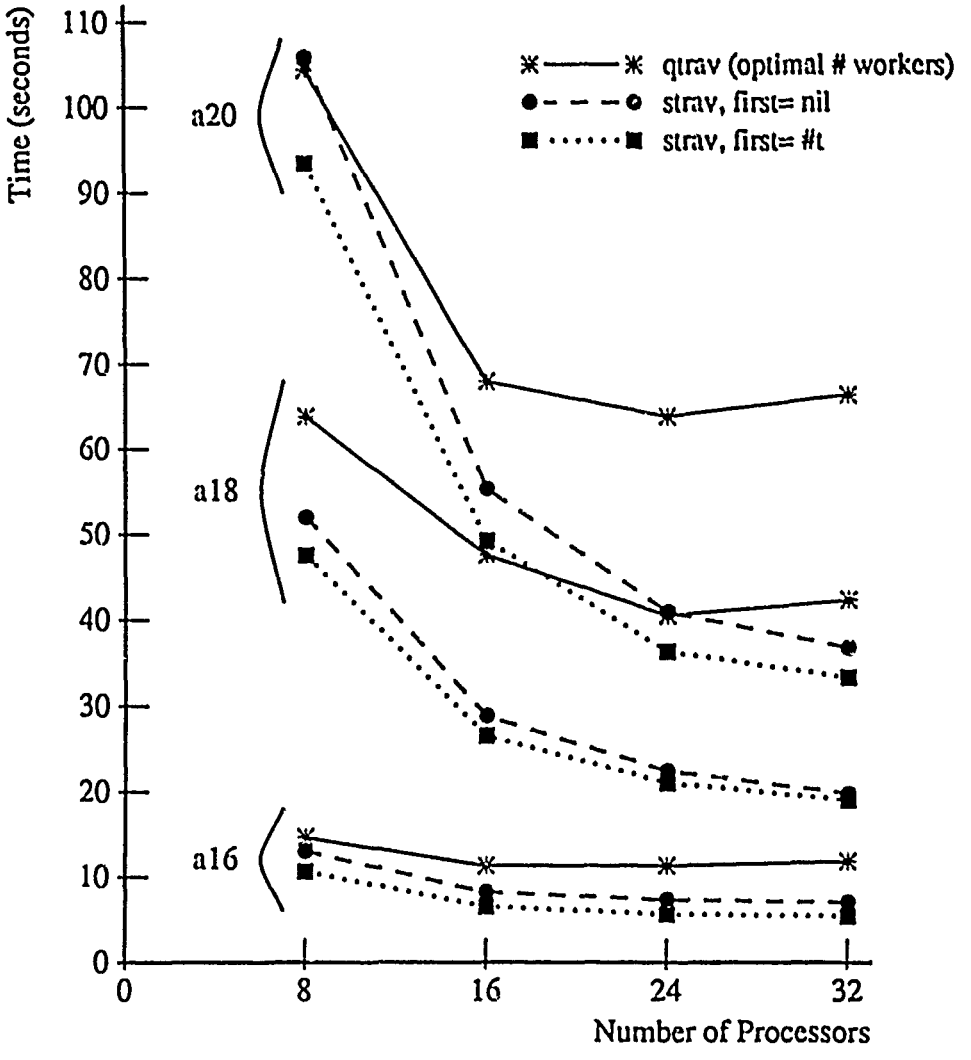


Figure 8.20: Execution time of Traveling Salesman Problem on the Concert Multiprocessor

strav continues to speed up as the number of processors increases to 32, whereas qtrav encounters a distinct knee around 24 processors and actually slows down (for a18 and a20) as the number of processors increases from 24 to 32. This behavior of qtrav is undoubtedly due to contention accessing the global priority queue.

2. strav is always faster with `first= #t` than with `first= nil`. We expected the opposite result. With `first= nil` we reasoned that strav would begin exploring some possibly useful routes with the free processors available until the first complete tour was found. With perfect preemption (zero time to notice a preemption condition and perform the preemption), this exploration of other routes should not interfere with the search for the first complete tour, which executes as a mandatory task. However, the benefit of precomputing these routes must be balanced with the cost imposed by generating superfluous nodes — nodes that would not have been generated if a pruning solution (from a complete tour) existed. Each superfluous node must be investigated later, compounding the cost. Obviously, this effect and non-perfect preemption must overwhelm the expected advantage of `first= nil`.

The fact that strav is always faster than qtrav is not surprising given that strav essentially pushes the priority queue overhead down one layer of interpretation, from the language level to the implementation. We gain more than speed by this in strav. What we have really done with strav is move a good deal of resource management (the management of task ordering in this case) from the user level to the implementation. Automating this resource management makes it easier to exploit resources in three ways:

1. We do not have to write (or even think about) the priority queue code. Instead, we can use the more efficient implementation code to perform task ordering.

2. We are insulated from concerns over the machine size. The implementation automatically distributes the priority queues to avoid excessive contention with a large number of processors.

3. We do not have to worry about appropriate parameter values, such as the number of workers. The underlying scheduling mechanism automatically determines the optimum tradeoff between inter-node and intra-node (expansion) parallelism. Table 8.15 would seem to indicate that this tradeoff is relatively unimportant for qtrav since the execution time is fairly flat around the optimum number of workers. However, we did not know this in advance and we certainly did not know the optimum number of workers in advance, so we still had to fiddle with various values.

Consequently, strav is easier to program and understand than qtrav.

Both versions of the branch-and-bound solution presented here for the traveling salesman problem demonstrate order-based speculative computation, in which the only speculation is in the ordering of computation and there is no explicit aborting of computation, so this application once again illustrates the importance of ordering. The only issue here is how to achieve the desired ordering. qtrav achieves this ordering by direct intervention by the

2	8	3
1	6	4
7		5

Table 8.16: The Eight-puzzle: a starting position

user. strav achieves this ordering by exploiting the priority scheduling mechanism of our support for speculative computation. This support provides two benefits for strav:

1. more efficient priority scheduling, and
2. easier exploitation of system resources.

As described above, these benefits lead to greater ease of programming and understanding.

There are two problems with strav, however. First, if it is stayed, as part of a larger computation, it cannot be restarted when touched since we broke the touch propagation paths. Touching a stayed strav will result in deadlock. We could solve this problem by making all the tasks members of some class sponsored by the tasks demanding the strav result. We omitted this fix for simplicity. In general, we would like a controller sponsor to solve this problem so that the original priority ordering can be preserved. Second, strav uses absolute priorities so the priorities cannot be rescaled if strav is part of a larger computation whose priority changes (such as if stayed), i.e. strav lacks modularity. We need to add modularity to our touching model to solve this problem.

8.6 Eight-puzzle Game

This application is basically a tree-structured por, like tree-equal, but with ordering. Consequently, the characteristics of the application are a cross between the characteristics of tree-equal and travsales.

The Eight-puzzle game has eight square tiles numbered 1 to 8 respectively arranged in some fashion on a three-by-three square game board, as in Figure 8.16. The tiles slide in the horizontal and vertical grids established by the board and cannot be removed from the board. Any tile adjacent to the one empty square can slide into that square. This constitutes a move. The object of the game is to arrange the numbered tiles in clockwise order around the perimeter of the board with the empty square in the center, as in Figure 8.17.

The application in this case is to solve the Eight-puzzle game: given a starting board, search the move tree to the given depth for the first path to the solution. (We are interested

1	2	3
8		4
7	6	5

Table 8.17: Eight-puzzle solution

```
(define (seq-solve starting-board search-depth)
  (solve-puzzle nil
                search-depth
                (cons starting-board (find-blank-tile starting-board))))

(define (solve-puzzle boards-seen depth board+blank)
  (let ((board (car board+blank))
        (blank (cdr board+blank)))
    (if (compare-boards board solution)                              ;1
        boards-seen
        (if (or (<= depth 0) (seen-board-before? board boards-seen))  ;2
            nil
            (mapcar (lambda (bb)                                     ;3
                      (solve-puzzle (cons board boards-seen)
                                    (- depth 1)
                                    bb))
                    (successors board blank))))))                    ;4
```

Figure 8.21: Sequential version of Eight-puzzle

in only the first path to the solution, not all paths and not the shortest path.) The code for this application is a Multilisp translation of the Id code in [Soley]. We consider four versions of this code — one sequential version and three parallel versions. Figure 8.21 shows the key part of the sequential version.

`seq-solve` takes a starting board (represented as an array) and a search depth, and calls `solve-puzzle` to start the game tree search. `solve-puzzle` takes three arguments: a list of the candidate boards examined so far, the current depth remaining to search in the tree, and a pair containing the current board and the position of the blank tile, i.e. empty square, in this current board. (All squares are numbered.) Line 1 checks if the current board is the solution board denoted by the variable `solution`. If not, line 2 checks both that the search depth is not exceeded and that the current board has not already been examined (this prevents cycles). Finally, line 3 calls `solve-puzzle` recursively on all the

successors of the current board. Line 4 generates these successors by trying all ways to swap the blank tile with a neighboring tile.

The first parallel version uses mandatory tasks. This mandatory version, shown in Figure 8.22, is basically the parallel or of the search tree branches. This code should be familiar from the mandatory version of por in Figure 8.1 (and the eager version of tree-equal in Figure 8.6). Line 1 initializes a lock cell for race officiating synchronization and line 2 initializes the result placeholder. As in Figure 8.1, line 6 spawns the tree search with a dfuture to remove task creation from the critical path of returning a result. solve-puzzle is the same as in the sequential version except for three changes — race officiating in line 3, task creation to examine the successor boards in line 4, and termination detection in line 5. touchlist performs termination detection by touching each child task. Note that child tasks are added to the "por" dynamically in line 4, so this is an another example of por with a dynamic number of disjuncts, the second generalization of por mentioned in Section 8.1.8.

The second parallel version uses speculative tasks. This version, which we call the spec version, differs from the mandatory version in only three lines. Figure 8.23 shows the code for the spec version with these three lines numbered. This spec version is much like version 1 of speculative por in Figure 8.2 except for centralized termination detection with touchlist. All the speculative tasks in the spec version have the same priority (*max-pri* in this case). This version cannot be restarted if stayed. We discuss this problem later.

The final parallel version is a refinement of the spec version with ordering, i.e. different task priorities. The task priorities in this version, which we call the spec2 version, are set to implement the the heuristic ordering of board examination suggested in [Nilsson]. Each board b has a heuristic cost $C(b)$ given by $D(b) + W(b)$ where $D(b)$ is the depth of board b in the search tree and $W(b)$ is the number of (numbered) tiles misplaced with respect to the solution board. For example, $W(b) = 4$ for the board in Figure 8.16. Boards should be examined in order of increasing heuristic cost.

To approximate this heuristic ordering, the spec2 version spawns a speculative task to examine each board b, as in the spec version, with priority *max-pri*$-C(b)$. Thus the spec2 version uses the built-in (distributed) priority queues to achieve a desired ordering, like the strav version in Section 8.5. The spec2 version aborts all remaining computation as soon as it finds a solution. Thus the spec2 version is also like the por and tree-equal applications. spec2 is an example of simultaneous multiple-approach and ordering-based speculative computation.

Like por, the spec2 version must perform termination detection since it may not find any solution within the given search depth. The most natural way to do this in Multilisp is by touching the descendant tasks. However, this gives rise to the same problem with touching termination detection as we discussed with strav in Section 8.5: how can we touch a task for termination detection without changing the touchee's priority (by the max combining-rule), hence distorting the desired ordering? We described one way in Section 8.5 to solve this problem by disabling touch propagation. Here, we describe another way to solve this problem by disabling the max combining-rule. The idea is simple: we simply force the

```
(define (mand-solve starting-board search-depth)
  (let ((lock (cons '*undetermined* nil))                              ;1
        (result (make-future)))                                        ;2

    (define (solve-puzzle boards-seen depth board+blank)
      (let ((board (car board+blank))
            (blank (cdr board+blank)))
        (if (compare-boards board solution)
            (if (rplaca-eq lock '*determined* '*undetermined*)         ;3
                (determine-future result boards-seen))
            (if (or (<= depth 0) (seen-board-before? board boards-seen))
                nil
                (let ((children
                       (mapcar (lambda (bb)
                                 (future                               ;4
                                   (solve-puzzle (cons board boards-seen)
                                                 (- depth 1)
                                                 bb)))
                               (successors board blank))))
                  (touchlist children)                                 ;5
                  nil)))))

    (dfuture (begin                                                    ;6
               (solve-puzzle
                nil
                search-depth
                (cons starting-board (find-blank-tile starting-board)))
               (if (eq (car lock) '*undetermined*)                     ;7
                   (determine-future result nil))))
    result))
```

Figure 8.22: Mandatory version of Eight-puzzle

```
(define (spec-solve starting-board search-depth)
  (let ((lock (cons '*undetermined* nil))
        (result (make-future)))

    (define (solve-puzzle boards-seen depth board+blank)
      (let ((board (car board+blank))
            (blank (cdr board+blank)))
        (if (compare-boards board solution)
            (if (rplaca-eq-mand lock '*determined* '*undetermined*)
                (begin
                  (determine-future result boards-seen)
                  (stay-group (group-id (my-group-obj)))))              ;1

            (if (or (<= depth 0) (seen-board-before? board boards-seen))
                nil
                (let ((children
                        (mapcar (lambda (bb)
                                  (spec-future                          ;2
                                    (solve-puzzle
                                      (cons board boards-seen)
                                      (- depth 1)
                                      bb)
                                    *max-pri*))
                                (successors board blank))))
                  (touchlist children)
                  nil)))))

    (make-group (begin                                                  ;3
                  (solve-puzzle
                    nil
                    search-depth
                    (cons starting-board (find-blank-tile starting-board)))
                  (if (eq (car lock) '*undetermined*)
                      (determine-future result nil)))
                *max-pri*)

    result))
```

Figure 8.23: Speculative (spec) version of Eight-puzzle

2	1	6
4		8
7	5	3

Table 8.18: board18

toucher task to have the minimum running priority, *min-pri*, before it touches tasks to check for termination. Thus the touchee priorities will not be affected, unless they are less than *min-pri* (which is 1), in which case the touchee tasks will go from being stayed (priority 0) to being unstayed. If the touchee tasks are stayed, they must be unstayed anyway so that spec2 terminates. The following two lines from spec2 demonstrate this solution.

```
(change-priority (my-future) *min-pri*)
(touchlist children)
```

The first line reduces the priority of the executing task to *min-pri*. The second line touches all the children tasks, as in line 5 of Figure 8.22, checking for termination.

We could refine the mandatory version to utilize the same ordering as the spec2 version. To enforce this ordering, this refined mandatory version would use an explicit priority queue written in Multilisp, as in the qtrav version in Section 8.5 (since there is no built-in support for priority queues in conventional Multilisp). Since the qtrav version is already an excellent example of the explicit queue approach, we do not consider the refined mandatory version further.

We ran the four versions with two different starting boards: board5, for which the shortest solution is five moves away, and board18 (from [Nilsson]), for which the shortest solution is 18 moves away. Figure 8.16 shows board5 and Figure 8.18 shows board18.

Table 8.19 shows the execution time on the Encore Multimax with 8 processors for these four versions and two starting boards.

For small search depths (depth = 4 for board5 and depth $\leq$ 10 for board18), the spec version is just slightly slower (than the mandatory version) whereas the spec2 version is about 1.3 times slower (cf. depth = 4 for board5 and depth = 10 for board18). For board5, the execution time of the mandatory and spec versions is erratic at greater depths because of random fluctuations in the ordering of board expansions. In contrast, the execution time of the spec2 version is consistent, due to control over the ordering, and usually smaller. Note that the speedup of the mandatory and spec versions in many cases, and the spec2 version especially, with respect to the sequential version is greater than eight, the number of processors. This superlinear speedup is a typical characteristic of multiple-approach speculative computation, as we saw in Section 8.2: the speedup depends on the position of

Start board	Search depth	Search version			
		Sequential	Mandatory	Spec	Spec2
board5	4	1.72	0.43	0.45	0.57
	5	3.27	0.60	0.50 - 0.75	0.60
	6	6.30	0.77 - 0.90	0.48 - 1.12	0.60
	8	20.7	1.1 - 2.2	1.3 - 2.8	0.60
board18	5	3.67	0.77	0.83	0.80
	10	77	11.1	11.7	14.5
	15	23 min.	192	crash	crash
	20	?	23 min.	crash	54

All times in seconds, unless otherwise indicated.

Table 8.19: Eight-puzzle execution time

the solution in the "subtrees" and the order in which these subtrees are searched.

For board18, the execution time of the spec2 version is far superior to that of the other versions, provided that the search depth is sufficient to include a solution (i.e. ≥ 18). For a search depth of 20, the sequential version took longer than we cared to measure and the spec version exhausted the heap (12MB on the Multimax) and crashed. The spec2 version, which did not exhaust the heap, was 26 times faster than the mandatory version! Thus careful control of the ordering reduced both the execution time and the memory requirements. There are two reasons why the mandatory version did not also exhaust the heap. The first reason is that mandatory tasks require considerably less storage than speculative tasks, as described in Chapter 10. The second reason is the unfair scheduling of mandatory tasks (see Section 1.4.2). As described in Section 10.2, once the machine saturates with mandatory tasks the child task created with `future` continues while the parent is inserted in a LIFO queue of runnable mandatory tasks. With the code for the mandatory version in Figure 8.22, this unfair scheduling will encourage a depth-first examination of the search space. In contrast, `spec-future` continues whichever of the parent and child has the higher priority (the parent in case of a tie) and inserts the other in a priority queue of runnable tasks. With the code for the spec version in Figure 8.23, all the speculative tasks have the same priority and thus `spec-future` will continue the parent task and process the runnable tasks in FIFO order. This scheduling will encourage a breadth-first examination of the search space which requires much more storage than a depth-first examination. The priorities in the spec2 version encourage a "directed" breadth-first search, focusing the breadth on the most promising search paths.

If the search depth is insufficient to include a solution, the spec2 version is only 1.3 times slower than the mandatory version for a depth of 10. For a depth of 15, the spec2 version exhausted the heap and crashed so we cannot say. The same two reasons as above account for why the spec2 version crashed in this case and the mandatory version did not.

This Eight-puzzle application demonstrates once again the importance of ordering (for execution time) and the expressive power gained by our support for speculative computation. As in Section 8.5, built-in support for ordering (in the form of priority queues) makes programming easier and improves efficiency by eliminating the layer of interpretation associated with explicit ordering (i.e. priority queues) at the language level.

This application also demonstrates the importance of ordering for controlling resource, i.e. memory, usage. The ordering for minimum resource use is not the necessarily the same as the ordering for minimum execution time, as demonstrated by the contrast between the mandatory and spec2 versions for a search depth of 15. For minimum execution time we want a breadth-first expansion of the P highest priority boards (where P is the number of processors) and for minimum memory use we want depth-first expansion of the boards on a single processor. In this fundamental tradeoff between execution time and memory use, we have focused on execution time and ignored memory use. This is a weakness of our work that must be addressed in the future.

The speculative version in Figure 8.23 has two problems. The first problem is that the speculative version cannot be restarted if it is stayed as part of a larger computation. Touching the result placeholder fails to restart all the descendant tasks spawned by `make-group` in line 3 since there is no demand continuity from this placeholder to the tasks. This is precisely the group incoherency problem discussed in Section 8.1.4. We can fix this problem in the same way as we did in Section 8.1.5: we can have the tasks blocked on the result placeholder sponsor a class containing the `make-group` task. (We omitted this fix for simplicity.) Then, since we did not break the touch propagation paths to perform termination detection, as we did with strav in Section 8.5, the termination detection touch will ensure that all tasks are eventually restarted. However, these tasks will only be restarted lazily, when actually touched. Hence, we would prefer a controller sponsor to solve this group incoherency problem. The second problem is really a generalization of the first problem: the speculative version uses absolute priorities so the priorities cannot be rescaled if it is part of a larger computation whose priority changes (such as being stayed). This problem results from a lack of modularity. It can be solved with a suitable mechanism for modularity which provides for local ordering relative to the group enclosing all the search tasks.

8.7 Summary

The applications in this chapter demonstrate three things: examples of speculative computation, benefits of speculative computation, and issues of speculative computation. We summarize each in turn.

8.7.1 Examples

Table 8.20 lists the inherent characteristics of each application. The applications cover all the different classifications of speculative computation mentioned in Section 1.1 except for

Application	Characteristics
por/pand	nondeterministic, multiple-approach speculative computation
Tree equal	application of pand
Emycin	nested pand, hence nested speculative computation side-effects sharing of computation
Boyer	ordering-based speculative computation producer-consumer relationship
Travsales	ordering
Eight puzzle	pand and ordering

Table 8.20: Characteristics of each application

precomputing speculative computation, which is not supported (very well) in our present implementation. Table 8.21 lists the contributions of each application.

8.7.2 Benefits

These applications demonstrate two different types of benefits of speculative computation. The first benefit is performance gain. Emycin demonstrates performance gains due to aborting useless computation and Boyer and Eight-puzzle demonstrate significant performance gains (a factor of about 2 with Boyer on 16 processors and a factor of 26 with Eight-puzzle on 8 processors) due to ordering speculative computation.

The second benefit is really the benefit of our support for speculative computation rather than the intrinsic benefit of speculative computation. Travsales and Eight-puzzle demonstrate how our support for speculative computation shields the user from resource management concerns and thus makes it easier to exploit machine resources. Tree equal demonstrates how our automatic naming of descendants makes aborting easier by solving the nesting and unknown function call problems. Emycin demonstrates how reversible "aborting" (a.k.a. staying) solves the problems with sharing of computation and side-effects. Finally, Boyer demonstrates how our touching model provides natural dynamic ordering to control producer/consumer scheduling, freeing the user from this burden.

8.7.3 Issues

Since we have covered the basic issues in speculative computation, like task scheduling, extensively in other chapters, we concentrate here on the issues these applications raise in our present implementation.

Application	Demonstrates
Tree equal	Importance of aborting useless computation - but how perform the aborting? How to use explicit checking and thus how speculative computation can be supported in a system offering only conventional parallelism
Emycin	Importance of aborting useless computation Explicit termination checking unsuitable because of side-effects and sharing of computation Touching model approach to side-effects
Boyer	Importance of ordering, particularly the dynamic ordering enforced by touching
Travsales	Importance of static ordering One way to perform touch termination
Eight-puzzle	Importance of ordering – yields speedup of 26 on 8 processors Another way to perform touch termination

Table 8.21: Contributions of each application

The key issue, demonstrated by all the applications, is group incoherency. This occurs when a group is stayed and later touched. Because of failure to propagate the demand, tasks in the group may not restart, leading in worst case to deadlock. We demonstrated with por how to use classes to solve this problem by ensuring demand continuity to all the required tasks in a group. This is an awkward solution, however. If deadlock is the only concern, we only need a solution like this if demand does not follow a data dependency path, such as with multiple-approach speculative computation like por. Otherwise, we can just wait for the tasks to be touched when demanded, so it is not necessary to eagerly restart all the tasks. This is the case, for instance, in Boyer and Eight-puzzle. However, this entails some performance disadvantage since all the tasks become lazy. To prevent deadlock and loss of eagerness, we would like automatic group coherency: all the tasks in a group should be restarted when the group is demanded.

Group incoherency is a symptom of the lack of modularity in our touching model. Staying destroys the ordering in a group, leading to group incoherency. Furthermore, the ordering is all absolute, leading to to unintended ordering conflicts that make it difficult to nest speculative computation, as we discussed with Boyer. We need modularity to provide local ordering so we can nest speculative computation arbitrarily.

We also need full controller sponsors so that we have a convenient way to reallocate sponsorship dynamically and a way to provide more complex sponsorship/control for applications like por.

We did not require modularity or full controller sponsors for the applications in this chapter because there was only one speculative activity in the system. While this is accept-

able for our purpose of testing the waters of speculative computation, future work must provide these mechanisms.

Chapter 9

Scheduling

In this chapter we consider optimal scheduling of speculative computation. Optimal scheduling is important for two reasons. First, and most obviously, the optimal schedule for a given problem provides the optimal control policy and hence the ideal control desired for that problem. In addition, the optimal control policy indicates the support — system scheduling capabilities and features — required for optimal scheduling. Second, the optimal schedule either provides or allows determination of the cost[1] of the optimal policy. The optimal cost provides

1. a yardstick with which to judge the merit of *ad hoc* and heuristic scheduling algorithms, and

2. an indication of the ultimate benefit of speculative computation.

Even if we cannot determine the optimal scheduling for a given problem, optimal scheduling is still important: the optimal scheduling for a simpler, related problem can present a guide for heuristic scheduling decisions in the original problem and suggest what support may be important for good scheduling.

In terms of our sponsor model, optimal scheduling is important for determining task ordering, rates, and controller sponsor policies.

In this chapter we restrict our scope to the scheduling of multiple-approach speculative computation and we focus just on optimal scheduling policies (rather than costs). We start by formulating the general scheduling problem for multiple-approach speculative computation. We then consider two specializations of this problem, `pif` and por/pand. We derive new results for the optimal scheduling of `pif` in simple cases and summarize some existing results for the optimal scheduling of por/pand. Then we consider examples of nested `pif`s, pors, and pands from which we draw some conclusions about the optimal scheduling of

[1] In terms of the optimality cost metric.

nested computations. Finally, we use the optimal scheduling for pif and por/pand as a springboard to discuss the system capabilities required for scheduling.

This chapter is intended to give a flavor of scheduling for speculative computation; this chapter is not an exhaustive treatment of the scheduling problem.

9.1 The Scheduling Problem

First we introduce some terminology which we use in the remainder of this chapter.

9.1.1 Terminology

As in the preceding chapters, a task is a sequential thread of computation. It is the smallest unit of computation which may be independently scheduled. We assume we have some number P of processors. Throughout this chapter, we assume that all these processors are identical. Each processor has, at each instant of time, one unit of processing power which it may allocate amongst zero or more tasks. Thus each processor may execute one or more tasks concurrently. The allocation of processing power to tasks is described by *rate* functions. $r_i\ (t)$ is the instantaneous rate of execution of task i on processor j at time t. Rate 0 corresponds to no execution (i.e. no processing power); rate 1 corresponds to the maximum execution rate (i.e. full processing power) that a processor can provide; and rate r, $0 < r < 1$, corresponds to execution with fraction r processing power. More formally, a rate r has the following interpretation. Let $C_i\ (t, \Delta t)$ be the total amount of processing received by a task i running at constant rate r in $[t, t+\Delta t)$ on processor j. Then $r_i\ (t) \equiv \lim_{\Delta t \to 0} \frac{C_{ij}(t,\Delta t)}{\Delta t}$. Thus, executing a task at (constant) rate r for an interval Δt is equivalent, in terms of the total computation performed by the task, to executing at rate 1 for an interval $r\Delta t$. We call rates 0 and 1, integer rates, and rates between 0 and 1, fractional rates. If some subset $T_j(t)$ of the tasks executing on processor j at time t have a fractional rate, we say that the tasks in $T_j(t)$ are *multiplexed.*

We have the following constraints on rates:

1. All rates are in $[0,1]$: $0 \leq r_i\ (t) \leq 1\ \forall i, j, t$
2. No processor allocates more than one unit of processing power: $\sum_i r_i\ (t) \leq 1\ \forall j, t$
3. No task executes on more than one processor simultaneously: $r_i\ (t) > 0 \Rightarrow r_i\ (t) = 0\ \forall j \neq k$

Constraints 1 and 3 imply that no task receives a total rate greater than 1, i.e. $\sum_j r_i\ (t) \leq 1$ $\forall i, t$.

If the execution rate of a task i changes from nonzero to zero at time t and the execution of task i is unfinished at time t, we say that task i is preempted at time t. A preempted task

is replaced by zero or more other tasks. If a task runs to completion on the same processor on which it started, without preemption, we say the task is non-preempted.

We can approximate fractional rates by round-robin preemption with integer rates. To approximate constant rates r_1, r_2, ..., r_n of respective tasks 1, 2, ..., n in some interval Δt, we just have to run each task i at rate 1 for time $r_i \Delta t$. (Any implementation would approximate fractional rates in this manner.) In the limit as $\Delta t \rightarrow 0$, we actually have fractional rates.

The execution of task i is fully described by the vector of rate functions $\mathbf{r}_i(t) = [r_i(t), \ldots, r_i(t)]$ (Since a task may not run on more than one processor simultaneously and all processors are identical, we will often just use the scalar function $r_i(t)$ to describe the task rate.) Given a set of tasks T, a *schedule* for execution of these tasks is a set of vector rate functions $\{\mathbf{r}_i(t), \forall i \in T\}$ which meet the above rate constraints. A *decision point* is a time t at which a scheduling decision may be made. A *scheduling decision* is a specification of a set of vector rate functions, i.e. a schedule, until the next decision point. A scheduling decision can only depend on the state at the decision point. The interval until the next decision point may depend on the scheduling decision at the most recent decision point and the state at that decision point. In any case, a decision point must occur at $t = 0$ and at any instant that a task completes execution. A scheduling *policy* is a function which, given a decision point, the current state at the decision point, and any constraints on decision points, yields the scheduling decision at that decision point. An *optimal schedule* is a schedule that minimizes the expected time to complete processing of some specified subset of tasks in T subject to any given constraints on decision points. An *optimal policy* is a policy which achieves an optimal schedule. We will be more precise about the input conditions and specification of optimality in the next section.

We are generally interested in an optimal schedule without any constraints on the decision points. In such schedules, a decision point may occur at each instant of time. We call such schedules *continuous decision* schedules. If the decision times are restricted to discrete points, the resulting schedule is a *discrete decision* schedule. For a given problem, a continuous decision schedule can always simulate a discrete decision schedule.

Finally, a schedule in which no task is ever preempted is said to be non-preemptive. Otherwise, the schedule is preemptive.

In general, fractional rates and preemption have nonzero overhead associated with them. Throughout this chapter, we assume this overhead is zero.

9.1.2 Scheduling problem formulation

We formulate the general scheduling problem for multiple-approach speculative computation as follows.

We are given a set of N activities denoted by A_i, $i = 1, \ldots, N$. Only some subset S of these activities may be required, such as the first activity to return a result or the first activity to return a true result. $p_i(t)$ denotes the probability, observed at time t, that

activity A_i will be required. We are given the *a priori* probabilities $p_i(0) \equiv p_i$. Each activity A_i consists of tasks. Rather than attempting to describe the interrelationship of tasks within an activity, we characterize (conceptually) each activity by its execution time as a function of the resources assigned to that activity. More precisely, the execution time t_i of activity A_i is given by the probability distribution function (PDF) $Prob(t_i \leq \tau) = F_i(\tau, \mathbf{M}_i)$ where $\mathbf{M}_i(\cdot)$ is the *resource allocation vector*. $\mathbf{M}_i(t) = [m_{1_i}(t), m_{2_i}(t), \ldots, m_{j_i}(t), \ldots]$ where $m_{j_i}(t)$ is the maximum processing rate allocated to activity i on processor j as a function of time. In other words, $m_{j_i}(t)$ is the time-varying fraction of processor j available to A_i. (The resource allocation vector should not be confused with the rate vector defined in the previous subsection. The rate vector describes the processor resources actually utilized by tasks whereas the resource allocation vector specifies the processor resources allocated to activities, but perhaps not fully utilized.) We call F_i the resource execution time PDF. Finally, we are given P identical processors, each which may execute tasks concurrently subject to the constraints (on rates) and assumptions (zero overhead) in Section 9.1.1.

The objective is to determine an optimal scheduling policy. We define an optimal schedule for this problem as a schedule for the tasks constituting the activities A_i which minimizes the expected execution time, defined as the time to fulfill the requirements of subset S. We allow continuous decision schedules and preemption.

The resource execution time PDF allows us to talk about activities without getting bogged down in a description of the activities (by precedence relationships for example) and it allows us to consider the interaction and nesting of activities in a uniform manner, simply by combining resource execution PDFs. We start by determining the resource execution time PDFs for leaf activities and propagate them back up the activity tree, combining the resource execution time PDF at each fork and interaction point to obtain a new resource execution time PDF and so on. We emphasize that resource execution PDFs are a conceptual way of viewing scheduling; we doubt any practical gain will result from this view since scheduling is difficult enough without the additional difficulty of trying to determine and manipulate such PDFs.

This formulation excludes partial results (as in Section 5.4.6) and resources other than processors. We excluded these factors to simplify the presentation. Their inclusion is straightforward.

In the following sections we examine two specializations of this general scheduling problem. In both cases we assume that all the activities are independent and that all the probabilities $p_i(t)$ are fixed at their *a priori* values for all t. We investigate both the optimal scheduling policies for these problems and the characteristics of these policies, such as whether they require fractional rates.

9.2 Parallel Branch Scheduling

In this section we derive the optimal scheduling for a simple specialization of the general scheduling problem involving parallel branch scheduling. We start with parallel `if` (`pif`)

and then later consider an n-ary generalization of pif which we call pbranch.

9.2.1 pif

As described in Section 1.2.2, (pif *pred consequent alternate*) evaluates *pred*, *consequent* (which we abbreviate by *con*) and *alternate* (which we abbreviate by *alt*) concurrently. If *pred* evaluates to true we accept the result of *con* and abort *alt*, and if *pred* evaluates to false, we accept the result of *alt* and abort *con*. We are interested in the scheduling of *pred*, *con*, and *alt* to minimize the expected execution time of pif.

In this section we restrict the predicate, *pred*, and each branch, *con* and *alt*, to be a single task. Thus the predicate and branches cannot create child tasks or be another pif. We model the execution time (at rate 1) of *pred*, *con*, and *alt* by random variables t_i, $i = pred, con, alt$, respectively, with given probability distribution functions (PDFs) $F_i(t)$, respectively. We model the outcome of the predicate by a random variable taking values true and false with *a priori* probabilities p and $1 - p$ respectively. We assume that 1) all these random variables are mutually independent, 2) tasks are always runnable until they complete or abort, and 3) *con* and *alt* do not produce the same value, so we must always evaluate *pred*. Under these conditions, a precise statement of the pif scheduling problem is as follows:

> Given PDFs for t_{pred}, t_{con}, and t_{alt} and the probability p, schedule the execution of the tasks on P processors to minimize the expected time to return the result. That is, specify execution rates $r_{pred}(t)$, $r_{con}(t)$, $r_{alt}(t)$ as a function of time for each task *pred*, *con*, and *alt*, respectively, to minimize the expected execution time.

The predicate task is in the critical path, so clearly we must complete it as soon as possible. Hence $r_{pred}(t) = 1$ for $0 \leq t \leq t_{pred}$ where t_{pred} is the execution time of *pred*. Thus this scheduling problem has two epochs: the interval $[0, t_{pred})$ before *pred* returns and the interval $[t_{pred}, \infty)$ after *pred* returns. The scheduling in this second epoch is obvious: if *pred* returns true (false) and *con* (*alt*) has not already completed, execute *con* (*alt*) at rate 1 until its completion.

Given our restriction to single tasks, the scheduling is non-trivial only for $P = 2$ (when *con* and *alt* must share a single processor). For this case, the optimal scheduling policy in the first epoch has the following simple form:

> Run *pred* at rate 1 and whichever of *con* and *alt* has the higher probability at rate 1 until either it completes or *pred* completes. If only one task remains, run that task at rate 1 until it completes or *pred* completes.

Note that we only need decisions at $t = 0$ and the points at which tasks complete. Thus the optimal continuous decision schedule reduces to a discrete decision schedule. The optimal schedule is

$r_{pred}(t) = 1$ for $t \in [0, t_{pred})$, and

$$\text{if } p \geq 1/2 \begin{cases} r_{con}(t) = 1 \text{ and } r_{alt}(t) = 0 & \text{for } t \in [0, min(t_{con}, t_{pred})) \\ r_{alt}(t) = 1 & \text{for } t \in [t_{con}, min(t_{con} + t_{alt}, t_{pred})) \end{cases}$$

$$\text{if } p < 1/2 \begin{cases} r_{alt}(t) = 1 \text{ and } r_{con}(t) = 0 & \text{for } t \in [0, min(t_{alt}, t_{pred})) \\ r_{con}(t) = 1 & \text{for } t \in [t_{con}, min(t_{con} + t_{alt}, t_{pred})) \end{cases}$$

In particular, neither preemption nor fractional rates can reduce the expected execution time further.

This result follows from a special case of the optimal scheduling for pbranch, considered in the next section. Under some conditions preemption and fractional rates may be optimal (such as when $p > 1/2$ and $t_{con} < t_{pred}$ or when $p < 1/2$ and $t_{alt} < t_{pred}$) but an equally optimal schedule with non-preemption and integer rates is always possible.

9.2.2 pbranch

(pbranch *pred* e_1 e_2 ... e_n) may evaluate *pred* and e_1, e_2, ..., e_n concurrently. If *pred* evaluates to an integer $i \in [1, n]$, pbranch returns the result of evaluating e_i. Otherwise, pbranch returns the result of evaluating e_n. All other e_j, $j \neq i$, may be aborted once *pred* selects e_i. We are interested in the scheduling of *pred* and all the e_i to minimize the expected execution time of pbranch.

As with pif, we restrict the predicate, *pred*, and each branch, e_i, to be a single task in this section. We model the execution time (at rate 1) of *pred* and each e_i, $i = 1, 2, \ldots, n$ by random variables t_i, $i = pred, 1, 2, \ldots, n$, respectively, with given probability distribution functions (PDFs) $F_i(t)$, respectively. We model the outcome of the predicate by a random variable taking values $1, 2, \ldots, n$ with *a priori* probabilities $p_1, p_2, \ldots, p_n$ respectively. We assume that 1) all these random variables are mutually independent, 2) tasks are always runnable until they complete or abort, and 3) not all e_i produce the same value, so we must always evaluate *pred*. Under these conditions, a precise statement of the parallel branch scheduling problem is as follows:

> Given PDFs for t_{pred} and all the t_i and the probabilities p_i, schedule the execution of the tasks on P processors to minimize the expected time to return the result. That is, specify execution rates $r_{pred}(t)$, $r_1(t)$, ..., $r_n(t)$ as a function of time for each task to minimize the expected execution time.

As with pif, the predicate task is in the critical path, so clearly we must complete it as soon as possible. Hence $r_{pred}(t) = 1$ for $0 \leq t \leq t_{pred}$ where t_{pred} is the execution time of *pred*. Thus, like pif, this scheduling problem has two epochs: the interval $[0, t_{pred})$ before *pred* returns and the interval $[t_{pred}, \infty)$ after *pred* returns. The scheduling in this second epoch is obvious: if *pred* returns i and e_i has not already completed, execute e_i at rate 1

until its completion. We define the time remaining in task e_i at time t_{pred} as the e_i excess time, which we denote by Δ_i.

Thus for a given scheduling policy, the expected execution time is $E[t_{pred}] + E[\Delta]$, where $E[\Delta]$ is the expected excess time given by

$$E[\Delta] = \int_{r=0}^{\infty} \left(\sum_{i}^{n} p_i E[\Delta_i \mid t_{pred} = r] \right) f_{pred}(r) dr$$

$f_{pred}(t)$ is the probability density function (pdf) for t_{pred}, which we assume exists.

Given our restriction to single tasks, the scheduling is non-trivial only for $n \geq P > 1$.

We need to determine the scheduling policy that minimizes $E[\Delta]$. Our strategy is to determine the scheduling policy that minimizes $E[\Delta \mid t_{pred}, \mathbf{t_{br}}]$ where $\mathbf{t_{br}} = [t_1, t_2, \ldots, t_n]$. If a given scheduling policy Γ for this conditional problem is optimal for every set of values of t_{pred} and $\mathbf{t_{br}}$, then policy Γ is optimal for the original problem, i.e. it minimizes $E[\Delta]$. (The existence of such a policy Γ is not guaranteed in general.)

We need some terminology before we proceed. Let x_i be the processing time (i.e. amount of processing power) received by e_i prior to time t_{pred}, i.e. $x_i = \int_{r=0}^{t_{pred}} r_i(r) dr$. We assume throughout the following that $r_{pred}(t) = 1$ for $0 \leq t \leq t_{pred}$ as discussed earlier. Then for the conditional problem, we have the following result for the optimal allocation of task processing times in the first epoch. (We follow Lemma 9.1 with a theorem relating optimal scheduling to this processing time allocation.)

Lemma 9.1: Optimal processing time allocation given task execution times

Given t_{pred} and $\mathbf{t_{br}}$, the optimal allocation of processing times in pbranch (for other than *pred*) is as follows:

> First, without loss of generality, arrange the e_i in an order such that $p_i \geq p_j$ $\forall i$. Then
>
> $$x_i = \min(t_i, t_{pred}, (P-1)t_{pred} - \sum_{j<i} x_j)$$
>
> (Either assign e_i processing time t_i to complete the task (if $t_i \leq t_{pred}$) or assign e_i all unallocated processing time up to t_{pred}.) If k p_i have the same value, any re-assignment of the respective x_i such that the sum of these k x_i remains unchanged is also optimal.

Proof:

We need to show that this allocation of processing times minimizes $E[\Delta \mid t_{pred}, t_{br}]$. (Clearly, this allocation is realizable since $\sum_i^n x_i \leq (P-1)t_{pred}$.) Since

$$E[\Delta \mid t_{pred}, t_{br}] = \sum_i^n p_i(t_i - x_i) = \sum_i^n p_i t_i - \sum_i^n p_i x_i$$

minimizing $E[\Delta \mid t_{pred}, t_{br}]$ reduces to the problem:

$$\begin{array}{ll} \text{maximize} & \sum_i^n p_i x_i \\ \text{subject to} & 0 \leq x_i \leq \min(t_i, t_{pred}) \ \forall i \\ & \sum_i^n x_i \leq (P-1)t_{pred} \end{array}$$

This is a linear program. From a well-known result of linear programming theory, the optimal solutions occur at the extrema of the constraint region. The extrema here occur for $x_i = 0$, $\min(t_i, t_{pred})$, and $(P-1)t_{pred} - \sum_{j \neq i} x_j$.

Now we have to show that the particular choice of these extrema mentioned in the lemma is optimal. There are two cases:

1. $\sum_i^n \min(t_i, t_{pred}) < (P-1)t_{pred}$

   In this case, there is excess processor capacity so the optimum solution is obviously $x_i = \min(t_i, t_{pred})$ for all i.

2. $\sum_i^n \min(t_i, t_{pred}) \geq (P-1)t_{pred}$

   In this case $\sum_i^n x_i = (P-1)t_{pred}$. Assume for the moment that no two p_i have the same value. For those familiar with linear programming, the proof is obvious from a geometric argument. We sketch an algebraic proof here.

   We have $U \equiv (P-1)t_{pred}$ to distribute amongst n x_i. Let c_m be the upper bound on x_m ($c_m \equiv \min(t_m, t_{pred})$) for $m = 1, \ldots, n$. Start with $m = 1$. Consider distributing the first c_m of the U amongst the x_i. Since $p_m > p_i$ for all $i \geq m$ (by hypothesis), the cost (i.e. the p_i weighted sum of x_i) with this first c_m is maximized by setting $x_m = c_m$. This leaves $U - c_m$ to distribute amongst the remaining $n - m$ x_i. We then continue recursively with $m \leftarrow m + 1$.

   To handle two or more p_i with the same value, we can transform our linear program into an equivalent one in which no two p_i' are the same by applying the following procedure (repeatedly as necessary): If p_i has the same value for all $i \in K$, let $p_i' = p_i$, replace $\sum_i \; p_i x_i$ in the cost equation by $p_i' x_K'$, replace all the constraints $0 \leq x_i \leq \min(t_i, t_{pred})$ for $i \in K$ by the single constraint $0 \leq x_K' \leq \min(\sum_i \; t_i, \mid K \mid \cdot t_{pred})$ (where $\mid K \mid$ is the cardinality of K), and replace $\sum_i \; x_i$ in the last constraint by x_K'. We can then apply the previous argument to this equivalent linear program. This equivalent linear program also makes it clear that in the optimal solution the x_i for $i \in K$ may have any value so long as they obey the constraints and $\sum_i \; x_i = x'_K$.

Theorem 9.1 indicates how these x_i lead to optimal schedules for the first epoch (when $0 \leq t \leq t_{pred}$).

Theorem 9.1: Optimal scheduling given task execution times

Given t_{pred} and t_{br}, any schedule for pbranch with $r_{pred}(t) = 1$ for $0 \leq t \leq t_{pred}$ and a set of rates $r_i(t)$ $(0 \leq t \leq t_{pred})$ which yields the optimal values of x_i $(x_i = \int_{\tau=0}^{t_{pred}} r_i(\tau)d\tau)$, as given by Lemma 9.1, and subject to the following constraints (explained in Section 9.1.1)

$$0 \leq r_i(t) \leq 1 \; \forall i, j, t$$
$$\sum_i r_i(t) \leq 1 \; \forall j, t$$
$$\forall k, t, r_i(t) > 0 \Rightarrow r_i(t) = 0 \; \forall j \neq k$$

is optimal.

Proof:

We need to show that any such schedule minimizes $E[\Delta \mid t_{pred}, t_{br}]$. We have already argued that an optimal schedule must have $r_{pred}(t) = 1$ for $0 \leq t \leq t_{pred}$. If the rates achieve the optimal x_i, then by Lemma 9.1 $E[\Delta \mid t_{pred}, t_{br}]$ is minimized.

Thus for the conditional problem, i.e. given t_{pred} and t_i $\forall i$, an optimal schedule is any schedule which achieves the optimal x_i. Such a schedule always exists: since $x_i \leq t_{pred}$ we never need a rate > 1 and we can always share the processing amongst all the processors. However, it is not always straightforward to determine such a schedule: preemption or fractional rates may be necessary to achieve the optimal x_i. See the discussion in Section 9.2.3.

In several special cases, an optimal scheduling policy which minimizes $E[\Delta \mid t_{pred}, t_{br}]$ also minimizes $E[\Delta]$ and thus constitutes a optimal scheduling policy for pbranch.

Case 1: t_{pred} and all t_i deterministic

In this case $E[\Delta] = E[\Delta \mid t_{pred}, t_{br}]$ and therefore an optimal schedule for pbranch in the first epoch is given by any rates which satisfy the optimal x_i as given previously.

Case 2: $P = 2$

From Lemma 9.1, the optimal x_i in this case for given t_{pred} and t_i is to order the e_i in non-increasing order of p_i, and set $x_i = \min(t_i, t_{pred}, t_{pred} - \sum_{j<i} x_i)$. One schedule which achieves these x_i is to run each of the e_i sequentially, in non-increasing order of p_i, at rate 1 until either it completes or *pred* completes. Thus an optimal scheduling policy for minimizing $E[\Delta \mid t_{pred}, t_{br}]$ in the first epoch is as follows:

> Run *pred* at rate 1 until it completes and, in non-increasing order of p_i, run each of the e_i sequentially at rate 1 until either it completes or *pred* completes.

Note that we only need decisions at $t = 0$ and the points at which tasks complete. Thus the optimal continuous decision schedule reduces to a discrete decision schedule. Since this scheduling policy is optimal given any set of values for t_{pred}, and t_i $\forall i$, this policy is also optimal for the original problem, i.e. it minimizes $E[\Delta]$. Of course, since this policy is optimal, neither preemption nor fractional rates can reduce the expected excess time further.

Thus we have shown the previously claimed result for pif.

Case 3: Restricted t_{pred} and t_i

In some cases the optimal policy for minimizing $E[\Delta \mid t_{pred}, t_{br}]$ is the same given any set of values for t_{pred} and t_i within certain ranges and therefore this policy is optimal for minimizing $E[\Delta]$, provided the random variables stay within these ranges. The following is one such case.

For the $P-1$ t_i with largest p_i, $t_i \geq t_{pred}$ for all possible values of t_{pred} and t_i:

In this case, for given t_{pred} and t_i we have $x_i = t_{pred}$ for the $P-1$ largest p_i. One scheduling policy that achieves these x_i is to assign e_i to the first available processor, to run at rate 1, in non-increasing order according to p_i. Since this policy is the same for all t_{pred} and t_i obeying the restrictions, this policy also minimizes $E[\Delta]$ and thus is an optimal scheduling policy for pbranch subject to the stated conditions on t_{pred} and t_i $\forall i$. Once again, this optimal continuous decision schedule reduces to a discrete decision schedule.

9.2.3 Preemption and rates

We need preemption and fractional rates, in general, for the optimal scheduling of pbranch. The reason is that we want to avoid fragmentation in assigning tasks to processors that could lead to idle processors while outstanding tasks remain. For example, suppose $P = 3$, and we have three identical e_i with $t_i = \frac{2}{3}t_{pred}$ and $p_i = \frac{1}{3}$. Then without fractional rates or preemption, the minimum expected excess time we can achieve is $\frac{1}{3} \cdot \frac{1}{3}t_{pred}$ with an idle period of $\frac{1}{3}t_{pred}$ on one of the processors. However, by either multiplexing all three tasks at rate $\frac{2}{3}$ for $t \in [0, t_{pred})$ or by running each task in round-robin fashion at rate 1 for some quantum like $\frac{1}{3}t_{pred}$, the expected excess time is zero. This illustrates that fractional rates are not necessary for deterministic task durations provided we can preempt tasks at specified intervals: we can always simulate fractional rates by running tasks in round-robin fashion for suitable quanta.

However, fractional rates are sometimes necessary for nondeterministic task durations. Consider the following modification of our previous example. Suppose now that the three e_i have $t_i = u_i$ where u_i is a uniformly distributed random variable in $[0, \frac{2}{3}t_{pred} + \epsilon]$, where $\epsilon > 0$ (t_{pred} remains deterministic.) Then multiplexing is necessary to avoid an unnecessary idle processor. Since we do not know u_i we cannot simulate multiplexing with preemption. no matter how small we choose the quantum, if it is not infinitesimal, multiplexing can still do better. (Of course, if the quantum is infinitesimal, we have multiplexing.)

In special cases we need neither preemption nor fractional rates. Two such special cases are:

1. deterministic tasks with $t_i \geq t_{pred}$ for the $P-1$ t_i with largest p_i

   Fractional rates are never optimal in this case since we need $r_i(t) = 1$ to get $\int_{\tau=0}^{t_{pred}} r_i(\tau)d\tau = x_i = t_{pred}$ for these i. An optimal schedule may include preemption — for example, we preempt two tasks on different processors and swap them — but preemption never improves the schedule. Thus for this case, neither fractional rates nor preemption are required. This means that the optimal continuous decision policy amounts to a discrete decision schedule with decision points at $t = 0$ and t_{pred}.

2. $P = 2$, deterministic or nondeterministic tasks

   We never need preemption or fractional rates for this case as discussed in Case 2 of Section 9.2.2. However, an optimal schedule for this case can include fractional rates.

9.2.4 Summary

We completely characterized the optimal scheduling for pif with single tasks. The optimal scheduling policy for pif is particularly simple: Assign priority a to the predicate task, priority b to whichever of the consequent and alternate is most likely to be chosen, and priority c to the other task where $a > b > c$. Now run the tasks in priority order at rate 1 on the available processors (until the predicate completes, at which time run just the consequent or alternate as appropriate). Preemption and fractional rates are never required for optimal scheduling of pif.

The optimal scheduling for pbranch with single tasks is more difficult to determine in general. We derived a necessary condition (Theorem 9.1) for optimal scheduling for deterministic task execution times. It is not always straightforward to determine an optimal schedule from this condition (but it is always possible). In special cases, the optimal scheduling for this deterministic case is also optimal for nondeterministic execution times. Finally, we showed that preemption and fractional rates are necessary, in general, for optimal scheduling of pbranch.

9.3 por/pand Scheduling

We gave informal definitions of por and pand in Section 1.2.1. (See Appendix B for formal definitions.) We consider only the scheduling of por in this section, since pand may be framed in terms of por by the identity

$$(\text{pand } E_1\ E_2 \ldots E_n) = (\text{not } (\text{por } (\text{not } E_1)\ (\text{not } E_2) \ldots (\text{not } E_n))$$

With por, unlike with pbranch, we do not know which expression will select the result. This key difference makes the por scheduling problem much more difficult than the parallel

branch scheduling problem.

In this section we restrict each E_i to be a single task. We model the execution time (at rate 1) of each E_i by a random variable t_i with a probability distribution function (PDF) $F_i(t)$, $i = 1, 2, \ldots, n$, respectively. We model the outcome of each E_i by a random variable with possible values true and false and *a priori* probability true of p_i for $i = 1, 2, \ldots, n$. We assume that all these random variables are mutually independent. We also assume that tasks are always runnable until they complete or abort. Under these conditions, a precise statement of the por scheduling problem is as follows:

> Given probabilities p_i and PDFs for all the t_i, schedule the execution of the tasks on P processors to minimize the expected time to return the result. That is, specify execution rates $r_1(t)$, $r_2(t)$, ..., $r_n(t)$ as a function of time for each task to minimize the expected execution time.

9.3.1 One processor

First assume that there are integer rates and no preemption so that each $r_i(t) = 1$ from the time E_i starts until it completes t_i later. The no-preemption assumption amounts to restricting decision points to occur only at $t = 0$ and when tasks complete. Under these conditions, the expected execution time with a task order of $i, j, k, \ldots$ is

$$E[t_i] + (1 - p_i)(E[t_j] + (1 - p_j)(E[t_k] + (1 - p_k)(\ldots)))$$

As first proven by [Mitten], the expected execution time is minimized by ordering the tasks for execution so that the ratio $E[t_i]/p_i$ is non-decreasing. This is a very pleasing solution for it amounts to assigning each E_i a priority $p_i/E[t_i]$.

However, (constant) integer rates are not always optimal, even given the above restriction on decision points. And, without this restriction, non-preemption is not always optimal either.

Multiplexing necessary

Suppose we have two tasks with identical execution time pdfs $f_i(t) \equiv f(t)$ and identical probabilities $p_i \equiv p$. Let t_i denote the execution time of these tasks and $E[t_i] \equiv E[t]$ their expected execution time. We assume, as always in this chapter, that the task execution times and task outcomes are all mutually independent. In addition, we assume that decision points may only occur at time zero and task completion instants. Then if we run the tasks at integer rates, the expected total execution time $E[t_{tot}]$ is

$$E[t_{tot}] = E[t] + (1 - p)E[t] = 2E[t] - pE[t] \tag{9.1}$$

Suppose now that we multiplex both tasks, i.e. run each at rate 1/2, until the first task completes. Let $z_{mi} = \min(t_1, t_2)$ and $z_{max} = \max(t_1, t_2)$. Then the expected time until

the first task completes is $2E[x_{min}]$ since both tasks run at rate 1/2 until that point. The expected remaining execution time at that point for the continuing task is $E[x_{max}] - E[x_{min}]$. Thus

$$E[t_{tot}] = 2E[x_{min}] + (1-p)(E[x_{max}] - E[x_{min}]) = (1-p)E[x_{max}] + (1+p)E[x_{min}]$$

Using the identity $E[x_{min}] + E[x_{max}] = 2E[t]$ and rearranging terms, we have

$$E[t_{tot}] = 2E[t] - 2p(E[t] - E[x_{min}]) \tag{9.2}$$

Comparing Equations 9.1 and 9.2, we see that multiplexing is superior if $E[t] > 2E[\min(t_1, t_2)]$.

Consider the following two examples.

1. Suppose t_1 and t_2 are uniformly distributed in $[0, 1]$. Then $E[t] = 1/2$ and $E[\min(t_1, t_2)] = 1/3$. Thus integer rates are superior.

2. Suppose t_1 and t_2 have the hyperexponential pdf $f(t) = \alpha a e^{-at} + (1-\alpha) b e^{-bt}$ with $0 < \alpha < 1$, $a > 0$, $b > 0$, and $a \neq b$. Then $E[t] = \frac{\alpha}{a} + \frac{1-\alpha}{b}$ and $E[\min(t_1, t_2)] = \frac{\alpha^2}{2a} + \frac{(1-\alpha)^2}{2b} + \frac{2\alpha(1-\alpha)}{a+b}$. Now $E[t] - 2E[\min(t_1, t_2)] = \alpha(1-\alpha)\frac{(a-b)^2}{ab(a+b)} > 0$, and thus multiplexing is always superior.

This last example demonstrates that multiplexing sometimes is optimal. Thus sometimes we need fractional rates for optimal scheduling. We cannot simulate fractional rates by round-robin preemption since, as we discussed in Section 9.2.3, we do not know which quantum size to choose (unless the quantum is infinitesimal, in which case we have fractional rates).

Preemption necessary

Preemption is sometimes optimal with continuous decision scheduling. Suppose we have two tasks E_1 and E_2 where E_1 has deterministic execution time $t_1 = 2$ and E_2 has execution time $t_2 = 1$ with probability .9 and $t_2 = 10$ with probability .1. Suppose also that $p_1 = p_2 = p$. Then since $E[t_2]/p_2 = 1.9/p$ is less than $E[t_1]/p_1 = 2/p$, the optimal non-preemptive schedule (i.e. continuous time schedule) is to run E_2 first, yielding an expected execution time of $1.9 + (1-p)2 = 3.9 - 2p$. However, if we run E_2 until time 1, followed by E_1 if necessary, and then E_2, if necessary, the expected execution time is $1 + .9(1-p)2 + .1(2 + (1-p)9)$ which simplifies to $3.9 - 2.7p$. Since this is less than $3.9 - 2p$, this latter schedule, with preemption, is superior.

Thus optimal scheduling of por on a single processor involves fractional rates and preemption in general.

9.3.2 More than one processor

With more than one processor, the solution of the scheduling problem is much more difficult because the system state must include the remaining time in each task on all the other processors, even if there is no preemption or fractional rates. With one processor the remaining time (at least with no preemption or fractional rates) is always 0.

There is a large amount of literature on scheduling to minimize the makespan, which is the total time to execute all the given tasks. These results are applicable to our scheduling problem when all $p_i = 0$ (so all tasks are required). We consider a number of special cases based on the literature and simple observations.

Case 1: All execution times deterministic and all $p_i = 0$.

With preemption, the total execution time of the optimal schedule is

$$\max(\frac{1}{P}\sum_{i}^{n} t_i, \max_{1 \le i} t_i)$$

[Muntz]. (As with pbranch, we do not need fractional rates if all tasks are deterministic and we have preemption: we can simulate fractional rates with preemption.) [Gonz78] shows that an optimal preemptive schedule requires at most $2(n-1)$ preemptions.

With non-preemption (i.e. decision points restricted to the instants at which tasks complete) this deterministic scheduling problem is the dual of the bin-packing problem. The bin-packing problem, expressed in terms of our framework, is as follows:

> Given n tasks with execution times t_i, $i = 1,2,\ldots,n$ and given a bound B, partition the tasks amongst P processors to minimize P subject to $T_p \le B$, $1 \le p \le P$, where T_p is the total execution time of tasks in the partition scheduled on processor p. That is, treating the tasks as items of size t_i, place the tasks in bins of size B to minimize the number of bins required.

The scheduling problem in terms of this bin-packing problem is: given n tasks with execution times t_i and given P, what is the smallest B such that the optimal number of bins required is less than or equal to P?

Since the optimal bin-packing problem is well-known to be NP-complete, the optimal scheduling problem for this special case without preemption is also NP-complete. This gives a hint of what makes the por scheduling problem so difficult.

The value of preemption in this deterministic scheduling problem is in eliminating any processor idleness before all the processors complete. Consider the following example. Suppose $P = 2$ and we have three tasks, each of length T. Without preemption, the best we can do is a schedule length of $2T$ with one processor idle for T. With preemption, we can get a schedule length of $3T/2$, a reduction of $T/2$. Thus preemption can be of significant benefit.

In general, we have the following result on the performance with and without preemption. Let C_P be the optimal execution time with preemption and let C_{NP} be the optimal execution time without preemption. Then $C_{NP}/C_P \leq 2 - 1/P$ [Gonz77]. The previous example indicates that this bound is tight at least for $P = 2$.

Case 2: All $p_i = 0$, all t_i identically distributed with the same PDF $F(t)$, and monotone hazard rate (defined below). (Or all $p_i = 0$ and all t_i exponentially distributed with possibly different parameters.)

Assuming that the probability density $f_i(t)$ exists (i.e. $F_i(t)$ is differentiable), the hazard rate for task i is

$$\rho_i(x_i) = f_i(x_i)/(1 - F_i(x_i))$$

where x_i is the amount of processing ($\int_{\tau=0}^{t} r_i(\tau)d\tau$) that task i has already received. The interpretation of $\rho_i(x_i)$ is as follows. If task i has already received an amount x_i of processing, then the probability that task i completes with infinitesimal δ further processing is $\delta\rho_i(x_i) + O(\delta^2)$. For an exponential distribution with parameter λ, $\rho(x) = \lambda$.

If all the tasks have the *same* PDF with monotone hazard rate, then [Weber] shows that the Least Hazard Rate (LHR) scheduling policy is optimal for continuous decision scheduling. At each point in time, the LHR policy assigns first the task(s) with the lowest hazard rate to processors at rate 1, then assigns the task(s) of the second lowest hazard rate to any remaining processors at rate 1, and so on. If the number of processors available at any point is less than the number of the lowest hazard rate tasks still unassigned, then (1) if the hazard rate is increasing, all these tasks are multiplexed on the available processors and (2) if the hazard rate is decreasing, these tasks are arbitrarily assigned processors at rate 1 until no processors are available.

If all the t_i are exponentially distributed (with possibly different parameters λ_i), a variation of the LHR policy is optimal: assign tasks non-preemptively to processors at rate 1 in order of increasing λ_i, i.e., largest $E[t_i]$ first. This is the opposite of the ordering we found with one processor (for $p_i > 0$).

Note that if the hazard rate is decreasing, the LHR policy is non-preemptive.

Case 3: All t_i are exponentially distributed

Because of the memoryless property of exponential distributions we need only consider preemption when tasks complete, i.e. an optimal schedule with decision points restricted to task completion times is an optimal continuous decision schedule. Assuming integer rates, we can write an expression for the expected execution time in this case quite simply as follows. Let A denote the current set of available, i.e. unfinished tasks, and let $D(A)$ denote the set of scheduling decisions given A, i.e. each element of $D(A)$ is a set of tasks that could be run given A. Let $E[A]$ denote the expected execution time, given A, that is achieved by an optimal schedule. Finally, let the exponential parameter of task t_i be λ_i (i.e.

$E[t_i] = 1/\lambda_i$). Then we have

$$E[A] = \min_{d \in D(A)} \left(\frac{1}{\sum_i \lambda_i} (1 + \sum_i (1 - p_i)\lambda_i E[A - \{i\}]) \right) \tag{9.3}$$

We have not been able to find a closed form schedule for this dynamic program formulation. Some special cases are very illuminating however. For all $p_i = 0$, the optimum scheduling, by Weber's result in Case 2 above, is to run the tasks with the largest $E[t_i]$ (i.e. at each point in time run the set of tasks with the largest $E[t_i]$). For all $p_i = 1$, the above expression simplifies, revealing that the optimal scheduling policy is to run the tasks with the *smallest* $E[t_i]$. Another case in which the optimal scheduling policy is obvious is when $\lambda_i = \lambda$ for all i. In this case the optimal policy is to run tasks with the largest p_i. Since all the $E[t_i]$ are equal, this optimal policy amounts to run the tasks with the smallest ratio $E[t_i]/p_i$, as we found for the one-processor case. However, ordering tasks by $E[t_i]/p_i$ is not optimal in general. Consider the following example.

Suppose we have two processors and three tasks with $E[t_1] = E[t_2] = 10$, $p_1 = p_2 = 0.5$, $E[t_3] = \alpha$, and $p_3 = p$. Then ordering tasks by $E[t_i]/p_i$ reduces to running tasks 1 and 2 first if $20 \leq \alpha/p$ and otherwise running task 3 (and task 1 or 2) first. However, if $16 < \alpha < 20$ and $p = 0.8$ (so that $20 < \alpha/p < 25$) the optimal scheduling (from Equation 9.3) is to run task 3 (and task 1 or 2) first (assuming integer rates). (When $\alpha = 20$ and $p = 0.8$, either running task 3 first or running tasks 1 and 2 first is optimal.) Interestingly, if all the tasks in this example have deterministic execution times, with the given means, rather than exponentially distributed execution times, the optimal scheduling is exactly the same.[2] Thus ordering by $E[t_i]/p_i$ is not optimal for deterministic tasks either.

With more than one processor, ordering tasks by $E[t_i]/p_i$ amounts to simply parallelizing the optimal scheduling policy for one processor (with integer rates and no preemption). However, as we have seen, we can obtain a better schedule by exploiting the additional degrees of freedom provided by additional processors.

We have not found any literature dealing with general por (i.e. $0 < p_i < 1$ and $P > 1$) scheduling, which appears to be very difficult. Scheduling in this case is the confluence of at least two phenomena:

1. shortest, most probable tasks first

   In this phenomenon, which is most common when the probability of a true result is large, optimal scheduling reduces to running the tasks first that minimize some metric favoring the shortest and most probable tasks. For example, with one processor and non-preemption, the metric is the expected task length divided by the task's probability of success.

[2] This is a coincidence; the optimal scheduling for exponential and deterministic tasks is not the same in general.

2. bin-packing

   In this phenomenon, which is most common when the probability of a true result is small, optimal scheduling reduces to ordering all the tasks to require the least total time. Of course, bin-packing becomes more difficult and less effective as the variability of task lengths increases. Weber's result in Case 2 above indicates that bin-packing for certain cases in which all $p_i = 0$ reduces to running the longest tasks first.

We saw the first phenomenon with one processor and with Case 3 above and the second phenomenon with the deterministic tasks in Case 1 above. The interplay between these different phenomena contribute to the difficulty with scheduling.

The difficulty of por scheduling suggests concentrating on special cases, bounds, and heuristics. One heuristic is to divide the scheduling into regimes roughly corresponding to the above phenomena and within each regime apply the solution consistent with its dominant phenomenon.

9.3.3 Preemption and rates

We need preemption in general. As we saw with one processor, preemption is sometimes necessary so we can switch to the most promising task. In addition to this phenomenon, with more than one processor we sometimes need preemption to ensure the best "bin-packing" of tasks, i.e. minimal processor idle time.

We also need fractional rates in general. We saw cases with both one and more than one processor in which multiplexing is sometimes necessary. Fractional rates are useful for minimizing idleness when competing with a deadline. With pbranch, branch tasks compete with the deadline established by the predicate task. With por, tasks compete with the deadline established by the completion time of other tasks. Such a competition may take the form of bin-packing, (as we saw with more than one processor) or multiplexing to return the first true value (as we saw with one processor). If the deadline is known, we do not actually need fractional rates; we can simply use integer rates and preemption as discussed in Section 9.2.3. However, if the deadline is nondeterministic, we do need fractional rates. As discussed in Section 9.2.3, we cannot simulate fractional rates with preemption, unless the preemption quantum is infinitesimal (in which case we have fractional rates).

9.3.4 Summary

We provided results for por/pand scheduling in special cases. General por/pand scheduling is a difficult problem. We demonstrated that preemption and fractional rates are necessary, in general, for optimal scheduling of por/pand.

9.4 Nested Computations

Our ultimate interest is the scheduling of nested computation, such as nested pifs, pands, and pors. The resource execution time PDF that we described in Section 9.1.2 provides a straightforward conceptual way to think about such problems by propagating the resource execution time PDFs for leaf activities up the activity tree as we explained in Section 9.1.2. Our examination of some "leaf activities" (simple pif and por/pand) suggests that this approach is, unfortunately, impossibly difficult except for special cases. In many (simple) cases we can solve the entire scheduling problem at the top level without following the resource execution time PDF approach. To determine the behavior of the optimal scheduling for nested computations, we consider four such special cases.

Case 1: Nested pors or pands

Provided that the "leaf" disjuncts (conjuncts) are all single tasks, we can treat this case by collapsing the nested pors (pands) into a single por (pand) with all disjuncts (conjuncts) single tasks. For example,

$$(\text{por } (\text{por } A\ B)\ C) \rightarrow (\text{por } A\ B\ C)$$

Then we can apply the results for single task por (pand) scheduling.

Case 2: Nested pifs

We consider two of the three possible cases for nesting pifs. First we consider

$$(\text{pif } A\ (\text{pif } B\ C\ D)\ E)$$

where A, B, C, D, and E are all single tasks. As in Section 9.2.1, A is in the critical path so clearly we must execute it at rate 1. Thus this problem has two major epochs, one before A completes, and one after A completes, as in Section 9.2.1. Once again, the scheduling in the second epoch is obvious.

If $P = 2$, we can treat the inner pif as a single task for the first epoch: it is never necessary to interleave E between B and C or between B and D. Thus the optimal scheduling policy is a simple composition of the optimal scheduling for each pif in isolation with single tasks.

However, this is not always the case. Suppose that all the tasks have deterministic execution time 1, $P = 3$, and $p_A = p_B = 0.6$ where p_i is the probability that task i returns a true value. Then if we compose the optimal scheduling for each pif in isolation, we have the following schedule: run tasks A, B, and C simultaneously followed by tasks D or E if necessary. (Since $p_A > 1/2$, we favor (pif B C D) over E and since $p_B > 1/2$, we favor C over D.) This schedule has an expected execution time of $p_A p_B + 2p_A(1 - p_B) + 2(1 - p_A) = 1.64$ However, a better schedule is to run tasks A, B, and E simultaneously followed by tasks C or D if necessary. This latter schedule has an expected execution time

of $2p_A + (1 - p_A) = 1.6$. Thus even though $p_A > 1/2$, it is better to run E rather than C or D of (pif B C D).

Now we consider

(pif (pif A B C) D E)

With the result of pif interpreted as a boolean value, we can no longer assume, as we did in our analysis in Section 9.2.1, that the consequent and alternate never return the same value. Consequently, with nested pifs there is the possibility that we may not need to evaluate the predicate of a pif. For example, in in the above expression, if task A has deterministic execution time 100, all the other tasks have deterministic execution time 1, and $p_B = p_C = 1/2$, the optimal schedule for $P = 2$ begins by evaluating tasks B and C simultaneously. If B and C yield the same value, the execution time is 2; otherwise it is 101. Thus the optimal scheduling of pif depends, in general, on the values returned by the consequent and alternate.

Composing the optimal scheduling for each pif in isolation leads to the following schedule for the above expression: Devote all processors to evaluating (pif A B C) and any remaining processors to evaluating D and E in order of their probability of being required. This schedule is is optimal, for example, if all tasks have deterministic execution time 1, $P = 4$, and $p_A = 0.5$, $p_B = 0.9$, $p_C = 0.9$. However, if we change the task probabilities to $p_A = 0.9$, $p_B = 0.9$, $p_C = 0.1$, a better schedule is to run tasks A, B, D, and E simultaneously, followed by C if necessary. Or, if $P = 2$ and $p_B = p_C = 1/2$, a better schedule is to run A and D (or E) on each processor, followed by either B or C (depending on the outcome of A) and E.

In summary, the optimal schedule for nested pifs depends on the type of nesting, the task parameters, and the number of processors available. In particular, the optimal schedule for pif is not necessarily the simple composition of the optimal schedule for each pif in isolation. Thus the optimal strategy is not *nestable* in general.

Case 3: Nested pifs and pors

The optimal scheduling strategy may or may not be nestable in this case as well. Consider the expression

(pif (por A B) C D)

Suppose $P = 2$ and A, B, C, and D are all single tasks with deterministic execution times of 1 for B, C, and D and 2 for A. Let A and B have probability 1/2 and p_B respectively of being true. Then if $p_B = 1/4$, the optimum strategy is to run all tasks at rate 1 in the following manner. Run A on one processor and B on the other processor. If B yields false, follow B by C and D in either order and if B yields true, follow B by D. This is a nestable policy: in isolation we would run the predicate of the pif first. However, if $p_B = 1/2$, the optimum strategy is run B followed by A on one processor and C followed by D on the other processor. This is not a nestable policy.

Case 4: Nested pands and pors

Once again the optimal scheduling strategy may or may not be nestable. Consider the expression

(pand (por A B) C)

Suppose $P = 2$ and tasks A, B, and C are all deterministic with length 1. If $p_A = p_B = 0.1$ and $p_C = 0.9$ the optimal strategy is to run A and B first on the two processors, at rate 1, followed by C (if necessary). This is a nestable policy. However, if $p_A = 0.9$, $p_B = 0.5$ and $p_C = 0.1$ the optimal strategy is to run A and C first on the two processors, at rate 1, followed by B (if necessary). This is not a nestable policy.

Thus, in general, the optimal scheduling of multiple-approach speculative computation depends on the context (e.g. pif vs. pand) as well as the parameters, such as the number of processors and the task PDFs and probabilities. In particular, the optimal scheduling policy for a pif or por in isolation is not necessarily optimal when the computation is embedded within other computation. One of the main areas for future work is to further investigate the scheduling of nested computations to determine the conditions under which nestable policies are optimal and the cost of employing them when they are sub-optimal. The results of such investigation will have significant impact on the support provided for modularity in multiple-approach speculative computation.

9.5 Scheduler Capabilities

The results and examples that we have presented in this chapter for optimal scheduling indicate that continuous decision scheduling, preemption, and fractional rates are important for scheduling speculative computation. Continuous decision scheduling is important so the scheduling can respond to changes in information, in the tasks scheduled and in the system — such as a change in the number of processors available for speculative computation. Preemption is essential to implement continuous decision scheduling decisions. Fractional rates are sometimes necessary when competing against an unknown deadline.

In terms of our sponsor model, continuous decision scheduling means that controller sponsors must be active agents monitoring the progress of computation. Priorities — the means for specifying preemption in the sponsor model — must be dynamic. We saw in this chapter that the optimal scheduling sometimes requires preemption at precise intervals. This requires some means of tracking the duration of computation in our sponsor model. We already mentioned that duration is a useful capability for controlling potentially infinite computation. Rates must also be dynamic. Finally, continuous decision scheduling introduces an new component into the scheduling problem. the cost of performing the scheduling itself. For our sponsor model, it is important that the controller sponsor policies be computationally cheap and thus simple.

An interesting question is whether we need (instantaneous) fractional rates in practice. We can always simulate instantaneous rates to a given degree of precision by round-robin preemption with a suitable quantum size. Thus, provided we are willing to pay a certain cost, we do not need instantaneous rates. The essential issue with rates revolves around three factors:

1. the benefits of rates,
2. the cost of preemptions to achieve rates, and
3. the ease of specifying the scheduling policy.

These first two factors are obvious. The last factor addresses the implementation of rates: it may be easier (and perhaps more cost effective too) to specify rates and let the scheduler manage the preemption schedule to achieve them, rather than forcing the user to build the preemption schedule into controller sponsor policies.

We saw in this chapter that nested scheduling is not optimal in general. This suggests that it is important for controllers to communicate and interact in scheduling, as described in Section 5.4.3. However, specification of such communication conflicts to some degree with the desire for modularity that we stated in Section 2.1. This conflict indicates there is a tradeoff between the performance of optimal scheduling and the complexity of optimal schedule specification and control.

Further work is necessary to investigate the cost/benefit tradeoff of the capabilities suggested by optimal scheduling.

9.6 Summary

In this chapter we gave a flavor of the optimal scheduling problem for multiple-approach speculative computation. That flavor is, for the most part, not very appetizing due to the intrinsic difficulty of the scheduling problems involved. For some simple cases we were able to make headway. We derived new results for the optimal scheduling of pbranch (the n-ary generalization of `pif`) on P processors. For por/pand we examined a few special cases and summarized the results by others for a few special cases. All these results are for simple cases with independent tasks and no descendants or nesting of computation. Future work must address the complications of dependent tasks, descendant tasks, and sharing of computation. We gave several examples illustrating that the optimal scheduling for isolated computation is not necessarily optimal when the computation is nested within other computation. Finally, we extrapolated from our results and discussed the support required for optimal scheduling.

Although difficult, the study of scheduling problems is very important to the foundation of speculative computation, both theoretically, and practically so we know (1) the ideal scheduling policies for these problems and what support these policies require, and (2) the

optimal cost of scheduling problems. With the optimal cost we can assess the merit of other scheduling policies and determine the ultimate benefit of speculative computation. The difficulty of the general scheduling problems suggests progress by examining special cases, bounds, and heuristics.

Chapter 10

Implementation Details

This chapter describes the implementation details of our touching model. As stated in Sections 1.5, our goals in this implementation were firstly, to minimize the changes to the existing Multilisp language definition and, in particular, retain future, and secondly, to minimize the impact of our support for speculative computation on the performance of conventional, mandatory computation.

Our implementation is an extension of the original Multilisp implementation by Halstead and Loaiza described in [Hals85]. Multilisp is compiled to a virtual stack machine language called MCODE, which is implemented by a byte-code interpreter written in C. A copy of this interpreter executes on each processor of the underlying multiprocessor — the Concert Multiprocessor or the Encore Multimax for this work, as described in the beginning of Chapter 8. Since Multilisp is interpreted, it is slow in absolute terms.

In adding our support for speculative computation we retained most of this original implementation, only changing the implementation to be consistent with our additions. This achieved our first goal.

To achieve our second goal, we distinguished mandatory tasks — the tasks in the original implementation — and speculative tasks — the tasks we added for speculative computation — as described in Section 6.2.

We discuss the four main additions to the original implementation, present some performance figures, and close with some thoughts on how to improve the performance of the implementation.

10.1 Speculative Tasks

Speculative tasks are created as described in Section 6.2. We represent speculative tasks in the implementation by speculative task objects which complement the mandatory task objects in the original implementation. In addition to the same six fields as a mandatory

task object (documented in [Hals86b]), a speculative task object has the following fields:

1. a source priority field,
2. an effective priority field,
3. four other attribute fields, which we describe later in Section 10.4,
4. four fields, which we describe in Section 10.2, for task state, position, location, and polling, and
5. a field for miscellaneous purposes which we won't describe.

These 17 fields — each a lisp object — make a speculative task object almost three times larger than a mandatory task object.

The source priority is initialized to the source priority argument specified by `spec-future` or `make-group`, or inherited from the effective priority of the parent task as described later. Subsequently the source priority may be changed by `change-priority` (see Appendix A) and staying (see Section 10.4). The effective priority is the maximum of the source priority and effective priority of the maximum priority task blocked on the task's future object. Tasks are scheduled for execution according to this effective priority, as described in Section 10.2. All priorities in this implementation are integers in the range $[0, MAND]$ where $MAND$ is $2^{22} - 1$, the largest integer representable as immediate data in a Multilisp object.

We distinguish mandatory and speculative tasks by their effective priorities. Speculative tasks, which are always represented by speculative task objects, have effective priority in the subrange $[0, MAX]$, where $MAX = MAND - 1$. (Source priorities are also restricted to this range.) Mandatory tasks have effective priority $MAND$. (The scheduling algorithm treats MAX and $MAND$ as equivalent though — see Section 10.2.) There are two types of mandatory tasks: "real" mandatory tasks created by a mandatory task and represented by a mandatory task object (which has implicit effective priority $MAND$), and quasi-mandatory tasks represented by a speculative task object. Quasi-mandatory tasks begin life as speculative tasks and subsequently become mandatory when touched by a mandatory t or promoted to mandatory status (via `promote-task` or `replace-eq-mand`). The following table should make these distinctions clear.

Task type	Effective Priority	Representation
(real) mandatory	$MAND$	mandatory task object
(quasi) mandatory	$MAND$	speculative task object
speculative	$\in [0, MAX]$	speculative task object

Task creation works as follows. When a mandatory task[1] executes `future`, `dfuture`, or `delay`, the child task is a mandatory task. When a processor running a mandatory task

[1]For now on, by mandatory task we mean either a real or a quasi-mandatory task unless stated otherwise.

executes future, the processor continues the parent task and spawns the child task, unless there are no other processors available, in which case the processor continues the child task and queues the parent task. (See Section 10.2 for details.) This slight variation from the original implementation provides cheap task creation (maintaining locality of reference) while yielding unfair scheduling when required, i.e. when the machine saturates. When a processor running a mandatory task executes dfuture, the processor always continues the parent task and spawns the child task, as in the original implementation.

When any task executes spec-future or make-group, the child task is a speculative task with initial source priority given by the source priority argument. When a speculative task executes future or dfuture, the child task is a speculative task with initial source priority given by the effective priority of the parent task (at that time). When a task creates a speculative task, the processor executing the task continues the parent task and spawns the child task unless the child has a greater effective priority, in which case the processor continues the child and spawns the parent. Finally, when a speculative task executes delay, the child task is a speculative task with source priority 0 so the task does not run until touched. The following table summarizes these task creation details.

Parent task	Parent executes	Child task	Source priority
mandatory	(future E), (dfuture E), or (delay E)	mandatory	N/A
	(spec-future E s) or (make-group E s)	speculative	s
speculative, with effective priority p	(future E) or (dfuture E)	speculative	p
	(delay E)	speculative	0
	(spec-future E s) or (make-group E s)	speculative	s

In our implementation we insist that a future object have not more than one associated speculative task object. This amounts to banning the execution of call/cc by speculative and quasi-mandatory tasks. In the original implementation (as in ours for real mandatory tasks) call/cc creates a copy of the task object. Consequently, multiple tasks — active simultaneously — can have the same future object. Limiting the number of speculative tasks per future simplifies the implementation tremendously: priority propagation involves only task chains, not trees (except for class sponsors) and futures name unique tasks. This latter consequence is useful for adding and removing tasks to and from classes. It is not at all clear what call/cc should mean in a parallel environment anyway (especially in the presence of side-effects). [Katz] presents one point of view on this controversy.

10.2 Preemptive Scheduling

The scheduling algorithm divides tasks into two types: non-preemptable and preemptable. Speculative tasks with priority[2] MAX and mandatory tasks comprise the first type and speculative tasks with priority in the interval $[1, MAX - 1]$ comprise the second type. Tasks with priority 0 are never scheduled for execution.

[2]By task priority in this section we mean the task's effective priority.

The scheduling algorithm treats priorities *MAX* and *MAND* as equivalent. This allows preemptability and stayability to be orthogonal within one simple priority propagation algorithm: tasks of priority $\geq$ *MAX* are non-preemptable, tasks of priority $\leq$ *MAX* are stayable, and the max combining-rule remains applicable for all priorities. For instance, if a non-preemptable, non-stayable (i.e. mandatory) task touches a non-preemptable, stayable task, the touchee automatically becomes non-stayable by the max combining-rule. Likewise, if a non-preemptable, stayable task touches a preemptable, stayable task, the touchee automatically becomes a non-preemptable, stayable task. The following table illustrates this orthogonality.

	Non-stayable	Stayable
Non-preemptable	mandatory task (priority *MAND*)	speculative task priority *MAX*
Preemptable	N/A	speculative task priority $\in [1, MAX-1]$

We want this orthogonality so we can run a speculative task at the same "priority" as a mandatory task, so it is non-preemptable, and still be able to stay the speculative task. For instance, if a mandatory task demands a por we want (one or more of) the disjunct tasks to run at top priority to obey demand transitivity but we still want to be able to stay the task(s) should some other disjunct return true first. (We have to be careful in such cases to sponsor the disjunct task with priority *MAX*, as discussed later, and not *MAND*.)

The scheduling algorithm groups processors into clusters (which map to *slices* on Concert [Hals86a]). Each cluster has the following data structures for scheduling:

1. a *mandq* — a LIFO queue for pending non-preemptable tasks

   Pending mandatory tasks are LIFO queued, as in the original implementation.

2. a *specq* — a priority queue for pending preemptable tasks

   Pending speculative tasks with the same priority are FIFO queued.

3. a *procq* — a priority queue of preemptable tasks running in that cluster

4. an array of *preempt slots*, one for each processor in the cluster.

A free processor, i.e. a processor not currently executing any task, continually polls, in the following order, its preempt slot, its mandq, and its specq, and the mandq of other clusters until it finds a task. If the task found is a preemptable task, the processor is preempted[3] and the task is entered in the priority queue of preemptable running tasks for that cluster.

A processor executing a preemptable task periodically checks its preempt slot and its specq. If the processor finds a task in its preempt slot, it injects the currently executing task

[3]Imagine a free processor as running a "make work" task with priority 0.

into the system (as described shortly) and executes the new task instead. If the priority of top priority task in the specq exceeds the priority of the currently executing task, the two tasks are swapped.

In addition, any processor executing a speculative task (any stayable task, preemptable or not) checks the task's "polling" field every "quantum" of instructions executed and just before spawning a child task. This polling field is used to signal a change in a running task's priority. If it indicates a change in priority, the task is scheduled according to its new priority. If the new priority is *MAND* or *MAX*, the processor injects the task into the system as a non-preemptable task (unless the old priority was *MAX*) and the processor becomes free. If the new priority is 0, the task is descheduled, i.e. stayed, and the processor becomes free. Otherwise, the new priority is compared with the priority of the top priority task in the specq and the tasks swapped if necessary. This polling is the run-time cost of priorities.

When a processor spawns a task or otherwise injects a (non-stayed) task into the system, the task is scheduled as follows:

- If the task is non-preemptable:
  1. If a free processor exists within the cluster, it is preempted with the task, that is, the task is inserted in that processor's preempt slot.
  2. If there are no free processors within the cluster and at least one processor is executing a preemptable task, the lowest priority preemptable task, according to the running task priority queue, is preempted by the new task.
  3. If all the processors within the cluster are busy with non-preemptable tasks, the other clusters are searched for one with a free processor. If one such processor is found, it is preempted with the new task. If none is found, the other clusters are again searched, this time for a free processor or any preemptable running task. The first such processor found is preempted with the new task. If still no suitable processor is found, there are two cases. If the new task is a mandatory task just created with `future`, the parent task is inserted in the local cluster's mandq and the processor continues the new task. Otherwise, the new task is inserted in the local cluster's mandq.
- If the task is preemptable:
  1. If the local cluster has a free processor, it is preempted with the new task.
  2. If the priority of the new task is greater than the lowest priority preemptable task running in the cluster, then that task is preempted by the new task.
  3. Otherwise, the other clusters are searched for a free processor to preempt. If none is found, the new task is inserted in the local cluster's specq.

Each cluster maintains a state variable cluster to enhance the efficiency of this scheduling algorithm. This variable allows a processor to deduce the state of another cluster with a single read. Cluster states are:

1. at least one free processor,

2. all processors busy, but at least one running preemptable task, and

3. all processors busy with non-preemptable tasks.

The mandq is a LIFO list consed out of heap, just as in the original implementation.

The specq is a straightforward heap queue (see, e.g., the section on heapsorts in [Knuth]) implemented with an array. The array size is adjusted dynamically to be between one and two times the number of tasks in the specq. When the number of tasks in the specq exceeds the array size, the specq is copied to an array of double the size. The garbage collector reclaims the array size by similar copying whenever the number of tasks in the specq is less than half the array size (except for a certain minimum array size).

The distributed specqs collectively approximate a global priority queue for preemptable tasks by periodically "balancing" the specqs. Whenever a cluster is in state 2 (all processors busy and at least one preemptable task running), it periodically compares the priority of the top two tasks in its specq with the priority of the top two in the specq of its neighbor and transfers tasks, if necessary, until no imbalance remains. Such balancing is not necessary in cluster state 1 since the scheduling algorithm does it implicitly while the cluster searches for a task. We avoid balancing in cluster state 3 to minimize the overhead on mandatory tasks. This can lead t imbalances but the scheduling algorithm minimizes these imbalances so long as creation and termination of preemptable tasks continues.

The procq is implemented as a small array, with size equal to the number of processors in a cluster, sorted by linear search.

To support the scheduling algorithm, each speculative task has the following three fields:

1. a state field, indicating the state of the task.

   A speculative task is in one of six states: running — the task is executing; runnable — the task is queued awaiting a free processor; blocked — the task is queued on an undetermined future object; stayed — the effective priority of the task is 0; staying — the task is staying other tasks; and terminated. The state and the following two fields give all the information necessary to reposition the task should its priority change.

2. a location field, describing the location of the task if it is runnable or blocked

   If the task is runnable, the location identifies the specq in case the task must be repositioned or removed from the specq. If the task is blocked, the location identifies the future on which it is blocked so the maxwaiter (see Section 10.3) can be updated if the task priority changes. If the task is running, the scheduling algorithm handles any change in task priority. Finally, any change in priority of a stayed task is handled by the task (i.e. toucher) which caused it to change. In fact, the scheduling algorithm maintains no pointers to stayed tasks, so that they may be garbage-collected when otherwise inaccessible.

3. a position field, indicating the position of the task if it is runnable or blocked

   If the task is runnable or blocked, the position identifies the task in the specq or toucher queue so the task may be repositioned or removed if its priority changes.

10.3 Priority Propagation

For priority propagation we added two additional fields to all future objects: a specq field and a maxwaiter field. The specq field is a queue for speculative task touchers, which complements the queue for mandatory task touchers retained from the original implementation. We distinguish the two queues for efficiency reasons, i.e. to avoid imposing the speculative task management overhead on mandatory tasks. The maxwaiter field contains the priority of the maximum-priority task touching the future. Thus maxwaiter is *MAND* if the mandatory task queue is non-empty.

The specq field is a list of speculative tasks headed by the maximum-priority speculative toucher. It is not a priority queue, to avoid excessive overhead. When a speculative task touches a future, it compares its priority with the future's maxwaiter field. If the task priority is less than or equal to the the maxwaiter, the task is simply linked in behind the maximum priority toucher task. If the task priority is greater than the maxwaiter, the maxwaiter field is updated and the task is consed on the head of the toucher list. The hitch comes if the priority of a toucher task changes. If the task is not the maximum priority toucher, then the two cases are straightforward: If the priority exceeds the maxwaiter, then the maxwaiter and maximum priority task are updated, otherwise nothing happens. If the task is the maximum priority toucher, then any increase in priority simply increases maxwaiter, but any decrease requires a linear search of the remaining touchers for possibly a new maximum priority waiter. Only in this last case is a priority queue cheaper — a priority queue is far more expensive in the previous three cases. Furthermore, this last case should be rather rare.

To aid in any required repositioning of tasks in this specq list, the position field of each task points to the previous cons cell in the list (or the future object for the maximum priority toucher task).

When a mandatory task touches a future, the task updates the maxwaiter field to *MAND*, if it is not *MAND* already, and adds itself to the head of the mandq list of touchers, as in the original implementation.

As mentioned in Section 6.1.1 we employ eager priority propagation, which works as follows. If a task's source priority changes (due to explicit change with `change-priority` or staying — see Section 10.4) or its maxwaiter changes (due to touching or a priority change in a toucher task), the task's effective priority is updated, if necessary, to the maximum of the source priority and maxwaiter priority, unless the effective priority is already *MAND*. To promote a speculative task to mandatory status we set its effective priority to *MAND* and to demote such a task we set its effective priority back to the maximum of its source and maxwaiter priorities (and inject the task into the scheduler). The other effective attributes

of the task (described in Section 10.4) are updated to the attributes of whatever task supplies the effective priority. If the effective priority changes, the subsequent action depends on the task state as follows:

1. if state is running or staying, set the task's polling field to indicate a change in priority
2. if state is runnable, adjust the position of the task in its specq to reflect its new priority, unless this new priority is 0, in which case remove the task from the specq and set its state to stayed
3. if state is stayed, inject the task in system, adjusting its state to running or runnable as appropriate
4. if state is blocked, perform the actions described previously, repositioning the task in the touchee's specq and if necessary, updating the maxwaiter priority and repeating this cycle until the end of the *touch* chain.

We implemented class sponsors using this same touching mechanism. To sponsor a task within a class we touch the task with a "fake" speculative task object with the effective priority of the class and to unsponsor such a task we "untouch" the task, removing the fake toucher task. (Only speculative and quasi-mandatory tasks can be members of a class; there is no need for real mandatory tasks to be members of a class since they cannot be stayed or preempted.) We represent a class sponsor as a task-like object with the class effective priority, the class effective attributes, a list of class members, and a list of fake task objects touching sponsored tasks. See Figure 10.1. The class effective priority never exceeds *MAX* to avoid inadvertently turning class-sponsored tasks into mandatory tasks.

Priority propagation deals with such a class sponsor object exactly the same as a speculative task, except the effective priority and effective attributes are copied to each of the fake task objects in the class sponsor list and then propagated, as before, down the touch chain headed by each fake task. Type `all` class sponsors can sponsor multiple tasks (as shown in Figure 10.1) and thus allow a fork in the touch chain. Our eager priority propagation follows the branches in such a fork one at a time in a depth-first fashion, keeping the class sponsor locked until all branches are traversed. This can be grossly inefficient. Type `pqueue` class sponsors maintain a priority queue (implemented the same as the specq in the scheduling algorithm) of class members, instead of a list, and propagate effective priority and effective attributes to the top-priority member of the class.

10.4 Staying

Only speculative tasks can be stayed, mandatory tasks, even quasi-mandatory tasks, cannot be stayed. This is a consequence of our two goals of minimal changes and minimal performance impact on conventional, mandatory computation. Thus the user must perform explicit checking to reclaim irrelevant mandatory tasks. In our view, any user generating

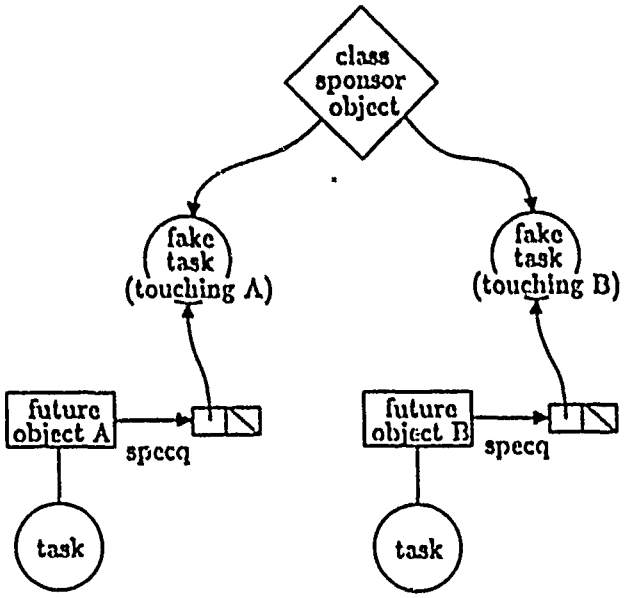


Figure 10.1: A class sponsor and sponsored tasks

irrelevant mandatory tasks is mis-using the system: mandatory tasks should be mandatory in the sense defined in Section 1.1.

Staying has two requirements:

1. we must be able to prevent run-away task phenomena, and
2. we must be able to stay all the descendants of a group. (We also have a `stay-task` construct — see Appendix A — which performs the staying operation on a task and all its descendants.)

We meet the first requirement by suspending the creation of any new tasks in a group while any staying is in progress in the group. We meet the second requirement by maintaining a data structure we call the tasknode tree that lists all the descendants of each task.

The tasknode tree consists of subtrees rooted by group objects, as depicted in Figure 10.2, with one group object per group. Each group object contains a variable called *staycount* which indicates the number of stayers active within that group. Whenever a speculative task or group is created, the staycount in the parent task's group (or root group if the parent is a mandatory task) is checked. If the staycount is zero, a new tasknode, or group object as appropriate, is created, linked into the tasknode tree, and the parent returns. Otherwise, the parent task backs out of the task or group creation, and becomes a "stay helper", assisting any stayers in the parent group. When the staycount in this group returns to zero, the stay helper checks the status of the original "backed out" task and resumes it

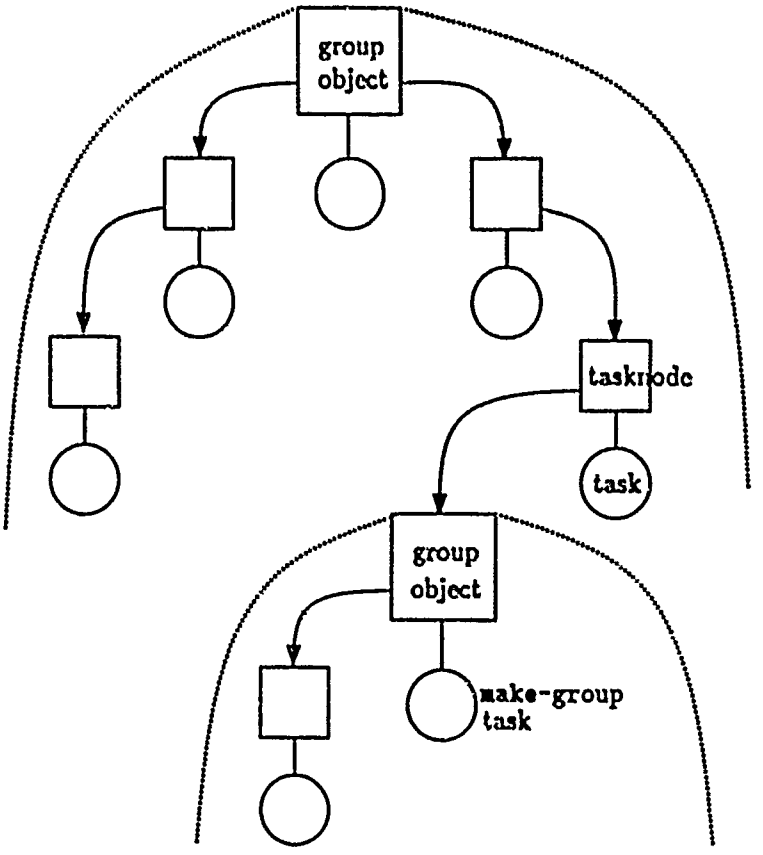


Figure .0.2: Example tasknode tree (backpointers not shown)

if it has not been stayed. Thus the staycount is the means by which we prevent new tasks being spawned in a group while we are staying in the group.

The staying algorithm performed by each stayer proceeds in three phases. The algorithm dove-tails nicely with groups to localize the impact of staying to a group and its descendants. The *source group* mentioned in the following is the group specified by stay-group (or the group to which the task specified by stay-task belongs).

Phase 1: the marking phase

In Phase 1 the staycount is incremented in the source group and each descendant group (i.e. all groups in the node tree rooted by the source group). Once a group's staycount is nonzero no new tasks or groups may be added to that group until staying is completed. Thus there can be no run-away task phenomenon within a marked group. However, there is still a potential run-away with groups: there is a race between marking groups and creating new groups.

This potential run-away is mitigated by the relative rate of marking and group creation. Marking is much faster than group creation. Marking involves locking the current group, incrementing the staycount in the group, unlocking the group, and cdring down the list of child groups, repeating the process recursively for each child group. The marking process cannot be interrupted, so this cycle can only be delayed by the time required to lock the group. Group creation involves creating and filling in a new group object, a new future object for the group, and a new task object for the future object. The parent group object must then be locked, the staycount checked for zero, the new group linked into the node tree, and the parent group unlocked.

Because of the large difference in propagation rates, the marking will catch up to run-away group creation eventually.[4] Thus we view group run-away as a solved problem.

Phase 2: the staying phase

In Phase 2 all the descendants of the source group (or task specified by stay-task) are traversed and the source priority of each descendant is set to 0. This change, if any, in the source priority is propagated as described in Section 10.3.[5] If the effective priority of the task is then zero, the task is *stayed*: it is ineligible to run and will be descheduled if it is running.

---

[4] Actually, the time to complete Phase 1 is bounded because of the consing associated with group creation. A garbage collection flip is prevented while any staying is in progress, so group creation must eventually cease. Once this happens, the time to lock a group is bounded (by a constant proportional to the number of processors) since (1) only a finite number of stayers compete for the group locks (all other potential group locking activity is halted pending the gc flip), (2) each stayer holds a group lock for a bounded time, and (3) each stayer makes forward progress. Thus the time to complete Phase 1 is bounded by a (large) constant proportional to the heap size and the number of processors.

[5] Thus a task temporarily promoted to mandatory status is not stayed until it is demoted.

Phase 3: the cleaning phase

Finally, in Phase 3 the staycount is decremented in the source group and in each descendant group, thus undoing Phase 1. If the staycount in a group returns to zero, any pending task creations may proceed (assuming their parent task was not stayed).

In the present implementation, an individual stayer performs the staying algorithm sequentially, although there can be more than one stayer active in a group simultaneously. Furthermore, stay helpers merely spin wait until the stayer completes. Both stayers and stay helpers are obvious sources of parallelism in the staying algorithm. All stayers and stay helpers have a task state of "staying."

As discussed in Section 6.3, we solved the intergroup touching problem in our implementation by assigning each task an effective "owner" which is responsible for staying the task Each speculative task has the following four attribute fields for implementing this effective ownership.

1. the group object of the task,
2. the tasknode object of the task,
3. the effective group, and
4. the effective tasknode.

When a task is created, the effective group and effective tasknode are initialized to the effective group and effective tasknode, respectively, of the parent task. (The effective group and effective tasknode of a mandatory task are the root group object and the effective tasknode of a make-group task is that task's group object.) Once a task's maxwaiter priority first exceeds its source priority, the effective group and effective tasknode are thereafter either (1) the group object and tasknode object, respectively, if the source priority contributes the effective priority, or (2) the effective group and effective tasknode, respectively, of the toucher task contributing the effective priority. Priority propagation updates the effective group and effective tasknode, as stated earlier. The effective tasknode is the "owner" of a task; the effective group merely gives the group of the owner. Thus the owner of a task's parent serves as a task's effective owner until some toucher task wrests ownership away.[6]

Whenever a task spawns another task and the parent task's group and effective group differ, the child task is added as an effective child to the parent task's effective tasknode. Then whenever the staying algorithm stays a task or group, the algorithm stays all the indirect descendants – the effective children — which have that effective group, as well as all

[6]Note that ownership can then revert back to the task's tasknode if the source priority subsequently exceeds the maxwaiter priority. Excepting changes in source priority via change-priority, this is not a problem since the objective is to solve the "unstayable" descendant problem, which occurs when the source priority is 0 due to staying the task's group and the task is touched by a task in another group.

the direct descendants of that task or group. This prevents any "unstayable" descendants, as described in Section 6.3. To support this change in the staying algorithm, task creation must check the staycount in both a task's group and effective group before spawning a task.

We maintain the tasknode tree with weak pointers. Weak pointers have the property that an object accessible only via weak pointers is garbage-collected at the next garbage collection flip.[7] See [Miller] for a good description of weak pointers and their implementation. Tasknodes and group objects are linked with weak pointers. Strong backpointers from task objects and descendant tasknodes and group objects force these tasknodes and group objects to be retained until they are no longer necessary.[8] We omit the complicated details. Note, though, that the only pointers that the system (as opposed to the user) maintains to stayed tasks are weak pointers so that such tasks may be garbage-collected.

10.5 Performance

Thorough investigation of the performance of our implementation is a project for future work. We provide three different performance measurements here. The following table compares the time to spawn mandatory and speculative tasks on the Encore Multimax. (We would have rather presented the times for spawning these tasks on Concert, but the failure of Concert, as noted in the beginning of Chapter 8, prevented the necessary data collection.)

Input	Execution time
(touch (future nil)) (executed by a mandatory task)	1.5-2.2 msec.
(touch (spec-future nil 100))	1.8-2.5 msec.
(touch (group-future (make-group nil 100)))	2.1-2.9 msec.
(touch (future nil)) (executed by a speculative task)	1.8-2.5 msec.

These figures give some idea of the spawn time cost of our speculative computation support. The times vary according to the steps of the scheduling algorithm executed due to machine load (see Section 10.2). For instance, (touch (future nil)) takes 1.5 msec if the machine is unsaturated and 2.2 msec if the machine is saturated, due to all the searching for a free or preemptable processor. In comparison, (touch (future nil)) takes 1.3 msec in the original Multilisp implementation (independent of loading), which is almost the same as in our implementation (when executed by a mandatory task) with the machine unsaturated.[9]

[7] Multilisp uses a parallel version of Baker's [Baker78b] incremental, copying garbage collector which we extended for weak pointers.

[8] A task, for example, has strong backpointers to its group and tasknode objects (making double use of these attribute fields).

[9] The time for this operation in the original implementation is independent of loading because it includes only the time to create and queue a task. The time in our implementation also includes a substantial portion of the time to match a task with a processor. This is the main reason for this 0.7 msec variation in execution time in our implementation.

This indicates that we largely succeeded in our goal of minimizing the impact of speculative computation support on the performance of mandatory tasks.

To get some idea of the run-time cost of our speculative computation support, we compared the running time of mandatory and speculative tasks executing the same sequential program on the Concert Multiprocessor. We found that polling imposes about a 5% slow-down and speeq balancing imposes about a further 5% slow-down in speculative task execution rate. This polling overhead does not occur in speculative tasks with priority *MAX* since these tasks do not need to check for preemption, only a change in priority. Consequently, such tasks run at about the same rate as mandatory tasks. This speeq balancing overhead only occurs in cluster state 2 (see Section 10.2).

We performed a preliminary performance investigation and found that speeq overhead is a substantial part of the spawn and run time overheads. On the Concert Multiprocessor, the time to insert and delete a task in a speeq with more than 3 tasks is about .4 and .8 msec respectively. (The corresponding time to evaluate (touch (future nil)) on Concert is 2.7 msec in the original implementation.)

10.6 Optimizations

The following is a list, in order of expected benefit, of what we consider the major optimizations to improve the performance of our present implementation. (Further performance analysis of the implementation may suggest different optimizations or change the expected benefit of these optimizations.)

1. priority queue improvements — Optimize the current priority queue implementation and perhaps try the parallel priority queue algorithm described in [Rao].
2. task node elimination — Eliminate the current indirection through tasknodes (we did not discuss this). This promises to make task creation and staying much faster.
3. parallel staying — Parallelize staying as described earlier.
4. parallel or incremental priority propagation — Make priority propagation along touch trees, such as resulting from type all classes, more efficient. Incremental propagation could also reduce useless updates and lessen the problem with cycles described in Section 6.1.1 but at the cost of greater response time in priority changes, as described in Section 6.4.4.

10.7 Summary

We retained the original Multilisp implementation and added four major additions to support speculative computation. speculative tasks, preemptive scheduling, priority propagation, and staying. We largely met our goal of limiting the overhead of this support to

speculative tasks. The performance of this speculative computation support needs improvement, particularly in the area of priority queue manipulation.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

The major contributions of this thesis are:

1. An elucidation of the issues involved and the requirements for supporting speculative computation in Multilisp.

   These issues and requirements are applicable to other computational paradigms as well.

2. A sponsor model for speculative computation which addresses these requirements.

   This model handles control and reclamation of computation in a single, elegant framework. This model can also handle side-effects[1], illustrating the power of this model.

3. Experimental evidence of the benefit of speculative computation and the benefit of our model and support for speculative computation.

   Our support for speculative computation adds expressive and computational power.

Minor contributions of this thesis are: a definition of speculative computation; a distinction of the different types of speculative computation; and a discussion of the optimal scheduling of speculative computation with a new result for `pif`.

We discuss each of the major contributions.

Issues and Requirements

Speculative styles of computation exploit excess resources in an attempt to reduce the critical path length and, hence, the execution time of applications. The success of this

[1] Actually, side-effect synchronization as discussed in Chapter 7.

approach depends on having excess resources (at least some of the time) and exploiting these excess resources *efficiently*. The key issues for efficient speculative computation are:

1. reclaiming computation, i.e. reclaiming the resources assigned to unnecessary computation, and
2. controlling computation, i.e. controlling the allocation of resources to computation.

The importance of reclaiming computation is obvious. All research concerned with speculative styles of computation (that we know about) addresses this issue in one way or another. We argued that this issue leads to the following requirement:

1. Explicit computation reclamation at the system level

   Implicit reclamation, i.e. garbage-collection of computation, is too inefficient.

With explicit reclamation we have the additional requirements:

2. Automatic naming of descendant tasks[2]

   Automatic naming of descendants solves the problem with unknown function calls and reduces the burden of managing descendant tasks for reclamation.

3. Reversible reclamation

   The possibility of shared computation (possible even without side-effects due to lazy thunks) and users mistakenly declaring computation irrelevant means that "reclaimed" computation may be relevant and thus need to be "unreclaimed."

Our position on explicit reclamation and its logical consequences — automatic naming of descendants and reversible reclamation — sets our work apart from other work.

Controlling computation is also very important, as our examples, particularly Boyer, demonstrate. However, this issue has received little attention by others. The major component of this issue is determining what control we want. This involves examining applications and optimal scheduling to determine the control policies and support required. However, even without knowing exactly what control we want in all cases, we argued that controlling computation has the following requirements. We need:

1. Ordering

   We need some way to allocate resources to computation in accordance with the relative promise of computations, i.e. some way to order computation. This is the most important control for speculative computation.

[2]Or whatever the paradigm calls the smallest controllable units of computation.

2. Demand transitivity

   Ordering must obey demand transitivity. If we use priorities to express the desired ordering, we want the priority of a demandee to be at least that of the demander.

3. Modularity

   We should be able to embed a group of speculative computations as a subcomputation in any other speculative computation and retain the local ordering within the group.

4. Other control

   Some desired control does not fit in the other categories of requirements. For example, sometimes we want complex, dynamic control capable of responding to changes in demand for speculative computation and changes in the allocation of resources.

A third issue in a computational paradigm like Multilisp is side-effects. Side-effects complicate computation reclamation since a computation may be relevant for the potential synchronization represented by its side-effects. This difficulty cements the requirement of reversible reclamation. Side-effects so complicate relevancy analysis — through the sharing of computation in non-obvious and insidious ways, for example — that it is easier to allow reclamation mistakes, undoing them when necessary, then attempting to avoid them. Also, reversible reclamation is useful as a safety net even without side-effects since it can be difficult to foresee all the interactions of computation, especially in a large system.

The Sponsor Model

To address the above requirements, we introduced a simple model based on the notion of sponsors. Sponsors supply tasks with attributes which control resource allocation. This sponsor model provides computation control with ordering controlled by external and controller sponsors, demand transitivity with the max combining-rule, modularity with groups, and more complex, dynamic control with controller sponsors. The model is a confluence of eager and demand-driven scheduling. This confluence allows the sponsor model to unify computation control and reclamation in one simple framework. Reclamation is explicit and reversible – reclamation occurs by simply unsponsoring computation. (Automatic naming of descendants is a requirement left to the implementation.) We demonstrated that we can also handle side-effects within the framework of the sponsor model. we simply have to ensure that (potentially) relevant computation receives sponsorship. Finally, the demand-driven aspect of the model provides an important safety net, as we argued above.

This sponsor model should furnish an archetype for speculative computation in other parallel language paradigms.

Experimental Evidence

The touching model we implemented, a subset of the sponsor model, was intended as a simple prototype and consequently suffers from a number of deficiencies. The most se-

rious deficiency is lack of modularity. Nevertheless, we were able to successfully exploit speculative computation with this implementation for several applications.

The applications we considered demonstrate the following:

1. the importance of aborting useless computation

   Tree equal and Emycin demonstrate the importance of aborting useless computation.

2. the importance of ordering computation

   Boyer, travsales, and Eight-puzzle all demonstrate the importance of ordering computation. In the case of Boyer, ordering sped execution by about a factor of about 2 (with 16 processors) and in the case of Eight-puzzle, ordering sped execution by a factor of 26 (with 8 processors) over a naive approach.

3. the expressive and computational power added by our support for speculative computation

   The expressive power comes from the ability to control computation — with priorities for ordering, explicit computation reclamation (staying), and controller sponsors — and the interaction of computation — with the max combining-rule. With our support for speculative computation we can realize a small gain with Emycin where other approaches either realize no gain, as with implicit reclamation, or fail, as with a naive approach like explicit checking. Furthermore, our support makes it easier to manage machine resources, and thus makes programming easier. The computational power comes from the significant improvement in performance — a factor of 2 with Boyer and a factor of 26 with Eight-puzzle.

We have successfully met our goals: we have provided a model and implementation support for speculative computation in Multilisp with which we demonstrated the benefit of speculative computation. Much work remains, however. In the short term we need to extend and improve the implementation, adding full controller sponsors and modularity. We present our thoughts for longer term work in Section 11.2.

11.1.1 Problems and limitations

In this section we discuss some negative aspects of our approach.

The Sponsor Model

The sponsor model increases the burden on the programmer. In addition to ensuring functional correctness, the programmer must now ensure correct sponsorship. In most cases suboptimal sponsorship or lack of sponsorship just results in performance inefficiency since computation is ultimately sponsored when it is demanded by touching - this is the

safety net resulting from the demand-driven aspect of the sponsor model. This tolerance is fortunate since it is often difficult to decide how to control computation.

However, correct sponsorship is critical in any situation in which implicit sponsorship, by touching, does not follow demand. Two examples of such situations are multiple-approach speculative computation, like naive por (e.g. version 1 of speculative por in Figure 8.2), and side-effects. Demanding the result of multiple-approach speculative computation, e.g. demanding the result of naive por, does not ensure that each approach or disjunct is sponsored. Likewise, requiring a side-effect, such as releasing a semaphore, does not ensure that the task(s) responsible for performing the side-effect is (are) sponsored. In both cases, the demand lies outside the implicit sponsorship channels and in both cases, lack of sponsorship can lead to deadlock. Consistently, the programmer must explicitly ensure sponsorship by, for example, using cl . We could make sponsorship implicit in the case of multiple-approach speculative computation by a "branching touch" — a touch which propagates down each branch of a special fork node representing multiple approaches. This would be just like our type `all` class. However, no one sponsorship distribution policy is optimal in such cases: should it be a type `all` class or a type pqueue class? Thus we prefer to leave such sponsorship explicit, to be determined as a component of the desired computation control.

Computation control and side-effect sponsorship are inherently imperative. The computational power that we gained with our support for speculative computation came from the ability to express the detailed control — ordering and aborting — that we wanted. Such expression is imperative and tricky to get correct but adds power not previously available in Multilisp. The question is: how much of this imperativeness does the user have to see? How much of this imperativeness can be hidden through appropriate interfaces (macros and user libraries) to make the model declarative at the user level? For example, we argued above that our support for speculative computation added expressive power since it moved some of the low-level resource management to the implementation and freed the programmer from these burdens. These questions are fodder for future work.

User Interface

The previous subsection highlights that our support for speculative computation is really an "assembly language" for speculative computation. This was a conscious design decision — we wanted to provide a flexible, low-level interface that others could build upon. For ordinary users we need to develop higher-level interfaces defined by macros and libraries. These libraries can house control details and optimizations.

Parallelism control

As we mentioned in Section 8.2, a major problem with Multilisp is how to avoid generating tasks in excess of machine parallelism. Such excess tasks represent inefficiency: their generation wastes time and consumes memory space. For conventional, mandatory computation

we want to control task generation to "impedance match" the application parallelism to the machine parallelism (and to control memory use).

However, parallelism control is fundamentally at odds with speculative computation. For speculative computation we want to encourage task generation so we have the flexibility to always be pursuing the most promising computation. In the Eight-puzzle application, for instance, we want breadth-first search, which produces an enormous number of tasks, for minimum execution time, but we want depth-first search to limit the number of tasks generated (and hence memory utilized). For speculative computation to be really successful we need to strike a compromise between the cost and benefit of excess tasks.

11.2 Future[3] Work

We see two components for future work. the theory of speculative computation and the practice of speculative computation.

11.2.1 Theory of speculative computation

We need to further understand the fundamentals of speculative computation and understand its fundamental benefits and costs. We need to develop this understanding in the context of an idealized, abstract model, like the one we assumed in Chapter 9 on scheduling. We can then extrapolate the understanding from this model to guide the practice of speculative computation. Within this abstract model we need to study optimal scheduling and applications. From optimal scheduling we can determine optimal scheduling policies for speculative computation — to determine what control is desired — and assess the ultimate benefit of speculative computation. From applications we can assess the requirements and benefits of speculative computation. We need to look at many examples to determine the ideal support for speculative computation. Finally, we need to examine the properties of our sponsor model in terms of this abstract model.

11.2.2 Practice of speculative computation

Our long term goal is an implementation for efficient and effective support of speculative computation. Our present implementation is a start. The path to this goal necessarily alternates between implementation and experimentation, and hopefully has the guidance of an abstract model, as mentioned above.

On the implementation side, we need to complete our approach. We need to further develop the sponsor model, adding full controller sponsors and modularity. We need to add parallel staying, other attributes — like duration so we can handle precomputing specu-

[3]Pardon the pun

lative computations like speculative streams, and like priority ranges. We need to make improvements in priority queues and the tasknode tree.

On the experimentation side, we need to further assess the benefits, difficulties, and costs of our present implementation and continue this assessment for future work. This understanding will help guide the work on future implementation. There are two components to this experimentation. The first component is implementation costs. We need to analyze the performance of the implementation to determine the implementation costs and how they can be reduced. The second component is applications. We need to experiment with many different prospective applications of speculative computation to determine the benefits and difficulties of speculative computation in our implementation. We need to determine the performance gain of speculative computation and the suitability of our sponsor model, the constructs, and the user/system interface. We need to consider large applications so we can understand how speculative computations interact within an application and also to serve as a "reality" check on our smaller toy applications. Finally, we need users to provide feedback.

On the implementation side again, we need to convert to a compiler-based implementation, such as Mul-T [Kranz], so we can obtain a more realistic idea of the benefits and costs of speculative computation without the artifacts in the present implementation. The interpreter in the present version biases the cost of some parts of the system relative to other parts and obscures certain costs, like that of priority queues, making it difficult to factor out the effect of the interpreter. A compiler-based implementation will eliminate such biases. In addition, any compiler-based implementation will be much faster, which will greatly facilitate investigation of large applications.

In the much longer term, an area for future work is architectural support for speculative computation, such as hardware priority queues and perhaps selection networks, as in MANIP [Wah], for distributing top-priority tasks around the machine.

11.3 Closing Comment

Speculative computation is a fundamental idea that will become very important in the future for searching-type applications as larger parallel machines proliferate.[4] Since search forms the core of most A.I. applications, speculative computation will be particularly important in the efficient application of A.I. technology, as A.I. systems become more commonplace. We have illuminated the issues concerned with speculative computation and presented a model for speculative computation in Multilisp. This model should be useful as an archetype for speculative computation in other languages. Much work remains in assessing and refining this model and studying speculative computation in general.

[4]Speculation has already become important in high-performance computer architecture in the form of branch prediction and multiple-approach speculation, as in VLIW architectures like the Multiflow Trace.

Appendix A

Language Features

This appendix lists all the language features we added to Multilisp to support speculative computation. This list is intended to serve both as a reference guide, with all the features organized in one place, and as a supplement to the description in Sections 6.2 and 7.4.

A.1 Task and future creation and manipulation

(`future` *exp* &`optional` *doc handler*) is exactly the same as described in [Hals86b] (it creates a task to evaluate *exp* and immediately returns a placeholder — called the goal future — for the result) except for the following changes:

1. if the parent task is a mandatory task, `future` creates a mandatory task, and
2. if the parent task is a speculative task, `future` creates a speculative task with source priority equal to the parent's effective priority.

doc is a document string, used by the debugger and Parvis, for identifying the placeholder and *handler* is an exception handler. These optional arguments are described in [Hals86b].

(`dfuture` *exp* &optional *doc handler*) incorporates exactly the same changes as for `future`.

(`delay` *exp* &optional *doc handler*) incorporates the same changes as for `future`, except that if it creates a speculative task, its source priority is 0. Thus, if evaluated by a speculative task, (`delay` E) is exactly equivalent to (`spec-future` E 0).

(`spec-future` *exp pri* &optional *doc handler*) creates a speculative task with source priority *pri* to evaluate *exp* and immediately returns a placeholder for the result, just as `future` does. *pri* must be an integer in the range $[0, MAX]$, where MAX is implementation-specific.

(`make-future` &optional *class*) creates and returns a placeholder and sponsors the class *class* (if specified) with the maximum priority task blocked on the placeholder. This sponsorship is removed when the placeholder is determined.

(my-future) returns the future object of the executing task.

A.2 Groups

(make-group *exp pri* &optional *doc handler*) is the same as spec-future except it returns a group object. A group object consists of a group identifier which is the name for a make-group task and all its descendant tasks and the placeholder object for the result of the make-group task. Again, *pri* must be an integer in the range $[0, MAX]$.

(group-id *grpobj*) returns the group identifier of group object *grpobj*.

(group-future *grpobj*) returns the placeholder, i.e. future, object of group object *grpobj*.

(my-group-obj) returns the group object representing the executing task's group. A task's group is always the newest ancestral group.

A.3 Staying and priority manipulation

(stay-task *obj*): if *obj* is an undetermined future object *f*, then performs the staying operation on, i.e. sets the source priority to zero for, the speculative task with goal future *f* and all its descendants. Otherwise, this construct does nothing. It returns an unspecified value.

(stay-group *group*) performs the staying operation on all members of the group *group*, i.e. it stays all group members. It returns an unspecified value.

(get-priority *f*): if *obj* is an undetermined future object *f*, then returns the effective priority of the speculative task with goal future *f*. Otherwise, this construct does nothing. Depending on implementation specifics, a mandatory task or a speculative task promoted to mandatory status may have a priority *MAND* exceeding *MAX*, the maximum priority for speculative tasks.

(change-priority *obj new-pri*): if *obj* is an undetermined future object *f*, then changes the priority of the speculative task whose goal future is *f* to *new-pri*. It returns an unspecified value. Otherwise, this construct does nothing. *new-pri* must be an integer in the range $[0, MAX]$, where *MAX* is implementation specific. Note: 1) This construct cannot be used to promote a task to mandatory status (priority and status are orthogonal at the language level), 2) It is an error to change the priority of a task promoted to mandatory status; and 3) If this construct is used to stay a task, by setting the task's source priority to 0, it does not stay the descendants of that task.

A.4 Classes

(make-class *class-type*) creates and returns a class object. A class is collection of tasks and a sponsor in the fashion of the groups mentioned in Chapter 5. Unlike the members of a group created with the make-group construct, the members of a class are arbitrary and not necessarily all descendants of a common parent. There are three types of classes, each of which corresponds to a different type of primitive group sponsor:

1. class-all, in which the class sponsor sponsors all the members of the class,
2. class-any, in which the class sponsor sponsors an arbitrary member of the class, and
3. class-pqueue, in which the class sponsor sponsors only the top-priority task in the class.

A class can be sponsored by the maximum-priority task blocked on a placeholder or semaphore or the maximum-priority task enqueued on a priority queue object.

(add-to-class *obj class*): if *obj* is an undetermined future object *f* or a class object *c*, then add-to-class adds either the task associated with *f* or the class object *c* to the class *class*. Otherwise, add-to-class does nothing. It returns an unspecified value.

(remove-from-class *obj class*) functions like add-to-class but instead removes *obj* from the class *class*. In addition, when a task terminates, it is automatically removed from any classes to which it belongs.

(enter *class*) adds the evaluating task to the given class.

(exit *class*) removes the evaluating task from the given class.

These last two constructs are macros which expand to (add-to-class (my-future) *class*) and (remove-from-class (my-future) *class*) respectively.

A.5 Semaphores

(make-sema &optional (count 1) *class*) creates and returns a semaphore object which consists of a count of the tasks which may still enter the critical region, a priority queue for tasks waiting to enter the critical region, and a class for waiting tasks to sponsor. The count field is initialized to count (which must be > 0), or 1 if count is omitted. If count is initialized to 1, the semaphore is a binary semaphore, otherwise it is a general semaphore. The maximum priority task in the priority queue sponsors the class in the class field, which is initialized to *class* (or nil if *class* is omitted). This field is accessible via the construct

(get-sema-class *sema*) and may be set via

(set-sema-class *sema class*)

(`wait-sema` *sema* &optional *cr-thunk*) is a standard semaphore wait operation augmented with a "critical region thunk". If present, the optional argument *cr-thunk* should be a procedure of zero arguments (otherwise an error will occur). The task executing `wait-sema` tests the count associated with *sema* and, if nonzero, decrements the count and promotes itself to mandatory status. Otherwise, the task enqueues itself in the priority queue of waiters for *sema* and suspends. The test and these subsequent actions (either decrement and promote or enqueue) occur indivisibly. If the count was nonzero, the task applies *cr-thunk* (if present), demotes itself to its proper status, and finally returns from `wait-sema`.

(`signal-sema` *sema* &optional *cr-thunk*) is a standard semaphore signal operation augmented with a "critical region thunk" like in `wait-sema`. If *cr-thunk* is present, the task executing `signal-sema` promotes itself to mandatory status, applies the zero argument procedure *cr thunk*, signals the semaphore *sema* (as described shortly), and then demotes itself. If *cr-thunk* is omitted, the task executing `signal-sema` signals *sema* as follows. If the count associated with *sema* is zero, the task dequeues the maximum priority (suspended) task from the priority queue of waiters for *sema*, promotes this task to mandatory status, and resumes it. Otherwise, the executing task increments count. The test and subsequent action in either case occur indivisibly.

A.6 Status manipulation

(`promote-task`) temporarily promotes the executing task to be a mandatory task. It returns an unspecified value.

(`demote-task`) demotes a task temporarily promoted to mandatory status. It returns an unspecified value.

(`rplac`*x*`-eq-mand` *pair new old*), *x* = a or d, performs the following eq check and possible swap atomically: If the c*x*r of *pair* is eq to *old*, the c*x*r of *pair* is replaced by *new*, the executing task is promoted to mandatory status, and *pair* is returned. If the c*x*r of *pair* is not eq to *old*, nil is returned.

Appendix B

Definitions of por and pand

We provide precise semantics of por and pand in this Appendix.

$$(\text{por } E_1\ E_2\ \ldots\ E_n)$$

por evaluates the expressions E_i in arbitrary order and returns the value of the "first" expression to evaluate to true, i.e. non-nil. In this case, any remaining evaluations may be aborted. If all of the n expressions $E_1, E_2, \ldots, E_n$ evaluate to nil, por returns nil.

By "first" we mean a nondeterministic choice among all expressions E_i which would evaluate to true (if evaluated). We can express this notion of "first" in terms of McCarthy's amb operator [McCarthy].

$$(\text{amb } a\ b) = \begin{cases} \text{either } a \text{ or } b & \text{if both } a \text{ and } b \text{ are defined,} \\ a & \text{if } a \text{ is defined,} \\ b & \text{if } b \text{ is defined, and} \\ \text{undefined} & \text{if both } a \text{ and } b \text{ are undefined.} \end{cases}$$

Thus the value of the "first" of E_1 and E_2 to evaluate to true is

$$(\text{amb } (\text{or } E_1\ \bot)\ (\text{or } E_2\ \bot))$$

where $\bot$ means undefined. We can extend this denotation of "first" to n expressions E_i by using an "amb" tree. Note that the specification of amb does not imply that both arguments must be evaluated. if one argument evaluates to a defined value, the other argument does not necessarily have to be evaluated. Thus a value may be returned without fully evaluating more than one E_i to a true value. This is consistent with the (usual) informal notion of "first". (Any such implied temporal property beyond our definition above is implementation dependent.)

We can define the semantics of por in terms of amb as follows:

$$(\text{por } E_1\ E_2) \equiv \texttt{(let ((a (delay } E_1\texttt{))}\\ \texttt{(b (delay } E_2\texttt{)))}\\ \texttt{(amb (touch (or a b))}\\ \texttt{(touch (or b a))))}$$

When one of the arguments to amb returns a value, the evaluation of the other argument may be aborted, which in this context amounts to merely halting such evaluations.

By aborted, we mean that any remaining evaluations are halted after perhaps any appropriate action for side-effects is performed. The exact definition of aborting depends on the computational model. In the context of our touching model aborting means staying; this is how we interpret aborting in this thesis.

Finally, note, as alluded above, that there is no guarantee any particular expression E_i is evaluated, unless all E_i evaluate to nil.

$$(\text{pand } E_1\ E_2 \ldots E_n)$$

pand evaluates the expressions E_i in arbitrary order and returns nil if any expression evaluates to nil. In this case, any remaining evaluations may be aborted. If all of the n expressions $E_1, E_2, \ldots, E_n$ evaluate to true, pand returns an arbitrary true value.

por and pand, as we defined them, are logical duals but not semantic duals: por is strictly more powerful than pand since por returns the actual value of the first E_i to evaluate to a true value. That is, por is both a "first-of" operator, returning the value of the first true-valued E_i, and a logical or. pand is not a first-of operator because of the asymmetry between false, which is the single value nil, and true, which is any non-nil value. (We could make por and pand semantic duals by restricting por to return boolean values (nil[1] and '#t). We chose not to do this because of the loss of semantic power. We could add a separate first-of operator to regain this semantic power, but such an operator seems superfluous in addition to por.)

We can define pand in terms of por as

$$(\text{pand } E_1\ E_2 \ldots E_n) \equiv (\text{not } (\text{por } (\text{not } E_1)\ (\text{not } E_2) \ldots (\text{not } E_n)))$$

but we cannot define por in terms of pand since the true value that pand returns is arbitrary.

[1]Technically '#f.

Appendix C

ParVis

ParVis is a program visualization tool for Multilisp developed by Bagnall [Bagnall] which is invaluable for understanding the performance of Multilisp programs. We used ParVis extensively to understand the potential and the effect of speculative computation in various programs.[1]

ParVis collects information on task state transitions and intertask communication during program execution. After some postmortem analysis, ParVis displays

1. the state of each task in the program execution as a function of time, and
2. intertask dependences

on a high resolution bit mapped terminal (a Symbolics Lisp Machine terminal) as depicted in Figure C.1.

ParVis considers a task to be in one of four states: running, queued, waiting, and no task, as indicated on the top of Figure C.1. The running state indicates that the task is executing; the waiting state indicates the task is blocked on an undetermined future or suspended; the queued state indicates that the task is runnable but queued, awaiting assignment to a processor; and the no task state indicates that the corresponding future object has no task. The no task state occurs in three cases:

1. After the creation of a future object but before the creation of its task object. For example, with `delay` a future object is created immediately but a task is not created until the future is touched.
2. If the task quits (via the Multilisp `quit` primitive).
3. A placeholder, which never has a task.

[1] ParVis also proved useful for debugging our implementation.

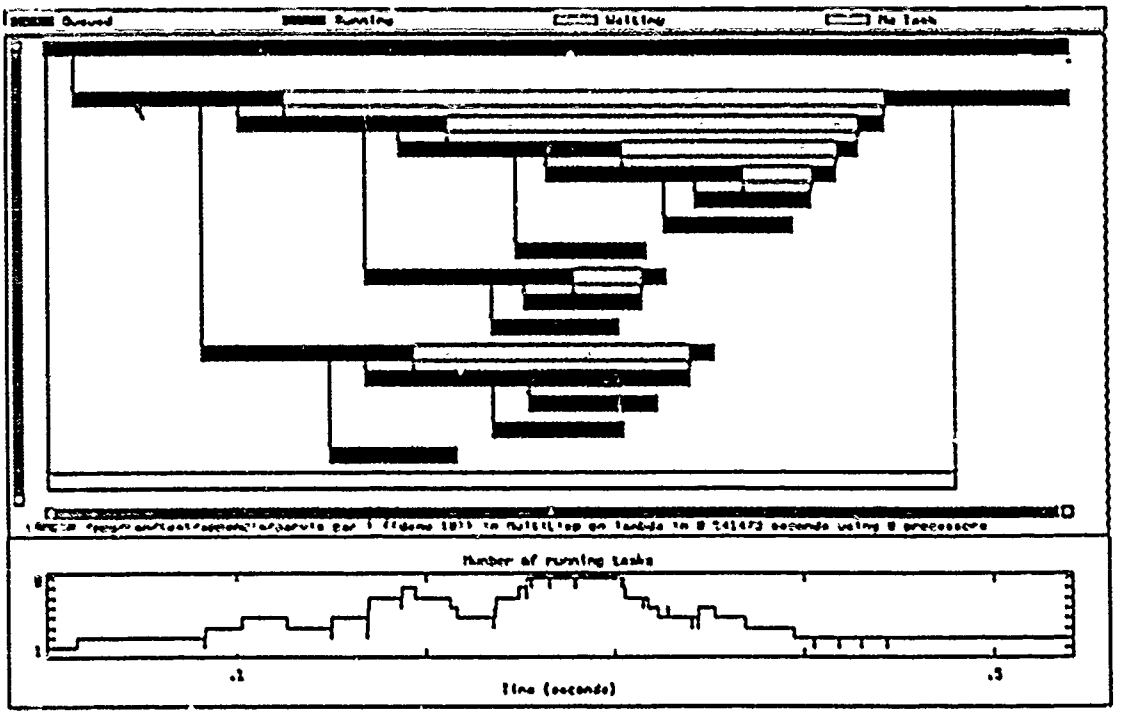


Figure C.1: An example ParVis display

In our implementation we also overload the no task state to mean that the task is in the stayed state. In this case the future object does actually have a task but it is not in any of the other three states that ParVis recognizes.

ParVis indicates intertask dependences with arrows between tasks, as in Figure C.1. There are four types of arrows:

1. create arrows – These are fat arrows from the creator of a future object to the future object.

2. touch arrows - These are narrow arrows from one task to another signifying that the first task blocked on the second, i.e. that the first task touched the undetermined future object associated with the task at the head of the arrow.

3. resume arrows – These are narrow arrows from the end of one task to other tasks. These arrows signify the resumption of waiting toucher tasks when a task is determined.

4. explicit determine arrows - These are dashed arrows from the determiner task to the determinee future object.

Figure C.1 contains each type of arrow.

The lower part of Figure C.1 contains a parallelism profile (number of running tasks versus time) based on the task state information collected and displayed in the upper part of the Figure C.1. All the parallelism profiles in this thesis were obtained from ParVis.

The raw state transition and intertask communication information for ParVis was collected by instrumenting our implementation to record events signifying state transitions and intertask communication. Example events are create-future, make-runnable, run, determine, touch (an undetermined future). This instrumentation contributes an overhead which, of course, varies with the characteristics of the program but is not worse than about 10% degradation in execution time.

We have eliminated a number of details from the above description. For further information on ParVis see [Bagnall].

Bibliography

[Abelson] Abelson, H. and G. Sussman
Structure and Interpretation of Computer Programs
M.I.T. Press, Cambridge, MA., 1984

[Agha] Agha, G.
Actors: A Model of Concurrent Computation in Distributed Systems
M.I.T. Press, Cambridge, MA., 1987

[Bagnall] Bagnall, Laura
ParVis: A Program Visualization Tool for Multilisp
S.M. Thesis, EECS, M.I.T., January 1989

[Baker78a] Baker, H.
List Processing in Real Time on a Serial Computer
Communications of the ACM, April 1978, p. 280-294

[Baker78b] Baker, H. and C. Hewitt
The Incremental Garbage Collection of Processes
A.I. Lab. Memo 454, M.I.T., March 1978

[Ben-Ari] Ben-Ari, M.
Principles of Concurrent Programming
Prentice Hall, 1982

[Bishop] Bishop, P.
Computer Systems with a Very Large Address Space and Garbage Collection
TR-178, Laboratory for Computer Science, M.I.T., May 1977

[Brinch] Brinch Hansen, P.
Structured Multiprogramming
Communications of the ACM, p. 574-578, July 1972

[Bubenik] Bubenik, R. and Zwaenepoel, Willy
An Operational Semantics for Optimistic Computations
TR89-85, Department of Computer Science, Rice University, February 1989

[Burt85a] Burton, F. W.
Controlling Speculative Computation in a Parallel Functional

Programming Language
Fifth Int'l Conf. on Distributed Computing, May 1985, p. 453-458

[Burt85b] Burton, F. W.
Speculative Computation, Parallelism, and Functional Programming
IEEE Trans. on Computers, Dec. 1985, p. 1190-1193

[Burt89] Burton, F. W.
Personal communication on March 30, 1989

[Chamber] Chamberlain, R., Edelman, M. Franklin, M., and Witte, E.
Simulated Annealing on a Multiprocessor
International Conf. on Computer Design, 1988

[Chika] Chikayama, T., Sato, H., and Miyazaki, T.
Overview of the Parallel Inference Machine Operating System (PIMOS)
Proc. of Int'l. Conf. on Fifth Generation Computer Systems, 1988, p. 231

[Clark] Clark, K. and Gregory, S.
PARLOG: Parallel Programming in Logic
Chapter 3 in Vol. 1 of *Concurrent Prolog: Collected Papers*
E. Shapiro, Ed., MIT Press, 1987

[Epstein] Epstein, B.
Support for Speculative Computation in MultiScheme
B.S. Thesis, Brandeis University, Waltham, MA., May 1989

[Gabr84] Gabriel, R. and McCarthy, J.
Queue-based Multiprocessing Lisp
Proceedings 1984 ACM Conf. on Lisp and Functional Prog., p. 25-44, 1984

[Gabr85] Gabriel, R.
Performance Evaluation of Lisp Systems
M.I.T. Press, Cambridge, MA., 1985

[Gabr88] Gabriel, R., and McCarthy, J.
Qlisp
J. Kowalik, ed., *Parallel Computation and Computers for Artificial Intelligence*,
Kluwer Academic Publishers, 1987

[Gold88] Goldman, R. and Gabriel, R.
Qlisp: Parallel Processing in Lisp
Draft, summer 1988

[Gold89] Goldman, R. and Gabriel, R.
Qlisp: Parallel Processing in Lisp
IEEE Software, July 1989, p. 51

[Gonz77] Gonzalez, M.
Deterministic Processor Scheduling
ACM Computing Surveys, Sept. 1977, p.173

[Gonz78] Gonzalez, M. and Sahni, S.
Preemptive Scheduling of Uniform Processor Systems
Journal of ACM, Jan. 1978, p.92

[Grit] Grit, D. and R. Page
Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System
ACM Trans. on Prog. Languages and Systems, October 1981, p. 49-59

[Hals86a] Halstead, R., T. Anderson, R. Osborne, and T. Sterling
Concert: Design of a Multiprocessor Development System
13th Annual Symp. on Computer Architecture, Tokyo, June 1986, p. 40-48

[Hals85] Halstead, R.
Multilisp: A Language for Concurrent Symbolic Computation
ACM Trans. on Prog. Languages and Systems, October 1985, p. 501-538

[Hals86b] Halstead, R., J. Loaiza, and M. Ma
The Multilisp Manual
Parallel Processing Group, Laboratory for Computer Science, M.I.T.,
June 1986

[Hals86c] Halstead, R.
Parallel Symbolic Computing
IEEE Computer, August 1986, p. 35-43

[Hals87] Halstead, R.
Parallel Computing Using Multilisp
J. Kowalik, ed., *Parallel Computation and Computers for Artificial Intelligence*,
Kluwer Academic Publishers, 1987

[Hals86d] Halstead, R.
An Assessment of Multilisp: Lessons From Experience
International Journal of Parallel Programming, Dec. 1986, Plenum Press, New
York

[Hausman] Hausman, B., Ciepielewski, A., and Calderwood, A.
Cut and Side-effects in Or-Parallel Prolog
Proc. of Int'l. Conf. on Fifth Generation Computer Systems, 1988, p. 831

[Heller] Heller, S.
Efficient Lazy Data-Structures on a Dataflow Machine
Ph.D. Thesis, EECS, M.I.T., January 1989

[Herlihy] Herlihy, M.
Optimistic Concurrency Control for Abstract Data Types

Proc. of Fifth ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 1986

[Hoare] Hoare, C.
Monitors: An Operating System Structuring Concept
Communications of the ACM, p. 549-557, October 1974
and Erratum in *Communications of the ACM*, p. 95, February 1975

[Hud82] Hudak, P. and Keller, R.
Garbage Collection and Task Deletion in Distributed Applicative Processing Systems
Proceedings 1982 ACM Conf. on Lisp and Functional Prog., p. 168-178, 1982

[Hud83] Hudak, P.
Distributed Task and Memory Management
Proceedings of Symposium on Principles of Distributed Computing
Lynch, N. et al Eds., p. 277-289, 1983

[Hud84] Hudak, P.
Distributed Applicative Processing Systems: Project Goals, Motivation, and Status Report
Technical Report TR-317, Yale University, May 1984

[Ito] Ito, T., and Matsui, M.
A Parallel Lisp Language PAILISP and its Kernel Specification
To be published in proceedings of US/Japan Workshop on Parallel Lisp
Sendai, Japan, June 5-8, 1989

[Jeffer] D. Jefferson
Virtual Time
ACM Trans. on Prog. Languages and Systems, July 1985

[Johnson] Johnson, D. and Zwaenepoel, W.
Recovery in Distributed Systems Using Optimistic Logging and Checkpointing
Proc. of Seventh ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 1988

[Katz] Katz, M. and Weise, D.
Continuing into the Future: On the interaction of Futures and First-class Continuations
To be published in proceedings of U.S./Japan Workshop on Parallel Lisp
Sendai, Japan, June 5-8, 1989

[Knuth] Knuth, D.
The Art of Computer Programming: Vol. 3, Sorting and Searching
Addison-Wesley, 1973

[Korn79] Kornfeld, W.
Using Parallel Processing for Problem Solving
A.I. Lab. Memo 561, M.I.T., December 1979

[Korn81a] Kornfeld, W.
The Use of Parallelism to Implement a Heuristic Search
A.I. Lab. Memo 627, M.I.T., March 1981

[Korn81b] Kornfeld, W. and C. Hewitt
The Scientific Community Metaphor
IEEE Trans. on Systems, Man, and Cybernetics, January 1981

[Korn82] Kornfeld, W.
Combinatorially Implosive Algorithms
Communications of the ACM, October 1982, p. 734-738

[Krall] Krall, E. and P. McGehearty
A Case Study of Parallel Execution of a Rule-Based Expert System
Int'l. Journal of Parallel Programming, January 1986, p. 5-32

[Kranz] Kranz, D., Halstead, R., Mohr, E.
Mul-T: A High-Performance Parallel Lisp
SigPlan 1989 Conf. on Prog. Language Design and Implementation

[Kung] Kung, H, and Robinson, J.
On Optimistic Methods for Concurrency Control
ACM Trans. on DataBase Systems, June 1981, p. 213-226

[Lieberman] Lieberman, H. and C. Hewitt
A Real-Time Garbage Collector Based on the Lifetimes of Objects
Communications of the ACM, June 1983, p. 419-429

[Lusk] Lusk, E. et al
The Aurora Or-Parallel Prolog System
Proc. of Int'l. Conf. on Fifth Generation Computer Systems, 1988, p. 819

[McCarthy] McCarthy, J.
A Basis for a Mathematical Theory of Computation
Computer Programming and Formal Systems
P. Braffort and D. Hirschberg, Eds., North-Holland, 1963

[Manning] Manning, C.
A Peek at Acore, An Actor Core Language
ACM SigPlan Notices, April 1989, p. 84-86

[Melle] Van Melle, W., A. Scott, J. Bennet, and M. Peairs
The Emycin Manual
Technical report STAN-CS-81-885, Stanford University, 1981

[Miller] Miller, J.
MultiScheme: A Parallel Processing System Based on MIT Scheme
TR-402, Laboratory for Computer Science, M.I.T., Sept. 1987

[Mitten] Mitten, L.
An Analytic Solution to the Least Cost Testing Sequence Problem
Journal of Industrial Engineering, January-February 1960, p. 17

[Mohr] Mohr, E.
Personal communication on June 19, 1989
Computer Science Dept., Yale University

[Muntz] Muntz, R. and Coffman, E.
Optimal Preemptive Scheduling on Two-Processor Systems
IEEE Trans. Computers, Nov. 1969, p.1014

[Nikhil] Nikhil, R.
Id (version 88.0) Reference Manual
Computation Structures Group Memo 284, Laboratory for Computer Science, M.I.T., March 1988

[Nilsson] Nilsson, N.
Principles of Artificial Intelligence
Morgan Kaufmann, 1980

[Osborne] Osborne, R.
Efficient Support for Speculative Computation in Multilisp
Ph.D. Thesis Proposal, EECS, M.I.T., May 1988

[Rao] Rao, V. and Kumar, V.
Concurrent Insertions and Deletions in a Priority Queue
Proceedings of Int'l Parallel Processing Conference, August 1988

[Rees] Rees, J. and W. Clinger (Eds.)
Revised^3 Report on the Algorithmic Language Scheme
ACM SigPlan Notices, December 1986, p. 37-79

[Shap1] Shapiro, E.
A Subset of Concurrent Prolog and its Interpreter
Chapter 2 in Vol. 1 of *Concurrent Prolog: Collected Papers*
E. Shapiro, Ed., MIT Press, 1987

[Shap2] Shapiro, E.
Concurrent Prolog: A Progress Report
Chapter 5 in Vol. 1 of *Concurrent Prolog: Collected Papers*
E. Shapiro, Ed., MIT Press, 1987

[Soley] Soley, R.
On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control
Ph.D. Thesis, EECS, M.I.T., June 1989

[Steele] Steele, G., Jr.
Common Lisp: The Language
Digital Press, 1984

[Strom] Strom, R. and Yemini, S.
Optimistic Recovery in Distributed Systems
ACM Trans. on Computer Systems, August 1985

[Take] Takeuchi, A.
Parallel Logic Programming Languages
Chapter 6 in Vol. 1 of *Concurrent Prolog: Collected Papers*
E. Shapiro, Ed., MIT Press, 1987

[Theriault] Theriault, D.
Issues in the Design and Implementation of Act2
TR-728, A.I. Lab., M.I.T., June 1983

[Ueda] Ueda, K.
Guarded Horn Clauses
Chapter 4 in Vol. 1 of *Concurrent Prolog: Collected Papers*
E. Shapiro, Ed., MIT Press, 1987

[Wah] Wah, B., and Ma. E.
MANIP — A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems
IEEE Trans. on Computers, May 1984, p. 377-390

[Warren] Warren, D.
The SRI Model for Or-Parallel Execution of Prolog: Abstract Design and Implementation Issues
Proc. of 1987 Sympos. on Logic Languages, p. 92

[Watt] Watt, S.
Bounded Parallelism in Computer Algebra
Report CS-86-12, Faculty of Mathematics, University of Waterloo, May 1986

[Weber] Weber, R.
Scheduling Jobs with Stochastic Processing Requirements on Parallel Machines to Minimize Makespan and Flowtime
J. Appl. Prob., Vol. 19, 1982, p.167-182

OFFICIAL DISTRIBUTION LIST

Director
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

2 copies

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. Gary Koop, Code 433

2 copies

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

6 copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

12 copies

National Science Foundation
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

2 copies

Dr. E.B. Royce, Code 38
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

1 copy