Speculative Computation in Multilisp*†

Randy B. Osborne

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/1 March 21, 1990

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 discuss these requirements in the parallel symbolic language Multilisp and present a *sponsor model* for speculative computation in Multilisp which 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 demonstrate that our support for speculative computation adds expressive and computational power to Multilisp, with observed performance improvement as great as 26 times over conventional approaches to parallel computation.

*This research was performed as part of the author's doctoral work at the M.I.T. Laboratory for Computer Science.

†To be published in *Parallel Lisp: Languages and Systems — Proceedings of U.S./Japan Workshop on Parallel Lisp, Sendai, Japan, June 1989*, Lecture Notes in Computer Science, Springer-Verlag.

1 Introduction

The `future` construct in Multilisp (described in Section 1.1) has proven very versatile and successful for achieving parallelism in symbolic computation [Hal86a]. 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` [Hal85, Hal86b, Hal86a]. 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 computation resources are generally limited, we would like to reclaim the 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 discuss these requirements in this paper and the issues with supporting speculative computation in Multilisp. We present a model for speculative computation in Multilisp which 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.

1.1 Multilisp

Multilisp [Hal85] 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). (`future` E) creates a task to evaluate E and immediately returns a placeholder for the result. This placeholder (known as a *future*) may be manipulated just as if it was the result of evaluating E. If a task attempts a strict operation on such a placeholder (i.e. an operation, such as $+$, requiring the value represented by the placeholder), the task suspends until the placeholder is *determined* with the result of evaluating E. This

[1]We put abort in quotes because, as we discuss later, we do not necessarily have to kill useless computation.

task is said to *touch* the placeholder. We generalize this notion to say that the strict operation task touches the task evaluating E.

(`delay` E) is exactly like `future` except the task created to evaluate E does not begin execution until the placeholder is touched. The Appendix describes two additional features of Multilisp which we use later.

1.2 Previous Work

We briefly review the most significant previous work in speculative computation.

Burton [Bur85] appears to be the first to have addressed both requirements — controlling computation and reclaiming unnecessary computation — for speculative computation. Burton proposed adding a parallel construct essentially like `future`, but with an additional priority argument, to a simple functional language. The "speculative" tasks created by this construct were to be scheduled for execution according to their priorities, thus providing a means to control computation. Burton suggested an approach like [GP81] for aborting useless tasks. Burton never implemented this proposed support for speculative computation.

There are two problems with Burton's proposed support. First, his priorities are fixed. Thus the computation resources allocated to speculative activities cannot be reassigned with the availability of better information. His priorities lack transitivity (see Section 4.1), for example. Second, his control via priorities lacks modularity. The priorities 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 (see Section 4.1).

Two other groups have investigated speculative computation in functional languages based on distributed graph reduction models. In his work on Distributed Applicative Processing Systems (DAPS), Hudak developed a model with distributed mark and sweep garbage collection to reclaim useless tasks [HK82] but with no priorities or means to control computation. More recently, Partridge and Dekker presented a scheme with integer priorities and distributed reference counting to reclaim useless tasks [PD89]. The priority of a task in their scheme is the maximum of the priority contributed to it by each of its (possibly multiple) parents. Let p denote the current priority of a parent task. Then, initially a parent task contributes priority $p-1$ to each of its child tasks. Upon demanding a child task, the parent contributes priority p to the child task. Priorities cannot decrease. Due to this constraint and the constrained relationship between parent and child priorities, these priorities lack flexibility in controlling computation. These priorities also lack modularity, as in Burton's work. One important difference between our work and these functional language efforts is that

Multilisp includes side-effects.

There has also been work on supporting speculative computation in the parallel Lisps MultiScheme [Mil87] and Qlisp [GG89]. MultiScheme has `disjoin` for spawning "first-of" styles of speculative computation (see Section 3.1) and garbage collection of tasks for reclaiming unnecessary computation. Recently, Epstein [Eps89] extended MultiScheme beyond this rudimentary support for speculative computation. Like Burton, Epstein added priorities for controlling computation, except Epstein's priorities are transitive (as in [PD89]): a computation has at least the priority of the computations blocked on it. Epstein also used this priority system to reclaim unnecessary computation by explicitly "downgrading" the priority of unwanted computation. Our ideas are similar, but go further. Epstein's ideas fall short in two ways. First, downgrading fails to reclaim all the descendants of an unnecessary computation. Epstein relied on garbage collection to reclaim such descendants, but this strategy suffers problems, as we discuss in Section 4.2.1, especially in the presence of side-effects. Second, Epstein's control via priorities lacks modularity. Finally, Epstein apparently did not investigate (due to time constraints) the performance of any applications utilizing speculative computation.

Qlisp's support for speculative computation centers around "heavyweight" futures which can implement "first-of" types of speculative computation. Qlisp also has garbage collection of tasks and explicit killing for reclaiming unnecessary computation. However, Qlisp lacks any means to control computation (other than aborting it). The Qlisp work is on-going.

Soley [Sol89] explored speculative computation in a functional dataflow language. His main emphasis was reclaiming unnecessary computation. He did add priorities to control computation but the associated overhead was large and priorities were not transitive.

Researchers in parallel logic languages have long been interested in a form of speculative computation they call Or-parallelism. However, to date all the work we know of has concentrated on aborting useless computation and lacked any means, such as priorities, to control computation. (See e.g. [Lus88, CG87, Ued87, Sha87].) To our knowledge, the one exception is the work of Chikayama et al [CSM88]. They are investigating shōens for speculative computation in logic languages. A shōen is a manager for controlling computation (through priorities) and the resources devoted to computation.

Our work addresses both requirements for supporting speculative computation — controlling speculative computation and reclaiming unnecessary computation — in an imperative environment. We argue that the implementation must support these requirements for suitable efficiency and expressiveness.

1.3 Overview

In Section 2 we give a precise definition of speculative computation and related terms. In Section 3 we present some examples of speculative computation to which we refer throughout the paper. In Section 4 we discuss the issues with speculative computation motivated by the examples. In Section 5 we present our sponsor model for speculative computation. In Section 6 we describe our implementation of a subset of this model which we call the touching model. Finally, in Section 7 we report the results of experiments with the examples in Section 3.

2 Definitions

2.1 Speculative Computation

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 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 if, with the given state of knowledge, the computation is definitely always necessary to meet the program specification. Likewise, a computation is not required if, with the given state of knowledge, the computation is definitely not ever necessary to meet the program specification. 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.

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.

Eager computation is computation that is started early, before it is required, but with certainty that it will be required.[2] 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 simpler. 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. For a nondeterministic operator like parallel `or` (described in Section 3.1), we define a **choice** computation to be any computation which such an operator may choose. When a demand-driven interpreter demands the result of such a nondeterministic operator, we define the interpreter to provisionally demand all the operator's choice computations. (To avoid problems with non-termination, we assume a *fair* demand-driven interpreter that provisionally demands all choice computations equally.) At the "choice time" when the nondeterministic operator chooses a choice computation, the demand-driven interpreter irrevocably demands the chosen computation and "undemands" all of the operator's non-chosen computations. The chosen computation is, of course, required and the non-chosen computations, provisionally demanded and then undemanded, are not required.

Thus for a functional program P we have the following simpler definitions. A 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.

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

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

variable, or releasing a lock or semaphore. Thus with side-effects we must use the general classification given earlier. However, side-effects raise some thorny 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 do not pursue these issues here.

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

2.1.1 Classifications of Speculative Computation

We distinguish three flavors of speculative computation based on the way in which resources are used for speculation:

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.

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.

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).

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.

2.2 Optimistic Computation

What most people mean by optimistic computation is subset of 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. That is, side-effects obey an atomic transaction or encapsulation [BZ89] model in which all or no side-effects persist. Our notion of speculative computation includes styles of speculation which do not fit into this all-or-nothing model.

3 Examples

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

3.1 Parallel search

Search is a particularly rich domain for speculative computation. In searching for a target in subspaces S_1, S_2, ..., S_n, we could search the subspaces concurrently (subject to availability of machine resources) and terminate all remaining searches when we find the target to avoid wasting machine resources. These subspace searches are examples of "first-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 in Section 4.2.2). 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 such as in Artificial Intelligence.

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 first E_i to evaluate 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 to evaluates to nil and true if all the E_i evaluate to non-nil. 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 is an example of multiple-approach speculative computation.

3.2 Branch prediction: parallel `if`

In the expression (`if` *pred consequent alternate*) 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*.[3] Computing *consequent* and *alternate* in parallel 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 the reduction of the critical path in an *efficient* manner.

3.3 Producer-consumer parallelism: The Boyer Benchmark

Written as a parallel Multilisp program, the Boyer Benchmark [Gab85] 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)` $\rightarrow$ `(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.) Thus the rewrites represent speculative computation.

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 7.2) because the machine becomes saturated with speculative rewrites which swamp 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

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

perform all rewrites lazily but then the execution time may be 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.)

3.4 Ordering: the traveling salesman problem

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. One way to achieve the desired ordering of node expansions is via an explicit priority queue (programmed in the language). However, this is an awkward solution. A better solution is to extend the static 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.

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. 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.

3.5 Precomputing streams

A stream is a possibly infinite list of objects [AS84]. 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 — the hope is to reduce the critical path by doing a certain amount of precomputing. Such element precomputing is an example of precomputing-based speculative computation. We do not pursue precomputing styles of speculative computation further in this paper.

4 Issues

In this section we describe the main issues with speculative computation. The idea behind speculative styles of computation is to use excess machine resources to reduce the average execution time. Because these excess resources are limited, it is important to use them efficiently. Therefore, in general, we must control speculative computation and we must reclaim irrelevant computation.

4.1 Controlling computation

Because not all speculative computation is necessary, the order in which we allocate resources to computation is important, as the Boyer example demonstrates. We want to allocate resources to computation to favor the most promising computations. This requires some sort of ordering of computation. For concreteness, let us assume priorities for specifying this ordering.

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

In general, the programmer must decide the relative promise of computations, as controlling computation relies on meta-knowledge about a program's function, inputs, and purpose as well as its operating environment. The optimal assignment of priorities is a scheduling problem which we mention briefly 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. We investigated optimal scheduling in some simple cases in [Osb89] and in current work we are analyzing optimal scheduling of nested `por`s and `pand`s.

Until we know more about the optimal control of speculative computation (at least in certain situations), control specification will have to be *ad hoc* and experimental. In the rest of this paper we focus on basic control mechanisms.

Finally, we wish to point out that there are other types of control, like controlling the duration of execution, and multiplexing computations, that we also want for speculative computation in addition to the ordering provided by priorities.

4.2 Reclaiming computation

To use machine resources efficiently we must reclaim irrelevant computation. The essential issue is the reclamation of processor resources devoted to irrelevant computation so that these resources may be recycled for other computation. (We assume that processors are expensive and few in number and that memory is cheap and plentiful. Hence our emphasis on processor resources.) Reclamation of computation state, i.e. the storage used by a computation, is an independent issue.

The two important issues for computation reclamation are:

1. Reclamation speed

   We must reclaim irrelevant computation quickly.

2. Run-away phenomenon

   This phenomenon occurs when an irrelevant computation spawns descendants faster than they can be reclaimed. It must be avoided.

There are two approaches to computation reclamation: implicit reclamation and explicit reclamation. Each approach has problems.

4.2.1 Implicit Reclamation

In implicit reclamation the system discovers that a computation is irrelevant and reclaims it. However, there is no operational test that the system can perform to determine if a computation is irrelevant. Instead, implicit reclamation systems approximate the relevance of a computation by its accessibility. (We ignore reference count systems since they cannot reclaim cyclic structures.) A computation is represented by a task object and a computation is considered relevant if and only if its task object is accessible from some root set. Therefore implicit computation reclamation amounts to the garbage collection of computation, an idea first suggested by [BH78].

There are two problems with implicit computation reclamation, both following from the assumption that a computation is relevant if and only if it is accessible:

1. A computation may be irrelevant but still accessible.

2. Inaccessibility implies irrelevance only if all the computation is functional.

   In the presence of side-effects, as in Multilisp, an inaccessible computation can still be relevant through the side-effects it may perform.

We also believe that implicit reclamation is too inefficient. With implicit reclamation, irrelevant computation is only reclaimed when a garbage collection occurs, which we argue is too infrequent and costly. The problem is that the frequency of implicit reclamation is tied to the frequency of garbage collection cycles, which depends on factors such as memory size and rate of storage allocation that do not necessarily have anything to do with the creation, rate of creation, or even presence of irrelevant computation. An application that does little consing, for instance, will have very infrequent computation reclamation.

There are two ways to address the problem of the infrequency of implicit reclamation:

1. Invoke garbage collection more frequently (than is required by storage concerns).

   This method is unattractive because it incurs the large cost of garbage collection (tracing all the objects), more frequently, whether or not there is any irrelevant computation present.

2. Invoke the garbage collector explicitly whenever we believe there is enough irrelevant computation to justify the cost.

   This alternative still suffers from the cost of tracing **all** objects, including non-task objects. Furthermore, it is unattractive if we are continually generating irrelevant computation.

3. Use generation garbage collection [LH83].

   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. However, this reduced cost does not entirely solve the problems with implicit reclamation. Three major problems still remain:

   (a) The reclamation time is still coupled to the consing rate.
   (b) We still must trace all accessible objects in a generation, whether or not the objects are task objects.
   (c) An irrelevant computation might still take a long time to be reclaimed if it or an inaccessible object which points to it (directly or indirectly) gets promoted to older generations.

Implicit reclamation does not have any run-away phenomena: all computation is stopped during a garbage collection and not restarted until proven accessible.

4.2.2 Explicit reclamation

In explicit reclamation the user identifies irrelevant computation and either the user or the system reclaims such computation.

The simplest form of explicit reclamation is explicit termination checking whereby the user both identifies and terminates irrelevant computation. In explicit termination checking, the programmer "wires" reclamation into the application. The programmer arranges for each computation he or she may 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" as in [Sol89]) or a nonsense value. In any event, computation is essentially short-circuited by a true termination condition. There are two important points with this type of reclamation. First, the reclamation is performed totally at the language level (in the source code), by merely changing the course of computation, and not by appeal to some extra-language facility, such as garbage collection. Second, computation deemed irrelevant terminates, returning a value, and is not aborted in mid-stream before returning a value.

Explicit checking has several problems. The main problem is inserting the termination checks in the source code. Every descendant computation and (potentially) every function must have termination checks. This checking is awkward to arrange in many cases, and it is impossible to arrange in some cases because not all the source code may be available. Some function calls may be "unknown" in the sense that the functions may be defined in modules outside a user's scope. For instance, a function may be defined previously in the environment, or defined subsequently in the environment by dynamic loading, or a function may be part of a system utility or library. How do we perform termination checking in these "unknown" functions? How do we do termination checking in the descendant computations that such functions may spawn? A static placement of termination checks in the source code cannot always track the dynamic execution of a program.

Three problems remain even if it is possible to insert all the necessary termination checks in the source code. The first problem is ensuring that reentrant source code remains reentrant. If the statically inserted termination checks in the source code check statically assigned locations, then all invocations of a function check the same location. This links the termination of all simultaneous invocations of a function even if the invocations are part of independent speculative activities. Such linking effectively prevents reentrant code — different speculative activities cannot call the same system utility simultaneously for instance. We can solve this problem by using dynamically bound variables for termination checks — then the statically inserted checks check dynamically bound locations. However, dynamic binding is expensive in parallel processing (because of the need for deep binding). The second problem is nested termination. Each nesting of speculative activities, e.g. each nesting of `por`, has to have an independent termination check. However, we cannot always statically determine the dynamic nesting behavior of computations and thus we cannot always insert the necessary termination checks. The third problem is sharing of computation. All parties that could potentially share the result of a computation must understand if that computation could terminate prematurely and must know what action to take if the computation does terminate prematurely. For example, a disjunct D 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 D is terminated? What value is returned to function F? In the presence of side-effects it is all too easy for this type of sharing to occur. Once again, we cannot always statically determine the dynamic nature — sharing in this case — of computation.

While explicit termination checking may be useful in simple cases where static analysis suffices, it lacks sufficient expressive power to be a general solution.

For general explicit reclamation we need to separate the identification of irrelevant computation from its reclamation. The user can "declare" computation irrelevant, but as just discussed, the user cannot always reclaim computation at the language level. For a general solution we need system support to reclaim computation.

First, we need source code-independent computation reclamation: we need to reclaim computation by (potential) checks below the language level in the dynamic execution of a computation, not by static termination checks embedded in source code. This addresses the problem with unknown functions. The checks could take the form of either interrupts or polling, as in explicit checking (but below the language level, unlike in explicit checking). With polling, the interpreter must perform such checks or the compiler must generate code to perform such checks in **all** code (user and system). Polling can take one of two forms. Either a speculative activity keeps one flag that all of its computations and their descendants poll, as in explicit termination checking, or each computation has its own flag which it polls and a speculative activity maintains a list of all the flag locations in its computations and their descendants. The former alternative means that a computation has, in general, a list of flag locations to poll (because of nested speculative activities and sharing of computation). Efficiency thus dictates the latter alternative which has a smaller runtime cost while performing a computation but a larger startup cost (to register the flag location) when initiating a computation. We therefore assume the latter alternative in the following discussion. One can view interrupts as merely an optimization of this latter alternative: an interrupt supplies a positive checking result without polling.

Second, we need dynamic naming of computations to solve the problems with reentrant code and nested computations. If using interrupts, we need dynamic naming of computations to uniquely identify computations for interrupts, and if using polling we need dynamic naming of checking locations so we can generate checking code statically but still perform checks dynamically. For polling, we can achieve the dynamic naming of checking locations by associating the checking location (i.e. flag location) with the (dynamic) name of the computation.

Third, we need automatic naming of descendant computations. We need to be able to name all the descendants of a computation to declare them irrelevant and cause their reclamation (by either interrupts or polling). The system must automatically record such names to solve the problem with unknown functions which spawn descendants.

Fourth, we need some form of reference counts for shared computation to avoid reclaiming such computations while at least one sharer is relevant.

Fifth, we need reversible reclamation. With declarations, a programmer may declare a relevant computation irrelevant. Although a programmer (or programming system) can potentially have an understanding of an entire program, the programmer may choose not to, may not be able to, or may simply make an error. Thus a relevant computation may demand a computation declared irrelevant, either directly, by demanding the result of the "irrelevant" computation, or indirectly, by requiring the side-effect(s) that the "irrelevant" computation will perform (e.g. release a lock or semaphore).

If a computation is killed — irreversibly reclaimed — when it is declared irrelevant, then *lockout* deadlock will result if some other computation should later demand its result. One solution to this problem is to signal an error when a killed computation is demanded, in effect declaring it an error to mistakenly declare a computation irrelevant.

This solution has two problems. First, a killed computation may be demanded 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 computations demand the same killed computation. Second, the user's idea of a computation's relevance may change with time. Initially, the best information available may suggest that a computation is irrelevant. However, information could later be produced (e.g. from new input data) which may suggest that the computation is in fact relevant. There is no way within this first solution to express this "imperfect" relevance knowledge.

A better solution is to merely suspend a computation when it is declared irrelevant, thereby reclaiming its computation resources (which we argued earlier is the important issue), while still maintaining its computation state. Thus the computation may be restarted later if a relevant computation demands its result (or if it must perform a relevant side-effect). The ability to resume computation declared irrelevant is important because it is often difficult to foresee the relevance of computations, especially in large systems, due to all the ways computation can interact.

Thus, we need reversible reclamation — we must be able to restart computation that is inadvertently reclaimed though user error or failure to (correctly) analyze side-effect dependencies.

Finally, to prevent run-away phenomena, the key requirement is to prevent the spawning of new descendant computations until reclamation reaches all computation.

In summary, the special system support required to reclaim computation is:

1. Source-code independent reclamation by either interrupts or polling below the language level. Interpreter or compiler support is required to automatically insert the polling checks.
2. Dynamic naming of computation (for interrupts and polling locations)
3. Automatic naming of descendant computations
4. Reference counts for shared computation
5. Reversible reclamation
6. Interlock of spawning and reclamation to prevent run-away phenomena

4.3 Side-effects

As with conventional parallel computation, the main issue with side-effects is proper synchronization. The emphasis on relaxing synchronization constraints and aborting computation give rise to three additional issues. The first is persistence of side-effects: when a computation is aborted, what happens to the effects it has performed? One possibility is to completely undo all side-effects performed by an aborted computation, yielding "all-or-nothing" semantics for side-effects. The second issue, which is very much related to the first, is 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 the consequent side-effected y and the alternate did not — which effect persists?). Both of these problems can be addressed (at considerable expense) by encapsulation methods in which each speculative thread has a copy of the store and the copies are merged afterwards.

We do not address these issues of persistence and interference in our work because we believe they are not important in our intended application domain (mostly functional applications — we believe side-effects should be used sparingly) and because we believe available solutions are too expensive for anything other than coarse-grained parallelism.

The third issue is the relevance of side-effects. Specifically, some side-effects may be relevant in the sense that deadlock may occur if they are not performed, e.g. if a computation holding a lock is aborted. This aspect of side-effects makes it tricky to determine if a computation is relevant, since it may only be relevant for its effect and not its result. In fact, this is precisely the subtlety that causes the problems with implicit reclamation.

4.4 Errors and exceptions

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

1. Control-related errors

   A computation controlling the resource use of other computations 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 computation or in a computation which later becomes irrelevant. Recognizing this, how are errors and exception conditions treated?

We have some ideas on how to address these issues based on the groups in Mul-T [KHM89], but we have not pursued them in our work.

5 The Sponsor Model

Our model for speculative computation in Multilisp is based on the notion of sponsors introduced by [KH81]. The allocation of resources to tasks is controlled by the *attributes* a task possesses. Sponsors supply these attributes, which may be priorities for example. 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. The attributes contributed by a task's sponsors are combined according to a *combining-rule* to yield the *effective attributes* of the task.

There are four types of sponsors in our model:

1. External sponsors — These sponsors supply absolute attributes. We discuss later why we call them external.

2. Toucher sponsors — When one task touches another (see Section 1.1), 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. Task sponsors — A task may sponsor 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 task to sponsor its children.

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 receive sponsorship and actively distribute it among the tasks in their control domain according to some built-in control strategy. Controller sponsors implement the complex and dynamic control that we sometimes want.

The collection of sponsors in an application form a sponsor network. 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 sponsors by touching, by passive task sponsors according to their explicit connection with other sponsors, 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, we call them external.

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 essentially introduces a new ordering space for tasks in that group. Thus the ordering of tasks in a group may be local to that group. Groups may be nested with all local ordering relative to the immediate parent group or to other groups, depending on the controller sponsor. Thus groups allow us to solve the modularity 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.

We have omitted some important details of this model, such as inter-group touching, for space reasons. See [Osb89] for further details of this model.

6 Implementation

6.1 The Touching Model

We implemented an initial subset of the sponsor model that we call the touching model. The only attributes in this model are priorities. Each task has an external sponsor which provides what we call the *source* priority of the task. The external sponsor, toucher sponsors, and controller sponsors of a task T combine to determine the task's effective priority according to the following *max combining-rule*:

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

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 $control(i)$ is the set of controller sponsors for task i. There are no task sponsors in this model and only a primitive form of controller sponsors which we describe in Section 6.2.

Any change in the effective priority of a task may lead to changes in the effective priority of the tasks sponsored by that task, and so on. The recursive process of updating the effective priorities according to the max combining-rule is called *priority propagation*. There can be cycles during priority propagation (see [Osb89]), but with our primitive controller sponsors all such cycles correspond to a "touch" chain cycle of tasks:[4] all the tasks in the touch chain cycle are deadlocked on each other. Because touch chain cycles are an error condition in Multilisp, we do not worry about the cycles they may cause during priority propagation.

All priorities are in the interval $[0, MAX]$. Only the tasks with the highest effective priority run at any point in time (i.e. scheduling is preemptive), except for tasks with effective priority 0. Such tasks are not runnable; they are *stayed*, meaning that their

[4]Recall that there are no task sponsors in our touching model.

processor resources have been reclaimed. (Thus the preemptive scheduling mechanism provides source code-independent reclamation.) In order for a task to be stayed, the source priority of the task must be 0 and the effective priority of all the task's touchers and controllers must be 0, i.e. the task must be un-sponsored. (Setting the sponsor priority to 0 accomplishes the same thing as removing the sponsor.) The computational state of a stayed task is retained (until it becomes inaccessible and is garbage collected). Thus a stayed task may be restarted by re-sponsoring the task so that the task's effective priority becomes > 0 — this provides reversible reclamation.

Computation reclamation is explicit, as suggested in Section 4.2.2. "Declaring a task irrelevant" means setting the source priority of the task to 0 and propagating any resultant change in the task's effective priority (to any task it may be sponsoring and so on). We call this process *staying*. 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 sponsors, indicating that the task is relevant from the viewpoint of other tasks demanding its value. Thus the effective priority provides a reference count.

The max combining-rule provides demand transitivity and solves the problem of mistakenly staying computation, such as might occur with shared computation and side-effects. If we mistakenly stay a task we automatically resume it when we later touch it. Extending this idea, we can even solve the problem with aborting relevant side-effects. We simply have to ensure that our demand for a side-effect ensures sponsorship of the computation that will perform the side-effect. We describe how to do this and give examples in [Osb89].

Thus the max-combining rule allows us to handle controlling and reclaiming computation in one simple and uniform framework.

6.2 Language Features

We had two important goals in this implementation. The first goal was to retain the `future` construct. 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 effective priority MAX 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 given earlier: 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. Thus `make-group` provides automatic naming of descendant tasks. Presently, there is no controller sponsor associated with these groups. (We plan to add full controller sponsors in the future.)

(`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. Thus a group provides a (dynamic) name for reclamation and `stay-group` provides automatic staying of descendants.

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

The primitives (`get-priority` *obj*) and (`change-priority` *obj new-pri*) provide a way to examine and change a task's source priority. See [Osb89] for details.

Sometimes we need to prevent a speculative task from being stayed. For example, we want to avoid deadlock if a task is stayed in a critical region. One way to achieve this goal is with the following constructs.

(`promote-task`) temporarily promotes the executing task to be a mandatory task.

(`demote-task`) demotes a task, if necessary, from temporary promotion to mandatory status.

(`rplacx-eq-mand` *pair new old*), x= `a` or `d`, has the same semantics as the `rplacx-eq` construct in Multilisp (see Appendix), except it temporarily promotes the task to mandatory status if the `eq` test succeeds. This construct is useful for spin-locks.

(`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 Section 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.

(`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*.

A class may be sponsored by either the maximum priority task blocked on a placeholder or the maximum priority task waiting on a semaphore.

(`make-future` &optional *class*) creates and returns a placeholder (as described in the Appendix) 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.

(`make-sema` &optional *class*) creates and returns a binary semaphore object and sponsors the class *class* (if specified) with the maximum priority task waiting on the placeholder.

Classes provide another way to prevent deadlock due to staying a task in a critical region. The idea is to ensure that any task in the critical region is the member of a class and ensure that any task requiring access to the critical region sponsors that class (e.g. by waiting on a semaphore for access to the critical region). This class in turn sponsors any tasks in the critical region. See [Osb89] for details.

6.3 Implementation Details

The implementation is an extension of the original byte-code interpreter implementation of Multilisp [Hal85]. The major features we added are preemptive scheduling based on priorities, touch propagation, and staying. The preemptive scheduling is implemented with priority queues and polling. In order to stay all the descendants of a group we maintain a "tasknode" tree of all tasks and their children. We mitigate run-away phenomena by preventing the creation of new tasks in a group while we are staying that group. See [Osb89] for further implementation details.

6.4 An Example: `por`

We present an implementation of `por` as an example of our language features. As a side benefit this example also illustrates some of the issues in speculative computation.

An implementation of `por` has four general requirements:

1. Initialization – create a task to evaluate each disjunct[5]
2. Race officiating – return the first true value
3. Termination detection – return nil if all the disjuncts evaluate to nil
4. Computation reclamation – abort any remaining (useless) tasks after the first true value is returned

Figure 1 shows an implementation of `por` meeting these requirements. The argument to this version of `por` is a list of (thunk . priority) pairs. The priority in each pair is the initial priority of the associated thunk. These priorities let the user convey the relative promise of the thunks.

First we give an overview of the implementation ignoring the class and group constructs. The implementation creates a placeholder for the result (line 3) and calls the procedure `spawn-tasks` (line 17) to spawn a task to evaluate each thunk in the argument list (line 6). 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 (line 7), the task determines the result placeholder to that value (line 8) and stays all the other tasks (line 9). If the thunk evaluates to nil, the task performs termination detection (line 11): if all the other thunks have already evaluated to nil, the task determines the result placeholder to nil (line 14).

[5]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.

```
(define (por thunk-pri-list)
  (let* ((lock (cons '*no-result* (make-future)))                  ; 1
         (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)                          ; 4
          (let ((thunk (get-thunk (car thunk-pri)))
                (priority (get-priority (car thunk-pri))))
            (add-to-class                                          ; 5
               (spec-future (eval-thunk thunk) priority)           ; 6
               class)
            (spawn-tasks (cdr thunk-pri) (+ n 1)))))

    (define (eval-thunk thunk)
      (let ((val (thunk)))
        (if val
            (if (rplaca-eq lock '*result* '*no-result*)            ; 7
                (begin
                  (determine-future result val)                    ; 8
                  (stay-group (group-block (my-group-obj)))))      ; 9
            (if (eq (car lock) '*no-result*)                       ;10
                (term-detect (cdr lock))))))                       ;11

    (define (term-detect n)
      (if (rplacd-eq lock (- n 1) n)                               ;12
          (if (= n 1)                                              ;13
              (determine-future result nil))                       ;14
          (term-detect (cdr lock))))

    (add-to-class                                                  ;15
      (group-future                                                ;16
        (make-group (spawn-tasks thunk-pri-list 0) *max-pri*))     ;17
      class)
    result))
```

Figure 1: An implementation of por

Now we explain the implementation in further detail. Line 1 initializes a lock cell. Its `car` is used for race officiating synchronization. Initially the `car` is `'*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. The `make-group` construct in line 17 serves two purposes. First, it creates a group in which to spawn all the `eval-thunk` tasks and thus provides a name (referenced in line 9 via (`group-block` (`my-group-obj`))) by which all the tasks comprising the `por` can be stayed. Second, it creates a new task to spawn the `eval-thunk` tasks so that `por` returns the result placeholder as soon as possible. The make-group task has maximum priority `*max-pri*` (equal to MAX) so that all the `eval-thunk` tasks are spawned as rapidly as possible.

Termination detection works in a distributed fashion as follows. The procedure `spawn-tasks` maintains a count of the number of thunks spawned and in line 4 determines the placeholder in the `cdr` of the lock cell with this number. Thereafter the `cdr` of this cell holds the number of thunks still running. When a thunk evaluates to nil, the thunk task first performs the optimization in line 10 and if necessary calls the `term-detect` procedure. This procedure spins until it can atomically decrement the count in line 12. If only one task remains when it eventually succeeds (line 13), the result placeholder is determined to nil as described before.

The class created in line 2 is used for two purposes. The first purpose is to cause any demand for the `por` result manifested by touching the result placeholder to be propagated to all the disjunct tasks by the class-all sponsor policy. This is the reason for the `add-to-class` in line 5. The second purpose is to ensure some amount of modularity. We want to be able to embed this `por` within other speculative computation, where it could be stayed and restarted, without danger of deadlock. If the `por` is stayed we must ensure that all the tasks comprising it are restarted when the result placeholder is touched. We presently lack full-fledged controller sponsors to perform this function so we achieve it instead by making sure all the tasks comprising the `por` are members of the class created in line 2. This is the reason for the `add-to-class` in line 15 (as well as line 5). The `group-future` in line 16 extracts the future object for the make-group task from the group object.

6.5 Problems

The main problem with the present implementation is the lack of full controller sponsors. Although classes are useful for solving deadlock problems with side-effects (see [Osb89]), they are an inadequate substitute for controller sponsors. For example, the class sponsor in the above `por` implementation fails to fully address the modularity problem. The desired relative priorities of disjuncts are not necessarily maintained if the por is stayed and restarted (because staying sets the source priority to 0 and restarting does not reset it) and descendants of the winning disjunct cannot continue once the por returns (because all descendants of the por are stayed when it returns). This later problem can be addressed by creating a group for each disjunct and staying only the non-winning disjunct groups. However, this solution is too awkward.

7 Results

7.1 pand: Emycin

This application is a follow-up on the Emycin work of Krall and McGehearty [KM86]. We started with the parallel kernel they developed for this rule-based expert system and investigated the use of pand for combining inference rules. We used the same gems database as Krall and McGehearty described in [KM86]. This database uses and heavily to combine inference rules.

We experimented with three versions of the Emycin kernel differing only in the version of and. The three versions were:

1. Sequential and
2. Speculative pand – the and analog of the speculative por function presented in Section 6.4.
3. Mandatory pand – the speculative pand version, implemented instead with mandatory tasks and without groups and staying, i.e. without the ability to abort unnecessary computation.

Table 1 shows the results on the Concert Multiprocessor [HAOS86] with 32 processors. The version with speculative pand is only slightly faster than the version with mandatory pand. The parallelism profiles in Figures 2, 3, and 4 are very informative. (The kernel is a parallel program; thus even the base sequential and version in Figure 2 has parallelism.)

The parallelism profile for sequential and in Figure 2 features three peaks of parallelism, each of which almost saturates the machine, separated by deep valleys of very little parallelism. The first peak in the mandatory pand version in Figure 3 is much longer due to all the rules that can fire in parallel. This additional parallelism mostly fills in the first valley. The increased computation delays the start of the second peak from 1.3 seconds to 1.6 seconds and changes the relative timing of computation so that the width of the second valley decreases. Thus the mandatory version fills in some of

Version	Inferencing time	
	Average	Std. Dev.
Sequential and	2.98	0.05
Mandatory pand	2.39	0.07
Speculative pand	2.28	0.13

Table 1: Emycin inferencing times (in seconds; statistics for 20 runs)

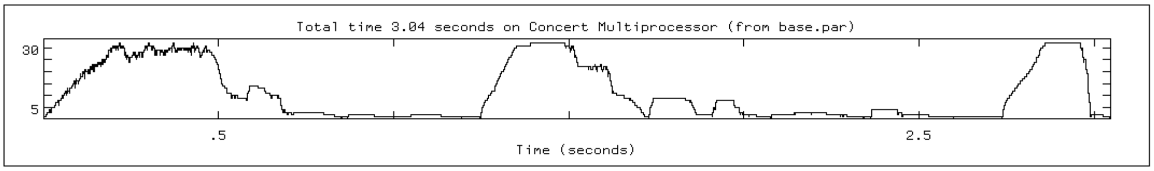

Figure 2: Parallelism profile with sequential and

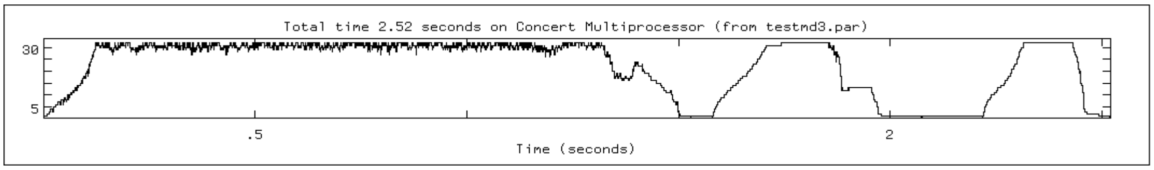

Figure 3: Parallelism profile with mandatory pand

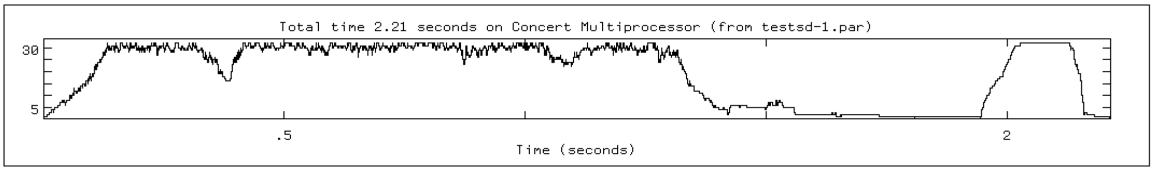

Figure 4: Parallelism profile with speculative pand

the valleys in the sequential version and thereby decreases the total execution time. In Figure 4 the speculative pand version completely fills in the first valley (though there is a little dip at 0.4 seconds about where the first valley previously started). This complete fill-in is due to aborting useless computation. The parallelism in the mandatory version only partly overlaps this valley because the useless computation lengthens the critical path. The result is that the first peak in the speculative version ends at about 1.3 seconds whereas the second peak in the sequential version ends at about 1.7 seconds. However, there is still a long sequential section in the valley, starting at about 1.6 seconds, with only one or two active rules. If this section with only one or two active processors were removed in all versions, the speculative version would be fastest by a significant margin. Although aborting useless computation does reduce the time until the end of the second peak, it does not significantly reduce the overall inferencing time for the gems database.

There are two reasons for this poor performance improvement. First, most conjunct tasks in the Emycin kernel are very short and thus the relative overhead of speculative pand is quite large. To separate the intrinsic merit of speculative computation from overhead artifacts, we artificially reduced the relative overhead of pand by adding delay loops to the four main procedures of the Emycin kernel. The speedup of the speculative pand version relative to the mandatory pand version increased with the delay amount, confirming the overhead factor, and reached an asymptote of about 15%.

The second, and more significant reason, is that many conjuncts are shared between ands with the gems database. Thus there is a great deal of dependency between pand invocations. For an artificial database with no shared conjuncts we observed about 50% speedup with speculative pand over mandatory pand.

Even with the poor performance improvement, the Emycin kernel with the gems database is a very interesting application of pand because many of the pands are nested and many conjuncts are shared among multiple pands. This nesting and sharing makes explicit termination checking too difficult (see Section 4.2.2) and makes outright killing of conjuncts unworkable. Finally, the short execution time of most conjuncts makes implicit reclamation ineffective: most useless tasks would simply run to completion before garbage collection occurred. (No garbage collection occurred during any of our runs.)

7.2 Boyer Benchmark

We wrote three versions of the Boyer Benchmark described in Section 3.3. All three versions were identical except for the parallelism construct in the rewriter, i.e. producer. The tautology checker, i.e. consumer, in these three versions uses mandatory

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 2: Boyer test cases

tasks created with `future`. Thus the tautology checker tasks run at top priority (*MAX*). The eager version uses `futures` in the rewriter (which run at *MAX* priority) and thus performs rewrites eagerly. The lazy version uses `delays` and thus performs rewrites lazily. The speculative version uses `spec-futures` with priority p[6] and thus performs rewrites eagerly subject to available processors. Thus the speculative version has two ordering levels: one for the tautology checker tasks and one for the rewriter tasks. If a tautology checker task touches a rewriter task, the rewriter task is promoted to priority *MAX*, thus providing the *dynamic* ordering that we mentioned in Section 3.3.

We ran these three versions on the Concert Multiprocessor with the three variations of *modus ponens* in Table 2 as inputs (which are all tautologies).

Table 3 shows the results. In every case, the speculative version is about the same or faster (and sometimes much faster) than both the eager and lazy versions.[7]

Table 4 makes the relative performance of the three versions clear. The speed of the speculative version relative to the speed of the eager and lazy versions displays two trends (though in opposite directions for each version). For the eager version, this speed ratio decreases as the number of processors increases and increases as the input size increases. As the number of processors increases with the eager version there are more "surplus" processors at every point in time to perform unnecessary rewrites, thus reducing the relative effect of unnecessary rewrites. As the input size increases the number of unnecessary rewrites increases, thus increasing the relative execution

[6]The priority p is unimportant as long as $0 < p < MAX$.

[7]The anomalies in the execution time of the eager and speculative versions for test-case2 and 32 processors are due to bus contention. The execution time for the eager version with test-case2 and 32 processors is also highly variable due to frequent garbage collection activity.

Input	Version	Number of Processors			
		8	16	24	32
tc2	eager	5.6	3.4	3.0	3.0
	lazy	8.0	7.9	8.0	8.2
	speculative	5.1	3.3	3.0	3.0
test-case	eager	39.8	22.1	16.3	15.1
	lazy	18.5	18.2	18.7	19.4
	speculative	14.8	10.6	10.3	10.3
test-case2	eager	335	185	170	211
	lazy	34.8	34.6	35.2	36.8
	speculative	30.2	28.1	27.8	30.6

Table 3: Execution time (in seconds) of Boyer Benchmark

Input	Version	Number of Processors			
		8	16	24	32
tc2	eager	1.1	1.0	1.0	1.0
	lazy	1.6	2.4	2.7	2.7
test-case	eager	2.7	2.1	1.6	1.5
	lazy	1.3	1.7	1.8	1.9
test-case2	eager	11	6.6	6.1	6.9
	lazy	1.2	1.2	1.2	1.2

Table 4: Speed of speculative version relative to speed of eager and lazy versions

time. As the number of processors gets very large, the execution time for the eager version should approach that for the speculative version. For the lazy version, the speed ratio increases as the number of processors increases and decreases as the input size increases. As the number of processors increases with the lazy version 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 of the lazy version should approach that for the speculative version.

Another way of looking at the results 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. 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 not 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.

7.3 Traveling salesman problem

We used two different programs to solve the traveling salesman problem using the same branch and bound algorithm. The qtrav program uses an explicit global priority queue (written in Multilisp) to achieve the desired ordering of node expansions. qtrav creates n worker tasks which continually pull nodes off this queue, expand them into child nodes, and re-insert them in the priority queue. The strav program instead uses

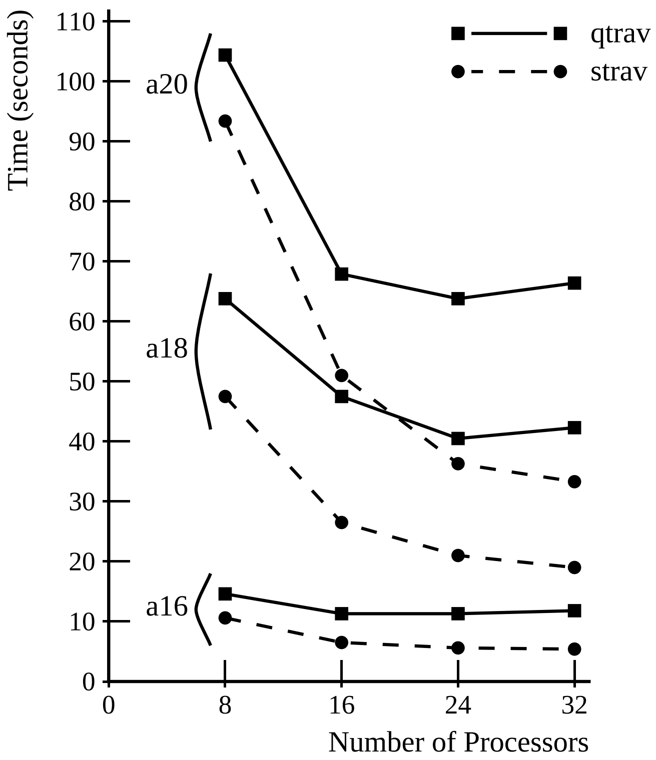

Figure 5: Traveling salesman results on Concert

the ordering mechanism for speculative tasks. It creates a speculative task for each node expansion with the priority set to effect the desired ordering of node expansions.

Figure 5 shows the execution time of qtrav and strav on the Concert Multiprocessor for three sets of cities, a16, a18, and a20, of 16, 18, and 20 cities respectively. The graph shows only the qtrav results for the optimum number of worker tasks.

The graph indicates that strav is always faster than qtrav. The difference in execution times increases as the number of cities and number of processors increases. 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 is caused by priority queue contention.

It is not surprising that strav is faster than qtrav given that strav essentially pushes the priority queue overhead down one layer of interpretation, from the language level to the implementation. strav provides more than speed. It also moves 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.

Consequently, strav is easier to program and understand than qtrav.

7.4 Eight-puzzle

The Eight-puzzle game is a popular children's game with eight tiles numbered 1 to 8 respectively arranged in some fashion on a three-by-three square game board, as in Figure 6. A move consists of sliding a numbered tile into the (adjacent) empty square. 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 7.

2	1	6
4		8
7	5	3

Figure 6: Starting position

1	2	3
8		4
7	6	5

Figure 7: Solution

We wrote four versions of a Multilisp program — one sequential and three parallel versions — to solve the Eight-puzzle. All the parallel versions create a task to examine each board position in the search tree. The first parallel version, which we call the mand version, uses mandatory tasks and thus cannot abort unnecessary computation. The second parallel version, which we call the spec version, uses speculative tasks all at the same priority — thus it lacks the ability to order computation. After the spec version finds the first solution, it aborts (i.e. stays) all the remaining search tasks. Thus the spec version is essentially like por. 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, implement the heuristic ordering

Search depth	Search version			
	sequential	mand (no staying)	spec (no ordering)	spec2 (heuristic order)
10	77 sec.	11.1 sec.	11.7 sec.	14.5 sec.
15	23 min.	192 sec.	crash	crash
20	?	23 min.	crash	54 sec.

Table 5: Eight-puzzle execution time

of board examination suggested in [Nil80]. 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, $W(b) = 7$ for the board in Figure 6. Boards are examined in order of increasing heuristic cost. Thus the spec2 version uses the built-in (distributed) priority queues to achieve a desired ordering, like the strav version in Section 7.3. The spec2 version is an example of simultaneous multiple-approach and ordering-based speculative computation.

Table 5 shows the time to find the first solution starting with the board position in Figure 6 for four different versions of game tree search. The shortest solution for this starting board is 18 moves [Nil80]. The indicated times are for eight processors on an Encore Multimax (Model 320).[8]

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 and crashed. The spec2 version did not exhaust the heap and was 26 times faster than the mandatory version![9] Thus careful control of the ordering reduced both the execution time and the memory requirements.

The mandatory version did not exhaust the heap because the scheduling of mandatory tasks encourages depth-first examination of the search space (see [Osb89]). The spec version, without ordering, encourages 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.

[8]Multimax is a trademark of Encore Computer Corporation.

[9]This superlinear speedup is a typical characteristic of multiple-approach speculative computation: the speedup depends on the position of the solution in the "subtrees" and the order in which these subtrees are searched.

If the search depth is insufficient to include a solution, the spec2 version is slower than the mandatory version. For a search depth of 10, the spec2 version is only 1.3 times slower. For a search depth of 15, the spec2 version exhausted the heap and crashed.

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 7.3, built-in support for ordering (in the form of priority queues) makes programming easier and improves efficiency by eliminating the layer of interpretation that would be associated with explicit ordering (i.e. priority queues) at the language level.

This application also demonstrates the importance of ordering for controlling 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 performance of the mandatory and spec2 versions for a search depth of 15. 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.

8 Conclusions

Speculative styles of computation exploit excess resources in an attempt to reduce the critical path length and hence execution time of applications. The success of this approach depends on exploiting excess resources *efficiently*. The key requirements for efficient speculative computation are:

- a means to reclaim unnecessary computation, and
- a means to control computation, i.e. control the allocation of resources to computation.

We discussed these requirements in the context of Multilisp and we presented a sponsor model for speculative computation which addresses these requirements. This model handles control and reclamation of computation in a single, elegant framework.

We implemented a subset of the sponsor model called the touching model. The touching model was intended as a simple prototype and consequently suffers from a number of deficiencies. The most serious 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

   Emycin demonstrates 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 1) the ability to control computation — with priorities for ordering, explicit computation reclamation (staying), and controller sponsors — and 2) 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 termination 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 about 2 with Boyer and a factor of 26 with Eight-puzzle.

Much work remains. In the short term we plan to add full controller sponsors and modularity to the implementation. In the longer term there are three main areas of work.

1. Computation control

   We need to further address the question: what control is desired? We need to study the optimal scheduling of speculative computation to determine the control mechanisms required and the ultimate benefit of speculative computation.

2. Applications

   We need to investigate many more applications to further assess the requirements and benefits of speculative styles of computation and the suitability of our model and language constructs.

3. Implementation

   We need to understand the fundamental costs of supporting speculative styles of computation. To address this, and facilitate experimentation with larger applications, we are (at the time of this writing) converting to a compiled implementation. We are integrating our sponsor model and support for speculative computation into the Mul-T [KHM89] parallel computing environment.

9 Appendix

In this appendix we describe two features of Multilisp used in this paper that we did not describe in Section 1.1.

Sometimes it is convenient to have a *future* without an associated task for write-once synchronization, like the I-structures in the dataflow language Id [Nik88].

(`make-future`) creates and returns an empty placeholder (*future*).

(`determine-future` *fut exp*) explicitly determines the undetermined placeholder (*future*) *fut* to the value of *exp*. It returns an unspecified value. Each task created by a `future` or `delay` ends with an implicit `determine-future`. It is permissible, though not encouraged, to explicitly determine a *future* created with `future` or `delay` with `determine-future`. It is an error to determine (explicitly or implicitly) a *future* more than once.

The following two atomic operations are extensions of the `set-car!` and `set-cdr!` 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` *pair new old*) performs the same `eq` check and possible swap as above atomically, but based on the `cdr` of *pair* rather than the `car`.

10 Acknowledgments

Bert Halstead provided numerous suggestions that contributed greatly to this work. We thank Ed Krall, Pat McGehearty, and the Microelectronics and Computer Technology Corporation (MCC) for permission to use the parallel kernel mentioned in Section 7.1. We also thank Laura Bagnall Linden for ParVis [Bag89], a wonderful program visualization tool which we used extensively during the experimental part of this work. Figures 2, 3, and 4 were extracted from ParVis displays.

11 References

[AS84] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* M.I.T. Press, Cambridge, MA., 1984.

[Bag89] L. Bagnall. ParVis: A program visualization tool for Multilisp. Master's Thesis, Electrical Engineering and Computer Science, M.I.T., January 1989.

[BH78] H. Baker and C. Hewitt. The incremental garbage collection of processes. A.I. Lab Memo 454, M.I.T., March 1978.

[Bur85] F. W. Burton. Speculative computation, parallelism, and functional programming. *IEEE Trans. on Computers*, pages 1190–1193, December 1985.

[BZ89] R. Bubenik and W. Zwaenepoel. An operational semantics for optimistic computations. Technical Report TR89-85, Department of Computer Science, Rice University, February 1989.

[CG87] K. Clark and S. Gregory. Parlog: Parallel programming in logic. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, Volume 1, Chapter 3. M.I.T. Press, 1987.

[CSM88] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Int'l. Conf. on Fifth Generation Computer Systems*, pages 230–251, 1988.

[Eps89] B. Epstein. Support for speculative computation in MultiScheme. Bachelor's Thesis, Brandeis University, May 1989.

[Gab85] R. Gabriel. *Performance Evaluation of Lisp Systems.* M.I.T. Press, 1985.

[GG89] R. Goldman and R. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[GP81] D. Grit and R. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Trans. on Prog. Languages and Systems*, pages 49–59, October 1981.

[Hal85] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[Hal86a] R. Halstead. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, December 1986.

[Hal86b] R. Halstead. Parallel symbolic computing. *IEEE Computer*, pages 35–43, August 1986.

[HAOS86] R. Halstead, T. Anderson, R. Osborne, and T. Sterling. Concert: Design of a multiprocessor development system. In *13th Annual Symp. on Computer Architecture*, pages 40–48, June 1986.

[HK82] P. Hudak and R. Keller. Garbage collection and task deletion in Distributed Applicative Processing Systems. In *Proceedings 1982 ACM Conf. on Lisp and Functional Prog.*, pages 168–178, 1982.

[KH81] W. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[KHM89] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *SigPlan Conf. on Prog. Language Design and Implementation*, pages 81–90, 1989.

[KM86] E. Krall and P. McGehearty. A case study of parallel execution of a rule-based expert system. *Int'l. Journal of Parallel Programming*, pages 5–32, January 1986.

[LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, pages 419–429, June 1983.

[Lus88] E. Lusk et al. The Aurora Or-parallel Prolog system. In *Proc. of Int'l. Conf. on Fifth Generation Computer Systems*, pages 819–830, 1988.

[Mil87] J. Miller. MultiScheme: A parallel processing system based on MIT Scheme. Technical Report TR-402, Laboratory for Computer Science, M.I.T., September 1987.

[Nik88] R. Nikhil. Id (version 88.0) reference manual. Computation Structures Group Memo 284, Laboratory for Computer Science, M.I.T., March 1988.

[Nil80] N. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann, 1980.

[Osb89] R. Osborne. Speculative computation in Multilisp. Technical Report TR-464, Laboratory for Computer Science, M.I.T., November 1989.

[PD89] A. Partridge and A. Dekker. Speculative parallelism in a distributed graph reduction machine. In Bruce Shriver, editor, *Proceedings of 22nd Annual Hawaii Int'l Conf. on System Sciences*, Volume 2, pages 771–779, 1989.

[Sha87] E. Shapiro. Concurrent Prolog: A progress report. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, Volume 1, Chapter 5. M.I.T. Press, 1987.

[Sol89] R. Soley. On the efficient exploitation of speculation under dataflow paradigms of control. Technical Report TR-443, Laboratory for Computer Science, M.I.T., 1989.

[Ued87] K. Ueda. Guarded Horn clauses. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, Volume 1, Chapter 4. M.I.T. Press, 1987.