

reprinted from

PARALLEL COMPUTING '91

Proceedings of the International Conference on
Parallel Computing '91, London, U.K., 3-6 September 1991

Edited by

D.J. Evans

Department of Computer Studies
University of Technology
Loughborough, Leicestershire, U.K.

G.R. Joubert

Aquariuslaan 60
5632 BD Eindhoven, The Netherlands

H. Liddell

Queen Mary and Westfield College
University of London
London, U.K.



1992

NORTH-HOLLAND
AMSTERDAM • LONDON • NEW YORK • TOKYO

Throttling Speculative Computation: Issues and Problems

Doug DeGroot
Texas Instruments
Computer Science Center
degroot@csc.ti.com

1. INTRODUCTION

Two of the lesser studied aspects of parallel processing are those of throttling and speculating. Loosely speaking, throttling may be viewed as a way of keeping dynamic parallelism under control, while speculating is a way of letting it get a bit out of control. It is frequently possible to significantly increase parallelism through less throttling or more speculation, and in certain application domains, we may want to do some of both. (In fact, there is at least one parallel computation model that *requires* both if any speedup is likely to occur, as is shown.) Unfortunately, however, the greater parallelism due to a lessening of throttling or to an increasing of speculation with the parallelism may lead to either significantly increased execution speedup or to significant execution slowdown, and surprisingly, it is only sometimes possible to predict ahead of time which will occur. Both throttling and speculation by themselves appear challenging but not overly difficult to implement. However, when speculative parallelism is employed within models of dynamic parallelism, throttling of speculative parallelism becomes necessary, and this appears impressively difficult. Problems with throttling are first described, then problems with speculative parallelism are described, and finally problems with throttling speculative parallelism are described. These problems are described within the dynamic parallelism execution models of "any solution", and-parallel, logic programming execution models. Such models involve inherently speculative computation. As such, they constitute a natural environment within which to study the effective combinations of throttling and speculation.

2. DYNAMIC PARALLELISM

There are a number of models of parallelism in which the potential for parallelism can be identified and/or extracted only during run-time. In these models, the amount of parallelism, the locations of the parallelism, and the types of the parallelism are all data-value dependent. These models of parallelism can be referred to as *dynamic* parallelism models.

Although there are many such models of parallelism, the most widely studied have been those dealing largely with non-numeric computing. Examples include parallel models of Lisp, logic programming languages, functional programming languages, and dataflow programming languages. In these models, parallelism is often extracted during run time through recursion, nesting, and looping. However, because the depth of recursion or the number of times a loop is iterated is usually data dependent, the compiler frequently cannot determine how many parallel tasks will be created during execution of the loop or recursion. (Nested expressions are less difficult to analyze, but they too present problems.)

In these dynamic models, the compiler may be able to determine (through data dependency analysis, for example) the code locations where parallelism *may* occur but not where it *should* occur. The compiler cannot, in general, determine the number of parallel code segments that will arise during program execution, the sizes of the parallel code segments, the beginning and end times of a parallel code segment, or even onto which processors to map a code segment.

Because of the inability of the compiler to determine the number of parallel code segments that will occur at any point of the execution, it is impossible to predict the resource requirements of the program. It is possible that parallel execution of certain applications may in fact produce so many tasks that the machine simply runs out of memory, out of task control blocks, or out of some other limited but necessary resource. In the worst case, the program may simply abort. Even if the program does not run out of required resources, the additional overhead involved in scheduling the numerous tasks and balancing the load across the multiple processors may so overwhelm the system that performance is degraded, even to the point of achieving overall slowdown rather than speedup. Consequently,

when dynamic parallelism is exploited, significant reliance is placed on the run-time system to assist the compiler in extracting, scheduling, and controlling the parallelism.

3. THROTTLING

Because the run-time systems of dynamic parallelism models are forced to share the responsibility of deciding when and where to invoke parallelism, there has been significant historical emphasis on designing run-times that can provide efficient support for rapid task creation, task destruction, and dynamic load balancing of tasks. More recently, however, some work has begun to investigate the incorporation of a "throttling" mechanism into the run-time systems of dynamic parallelism models.

The primary goal of throttling is preventing a dynamically unfolding parallel execution from creating so many parallel code segments that either: 1) the system runs out of resources with which to manage the parallel segments, 2) the system begins to thrash as a result of the increased process management and communication requirements, or 3) the system simply wastes CPU time creating extra code segments which end up being executed sequentially anyway.

This aspect of throttling is responsible for ensuring that "too much" parallelism does not occur; it allows all processors to become busy, sets up a pool of "spare" work, and then "shuts down" the creation of parallelism until more work is needed to replace previously completed work, at which point it turns creation of parallelism back on. It is a bit related to the job of the medium-term scheduler of a multiprogramming system whose job is to control the degree of multiprogramming for similar reasons.

The secondary goal of throttling is properly controlling the reduction of granularity in self-decomposing parallel procedures. Here, the throttler is responsible for ensuring that tasks do not decompose into two or more tasks of too small a size. Because there is a not insignificant cost involved in setting up, distributing, and then reclaiming the resources of the parallel code segments, it must be ensured that this cost does not exceed the performance gains returned by having executed the code segment in parallel in the first place. This aspect of throttling is not yet well understood, and it has certainly received far less attention.

Several software approaches to throttling have been explored recently, at both the system level and the application level [Bush, Roberts, Watson, Tucker]; some report as many failures in solving the problem as successes [Ruggiero]. Certainly it can be easily demonstrated that programmer control of throttling through program annotation is impractical [Goldman]. A significantly smaller number of hardware approaches to throttling have been examined; these too have met with occasional failure [Ruggiero]. It is generally agreed that only appropriate combinations of both hardware and system-level software control appear capable of effecting satisfactory throttling over a wide domain of different problem types and problem sizes [Ruggiero, Sargeant].

4. SPECULATIVE PARALLELISM

Speculative parallelism involves the parallel execution of code segments which are not yet known to be required. For example, consider an "if" statement:

```
if conditional then
    consequent
else alternate
```

An "if" statement imposes a conditional flow of control on a program's execution. If control reaches an "if" statement, the conditional must be evaluated in order to determine whether to execute the consequent or the alternate. Because of this, traditional parallel processing approaches use conditional branch instructions (such as "if" statements) as places to synchronize, and indeed serialize, execution. An "if" statement is thus classically viewed as an inherently sequential control flow construct.

Surprisingly, it is not necessary to execute an "if" statement sequentially. Instead, we may choose to execute the conditional, consequent, and alternate all at once, in parallel. Then, once the result of the conditional becomes known, it is known which of the two paths is required to complete the execution. However, because execution of that path has already begun, speedup may possibly be

obtained by being able to complete its execution sooner than if we had waited until evaluating the conditional to begin executing the required path.¹

Until the conditional test completes, neither path can be said to be *required*. Thus invoking their executions is somewhat premature and speculative. Work performed before it is known to be required is called *speculative work*, and parallelism resulting from executing speculative work is called *speculative parallelism*. In the case where both the consequent and alternate are executed in parallel, both are speculative work until the conditional is complete. Once the test is complete, however, one of the two speculative pieces of work becomes *necessary* (or, *mandatory*), while the other is now known to be *unnecessary*. Any work expended on the unnecessary path has become *wasted work*. Work expended on the now necessary part has not been wasted, although if evaluation of this necessary part is not yet complete, it may itself be engaging speculative parallelism to speed up its self-evaluation.

Clearly, speculative work can be engaged in hopes of obtaining execution speedups greater than possible without it. Ensuring this is very difficult, however, as will be shown; and if not properly done, it can result in significant slowdown of an execution or even deadlock [Burton, Finkel]. Consequently, many forms of speculative parallelism, particularly parallel "if" statements, have been shied away from [Page]. Because speculative parallelism involves parallelizing what have traditionally been considered to be inherently sequential control constructs, we can measure the success of our efforts (e.g., speedup) against the non-speculative execution of the same program.

5. TYPES OF SPECULATIVE PARALLELISM

Speculative parallelism can be exploited in two basic forms: parallel search and parallel test. Parallel search techniques have been studied for years, but not always within the speculative framework. "All solutions" parallel searches, as typified by many parallel relational or logic database query schemes, for example, are usually non-speculative. On the other hand, when only one solution to a problem or query is desired, as when finding some aspect of an object that satisfies some abstract property, or when parsing a natural language phrase, for example, parallel searches often prove speculative.

Parallel test, as a form of speculative parallelism, seems to have almost been ignored. Unlike speculative search, which evaluates a number of disjoint, alternative execution paths simultaneously, speculative test focuses on only one continuous execution sequence. It bets totally on the selected sequence and attempts to preprocess its parts even before they are known to be needed. Speculative test may be applied when speculative search is inappropriate, or in addition to speculative search.

Consider a complex "if" statement, such as that shown in Figure 1. Using speculative parallelism, it is possible to execute any number of the conditionals, consequents, or alternates in parallel. For example, if we assume that A will succeed, we may wish to execute A and B in parallel; if we assume that A will fail, we may execute A and H in parallel;

and if we are unable or unwilling to make an assumption about A's outcome, we may choose to execute A, B, and H all in parallel. Of course, any combination of conditionals might be executed in parallel, such as A, F, and R, although it is far from clear when this would make sense. Clearly, some combinations make more sense than others.² Two of these are now defined.

First, any number of conditionals along a directly connected path of consequents, possibly in conjunction with the final consequent at the end of the path, is defined as a *speculative test*. In the

```

if A then
  if B then
    C
  else if D then
    E
  else if F then
    G
else if H then
  if J then
    K
  else if L then
    M
  else if N
    then P
else if Q then
  if R then
    S
  else if T then
    U
  else if V then
    W

```

Figure 1
A Complex "if" Statement

¹ Of course, data dependencies must still be respected.

² but see [Steinberg] for an interesting example

example above, the possible speculative tests include the following expressions (plus any prefix or suffix containing two or more subexpressions):

A, B, C	N, P
D, E	Q, R, S
F, G	T, U
H, J, K	V, W
L, M	

Second, any number of conditionals along a directly connected path of alternates, possibly in conjunction with the final alternate at the end of the path, is defined as a *speculative search*. In the example above, the possible speculative searches include the following expressions (plus any prefix or suffix containing two or more subexpressions):

A, H, Q	J, L, N
B, D, F	R, T, V

Speculative tests can be engaged to quickly test whether the final consequent in a chain is a valid solution, while speculative searches can be engaged to explore with equal attention and speed a set of mutually exclusive alternate solutions. (In logic programming, speculative test is embodied in "don't know" and-parallelism, while speculative search is embodied in "don't know" or-parallelism.)

These are not the only types of speculative parallelism — several more obvious types are described in [Soley], [Osborne], and [Burton]. Instruction prefetching beyond unexecuted branch instructions by a CPU is a form of micro-level speculative parallelism. Another, less obvious, but interesting form of speculative parallelism is found in the Time Warp model of discrete-event simulation [Jefferson].

6. BOOLEAN AND-EXPRESSIONS

Another common conditional control-flow expression is the boolean *and* and its variants. For example, in Common Lisp, the expression

```
(and form1 form2 . . .)
```

is defined as follows

"(and *form1 form2 . . .*) evaluates each *form*, one at a time, from left to right. If any *form* evaluates to *nil*, the value *nil* is immediately returned without evaluating the remaining forms. If every form but the last evaluates to a non-*nil* value, *and* returns whatever the last form returns." [Steele]

Because the failure of one form (i.e., its evaluating to *nil*) precludes the evaluation of all following forms, this type of and statement has been called a "short circuiting" and. The effect is no different than if

```
(and A B C)
```

were written as

```
if A then
  if B then
    return C
  else return nil
else return nil
```

From this viewpoint, it is apparent that the short-circuiting boolean-and expression is classically a sequential expression. If we choose to execute it in parallel, using a parallel-and expression such as MultiLisp's *pand* [Halstead], we simply write

```
(pand A B C).
```

Now, instead of executing sequentially, A, B, and C execute in parallel (barring side-effects and data dependences, of course). However, the resulting parallelism is clearly speculative parallelism.³

Suppose A, B, and C begin executing in parallel and that C fails (evaluates to nil) before the evaluation of either A or B is complete; if this occurs, some execution models might allow the failure of C to short circuit *backwards* the execution of A and/or B. In such models, the evaluation of the *pand* expression might consume less time/resources than the sequential execution would, thereby possibly yielding superlinear speedup (see Section 10). Note, however, that special care must be taken to properly handle side-effects [DeGroot87]. Also note that when short-circuiting occurs during parallel execution, the tasks to be short-circuited should be tracked down and killed to prevent them from consuming additional resources.

7. HORN CLAUSES AND AND-PARALLELISM

Given a Horn-clause logic programming statement such as

$$f(X) :- g(X), h(X), k(X).$$

we can read this declaratively as, "f(X) is true if g(X) and h(X) and k(X) are all true. [Kowalski]" We can also read this procedurally as, "To prove f(X), prove g(X) and prove h(X) and prove k(X)." In the declarative reading, we interpret the word "and" traditionally, i.e., as inclusive and not as short-circuiting. However, the procedural definition is less clear and must be interpreted within the context of a given execution model. For Prolog [Sterling], for example, we interpret the "and" as a short-circuiting "and", with execution proceeding from left to right. As a result, Prolog is not guaranteed to execute all three subgoals in the clause above. As soon as one is found to fail, execution backs up to the previous subgoal, if any, to try again. If the first subgoal ever fails, either initially or upon being backtracked into, the whole clause is known to be false; execution of the clause then terminates, and backtracking is invoked, even if there remain unexecuted subgoals to the right of the failing subgoal(s).

Because the "and" in the Prolog procedural definition is a short-circuiting "and", the meaning of the above clause is approximately equal to the following "if" statements:⁴

<pre> if g(X) then if h(X) then if k(X) then f(X) else false else false else false </pre>	or	<pre> L1: A = g(X) if (A ≠ 'fail') then { L2: A = h(X) if (A ≠ 'fail') then { A = k(X) if (A ≠ 'fail') then return true else unwindto L2 } else unwindto L1 } else unwind </pre>
---	----	--

Considering this clause, it is interesting to try to identify the potentials for parallelism. From the classical point of view, there is none. However, as described above, we can clearly evaluate this clause with speculative-test parallelism, evaluating g(X), h(X), and k(X) in parallel. Doing so constitutes the basic approach of most *and-parallelism* execution models of logic programming [Conery, DeGroot84, Hermenegildo, Ci, etc.].

There are two general methods of execution of logic programs: 1) "all solutions", in which every possible solution (actually, every possible method of producing a solution) is found, and 2) "any solution", in which only the "first" solution found is desired. "All solution" execution models must search the entire execution tree, so they exploit very little speculative parallelism. Most and-parallelism models proposed have been any-solution models; because they need search only one part of the search tree containing a solution, and because that part is not generally known before execution

³ unless it is known that A and B cannot fail; see Section 17.

⁴ Backtracking and unification, of course, make this equivalence less obvious.

begins, they fundamentally exploit speculative parallelism. These any-solution, and-parallelism execution models are the ones being considered here.

Because these models are inherently speculative, they provide a unique parallel execution model with which to study some of the fundamental issues of invoking and controlling speculative parallelism. The lack of other major control constructs in logic programming makes the languages devoid of other parallel execution issues, allowing all speedup arguments to be centered squarely on the success or failure of exploiting speculative parallelism.

8. DYNAMIC PARALLELISM WITHIN AND-PARALLEL LOGIC PROGRAMMING

As noted above, the backtracking execution model of sequential Prolog makes it impossible to ascertain how much work is represented by a given logic programming clause. The compiler cannot tell which goals of the clause, other than the first, will be executed; nor, due to backtracking, can it tell how many times a particular subgoal within the clause will be executed. Further, because the execution semantics of each subgoal are defined via other clauses, we can in general make no claims about the amount of work represented by each subgoal (but see [Tick] or [Debray], for example).

To exacerbate the problem, the compiler cannot, in general, even determine which subgoals in a clause may execute in parallel. It can only tell which possible parallel executions might result, depending on certain data-dependencies existing at run-time. As examples of the types of execution graph expressions generated for typical Prolog clauses, consider the following examples. The clause:

$$f(X) :- g(X), h(X), k(X).$$

may be compiled into the execution graph expression [DeGroot84]:

$$f(X) :- (GPAR(X) g(X) (GPAR(X) h(X) k(X))).$$

The clause:

$$f(X,Y) :- p(X), q(Y), s(X,Y), t(Y).$$

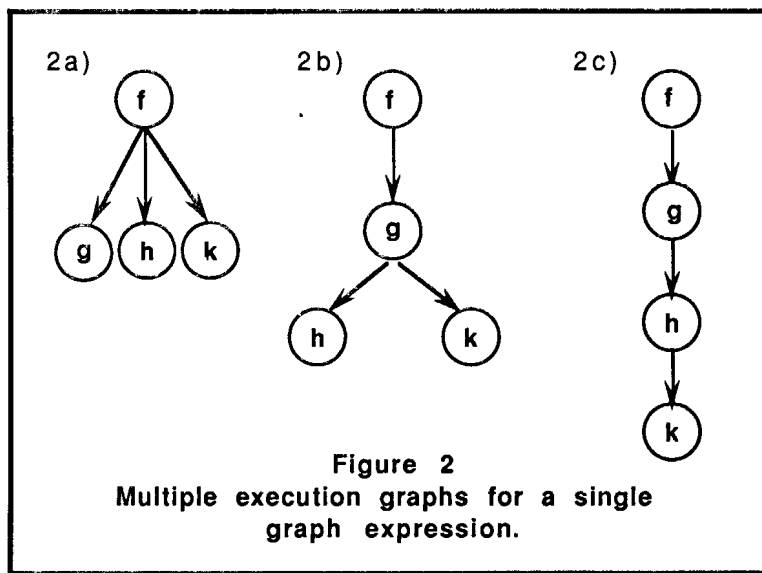
may be compiled into the execution graph expression:

$$f(X,Y) :- (GPAR(X,Y) (IPAR(X,Y) p(X) q(Y)) (GPAR(Y) s(X,Y) t(Y))).$$

or even into:

$$f(X,Y) :- (GPAR(X) p(X) (GPAR(Y) q(Y) (GPAR(Y) s(X,Y) t(Y)))).$$

Even though every Prolog clause is compiled into a single execution graph expression, and this is done at compile-time, the actual execution graph may vary considerably depending on the results of any IPAR or GPAR tests embedded within it. This is a significant advantage, as a single compile-time expression can represent a number of possible dynamic, run-time execution graphs. For example, the first execution graph expression above can actually result in any of the run-time execution graphs shown in Figure 2.



Because recursion is heavily used in logic programming languages, it is also quite easy to have a given clause repeatedly invoked, perhaps with smaller and smaller data segments on each invocation. The number of parallel code segments resulting from parallelizing such a recursive clause can be a function of both the size of the data segment and the particular values of the data elements within the data segment.

For example, consider the simple naive Fibonacci procedure shown in Figure 3. When this procedure is invoked, a recursive execution tree results with size $O(N^2)$, where N is the value of the data element passed into the procedure. Parallel execution of this procedure will result, then, in $O(N^2)$ parallel tasks. If N is very large, this number of tasks may overcome the system and cause either erroneous termination or thrashing so serious that parallel execution speed becomes slower than sequential execution speed. Clearly some form of throttling is required here.

```

fib (1,1) :- !.
fib (2,2) :- !.
fib (N,V) :- A is N-1, B is N-2,
             fib(A,S1), fib(B,S2),
             V is S1+S2.

```

Figure 3
Naive Fibonacci

If N is fairly small, say even as small as 10, the size of the resulting tasks may be too small to warrant incurring the costs of creating the tasks, distributing the tasks, and collecting their results. But even if 10 is not too low a value, the next clause invocations will not be with the value 10 but will be with the values 9 and 8, the next ones will be with the values 8, 7, and 6, and so on. So execution will eventually reach the point where the work to be expended in executing the task is less than the work expended in creating and managing the task, and slowdown will have been introduced into the parallel execution.

Here too we would like the throttler to kick in and prevent this slowdown. We can refer to this problem as the *minimal granularity problem*. Clearly, execution speed will be enhanced if we can stay far from the minimally acceptable grain size. This is because the ratio of work returned by the parallel task to the work expended in setting up the task indicates the gain to be expected from spawning the parallel task. If the ratio sinks below 1, a processor introduces more overhead into the execution than it saves by spawning off the extra work. If the ratio is 1, we might expect a break-even behavior, but this is really far from certain, as the spawned task occupies resources that might have been available for some other very large task to use, bus bandwidth and latency might not have become so high, etc. At ratios greater than 1, we begin to hope for an overall reduction in execution speed, and the greater the ratio, the greater our expectations. However, note that no matter how high the ratio, it may make little sense to spawn additional tasks if all processors are already sufficiently loaded, and in fact, doing so may hurt performance. As mentioned, most work on throttler mechanisms has failed to adequately address this *minimal granularity* problem.

9. SPEEDUP

Speculative parallelism can be used when there are idle resources and the task's currently executing set of code is not expected to yield additional, mandatory parallel threads of execution. Then, in order to both more fully utilize the available resources and to possibly complete one of the tasks faster than normal, parts of the execution sequence may be activated before it is known whether control will ever reach those parts. Speculative parallelism is thereby employed.

When properly performed, speculative search is guaranteed to be wasting part of its efforts. Speculative test, on the other hand, when properly performed, may actually result in no wasted work, although this would be rare. Most importantly, speculative search and speculative test techniques can both lead to either incredible speedups or to incredible slowdowns, and again, it is difficult to predict ahead of time which will occur, unless, of course, certain precautions are taken in the parallel execution of speculative work. While this sounds simple, it is in fact quite difficult to ensure in some dynamic parallelism models; at a minimum, an effective throttling mechanism is required to pull it off. Coupling throttling intimately with speculative parallelism offers a unique approach to achieving execution speedups in dynamic parallelism execution models greater than with traditional approaches. However, the effective combination of the two presents major challenges.

Given a logic program, we can consider many different sequential and parallel execution models for evaluating the program; some of these will incorporate deterministic execution orderings, and some

will incorporate nondeterministic orderings.⁵ For a given program and input data set, each deterministic model results in a particular execution time and a particular set of required work; but each nondeterministic model results in a *set* of execution timings and a *set* of different sets of required work. It is thus meaningful to ask for the *specific* run time of a deterministic model, but for the *average* run time of a non-deterministic model.

Given that the goal of parallelism is speedup, it is meaningful to ask, "Speedup with respect to what?" At a minimum, one would expect the average parallel execution time to be faster than any *specific*, deterministic, sequential execution time. In fact, it is not unreasonable to expect a parallel execution model to deliver faster execution times for *any* parallel execution compared to any specific, deterministic, sequential execution. Furthermore, for "any solution", "don't know" and-parallel models, we expect the parallel execution to be faster than the sequential for each answer demanded [Saletore]. Finally, we would also hope for high probabilities that the parallel execution would be faster than the *average* run time exhibited by any nondeterministic sequential execution.

There are clearly a number of meaningful comparisons that can be made to evaluate the effective speedup of a given parallel execution model [Eager]. In particular, for logic programming models, one obviously meaningful comparison is the speed of a selected program using the parallel logic programming execution model compared to the speed resulting from the use of some well-known, high-performance sequential Prolog. In fact, this particular definition of speedup is the desired one, but one that is frequently "avoided".

In addition, one would hope that a new model of parallel execution would exhibit better speeds than other parallel execution models. Nondeterministic parallel execution models that do not yield single execution times for given programs and given input data sets cannot easily be compared to other parallel execution models, especially if these other execution models are also nondeterministic parallel execution models. ("Toy" benchmarks do not count, as they are not representative of the types of problems for which we are going to all the trouble. It is the very large applications that are most likely to exhibit large variances in run times under non-deterministic executions. With such applications, it can be very costly to obtain the average run time, much less compare them to alternative approaches.)

The remainder of this paper considers only parallel execution models that exhibit deterministic run-times (or at least run-times that exhibit small variances). Speedup is defined as the ratio of the time required by the best sequential algorithm using a sequential execution model to the required time of the parallel algorithm running under the parallel execution model. In other words, for N processors, we define the speedup using these N processors as speedup(N), which is

$$S(N) = \frac{\text{best deterministic sequential speed}}{\text{parallel speed on N processors}}$$

or perhaps even as

$$S(N) = \frac{\text{average non-deterministic sequential speed}}{\text{parallel speed on N processors}}$$

Because it has so often been incorrectly applied and has been noted as being somewhat "less honest" [Finkel], we are not interested here in the more frequently used speedup equation

$$\text{speedup} = \frac{\text{speed of parallel code on 1 processor}}{\text{speed of parallel code on N processors}}$$

Clearly, we need speedup to be greater than 1, and the greater the better. If S(N) is less than 1, the parallel execution is slower than the sequential execution, and slowdown will have occurred rather than speedup. A great many parallel logic programming schemes reported in the literature are capable of exhibiting unpredicted slowdown.

⁵ The determinism of a program execution should not be confused with the determinism of the program itself.

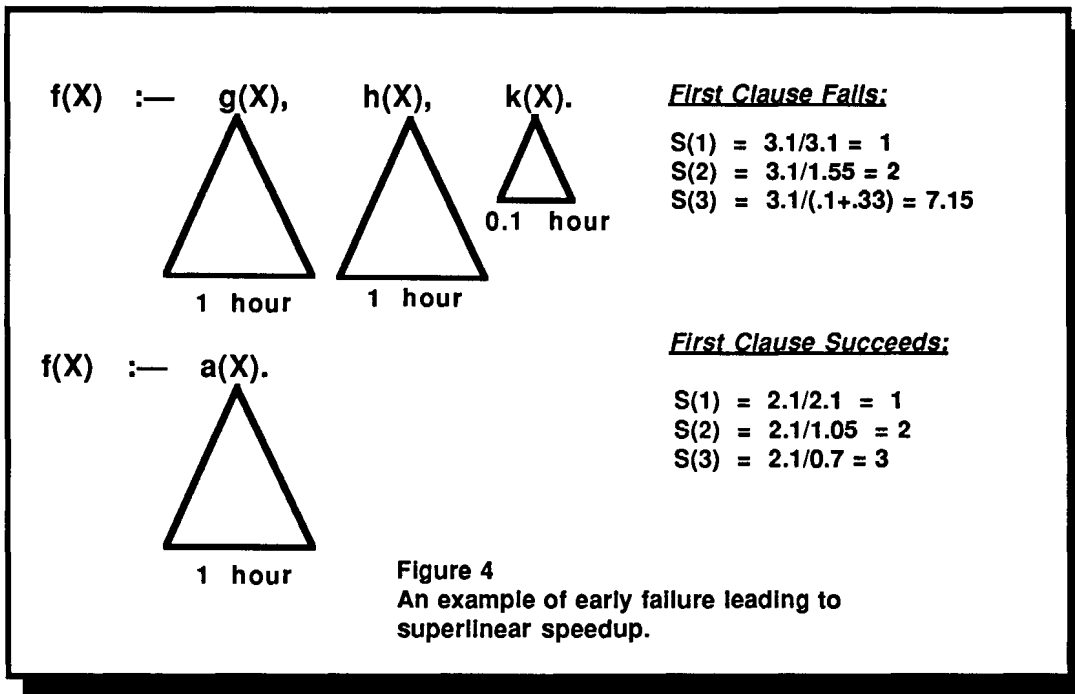
10. SUPERLINEAR SPEEDUP

If $S(N)$ is greater than N , the result is *superlinear* speedup. Such speedups have generally been called *speedup anomalies* [Li, Finkel, Lai], though they can be a direct result of speculation, and thus from a speculative point of view, should be considered quite normally achievable. In and-parallelism, superlinear speedups can occur as a result of *early failure detection* of a clause. For example, consider Figure 4 and the clauses

$$f(X) :- g(X), h(X), k(X).$$

$$f(X) :- a(X).$$

If upon entry into f the variable X is ground, all three subgoals of the first clause can execute in parallel [DeGroot84]; doing so is clearly speculative, however, as has been shown. In any event, assume that g , h , and a all succeed with the given value of X , and that all three require 1 hour each to compute. Further, assume that k requires 0.1 hours to compute and that it fails right near the end of its computation. Also assume that sufficient resources exist to compute all three in parallel with little or no interference from each other. Finally, assume that perfect linear parallelism is obtainable within each subgoal when extra processors are available.



The normal, left-to-right, sequential execution of the first clause consumes 2.1 hours of processing, only to fail at the end. Execution then resumes at the next clause, at which time 1 more hour is consumed executing subgoal a . The total sequential execution time required is 3.1 hours. As listed in Figure 4, however, the time required for 2 processors is only 1.55, yielding a speedup of 2, under the above assumptions of perfect parallelizability and no interference. For 3 processors however, the time required is only $\cdot 1 + \cdot 333 = \cdot 433$; this gives a speedup of 7.15 for only three processors.

To better see this, when three processors are available, we may assign processor 1 to solving $g(X)$, processor 2 to solving $h(X)$, and processor 3 to solving $k(X)$. While processors 1 and 2 are busily

solving their subgoals, processor 3 suddenly (after 0.1 hours) announces failure of $k(X)$. Now, regardless of the outcome of processor 1's or 2's efforts, the whole clause must fail due to its conjunctive nature. Processors 1 and 2 may therefore be immediately interrupted, using backwards short-circuiting (assuming they have no pending side-effect operations to complete). Execution then proceeds to the next clause, whereupon all three processors cooperate to execute in parallel the subgoal a in 1/3 hours. The early failure of $k(X)$ reduced the real time expended in the first clause by 0.9 hours, or 90% of the full time, and it saved 1.8 hours of processor time.

Unfortunately, if the first f clause succeeds, superlinear speedup does not occur, as the critical path of the longest computation continues to dominate the time requirement; thus superlinear speedups are most likely to occur in what are considered nondeterministic application codes that exhibit early failure detection. Additionally, if the first subgoal (here, it is g) is the shortest instead of the last subgoal (k), then superlinear speedup will again not occur. Finally, if the goals are all speculative computations that turn out to be wasted, no speedup occurs period. Clearly superlinearity exhibits itself only under certain circumstances in a typical logic program.

The point of greatest interest is that the chances of encountering superlinear speedup using speculative test are greatest when a goal to the right of the mandatory goal (the farther away the better) is allowed to execute in parallel with the mandatory goal, and that goal fails before it has become mandatory; the sooner it fails before becoming mandatory, the greater the speedup, as it will have prevented greater amounts of work. The above example is slightly misleading, as it is not necessary for the rightmost goals to be the shortest — they should just fail earlier. So using granularity analysis to predict early failure is the wrong approach — failure analysis is needed [Stone].

11. SLOWDOWN

Unfortunately, due to the speculative nature of "any solution", and-parallel execution models, there are also numerous ways in which slowdown can occur in a parallel execution of a logic program. Again, consider the clause:

$$f(X) :- g(X), h(X), k(X).$$

Suppose that on some particular clause invocation $g(X)$ is doomed to failure. Then in a sequential execution model, control will never reach h or k . The amount of work expended by the sequential execution is limited to the (possibly partial) execution of g .

A parallel execution model, however, might fire off all of g , h , and k in parallel. If h and/or k rapidly unfold and generate a great many parallel subprocesses, these processes may be spread about the system (by the dynamic load balancer, for example), with some being placed on the same processor executing g . This can lead to slowdown if any of the following happen:

1. If g 's processor time-slices its collection of active processes, where some of these processes are not descended from g , then g will be slowed down by the ratio of the total number of tasks to the number of tasks descended from g .
2. If g (or one of its descendants) blocks for I/O or memory management, a process descended from either h or k may then be scheduled onto g 's processor. When g (or its descendant) again becomes ready, the processor may not preempt the other process for some time, perhaps until its time-slice runs out. The progress of g is delayed during this time.
3. Assume g begins executing on processor P , unfolds quickly, and spreads out across the system. Processor P keeps some of the descendant processes, but the remainder are distributed among the other processors. Assume that the tasks kept by processor P complete; then the root of the task tree for g attempts to collect the partial results and return a final result. Suppose now that at least one of the partial results is not yet computed on one of the remote processors. At this point, processor P has no other work to do and so likely will be assigned a descendant task from one of the other processors. This task might in fact, be a descendant of g , but it might also be a descendant of h or k . Suppose it is descended from h . Processor P then begins executing this descendant of h . If and when the final pieces of g complete, P may no longer be in a position to combine all the partial results of g due to its paying attention to its newly received task from h or to

the memory pages belonging to g having been swapped out. Then clearly the report of failure of g is again delayed.

Each of these scenarios is offered as a quite realistic possibility and not as some sort of anomalous behavior. There are obviously a great many other similar scenarios in which the execution of g can be slowed relative to the speed which would occur in a deterministic sequential execution. What happens in each of these is *late failure detection* — the failure of a subgoal is delayed by speculative subgoals to its right.

Interestingly, both superlinear speedup and slowdown are achieved in "don't know" and parallelism the same way — through speculative-test parallelism. While guaranteeing speedup is in general impossible, whether superlinear or not, it is fortunately not impossible to guarantee the absence of slowdown; it is, however, very difficult.

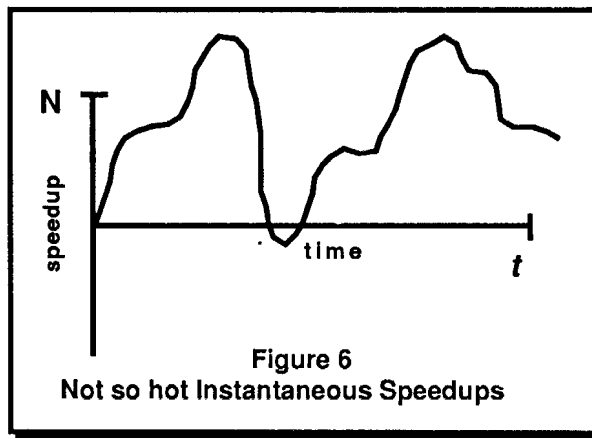
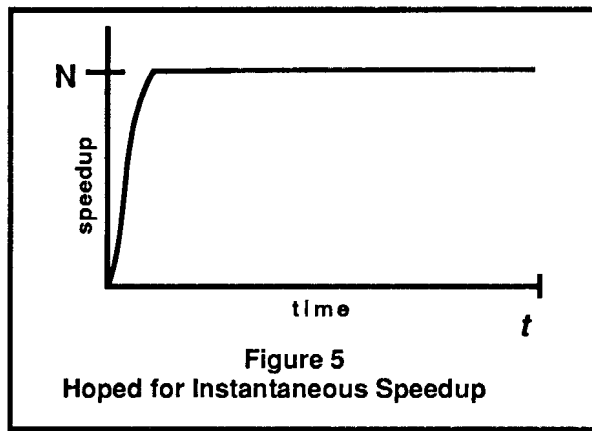
12. INSTANTANEOUS SPEEDUP

Normally, speedup figures are reported as the *average* or *final* speedup; in other words, the total time required to complete the parallel execution is compared to the total time required to complete the sequential execution. It is useful to also consider the speedup obtained over time. Let $c(t)$ be the code location in the program that the sequential execution has reached by time t ; let $p(x)$ be the time required by the parallel execution to have completed execution of program location x and all points preceding it that would have been executed by a sequential program. In other words, t time units into the computation, the sequential execution would have reached code point $c(t)$, having executed all points along the mandatory path to $c(t)$. If we define $p(c(t))$ as the time it takes the parallel execution to have executed all those same points up to $c(t)$ and maybe even more beyond $c(t)$, then plotting

$$S = \frac{t}{p(c(t))}$$

lets the behavior of the speedup function be observed over time.⁶ The final speedup is the value of S at the final value of t , the time required by the sequential execution.

We always hope for graphs similar to that shown in Figure 5, as they are representative of perfect linear speedup; unfortunately, they are not easily obtained.



⁶ Using speculative parallelism, it is possible for some code location x to be executed before all points along the mandatory execution path to x are executed. The definition of $p(x)$ involves the first time at which x and all points along the mandatory execution path to x have been executed.

For a typical and-parallel execution of a highly non-deterministic logic program, we might expect to see speedup graphs similar to that shown in Figure 6. Even though the final speedup plotted indicates that a respectable, overall speedup was obtained, it can be seen that there were two regions of significant slowdown and two regions of superlinear speedup⁷. Decreasing the regions of slowdown and/or increasing the regions of speedup would result in an improved final speedup. Furthermore, it is mandatory that the final speedup reported never represent slowdown. While it appears difficult to identify and properly schedule during run-time the points of superlinearity, preventing points of slowdown is possible, even if difficult.

13. PREVENTING SLOWDOWN

Parallel execution models are obviously best when it can be ensured that a parallel execution of a given program will never be slower than a sequential execution of the same program. One sufficient condition to ensure this in "don't know", and-parallel execution models is the following:

Slowdown Prevention Rule:

If at all times during a parallel execution the set of work which has been accomplished by the parallel execution includes all that work which would have been accomplished by a sequential execution in the same amount of time, then no slowdown will occur.

Clearly, this is not easily accomplished, as the mere action of splitting a computation to achieve parallelism consumes CPU cycles, and by definition places the parallel execution behind, even if only slightly so. One way to obey the Slowdown Prevention Rule is to obey the following rule:

Mandatory Task Execution Rule

To ensure the absence of (or at least the boundedness of) slowdown, it is sufficient to ensure that at all times during parallel execution all mandatory tasks are in execution.

Any parallel execution model that does not embody the Slowdown Prevention Rule is susceptible to potential slowdowns, perhaps even to unbounded slowdowns. It is in fact easy to construct *realistic* examples which should demonstrate significant slowdown on any system not obeying the Slowdown Prevention Rule once the mechanics of the scheduler and load balancer are understood.

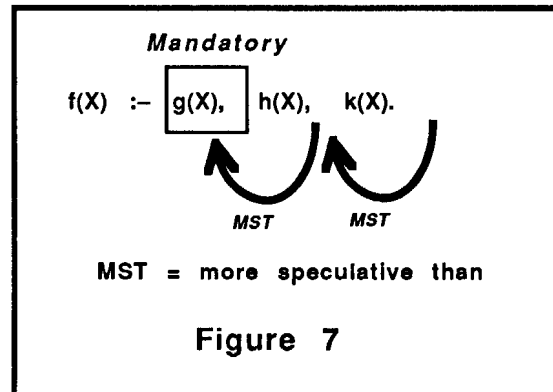
14. DEGREES OF SPECULATION

Consider once more the clause

$$f(X) :- g(X), h(X), k(X).$$

When this clause is first invoked (in a mandatory manner), the only piece of mandatory work is the first (leftmost) subgoal, namely $g(X)$. All the rest are speculative. However, we might choose to consider $k(X)$ more speculative than $h(X)$. Why? Because in order to reach $k(X)$, execution in a sequential model must first pass through $h(X)$. Since $h(X)$ cannot be guaranteed to succeed, $k(X)$ may never be reached. Thus in the sequential execution model, reaching $h(X)$ depends only upon the success of $g(X)$, whereas reaching $k(X)$ depends on the success of both $g(X)$ and $h(X)$. It may be viewed then as somewhat *less* likely that execution will reach $k(X)$ than $h(X)$. Accordingly, we say that $k(X)$ is "more speculative" than $h(X)$. The further to the right a subgoal is from the mandatory goal, the more speculative it is likely to be (see Figure 7).

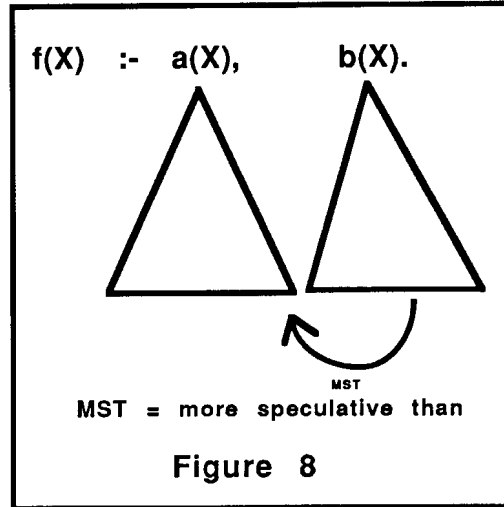
It is interesting to note that as time progresses during a parallel execution, a speculative task tends to become more speculative, then less, then perhaps more, then less, and so on, until it eventually becomes either mandatory or wasted. This is because the frontier of the search tree is generally being



⁷ These points would be rare in traditional, parallel procedural language execution models.

expanded in a non-monotonic manner. The 30'th leaf of the search tree may become the 28'th, then the 40'th, the 41'st, and then the 16'th, as the tree expands and nodes get solved. As nodes to the left of a given goal become solved, the node becomes less speculative; as nodes to the left of the given goal expand into two or more subgoals, the given node becomes more speculative. Figures 8, 9, and 10 partially illustrate this phenomenon.

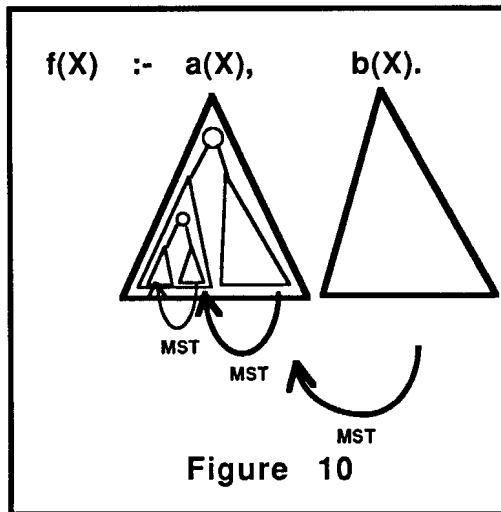
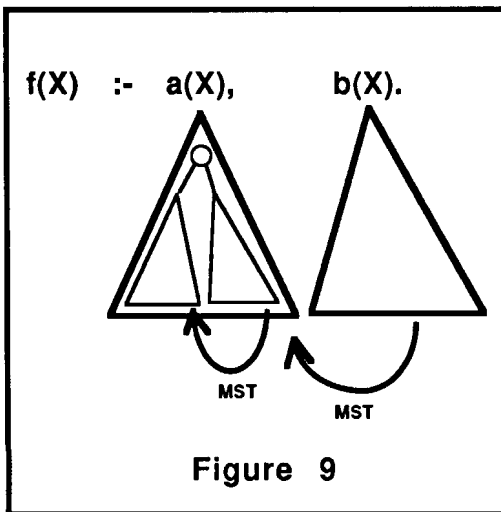
Also, note that mandatory work called from within a speculative task becomes speculative and not mandatory. For example, the first subgoal in a Horn clause will always be executed, and so execution of that subgoal is mandatory if that clause is to be executed. But if the clause is invoked by a speculative subgoal, the execution of the entire clause is speculative, including the execution of the first subgoal. It is clear, then, that at all times in and-parallel, "any solution" logic programming, there is one and only one piece of mandatory work; all the rest are speculative.⁸ Furthermore, there are no two tasks of the same degree of speculation. As a result, load balancing schemes based on using the degree of speculation as an indication of priority must deal with a continuum of priorities; this phenomenon has been observed in speculative parallelism before [Ruggiero, Osborne]. If this continuum is ignored, by assigning all goals in a conjunction the same priority for example [Saletore], slowdown may occur.



Suppose that in an attempt to reduce the number of points of execution slowdown it is desired to minimize at all times the total amount of speculation being engaged. This can be accomplished by observing the following rule:

N-leftmost Tasks Rule

If at all times during execution it can be ensured that the N processors in the system are executing only the N leftmost tasks in the and-execution tree, then the amount of speculation engaged will be minimized. Consequently, either slowdown will not occur, or it can be bounded.



⁸ See Section 17.

Observing this rule also observes the Mandatory Task Execution Rule, and therefore also observes the Slowdown Prevention Rule. Note too that when N is 1, obeying this rule implies that totally non-speculative, sequential execution results, as desired.

Unfortunately, the N -leftmost Tasks Rule is not easily observed. The reader can easily convince himself or herself, for example, that having balanced the system so that the N leftmost tasks are being executed on the N processors, the system will become out of balance following one parallel goal reduction by each processor. The required cycle then is reduce, balance, reduce, balance, etc. Clearly the overhead of load balancing will inhibit effective exploitation of the N -leftmost Tasks Rule.

It should be pointed out that the goal of minimizing speculation is to minimize the amount of wasted work encountered, thereby significantly increasing the possibilities of speedup. Unfortunately, minimizing the speculation also reduces the payoffs of any early detection failures, thereby lessening the possibilities of superlinear speedups.

15. ADDING THROTTLING TO SPECULATION

Not only is it reasonable to try to minimize the amount of overall speculation, it is also reasonable to think that we should devote as much attention as possible to the one and only mandatory subgoal in execution. At a minimum, this means we should always have this subgoal in execution on some processor (viz., the Mandatory Task Execution Rule); doing so will prevent (or at least bound) slowdown. However, if the possibility of speedup is to be increased, we would also expect that parallelizing the execution of this mandatory subgoal would be the most likely way to increase "normal" speedup (i.e., speedup resulting from speeding up the mandatory work rather than speedup achieved through early failure detection). This would imply that whenever the mandatory subgoal reaches a point of potential parallelism, that parallelism should be invoked, and the newly created subtask should preempt any speculative task in execution. But this implies that the continued extraction of parallelism from the one mandatory task will continue to reduce this task's granularity until possibly the point of minimal granularity is reached. As this point, the throttler would be expected to kick in and prevent the mandatory task from further decomposition. But by this point, the ratio of work returned to overhead will be nearly minimal, and as argued in Section 8, we really don't want to get anywhere near this low a ratio. Staying suitably above a ratio of 1 would argue then for *not* breaking down the one and only mandatory subgoal but letting it stay substantially sized. In other words, we might like the mandatory goal to be decomposed somewhat into multiple, parallel tasks, but once these tasks reach some lower cutoff point, a point suitably a work/overhead ratio of 1, throttling should kick in and prevent further decomposition of the mandatory subgoal. At this point, there will still be only one mandatory goal; all the rest will be speculative. If speculative parallelism does not pay off very much, parallel execution speed will have been effectively reduced to the speed of sequential execution.

Consider now that all subgoals will not unfold into parallel executions at the same rate. Suppose that when a clause is invoked several speculative subgoals to the right of the clause are the first to unfold, while those to the left of the clause are prevented from unfolding due to existing data dependencies. As the subgoals to the right unfold, their subtasks will be spread throughout the system, occupying the various system resources. As the resources become used up, the throttler may kick in and decide to prevent any further task decompositions. If the less speculative tasks to the left of the clause, and in particular, the one mandatory task now reach the point where they may unfold, they will be prevented from doing so by the throttler. Execution will have reached a point where the majority of the system resources are being devoted to the most speculative tasks, while the least speculative tasks are receiving the least amount of system resources. Even if the N -leftmost tasks are in execution, they may be prevented from unfolding into parallel subtasks, and their execution speeds may become slower than the speeds that would have resulted from preventing the more speculative tasks from unfolding so early in the execution. In any event, the overall degree of speculation is hardly being minimized. The opportunity to achieve early failure detection, however, might be being maximized, but it would be hard to tell.

If throttling is to work effectively with the goal of minimizing speculative computation, it can be argued that more speculative goals should be throttled differently than less speculative goals. But because each subgoal is less speculative than all those to its right, no two subgoals are of the same degree of speculation. If we consider the degree of speculation of a task to be its number in a left-to-right ordering of the leaf nodes in the execution tree, we can assign a priority to each task based on its degree of speculation, or its leaf number. Given two tasks, the throttler must somehow throttle the

higher priority (less speculative) task differently than the higher priority (more speculative) task. But because no two tasks have the same priority, the throttler must be prepared to deal with an unbounded number of different priorities, and the yet the throttling policy for a given task should be a function of at least the task's priority.

This may not be easy, as a task's priority will tend to shift up and down, with possibly great rapidity. For example, consider an attempt to allocate system resources proportionally to the degree of speculation of a task. A task of priority level 2 may be deemed worthy of receiving many resources, allowed to unfold to some significant degree, and then reduced to a priority level of 102 as the subgoal of priority 1 suddenly and rapidly unfolds into many subtasks. This task of priority 102 would never have been granted so many resources had it been known at the time of its unfolding that its priority would be so greatly reduced; yet there is no way to predict the shifting priorities of a task. An interesting approach to this problem is described in [Zink].

A significant number of other important and challenging problems exist in this area

16. ORPHAN ELIMINATION

Both speculative search and speculative test can lead to the existence of abandoned code (through short-circuiting, for example). Once determined to be abandoned, this code must be prevented from consuming any further system resources, hunted down, and, together with all its descendants, eliminated [Baker, Grit, Hudak]. If they are not eliminated, they will continue to occupy resources beyond their point of need, thereby possibly preventing the unfolding of parallelism within more recently created tasks, and especially within the mandatory task. This can also contribute to potential slowdown. Sadly, whenever a particular subgoal fails in and-parallel logic programming, all subgoals to the right of it in the execution tree are now unnecessary and must be eliminated — *all* of them. Eliminating these now unnecessary tasks will not be without its overhead costs, and so must be handled expeditiously. Note too that if two subgoals fail at the same or nearly the same time, two requests might be generated to eliminate the same task. Thus elimination requests can collide and so must be properly handled.

Related "orphan elimination" requirements have been encountered in other domains, but with differing performance and functional implications [Herlihy].

17. MULTIPLE MANDATORY SUBGOALS

Even though it has been argued above that there is never more than one mandatory goal in "don't know" and-parallelism, there are occasions when this is in fact not true; unfortunately, these occasions are so rare that for all *practical* purposes it can be considered that in fact there will almost always be only one mandatory goal. Optimizing a parallel execution model to this case is therefore appropriate.

If the mandatory goal cannot fail, execution *must* proceed to the goal following it; this following goal is then clearly part of the required work. However, unless it is *known* that the mandatory goal cannot fail, the following goal cannot be classified as mandatory. Thus if the mandatory goal cannot fail and it is *known* that it cannot fail, then the goal immediately following it is also known to be mandatory. If that goal also cannot fail, and this is known, then the goal following this goal is also mandatory, and so on.

Knowing that a goal cannot fail allows the compiler to invoke parallelism differently than it does for speculative parallelism; additionally, the load balancer, scheduler, and throttler can respond differently as well. One trivial case in which it can be determined that a goal cannot fail involves evaluable predicates ("built-ins"), as certain evaluable predicates cannot fail when properly called.⁹ With the use of automatic mode analysis and type inferencing, these are easily identified by the compiler. Unfortunately, it is unlikely to ever prove desirable to execute an evaluable predicate in parallel with anything else anyway since the grain size of these predicates is too small, and so knowing they are mandatory does not help parallel execution.

It remains to be seen whether effective compile-time analysis techniques can be developed to prove that certain complex, user-defined goals cannot fail. To do this, the compiler must prove that the predicate invoked by the goal defines a total function over its domain; this is considerably more difficult than mode or type analysis, and may prove fruitless for any but modest size predicates. Fortunately, however, it is not necessary to prove that the goal will halt, only that it cannot fail.

⁹ They may abort, but that is another matter.

Failing this, one possible empirical approach involves the use of trace-driven compilations made possible by recording the failure rates observed during previous executions; here, the *probability of failure*, coupled with the expected, observed costs [Stone], can possibly be used to limit the speculation engaged.

Finally, note that a clause containing only goals that cannot fail is not necessarily wholly mandatory; if invoked by a speculative goal, then all goals within the clause inherit the speculative property. Only when the first goal of the clause becomes the leftmost unsolved goal in the and-tree would all the goals in the clause become mandatory. And clearly, a mandatory goal can exhibit unlimited internal speculative parallelism. Thus just because a goal becomes mandatory, it is incorrect to treat all subgoals derived from that goal as mandatory.

18. I/O AND SPECULATIVE PARALLELISM

Performing I/O within speculative computations is more difficult than within mandatory computations. Given two parallel, mandatory tasks, as soon as the first completes all its I/O, the second may begin its I/O, barring data dependences. Given a mandatory task followed by a speculative task, however, the second task cannot begin its I/O until the first task has completed its entire execution (or has gone past all points where it may fail) and until all portions of the second task that precede the I/O in the second task and that may fail have also completed. This problem has been solved for speculative-test and-parallelism [DeGroot87, Muthukumar, Chang] and for or-parallelism [Hausman], although doubtfully in the most efficient manner possible.

19. OTHER PARALLEL LOGIC PROGRAMMING MODELS

It is important to remember that the preceding discussion has focused strictly on "don't know", any-solution models of and-parallelism. "Don't care" execution models embodying and-parallelism [Shapiro] exhibit speculative parallelism in the guards, but only mandatory parallelism in the clause bodies. Flat concurrent execution models, due to the simplicity of the guards, exhibit the least speculative parallelism. All-solution execution models are, by and large, non-speculative; the "cut" operator can, however, introduce speculative parallelism under certain circumstances. Or-parallel execution models may or may not be speculative [Hausman]. As with and-parallel execution models, all-solution or-parallel execution models are largely non-speculative; again, however, the cut operator can introduce speculation. Many-solution, or-parallel execution models exhibit significant speculation. Committed-choice languages executed with or-parallelism exhibit speculative parallelism in both the guards and the body. Because of the differing natures of or-parallelism, it should be clear that all parallel search is not speculative search. Surprisingly, even parallel unification is speculative, at least with respect to sequential execution models like Prolog.

20. SUMMARY

It has been shown that "any solution", and-parallel logic programming execution models involve dynamic parallelism, that is, parallelism in which the amounts, types, and locations of the parallelism are only partially determinable at compile time — the actual invocation of parallelism involves run-time decisions based on current system states. It has also been shown that these and-parallel execution models are also inherently speculative, embodying a form of speculative computation called *speculative test*. During a parallel speculative-test execution, there is almost always only one piece of mandatory work; all the other work is speculative. When properly used, speculative test can lead to significant speedups, even superlinear speedups. However, failing to ensure that the mandatory work always receives precedence over speculative work can result in several forms of slowdown. Yet ensuring this in a way that maximizes the opportunities for execution speedup appears extremely difficult. Several significant problems in this area have been discussed, as it is believed that much interesting research remains to be pursued before these problems can be claimed to have been solved, and in fact, before and-parallel logic programming can be claimed to be broadly successful.

Although the problems and discussions were motivated in terms of and-parallel logic programming, these problems are equally applicable to other, more traditional languages, including Lisp, C, Pascal, and Fortran, as speculative parallelism and throttling may be applied to them as well [DeGroot90]. Thus

it is believed that solutions to the problems discussed will have significance and applicability far beyond the limited domain of logic programming.

21. BIBLIOGRAPHY

- [Baker] "The Incremental Garbage Collection of Processes," Henry G. Baker, Jr. and Carl Hewitt, AI Memo 454, MIT AI Laboratory, Cambridge, MA, Dec. 1977.
- [Burton] "Controlling Speculative Computation in a Parallel Functional Programming Language," F. Warren Burton, *Procs. of the Fifth Int'l Conf on Distributed Computing Systems*, pp. 453-458, Denver, CO, May 1985.
- [Bush] "Transforming Recursive Programs for Execution on Parallel Machines," V.J. Bush and John R. Gurd, LNCS, Vol. 201, pp. 350-367, Sept. 1985.
- [Chang] "Restricted And-Parallelism Execution Model with Side-Effects," Si-En Chang and Y. Paul Chiang, *Procs. of the North American Conference on Logic Programming, Vol. 1*, MIT Press, pp. 350-368.
- [Ci] *Research on Frontiers in Computing*, Ci Yungui, Tsinghua Univ. Press, 1989.
- [Conery] *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, John S. Conery, Kluwer Academic Publishers, Boston, MA, 1988.
- [Debray] "Task Granularity Analysis in Logic Programs," Saumya Debray, N.-W. Lin, and M.V. Hermenegildo, *Procs. of the 1990 ACM Conf. on Programming Language Design and Implementation*, ACM Press, June, 1990.
- [DeGroot84] "Restricted And-Parallelism," Doug DeGroot, *Procs. of the Int'l Conf on Fifth Generation Computer Systems*, pp. 471-478, Ohmsha, Tokyo, Nov. 1984.
- [DeGroot87] "Restricted And-Parallelism and Side-Effects," Doug DeGroot, *Procs. of the 4'th Int'l Symp. on Logic Programming*, IEEE, 1987, pp. 80-89.
- [DeGroot90] "Control of Dynamic Parallelism", tutorial slides, 1990 Int'l Conf. on Parallel Processing, IEEE, 1990, available upon request.
- [Eager] "Speedup versus efficiency in parallel systems," Derek L. Eager, John Zahorajan, and Edward Lazowska. *IEEE Trans. on Computers*, 38(3):408-423, March 1989.
- [Finkel] "Large-grain parallelism - Three case studies," Raphael A. Finkel, in *The Characteristics of Parallel Algorithms*, Leah Jamieson, Dennis Gannon, and Robert Douglass, Eds, MIT Press, 1987.
- [Goldman] "Qlisp: Parallel Processing in Lisp," Ron Goldman and Richard P. Gabriel, *IEEE Software*, July 1989, pp. 51-59.
- [Grit] "Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System," Dale H. Grit and Rex L. Page, *ACM TOPLAS*, Vol. 3, No. 1, January, 1981, pp. 49-59.
- [Halstead] "An Assessment of Multilisp: Lessons from Experience," Robert H. Halstead, *Int'l Journal of Parallel Programming*, Vol. 15, No. 6, pp. 459-501, Dec. 1986.
- [Hausman] *Pruning and Speculative Work in OR-Parallel Prolog*, Bogumil Hausman, Ph.D. dissertation, Dept. of Telecommunications and Computer Systems, The Royal Inst. of Technology, Stockholm, Sweden, March, 1990.
- [Herlihy] "Timestamp-Based Orphan Elimination," Maurice P. Herlihy and Martin S. McKendry, Tech Report CMU-CS-87-108, Carnegie Mellon Univ, CS Dept., Dec. 31, 1987.

- [Hermenegildo] "The &-Prolog System: Exploiting Independent And-Parallelism," Manuel V. Hermenegildo and K.J. Greene, *New Generation Computing*, Vol. 9, Nos. 3 and 4, 1991, pp. 233-256, Ohmsha and Springer-Verlag.
- [Hoare] "Communicating Sequential Processes", C.A.R. Hoare, *CACM*, 21(8):666-677, Aug. 1978
- [Hudak] "Garbage collection and task deletion in distributed applicative processing systems," Paul Hudak and Robert M. Keller, *Procs. ACM Symp. on LISP and Functional Programming*, Pittsburgh, PA, Aug. 1982, pp. 168-178.
- [Jefferson] "Virtual Time", David R. Jefferson, *ACM TOPLAS*, 7(3):404-425, July 1985.
- [Kerschberg] *Expert Database Systems*, Larry Kerschberg, Editor, *Procs. of the First Int'l Workshop on Expert Database Systems*, Benjamin Cummings, 1986.
- [Kowalski] *Logic for Problem Solving*, Robert A. Kowalski. Elsevier North-Holland, New York, 1979.
- [Kuck] "Dependence Graphs and Compiler Optimizations," D. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, *Procs. 8'th ACM Symp on Principles of Programming Languages (POPL)*, ACM, 1981, pp. 207-218.
- [Lai] "Anomalies in Parallel Branch-and Bound Algorithms," T.H. Lai and Sartaj Sahni, *CACM*, pp. 594-602, June 1984.
- [Li] "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," Guo-Jie Li and Benjamin Wah, *IEEE Trans. on Computers*, C-35(6):568-573, June 86
- [Muthukumar] "Complete and Efficient Methods for Supporting Side-effects in Independent-Restricted And-parallelism," K. Muthukumar and M. Hermenegildo, *Procs. of the 1989 Int'l Conf. on Logic Programming*, MIT Press, 1989.
- [Osborne] *Speculative Computation in Multilisp*, Randy Osborne, Ph.D. Thesis, MIT/LCS/TR-464, MIT, Cambridge, Mass, Dec 1989
- [Page] "If-then-else as a Concurrency Inhibitor in Eager Beaver Evaluation of Recursive Programs," Rex L. Page, Martha G. Conant, and Dale H. Grit, *ACM Procs. of the 1981 Conf. on Functional Programming Languages and Computer Architectures*, Portsmouth, New Hampshire, Oct 18-22, 1981, pp. 179-186.
- [Roberts] "WorkCrews: An Abstraction for Controlling Parallelism," Eric S. Roberts and Mark T. Vandevoorde, Technical Report 42, Digital Systems Research Center, Palo Alto, CA, April 2, 1989.
- [Ruggiero] "Control of Parallelism in the Manchester Dataflow Machine," Carlos A. Ruggiero and John Sargeant, *Procs. of the Conf. on Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS-274, Sept. 1987.
- [Saletore] "Obtaining First Solutions Faster in AND-OR Parallel Execution of Logic Programs," Vikram A. Saletore and L.V. Kale, *Procs of the North American Conference on Logic Programming*, 1989, Vol 1, pp.390-406, MIT Press, 1989
- [Sargeant] "Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines," John Sargeant, Technical Report UMCS-86-11-5, Dept. of CS, Univ. of Manchester.
- [Shapiro] "The Family of Concurrent Logic Programming Languages," Ehud Shapiro, *ACM Surveys*, 21(3):412-510, Sept. 1989.

- [Soley] *On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control*, Richard Mark Soley, Ph.D. Thesis, MIT/LCS/TR-443, MIT, Cambridge, Mass, May 1989.
- [Steele] *Common Lisp: The Language*, Guy L. Steele, Jr., Digital Press, 1984.
- [Steinberg] "Searching Game Trees in Parallel," Igor Steinberg and Marvin Solomon, *Procs. of the 1990 Int'l Conf. on Parallel Processing*, Aug. 1990, Penn State Univ. Press, pp. III-9-17.
- [Sterling] *The Art of Prolog*, Leon Sterling and Ehud Shapiro, MIT Press, Cambridge, MA, 1986
- [Stone] "The average complexity of depth-first search with backtracking and cutoff," Harold S. Stone and Paolo Sipala, *IBM Journal of Research and Development*, 30(3):242-258, May 1986
- [Tick] "Comparing Two Parallel Logic-Programming Architectures," Evan Tick, *IEEE Software*, July 1989, pp. 71-80.
- [Tucker] "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," Andrew Tucker and Anoop Gupta, *Operating Systems Review, Special Issue, Procs. of the 12'th ACM Symp. on OS Principles, ACM SIGOPS*, ACM, Vol. 23, No. 5, pp. 159-166.
- [Watson] "Lager - Interim Report: Issue No. 1," Ian Watson, Technical Report, Nov. 25, 1988, Flagship Project, The Univ. of Manchester, Manchester, M13 9PL, United Kingdom.
- [Zink] "Engines as a Method of Controlling Speculative Evaluation," Ken Zink and Steven Tighe, MCC Technical Report No. ACT-HI-108-89, MCC, Austin, Texas, March 1, 1989.

•

•

•

•