On the Inherently Speculative Nature of Nondeterministic And-Parallelism

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

ABSTRACT

In Horn-clause logic programming, it is necessary for all subgoals in a clause to succeed for the clause itself to succeed. As a result, many models of and-parallel execution of logic clauses have generally considered it desirable to execute as many subgoals as possible in parallel. Contributing to this idea has been the belief that and-parallelism is not speculative, as is or-parallelism. This is, unfortunately, not true. This paper shows that "don't know" and-parallelism does exhibit speculative parallelism; it is in fact *inherently* speculative. Due to its speculative nature, executing subgoals in parallel can lead to superlinear speedups in a regular manner. Unfortunately, its speculative nature can also lead to serious slowdown rather than to speedup. Examples of scheduling activities which would be considered quite normal (even desirable) in non-speculative parallel systems are described and shown to be able to lead to slowdown. A "Slowdown Prevention Rule" is then described which, if adhered to, can ensure the absence of slowdown, or to at least bound its damage. Several additional aspects of speculative parallelism are also discussed.

1. Introduction

There have been a great many models of parallel execution for logic programming languages based on Horn-clauses. The two most common approaches on which these models have been based are those of and-parallelism and or-parallelism. Given the type of parallelism to be employed, there is another significant aspect of computation that must be considered by any proposed parallel execution model. This aspect concerns whether the model is to return: 1) one and only one answer to a query, regardless of how many exist, 2) any number of answers - perhaps even all, but only one at a time, or 3) all answers to a query all at once.

Models of the first type are "single solution" models; they exhibit "don't care" nondeterminism [Kowalski]; and they are typified by "committed choice", concurrent logic programming languages [Shapiro]. The second type are "any solution" (or, "many solutions") models; these models all exhibit "don't know nondeterminism" [Kowalski]. Parallel versions of sequential logic programming models, such as parallel Prologs, are included in this second type. Models of the third type are called "all solutions" models. These "all solutions" models have frequently been pursued for use in database query and/or implementation models, especially expert or knowledge data bases [Kerschberg].

This paper is concerned with the second type of parallel execution models — "any solution", "don't-know" parallel execution models. These models may exhibit and-parallelism, or-parallelism, or both [Conery]. While both have proven useful, it has generally been felt that and-parallelism was much better suited for highly deterministic programs. It has also been widely believed that, unlike or-parallelism, and-parallelism in logic programs does not exhibit speculative computation (see [Tick], for example).

This paper restricts its attention to "any solution", "don't know" and-parallel execution models. It shows clearly that and-parallel execution of such models is in fact *inherently speculative*. In fact, at any time during the and-parallel execution of a logic program there is one and only one

piece of non-speculative work to be performed; all other work is necessarily speculative. Furthermore, no two speculative tasks exhibit the same degree of speculation. Scheduling speculative computation with non-speculative computation places severe restrictions on the system. One possible negative outcome that may result if these restrictions are not adhered to is that the parallel execution may result in significantly slower execution than a comparable sequential execution; thus slowdown will have been achieved rather than speedup. Ensuring speedup is difficult; but it is shown that ensuring the absence of slowdown is also difficult.

This paper describes speculative parallelism and then shows how it arises in and-parallel, "don't know" execution models. Ways in which slowdown may occur as a result of the speculative nature of this parallelism are discussed. Then, several rules which may be obeyed in order to ensure the absence of slowdown are presented. Finally, implications to load balancing algorithms, I/O, and throttling are presented.

2. Speculative Parallelism

Nondeterministic programming languages can be executed either nondeterministically (as are CSP [Hoare]) or deterministically (as are most sequential Prologs). Nondeterministic executions of a nondeterministic program result in differing amounts of work being expended from one execution to another. If a particular nondeterministic program is executed in a deterministic, sequential manner with a given data set (as is the case with most sequential Prolog systems), a prescribed set of instructions is executed in a particular order. Given the identical data set, further deterministic, sequential executions of the program yield an identical set of executed instructions with the identical execution ordering. These ordered instructions are referred to here as the set of ***required*** work — that work that the sequential program *must* engage in in order to compute its results.

Unfortunately, except in the rarest of circumstances, determining the required set of work requires executing the program. Knowledge of this set of work is thus not usually known by the compiler, and so it cannot be used to optimize the parallel execution of the program. However, at certain times during execution, certain instructions are known to be necessary to make forward progress. For instance, when a sequence of unconditional assignment statements is encountered, it is known that all of these must be executed, with control to resume following the last of these. At each point during the execution, the set of work that is known to be necessary to make forward progress is called ***mandatory*** work. Clearly all mandatory work is included in the set of required work.

At most points in the execution, required work exists which is not known to be mandatory. For example, consider the "if" statement:

if ***conditional*** *then*
    consequent
else ***alternate***

An "if" statement imposes a conditional flow of control on a program's execution. If control ever reaches the "if" statement, it becomes mandatory to evaluate the conditional in order to make forward progress, and thus the conditional is known to be part of the required work. Prior to execution of the conditional, however, it is unknown whether the consequent or the alternate part will be executed, and thus neither can be classified as mandatory; we will have reached the end of our current knowledge of which work is required and which is not.

Because of this, traditional approaches to the parallelization of computer programs use conditional branch instructions (such as "if" statements) as places to synchronize, and indeed serialize, execution [Kuck]. 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; completing that

path is now mandatory. 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.

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 mandatory, while the other is now known to be ***unnecessary***. Any work expended on the unnecessary path becomes ***wasted***. Neither path can any longer be classified as speculative.

Clearly, speculative work can be engaged in hopes of obtaining greater execution speedup 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 has traditionally been considered to be inherently sequential control constructs, we can measure the success of our efforts against the non-speculative execution of the same program.

3. Types of Speculative Parallelism

Consider a complex "if" statement, such as that shown in Figure 1.

```
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

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 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. 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 example above, the allowable speculative tests include the following expressions (plus any legal 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 allowable speculative searches include the following expressions (plus any legal 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. Speculative searches can be engaged to explore 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]. Another, less obvious, but interesting form of speculative parallelism is found in the Time Warp model of discrete-event simulation [Jefferson].

4. 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*) can preclude 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
      C
   else false
else false
```

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 Multi-Lisp'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.[1]

5. Horn Clauses and And-Parallelism

Given a Horn-clause logic 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 be false, execution backs up to the previous subgoal, if any. 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 sub-

[1] unless it is known that A and B cannot fail; see Section 12.

goals. Clearly then, the amount of work represented by this clause is not constant.

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" statement:

```
if g(X) then
    if h(X) then
        if k(X) then
            f(X)
        else false
    else false
else false
```

[2]

Considering this clause, it is interesting to try to identify the 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. In fact, it should be clear that evaluating the subgoals in parallel is indeed *only* speculative. While this is rather obvious, it constitutes one of the main points of this paper; unfortunately, this point has been missed by a large number of "any solution", and-parallel execution models. In the remainder of this paper, several implications of this fact are explored.

6. Speedup

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.[3] 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 a parallel execution to be faster than any *specific*, deterministic, sequential execution time, especially given approximately equal-performance of the CPUs. 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 times 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, one obviously meaningful comparison is the speed of a selected program using the parallel 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 "overlooked".

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.

The remainder of this paper considers only parallel execution models which exhibit deterministic executions. 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

---

[2] Backtracking and unification, of course, make this equivalence less imediately obvious.

[3] The determinism of a program execution should not be confused with the determinism of the program itself.

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

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

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

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 slowdown.

7. 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 2 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.

Then 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 then 3.1 hours. As listed in Figure 2, 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 .1 + .333 = .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. 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.

8. Slowdown

Unfortunately, due to the speculative nature of "any solution", and-parallel execution models, there are also numerous ways in which slowdown (even superlinear 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 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 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 held up 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 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.

9. 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. Then plotting

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

lets the behavior of the speedup function be observed over time. The final speedup is the value of S at the final value of t. We always hope for graphs similar to that shown in Figure 3, as they are representative of perfect linear speedup; unfortunately, they are not easily obtained. However, for a typical parallel execution of a logic program, we might expect to see speedup graphs similar to that shown in Figure 4. Even though the final speedup plotted indicates that a respectable, overall speedup was obtained, it can be seen that were two points of significant slowdown and three points of superlinear speedup[4]. Decreasing the number of slowdown points and/or increasing the number of superlinear speedup points 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.

10. Preventing Slowdown

Parallel execution models are best when it can be ensured that the parallel execution of a given program will never be slower than the 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 place 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 slowdown, it is sufficient to ensure that at all times during parallel execution all mandatory tasks are in execution.

Obeying this rule is not as easy as it might appear [DeGroot90b].

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.

11. 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 successes 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).

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.

---

[4] these points would be rare in traditional, parallel procedural languages

Figures 5 through 8 partially illustrate this phenomenon. Also, note that mandatory work called from within a speculative task becomes speculative and not mandatory.

It is clear too, then, that at all times there is one and only one piece of mandatory work; all the rest are speculative.[5] 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 [DeGroot-90b],[Ruggiero]; this phenomenon has been observed in speculative parallelism before [Osborne]. If this continuum is ignored, by assigning all goals in a conjunction the same priority for example [Saletore], slowdown may occur.

Suppose 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.

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 on the N processors, the system will become out of balance following one parallel goal reduction by each processor.

One approach to embodying the N-leftmost Tasks Rule is described in [DeGroot90b. While this approach does not, and indeed cannot, obey the N-leftmost Tasks Rule at all times, except in purely sequential programs, it can come within a bounded percentage performance degradation of doing so.

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.

12. 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 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 [DeGroot90a]. 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.[6] 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

[5] See Section 12.

[6] They may abort, but that is another matter.

since the grain size of these predicates is too small.

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.

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, the 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.

13. 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 [Cepielewski].

13. Other Parallel Logic 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 gaurds, 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. 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.

14. Conclusions

It has been shown that "don't know" and-parallelism is 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 rest are 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 slowdown. Three rules have been presented which can be incorporated into a parallel system's scheduler and load balancer to prevent slowdown (or at worst, allow only bounded slowdown).

15. Acknowledgements

I am indebted to Gary Lindstrom and David Haug for their patience and interest in assisting me in the early stages of this work.

Bibliography

[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.

[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.

[Conery] *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, John S. Conery, Kluwer Academic Publishers, Boston, MA, 1988.

[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.

[DeGroot90a] "Throttling and Speculating on Parallel Architectures", Doug DeGroot, lecture slides (abstract in *Procs. of Parbase '90*), IEEE, March 1990.

[DeGroot90b] "Limiting Speculation in And-Parallelism", Doug DeGroot and Y. Paul Chiang, in preparation.

[Eager] "Speedup versus efficiency in parallel systems," Derek L. Eager, John Zahorjan, 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.

[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.

[Hoare] "Communicating Sequential Processes", C.A.R. Hoare, *CACM*, 21(8):666-677, Aug. 1978

[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.

[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

[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.

[Sterling] *The Art of Prolog*, Leon Sterling and Ehud Shapiro, MIT Press, Cambridge, Mass, 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