

**THE UNIVERSITY OF CALGARY**

# **An Optimistic AND-Parallel Prolog Implementation**

**BY**

**Ian William Olthof**

**A THESIS**

**SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE**

**DEPARTMENT OF COMPUTER SCIENCE**

**CALGARY, ALBERTA**

**April, 1991**

**© Ian William Olthof 1991**



National Library  
of Canada

Bibliothèque nationale  
du Canada

Canadian Theses Service    Service des thèses canadiennes

Ottawa, Canada  
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.


ISBN 0-315-66893-8

Canada

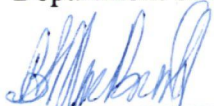
THE UNIVERSITY OF CALGARY


**Faculty of Graduate Studies**


The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Optimistic AND-Parallel Prolog Implementation," submitted by Ian William Olthof in partial fulfillment of the requirements for the degree of Master of Science.

  
John Cleary, Supervisor  
Department of Computer Science

\_\_\_\_\_  
Jun Gu  
Department of Electrical Engineering

  
\_\_\_\_\_  
Bruce MacDonald  
Department of Computer Science

  
\_\_\_\_\_  
Mike Williams  
Department of Computer Science

  
\_\_\_\_\_  
Ian H. Witten  
Department of Computer Science

Date 1991-04-23

## Abstract

This thesis documents the implementation and testing of a distributed Prolog interpreter that exploits one of the major forms of parallelism available in Prolog: AND-parallelism. Previous AND-parallel systems have had to make a tradeoff, sacrificing potential parallelism to retain the semantics of sequential Prolog, or giving up those semantics (in particular, *completeness*) in order to achieve maximum parallelism. This tradeoff is seen to be unnecessary when a total ordering is imposed on the goals executed by the parallel system.

The implementation described in the thesis is based on an algorithm due to Cleary *et al*, which uses Virtual Time to impose this total ordering. In the course of implementing the distributed Prolog system, several points of the algorithm were clarified and a number of small errors corrected. As well, an apparently obvious optimization to the algorithm was found instead to be an over-optimization that caused the system to miss solutions. This work resulted in a working parallel Prolog system blemished only by the over-optimization; as well, a safer optimization is outlined.

The system was tested using a variety of Prolog programs—some that featured independent AND-parallelism, some that offered stream AND-parallelism between dependent goals, and some that were highly nondeterministic. For each program, different goal-ordering strategies were tested. Results showed that several programs ran well under the parallel system, and suggested that others could be made to run well with the addition of two optimizations; proposed implementations for these optimizations are described in detail.

## Acknowledgements

Well, this thesis has been a lot of hard work. I dread to think how much harder it would have been without the help and encouragement of many people. The most prominent of these kind souls (and soulless institutions) are listed below.

**John Cleary** my supervisor, for helpful discussions about the algorithm and its implementation, for prompt and encouraging reviews of my scribblings, and for financial support through the darkest days of thesis.

**Lita Martin** for holding the home front together and being an all-around sweetie.

**Big Rock Brewery** for producing fine creative juices that helped keep *my* creative juices flowing.

**Richard Esau** for listening to all of those boring implementation details so I could explain my latest bug, as well as for the weekly squash games.

**Mike and Mark** for the opportunities they gave me to get away from my thesis for a while, to enjoy instead scintillating conversations (or was that incoherent babble?) about hardware, motorcycles, politics, and/or beer.

**David Pauli** for showing the rest of the gang how to get it done.

**RMR** and all of its members, for all of the kilometers we've spent together on the road pedalling to the next checkpoint, and for expanding my knowledge of sleep deprivation.

**NSERC** for funding the early years of my research.

## **Dedication**

To my parents, for their quiet but unmistakable support throughout my educational career.

# Contents

<b>Approval</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Dedication</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Parallelism in Prolog</b>	<b>5</b>
2.1 Prolog . . . . .	5
2.2 OR-Parallelism . . . . .	10
2.3 AND-Parallelism . . . . .	11
2.4 Other Forms of Parallelism . . . . .	12
<b>3 AND-Parallelism</b>	<b>15</b>
3.1 Naive AND-Parallelism . . . . .	16
3.2 Independent AND-Parallelism . . . . .	17
3.3 Concurrent Logic Programming Languages . . . . .	20
3.4 Backtracking Stream AND-Parallel Systems . . . . .	22
<b>4 AND-Parallelism Using Virtual Time</b>	<b>26</b>
4.1 Virtual Time and Time Warp . . . . .	26
4.2 Stacks and Frames . . . . .	28
4.3 Messages . . . . .	29
4.4 Mixing Prolog and Time Warp . . . . .	30
4.5 Execution Model . . . . .	31
4.5.1 Forward Execution . . . . .	31
4.5.2 Local Backtracking . . . . .	32
4.5.3 Rollback . . . . .	32
4.5.4 Remote Backtracking . . . . .	33
4.5.5 Example . . . . .	35
4.6 Optimizations . . . . .	39
4.6.1 BIND Optimization . . . . .	40
4.6.2 ANTI-BIND Optimization . . . . .	41
4.6.3 FAIL Over-optimization . . . . .	41

<b>5</b>	<b>Implementation</b>	<b>45</b>
5.1	Overview of the System . . . . .	45
5.2	Solver Processes . . . . .	48
5.3	Ear Processes . . . . .	48
5.4	Binding Environment . . . . .	52
5.4.1	Naming . . . . .	52
5.4.2	Local and Shared Variables . . . . .	53
5.4.3	Incomplete Data Structures . . . . .	54
5.5	Delayed Predicates . . . . .	54
5.5.1	User Predicates . . . . .	54
5.5.2	Builtin Predicates . . . . .	56
5.6	Termination Detection . . . . .	61
5.7	Optimizations . . . . .	64
5.7.1	BIND Optimization . . . . .	64
5.7.2	ANTI-BIND Optimization . . . . .	68
5.7.3	FAIL Optimizations . . . . .	69
<b>6</b>	<b>Testing and Results</b>	<b>73</b>
6.1	Testing Environment . . . . .	73
6.2	Testing . . . . .	74
6.2.1	Determinism and Nondeterminism . . . . .	75
6.2.2	Delay Characteristics . . . . .	76
6.2.3	Goal-ordering Strategies . . . . .	77
6.3	Results . . . . .	80
6.3.1	Summary of Test Runs . . . . .	81
6.3.2	Deterministic Programs . . . . .	82
6.3.3	Shallow-nondeterministic Programs . . . . .	88
6.3.4	Deeply-nondeterministic Programs . . . . .	92
6.4	Analysis . . . . .	98
6.4.1	Comparison with the Sequential Interpreter . . . . .	98
6.4.2	Comparison Between Parallel Versions . . . . .	99
6.4.3	Conclusion . . . . .	101
<b>7</b>	<b>Conclusion</b>	<b>102</b>
7.1	Future Work . . . . .	104
7.2	Summary . . . . .	105
	<b>Bibliography</b>	<b>107</b>
<b>A</b>	<b>Pseudocode for AND-Parallel Prolog Algorithm</b>	<b>110</b>





## **List of Tables**

6.1	Messages processed under combinations of goal-scheduling parameters. .	79
6.2	Summary of comparisons of unifications. . . . .	99
6.3	Summary of comparisons of message counts. . . . .	99

## List of Figures

2.1	Prolog program and query. . . . .	8
2.2	Goal tree for execution of a program and query. . . . .	9
4.1	Forward execution . . . . .	36
4.2	Rollback . . . . .	37
4.3	Remote backtracking . . . . .	37
4.4	Termination with solution . . . . .	38
4.5	Backtracking among two processes . . . . .	42
4.6	Backtracking to a third process (unoptimized) . . . . .	43
4.7	Backtracking to a third process (optimized) . . . . .	43
5.1	Relationship between ear and solver processes . . . . .	50
5.2	Compatible BIND: no local binding . . . . .	65
5.3	Compatible BIND: local binding has precedence . . . . .	66
5.4	Compatible BIND: incoming binding has precedence . . . . .	66
5.5	Incompatible BIND: message is rejected . . . . .	67
5.6	Incompatible BIND: message is accepted after partial rollback . . . . .	68
5.7	Pathological case for FAIL “optimization” . . . . .	71
6.1	Average, minimum, and maximum unifications for <code>mmult</code> . . . . .	83
6.2	Tokens, BINDs, ANTI-BINDs, and message annihilations for <code>mmult</code> . . .	84
6.3	Average, minimum, and maximum unifications for <code>fib</code> . . . . .	85
6.4	Message counts for <code>fib</code> . . . . .	85

6.5	Unification average, minimum, and maximum for <code>tak</code> . . . . .	87
6.6	Message counts for <code>tak</code> . . . . .	87
6.7	Unification average, minimum, and maximum for <code>qsort</code> . . . . .	89
6.8	Message counts for <code>qsort</code> . . . . .	90
6.9	Unification average, minimum, and maximum for <code>union</code> . . . . .	91
6.10	Message counts for <code>union</code> . . . . .	91
6.11	Unification average, minimum, and maximum for <code>inter</code> . . . . .	93
6.12	Message counts for <code>inter</code> . . . . .	93
6.13	Program and query for <code>x15</code> . . . . .	94
6.14	Program and query for <code>xy15</code> . . . . .	94
6.15	All-solutions averages, minima, and maxima for <code>x15</code> . . . . .	96
6.16	All-solutions message counts for <code>x15</code> . . . . .	97
6.17	All-solutions averages, minima, and maxima for <code>xy15</code> . . . . .	97
6.18	All-solutions message counts for <code>xy15</code> . . . . .	98

# Chapter 1

## Introduction

One of the constants of the computing field is the insatiable need for greater computer power to solve ever-larger problems. On the hardware end, this need has been met in two ways: through advances in computer architecture, but mainly by producing ever-faster hardware devices [Hwang & Briggs 1984]. As the physical (and economic) limits to device speeds have been approached, attention has shifted more toward producing better architectures. In this light, parallel architectures have become increasingly attractive; a single-CPU supercomputer costs far more than a parallel system of comparable power that uses many less-advanced CPUs.

With the shift in hardware focus comes a similar shift in software focus. In order to exploit the power of a parallel computer, a program must itself be parallelizable. The sequential codes of yesteryear rarely meet this criterion; thus the widespread interest in developing parallel algorithms. (Thus also the continued interest in fast sequential systems, which allow “dusty-deck” sequential codes to be run more quickly without modification.)

Designing parallel algorithms is itself no easy task, particularly when they must be expressed in a programming language or pseudocode which is essentially sequential, even though it may incorporate parallel constructs. The economic savings of running programs on an inexpensive multiprocessor platform rather than on an expensive uniprocessor machine may well be outweighed by the cost of rewriting software to run in parallel.

An attractive alternative is to use a nonprocedural language, since such languages are by definition asequential; where statements in a procedural language program imply a specific

order of execution, statements in a nonprocedural language do no such thing. A good example is Prolog, a declarative language based on first-order predicate logic. Because logic is asequential, so by extension is (pure) Prolog. The conjuncts and disjuncts of a Prolog program imply no particular order of execution (though sequential Prolog implementations require that some ordering be imposed), so they should be readily parallelizable.

The use of Prolog or another nonprocedural language offers an additional benefit: since it is higher-level than a procedural language, programmers can concentrate more on algorithms and less on low-level details. Thus, the cost of parallelizing software would be reduced.

Two main forms of parallelism are available in Prolog, corresponding to the conjuncts and disjuncts of a Prolog program. *OR-parallelism* refers to the concurrent execution of a group of disjuncts; *AND-parallelism* refers to the parallel execution of conjuncts.

Many parallel implementations of Prolog and other logic programming languages already exist. Of those exploiting AND-parallelism, most either fail to extract as much parallelism as exists in many programs, or in attempting to extract that parallelism, they give up the logical semantics of Prolog. The system described in this thesis is (to my knowledge) the first implementation to aspire to the best of both worlds. Jefferson's *Time Warp* [Jefferson & Sowizral 1985] model is used as a basis for combining *independent* and *dependent* AND-parallelism with *backtracking*, thus achieving maximum parallelism while retaining Prolog's logical semantics.

The rest of the thesis examines the development of a fully AND-parallel Prolog. It begins by examining sequential Prolog, after which several parallel execution models are described. A backtracking algorithm allowing both dependent and independent AND-parallel execution is outlined, and its implementation is discussed. Finally, test results and

an evaluation of the implementation are given. The thesis is structured as follows:

**Chapter 2** begins by introducing the Prolog language, giving its syntax and a sequential execution model. The various forms of parallelism available in Prolog are then discussed: OR-parallelism, AND-parallelism, and low-level parallelism.

**Chapter 3** focuses on AND-parallel execution, beginning with a look at a naive approach. Next, the two most common approaches for exploiting AND-parallelism are described: the independent AND-parallel models and the concurrent logic programming languages. Finally, an approach that combines the strengths of these approaches while avoiding the weaknesses is introduced: the backtracking stream AND-parallel models.

**Chapter 4** introduces an algorithm for fully AND-parallel Prolog execution that incorporates backtracking. For backtracking to work correctly, some notion of goal ordering is necessary; *Virtual Time* [Jefferson 1985] is presented as a basis for such an ordering. Next, the stack structures and message types necessary for parallel execution are given, and an interface to the underlying virtual time system is suggested. At last, the execution model is presented, and several optimizations to the basic algorithm (as well as an over-optimization) are discussed.

**Chapter 5** outlines an implementation of the algorithm given in the previous chapter. This outline begins with an overview of the system, followed by a look at *solver* and *ear* processes. Next, several features of the system are examined in detail: the shared variable binding environment, the use of delay annotations on predicates, and the detection of system termination. Finally, a discussion is presented on how to implement the optimizations discussed previously.

**Chapter 6** looks into the results of testing the system. First, the testing environment is described, and the expected effects of several factors that varied in the testing are discussed. Eight test programs are described, and results for each are given and analyzed. The results are then summarized and interpreted.

**Chapter 7** summarizes the work done on the system and suggests possible future work. Finally, an analysis of the contributions of the thesis is given.

**Appendix A** presents an updated version of the algorithm given by [Cleary *et al* 1987].

**Appendix B** lists the programs used in testing the system and the top-level queries for each.



## Chapter 2

### Parallelism in Prolog

Parallelism in Prolog comes in several forms. Each of these forms has its own unique characteristics, and its own strengths and weaknesses, but all are based on sequential Prolog. Thus, we begin with a look at sequential Prolog and the algorithm for executing a sequential Prolog program. Then we examine each parallel variant in turn, beginning with OR-parallelism, moving on to AND-parallelism, and concluding with a look at other, lower-level, forms of parallelism.

#### 2.1 Prolog

A Prolog system is a *resolution* [Lloyd 1984] system. A Prolog execution consists of the runtime system accepting a *query* from the user and trying to resolve it with respect to some *program*.

A Prolog program is made up of one or more distinct *predicates*. Each predicate is a disjunction of one or more *clauses*. A clause consists of a *head* and a *body*, either or both of which may be empty. A clause with neither head nor body is the *empty clause*; a clause with a head but no body is known as a *unit clause*. A query from the user is a clause with no head.

$\square$	% empty clause
head.	% unit clause
$:-$ body.	% query

```
head :- body.      % "normal" clause
```

The head of a clause, if it exists, is a positive *literal*; the body, if it exists, is a conjunction of one or more positive or negative literals. A literal consists of a *functor* (with some *arity*) and its *arguments*.

```
f(3, a)           % functor f/2, with arguments 3 and a
```

Overloading functor *names* is permitted; thus, a functor  $f$  with arity 2 (written  $f/2$ ) is distinct from the functor  $f/3$ . Each argument is a *term*; a term is a literal, a *constant*, or a *variable*.

```
2                % constant
x                % variable
a, f(Y)          % literals
```

In a Prolog system, execution cycles through three phases: *goal selection*, *clause selection*, and *unification*. The goal-selection mechanism chooses a goal to be resolved from a goal list; initially, this list is composed of the literal(s) in the user's query. The clause selection mechanism then attempts to find a clause with a head whose functor matches that of the selected goal. Then, the arguments of the selected goal and selected head are unified pairwise. If the unification succeeds (i.e. the goal and head are consistent with each other), any variable bindings made are recorded, and the selected goal is replaced in the goal list by the body of the selected clause.

In unification, a variable may be *bound* to some other term. If the variable is bound to a constant, it may not be further instantiated, or mutated in any way. If it is bound to a literal, the same is true, although any *free* variables within the literal may be bound later. If

it is bound to another variable, it becomes an alias for that other variable. *Dereferencing* it yields that other. Unifying several variables with each other may lead to a long dereference chain.

In the standard algorithm, the selected goal is the leftmost in the goal list, and a clause body replacing its parent goal is placed leftmost. (Other algorithms, such as those in systems that feature *delays*, may alter this rule somewhat.) Similarly, clauses are normally selected from top to bottom as they appear in the input program text. When one clause of a predicate has been selected, no other clause in the predicate can be selected until the first has been *backtracked*.

Backtracking occurs when a unification fails, that is, when the selected goal and selected clause are found to be mutually inconsistent. At this point, the clause-selection mechanism tries to provide another clause. If it can find an alternate clause, forward execution begins again with unification. Otherwise, the selected goal fails, and the execution must back up to the previous goal. The selected clause for this previous goal must now be rejected, and another selected. Whenever a clause is rejected in this manner, the bindings associated with it are undone; variables that were bound when the clause head was unified are now free again.

Backtracking continues only until an alternative clause is found for some goal. Then, forward execution is restarted. Execution terminates with success when there are no more goals left in the goal list; when this occurs, all top-level variable bindings are printed out to the user. If the user requests another solution, the system behaves as though its last (successful) unification had failed instead, and backtracks from that point. Through repeated backtracking, all solutions to a query may be found (assuming that the program does not loop). When backtracking reaches the leftmost goal of the initial goal list and

```

f(1) .      g(2) .
f(2) .      g(1) .

:- f(X) , g(X) .

```

Figure 2.1: Prolog program and query.

no more clauses can be found, the execution terminates with failure; all possible solution paths have been searched.

Consider the program and query given in Figure 2.1. From the query, we see that the initial goal list is

```
f(X) , g(X) .
```

Taking the goal  $f(X)$  and the clause  $f(1) .$ , unification succeeds and gives the binding  $X = 1$ . The (empty) body is put on the goal list; the next goal selected is  $g(1) .$  (Note that the binding  $X = 1$  is reflected in this goal.) The clause  $g(2) .$  is selected, and unification fails. On backtracking, the clause  $g(1) .$  is selected; unification now succeeds. The goal list is now empty, so the system reports the binding  $X = 1$  to the user.

Suppose that the user wants another solution: the last unification is then treated as a failure and another clause for  $g/1$  is sought. Since there are no more such clauses, backtracking continues, so that the first clause of  $f/1$  is rejected. The binding  $X = 1$  is rescinded, and the second clause,  $f(2) .$ , is selected. Unification succeeds,  $X$  is bound to 2, and  $g(2)$  is chosen as the next goal. The clause  $g(2) .$  is chosen and unification again succeeds; since no goals remain, the binding  $X = 2$  is printed. If the user were to ask for a third solution, the system would eventually backtrack to the initial goal, find no more clauses to try, and terminate with failure.

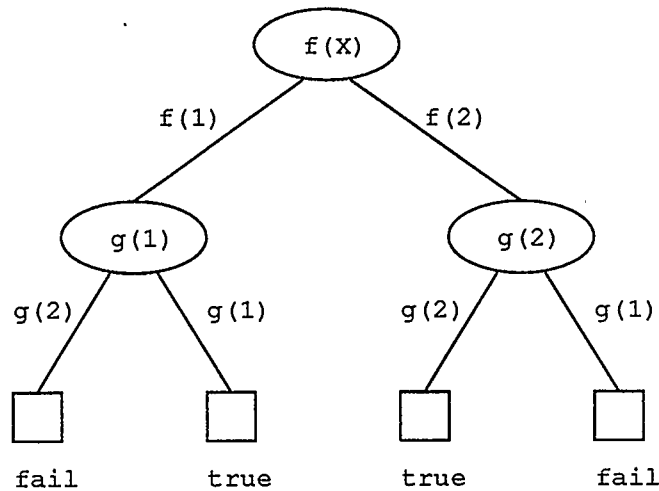


Figure 2.2: Goal tree for execution of a program and query.

Prolog execution can be represented as a tree traversal—specifically, a depth-first traversal. Figure 2.2 shows the tree for the program and query of Figure 2.1. Each node in the tree corresponds to a goal; each edge below a node for some goal corresponds to a clause matching that goal. Thus, each path from the root to a leaf represents a possible solution path. A successful path indicates a solution; a failed path represents a failed unification. The tree is thus disjunctive in its breadth and conjunctive in its depth: for a goal to be true, one or more branches below it must be true; for a branch to be true, each of the goals along its path must be true.

In examining the tree, it is apparent that there are two main types of parallelism available. The first is known as OR-parallelism, in which the clauses of a predicate, each of which may produce a solution independently (i.e. branches in the tree) are searched in parallel. The other is known as AND-parallelism, in which several goals, all of which contribute to one solution (i.e. nodes from one path in the tree) are executed concurrently.

## 2.2 OR-Parallelism

An OR-parallel system attempts to improve on sequential execution speed by parallelizing the clause selection component of the sequential algorithm. Rather than selecting one clause when trying to solve a goal and backtracking to try the others, an OR-parallel system selects all the clauses of a predicate at once and schedules them as separate processes. Subgoals within a clause are still executed sequentially.

Because each possible solution path is allocated its own process, backtracking becomes unnecessary. If a process fails at an attempted unification, it need at most report its failure before it terminates. When a process has no more goals to solve, it succeeds and reports its solution.

For all its potential, however, this scheme is not without its problems. One problem is simply that of *too much* parallelism; attempting to run each process as soon as it is created can cause a combinatorial explosion in the number of processes vying for processor time. A workable OR-parallel scheme must include some scheduling algorithm to maximize processor usage without introducing too much additional overhead.

Another problem lies in maintaining a suitable binding environment. For example, given the program

```
f (1) .
```

```
f (2) .
```

and the query

```
:- f (X) .
```

two processes are created. One process attempts to bind *X* to the value 1; the other

tries to bind  $X$  to 2. In this case, the system must be able to maintain separate binding environments for each process, yet still allow the top-level process that interacts with the user and outputs results to access each binding.

An OR-parallel system lends itself well to search-oriented applications—game-playing programs, for example, or map-coloring systems—in short, any application that is highly nondeterministic. More deterministic programs, such as compilers or operating systems, are poorly suited to OR-parallel computation and would leave such a system's potential largely untapped.

### 2.3 AND-Parallelism

An AND-parallel system attempts to improve on sequential execution speed by parallelizing the goal selection component of the sequential algorithm. Instead of choosing the single leftmost goal from the goal list, a process may choose multiple goals, running each as a new process—each maintaining its own goal list—before itself executing a goal.

Because alternative clauses are still examined sequentially, backtracking is still necessary in an AND-parallel system. This leads directly to the central problem of AND-parallel execution: controlling parallel backtracking, particularly in the context of resolving binding conflicts in shared variables. For example, consider the program of Figure 2.1, running the two goals

$$f(X), g(X)$$

in parallel.

When a binding conflict arises (process  $f$  wanting to bind  $X$  to 1,  $g$  wanting to bind  $X$  to 2, the problem arises of which goal to backtrack. In sequential Prolog, the last goal to

execute would backtrack first, but in the parallel case, the question of which goal executed “last” becomes moot.

There are a number of different ways around this problem, each of which gives rise to a different class of systems. One approach is that taken by the *committed-choice* languages. Another tack is that of the *independent AND-parallel* (IAP) models. Finally, the *backtracking stream AND-parallel* methods attack the problem from a newer angle. Each of these approaches is examined in greater detail in the next chapter; the system that is the basis of this work belongs to the third group.

## 2.4 Other Forms of Parallelism

Other forms of parallelism are found at a lower level; in these forms, goal and clause selection both remain sequential. One possible source of parallelism comes in the unification step of the sequential algorithm. Sequential Prolog unifies terms pair by pair; clearly, this could be parallelized by unifying all pairs of terms at once.

Again, this form of parallelism has problems to overcome. Grain size is a distinct factor; AND- and OR-parallel strategies can rely on the average cost of a sequential unification to provide a grain size large enough to prevent prohibitive overhead. With further decomposition into parallel unification, the average grain size may become very fine indeed.

Another problem with parallel unification echoes that of the binding conflicts seen in AND-parallel execution. Unifications are interdependent with respect to the *occur check* [Lloyd 1984]: for example, two independently acceptable unifications

$$A = f(B)$$



$$B = g(A)$$

are unacceptable when taken together—in standard first-order semantics, a variable may not occur within a structure to which it is bound, so that infinite terms cannot be created. Since most sequential Prolog implementations omit the occur check, this point could be ignored, but it should certainly be considered by any serious implementor.

Finally, the parallelism available to be exploited within a single unification may be severely limited. This is similar to pipelining machine instructions in hardware: the speedup over sequential execution is significant, but it is *not* scalable; adding more processors will not reduce the execution time further.

Another form of parallelism involves pipelining sequential execution [Beer 1990]. The observation that supports this idea is that the only “real” work done by a Prolog system is unification; the rest is overhead: calling procedures, stacking arguments, setting up environments, etc. Consider a root-to-leaf path in a goal tree. The goals that make up this path can be pipelined in the following manner:

- the root goal spawns as a child the leftmost goal of its first clause;
- the root goal performs its own unifications; meanwhile, the child sets up its environment and spawns its own child—the leftmost goal of its own first clause;
- the root goal begins passing arguments to its child; either they will be early, or the child will have to wait for them before it begins its own unifications.

The pipeline only stays full as long as each successive unification succeeds. When a unification fails, the processors “ahead” of the failure must restart with a different goal.

This strategy has at least one benefit: it should be able to speed up any Prolog program, even a highly-sequentialized “dusty-deck” one. The main drawback to this scheme is also speedup: the speedup in any one program is bounded by the number of pipeline stages. Thus, pipelining is not scalable; the only way to use more processors effectively would be to run several programs at once.

Even with this small amount of parallelism, there are losses. A processor may often have to wait for the arguments it needs to perform its unification. Also, parallelism is lost when a unification fails, since the pipeline must then be refilled with goals from an alternate path.

## Chapter 3

### AND-Parallelism

As noted in the previous chapter, the major hurdle in the successful exploitation of AND-parallelism is in handling backtracking in the presence of shared variables. The naive method of attacking this problem would be to avoid sharing altogether, having each goal compute its own solutions independently and combining them afterward to find the complete solutions. This naive method is seen to be infeasible; thus, some other approach must be taken.

One approach—that taken by the *independent AND-parallel* systems—achieves parallelism by running independent goals concurrently. Dependent goals—that is, goals that share variables—are run sequentially, usually using some form of backtracking to produce all solutions.

The *concurrent logic programming* languages take the opposite approach, trying to maximize parallelism by allowing all goals—including those that share variables—to execute concurrently, while disallowing backtracking. The form of parallelism exploited by these languages, combining independent and dependent parallelism is known as *stream AND-parallelism* [Conery & Kibler 1985]: shared variables act as communication channels between processes (goals).

Finally, the *backtracking stream AND-parallel* models feature both parallel execution of dependent and independent goals *and* backtracking, at the cost of execution algorithms rather more complex than those of languages in the other two classes. The effort here is to gain the best of both worlds, achieving maximum parallelism but still allowing all

solutions to be found.

This chapter examines each of these approaches to AND-parallelism and describes in detail representative systems from each class. The relative strengths and weaknesses of each system are considered.

### 3.1 Naive AND-Parallelism

A simple approach would prevent shared variables from occurring at all by having each process keep its bindings local. Backtracking could be kept completely local, and each process could compute its own set of partial solutions. The set of complete solutions could then be generated by taking the cartesian product of all sets of partial solutions and discarding elements with inconsistent bindings. Consider a simple example program

$$\begin{array}{ll} a(1) . & b(2) . \\ a(2) . & b(3) . \end{array}$$

with respect to the query

$$:- a(X), b(Y) .$$

For  $a(X)$ , the set of partial solutions is  $\{ \{X = 1\}, \{X = 2\} \}$ ; for  $b(Y)$ , it is  $\{ \{Y = 2\}, \{Y = 3\} \}$ . The set of complete solutions, given by the cartesian product, is thus  $\{ \{X = 1, Y = 2\}, \{X = 1, Y = 3\}, \{X = 2, Y = 2\}, \{X = 2, Y = 3\} \}$ .

This method provides a high degree of parallelism, but as Conery and DeGroot point out, it suffers from a number of drawbacks [Conery 1987, DeGroot 1984]. One such drawback is that since each process runs and produces bindings independently, much

work can be wasted; the bindings made by one process do not constrain those made by another. Consider the program above with respect to the query

$$:- a(X), b(X).$$

The resulting cartesian product is  $\{\{X = 1, X = 2\}, \{X = 1, X = 3\}, \{X = 2, X = 2\}, \{X = 2, X = 3\}\}$ . Discarding inconsistent and redundant bindings, we come up with  $\{\{X = 2\}\}$  as the solution set. Surely much of the computation, particularly in the construction of the cartesian product, was wasted. In more complex examples, computation in generating sets of partial solutions would also be wasted.

Another problem is that not every goal will generate bindings; some (for example  $>/2$ ) require one or more ground inputs. How can such a goal produce any partial solutions? Surely we could not ask  $>/2$  to generate all possible sets of bindings.

Thus, we must conclude that the local-binding method is infeasible, given the problems associated with dependent goals. Some other approach is necessary that allows dependent goals to be handled gracefully and (more) efficiently.

### 3.2 Independent AND-Parallelism

To avoid binding conflicts, the independent AND-parallel systems must determine which goals can safely be run in parallel. This may be done statically, at compile time; alternatively, it may be done dynamically, at run time. In both cases, the object is the same: imposing some (partial) ordering of the goals to maximize parallelism, while still preventing binding conflicts. The rule is simple: *no variable may be bound by more than one goal*. Thus, goals with non-overlapping sets of variables may safely be run in parallel; goals with variables in common must be serialized. A goal that binds a variable is a *producer* of that

variable; other goals sharing the variable are *consumers*. A consumer of some variable must wait for the producer to bind it completely; then the consumer may begin execution and produce bindings of its own, for some other consumer.

Static analysis, such as that used to produce the data join graphs of Kalé's REDUCE-OR Process Model [Kalé 1985], is attractive because it adds no run-time overhead; decisions about which goals to run in parallel are all made at compile time. Because less information about the variables is available at compile time than at run time (for example, two variables, apparently distinct at compile time, may be aliased to one another during execution), static analysis must be conservative. Thus, potential parallelism may be missed; for example, two goals may share a variable, but if that variable is bound, both goals may safely execute concurrently. Mode annotations are often used to help determine a producer for each variable, thus making static analysis easier and more precise.

On the other hand, dynamic analysis, such as that used in Conery's AND/OR process model [Conery 1987], needs no annotations; all the information necessary to determine goal dependence or independence can be found by examining the state of the variables involved. Thus, (independent) parallelism is maximized. The main problem with dynamic analysis lies in the run-time overhead that results from continually having to check on the state of variables as the computation progresses.

DeGroot's Restricted AND-Parallelism (RAP) model [DeGroot 1984] takes a hybrid approach: static analysis determines potential parallelism, while simple run-time checks determine whether that potential parallelism may be exploited. Static analysis may determine that two unbound variables could be independent; if a run-time check confirms this, their respective goals are run in parallel. Similarly, a run-time check determines whether a shared variable is already bound; if so, goals that share it need not be sequentialized.

Backtracking in the IAP context is as a rule sequentialized. If an independent goal fails, it may backtrack on its own, but in the general case, failure of a dependent goal must be handled as well. For example, the consumer of some variable may fail. This failure may be local, so this possibility must be examined first; only when it is determined that the failure is due to the externally-bound variable can the producer be made to backtrack. After backtracking locally, this producer may in turn need a producer for some other variable to backtrack . . . and so it goes.

An exception to this general rule is the SYNC model [Li & Martin 1986], in which variable bindings are in effect pipelined from producers to consumers, with each set of distinct bindings separated by a synchronizing marker. Once a goal is solved, it may backtrack immediately to find more solutions. Solutions must therefore be buffered, and “shared” variables must be kept local. This is somewhat reminiscent of the naive algorithm presented in Section 3.1, except that the pipelining of results from producers to consumers has the desirable effect of discarding inconsistent sets of bindings as soon as they are detected, rather than first generating them completely. On the other hand, this method may not exhibit much more parallelism than sequentialized backtracking, particularly for highly deterministic problems or for programs in which sibling goals differ greatly in computation time (i.e. the speed of a pipeline is determined by the speed of its slowest stage).

From this discussion, it is clear that IAP systems are not well suited to highly deterministic problems, except those with a high degree of data independence (matrix multiplication, for example). They are better suited to nondeterministic problems, in which a potential solution may have to be discarded and another tried.

### 3.3 Concurrent Logic Programming Languages

The concurrent logic programming (CLP) languages offer stream AND-parallelism, in which shared variables may be thought of as *streams* which communicate bindings between processes. All goals—even those with variables in common—may execute concurrently. Languages in this class include PARLOG [Clark & Gregory 1986], Concurrent Prolog [Shapiro 1983], and GHC [Ueda 1985] and their “flat” versions; newer languages like Strand [Foster & Taylor 1990] have recently come on the scene as well.

Like many independent AND-parallel systems, the concurrent logic programming languages reduce the problem of binding shared variables by requiring that each variable have exactly one producer; other processes sharing a variable are consumers. Unlike the case in IAP systems, however, a variable need not be completely ground before a consumer in a CLP system can use the binding. For example, consider the code

<pre>produce ([X L]) :-     generate (X) ,     produce (L) .</pre>	<pre>consume ([X L]) :-     use (X) ,     consume (L) .</pre>
--	---

with respect to the query

```
:- produce (L) , consume (L) .
```

In an IAP system, `produce/1` would have to terminate before `consume/1` could begin; in a stream AND-parallel system, `consume/1` can use each list element as it is produced. That is, a consumer can execute as soon as the variables it is waiting for are sufficiently bound; if a variable is bound to a structure, variables within that structure may still be free, and indeed may be bound by another process. The initial producer may even wait



for some consumer to bind the structure further; this is known as *back communication* [Clark & Gregory 1986].

In PARLOG and Strand, the designation of producers and consumers is done via *mode declarations*: for each predicate, arguments are specified as inputs or outputs. If a variable designated as an input is not bound when the predicate is called, the call suspends until the variable has been bound. In Concurrent Prolog, suspension is done instead through the use of *read-only variables* as goal arguments: if solving a goal would bind a read-only variable, then the goal is suspended until the variable has been bound by some other goal. In GHC, suspension occurs when a *guard goal* attempts to bind a variable; the goal then waits until that variable has been bound.

The ability to have processes cooperate on constructing a solution, rather than requiring a producer to complete its entire computation before any consumer may begin, is the essence of stream AND-parallelism; it exploits parallelism inaccessible to IAP systems.

Because processes can cooperate in creating the full binding of a variable, backtracking can be much more complicated than in the IAP case. To avoid this, CLP languages give up the backtracking of Prolog and the *don't-know nondeterminism* associated with it. These are replaced by *guarded clauses* and *don't-care nondeterminism*, in which the system *commits* to the first clause found to be acceptable; all other clauses are discarded. The concurrent logic programming languages are thus also known as “committed-choice” languages.

A guarded clause has the form

$$\text{head} :- \text{guard} \mid \text{body}.$$

where “ $\mid$ ” is the *commitment operator*. When a goal is executed, *argument matching* (a restricted form of unification) is attempted with a clause head. If a clause head matches

successfully (e.g. all input variables are bound (PARLOG), or no attempt is made to bind a read-only variable (Concurrent Prolog)), the clause's guard goals are executed. When all of the guard goals have succeeded, the commitment operator is executed, and the system commits to that clause. At this point, any guard computations for alternate clauses are aborted, and the body goals of the committed clause are executed in parallel. Also, only after commitment are any bindings made in the goal/head matching actually made and transmitted.

This non-backtracking approach eases the shared-variable binding problem considerably. Once a variable has been bound, the binding is permanent, and no information need be kept to tell which process bound it. Thus, goals need not be ordered as they are in the independent AND-parallel case.

In terms of application areas, the CLP languages are orthogonal to the IAP systems. The committed-choice systems are well suited to deterministic problems, because the focus is more on exploiting maximum parallelism within a solution path than on finding multiple solutions or on having to search for the correct solution path. For nondeterministic problems, the CLP systems fare poorly, since they do not allow backtracking. Some nondeterminism is available via the use of OR-parallel all-solutions predicates, but these are limited.

### 3.4 Backtracking Stream AND-Parallel Systems

These systems attempt to combine the advantages of both the independent AND-parallel systems and of the concurrent logic programming languages: respectively, backtracking and stream AND-parallelism. For several years, combining the two was deemed impracti-

cal. Recently, however, a number of algorithms have been published that efficiently combine stream AND-parallelism and backtracking [Cleary *et al* 1987, Somogyi *et al* 1988, Tebra 1987].

Central to all of these algorithms is the notion of imposing a total ordering on the goals executed by the system. The natural temporal goal ordering of sequential Prolog is what allows it to backtrack successfully, but in a distributed environment, there is no such ordering readily available. Thus, some ordering must be imposed artificially.

The ordering is used to determine the *priority* of a goal: the earlier it appears in the ordering, the higher its priority. If two goals disagree on the value of a binding, the binding made by the higher-priority goal is accepted; the lower-priority goal must retract its binding and recompute. When a lower-priority goal can find no solution, it may ask a higher-priority goal to recompute its bindings.

[Tebra 1987, Tebra 1989] in fact imposes on the computation Prolog's standard depth-first ordering. The main advantage of this ordering is that it ensures that solutions are delivered in the same order as they would be by a sequential system. There is, however, the potential for a lot of "wasted" work: if a binding made deep in some branch of the search tree conflicted with one made to the left of it, it would have to be retracted since the binding to the left would have higher priority. Any work done below the right-hand binding would have to be undone. If it were later found that the left-hand binding was incompatible with any right-hand binding, work on the left-hand side would have to be undone and work on the right-hand side actually *redone*. Tebra calls his system *optimistic* in that it assumes that allowing processes to compute ahead will more than offset losses in having to undo and redo work.

[Somogyi *et al* 1988] orders goals on the basis of their producing or consuming vari-

ables; as with many of the IAP and CLP systems mentioned before, each variable has exactly one producer. The ordering is such that producers always come before consumers. Assuming that the producers have been chosen correctly, there will be no wasted work as there may be in Tebra's system: no lower-priority goal will have to retract a binding incompatible with that of a higher-priority goal (though it may still have to ask a higher-priority goal for a new binding). On the other hand, this system is *conservative*, since consumers are delayed, waiting for variable bindings rather than computing ahead. Another drawback is that *every* producer must be known at compile time, requiring the programmer to supply sophisticated mode declarations.

The Delta Prolog system [Pereira *et al* 1986] takes an approach similar to Tebra's system, except that interprocess communication and synchronization is achieved explicitly via *event goals*, rather than implicitly via shared variables. An event is a synchronous communication between two processes. Thus, processes may not compute ahead, and the system must be classified as conservative.

Finally, [Cleary *et al* 1987] presents an algorithm in which a Virtual Time system [Jefferson 1985] is used to impose an ordering on the goals. This too is an optimistic system, requiring no prior knowledge of producers and consumers. Thus, mode declarations are not necessary—though they are allowed, and can be useful in clear producer/consumer situations, for example. The ordering is explicitly not depth-first; rather, goal priorities are distributed more evenly through the breadth of the tree. Processes should therefore stay more closely synchronized (in terms of goal priority), and the depth of computations to be undone and/or redone should be less than the corresponding depth under Tebra's scheme.

The backtracking stream AND-parallel systems are designed to work well for both deterministic and nondeterministic problems—that is, as well as the CLP systems for

deterministic problems and as well as the IAP systems for nondeterministic ones. The purpose of this thesis is to report on an implementation of the algorithm given in Cleary *et al* and to demonstrate that this algorithm and the implementation live up to this standard.

## Chapter 4

### AND-Parallelism Using Virtual Time

The distributed Prolog system consists of a number of components. Each of these—the underlying Time Warp system, the stack structure of each process in the system, the messages used by these processes to communicate—are examined in detail, after which the relationship between these components is explored. Next, the algorithm presented in [Cleary *et al* 1987] is considered as a whole; finally, a number of possible optimizations are discussed.

#### 4.1 Virtual Time and Time Warp

A virtual time system [Jefferson 1985] imposes on a computation a temporal coordinate system; all events in the computation are viewed in terms of this coordinate system. Each process has its own *local virtual time* (LVT); each event (in a Prolog system, each goal-head unification) receives its own timestamp based on the current LVT. Time increases with each event, and execution is finished when all processes have a local virtual time of  $+\infty$  (i.e. the *global* virtual time (GVT) is  $+\infty$ ).

Virtual time is domain-specific and need not be related to real time. For example, in distributed simulation, the natural basis for virtual time is simulation time. In a Prolog system, an ordering based on the search tree maps easily onto a virtual time system.

What distinguishes virtual time from other strategies is that it is *optimistic* rather than *conservative*. A process in a conservative system, before it can receive a message from

some other process, must be sure that no other message should have arrived earlier. In an optimistic system, on the other hand, a process assumes that messages will arrive in the correct order, and receives them immediately. If the correct-order assumption holds, an optimistic system clearly wins. When the assumption is false, however (i.e. when a message arrives out of order), the computation will be incorrect unless the ordering is repaired. In this case, the optimistic system is little worse off than the conservative one: computation time wasted by an optimistic process will equal blocking time wasted by a conservative process performing the same computation; the optimistic method has only the extra overhead of undoing the incorrect computation.

The virtual time definition does not specify how this order repair is to be carried out; this is left up to the individual implementation. The first implementation of virtual time was the *Time Warp* mechanism [Jefferson & Sowizral 1985]. It was designed with parallel simulation in mind, but the ideas behind it can be applied as well to parallel Prolog.

The key component of the Time Warp mechanism is *rollback*; this is used to return the computation to an earlier state. When a message arrives out of order at a Time Warp process, the process performs a rollback to the virtual time of the message (given by its timestamp); then, forward execution starts again, processing the incoming messages in the correct order. To accomplish a rollback to a given time, a process must perform several operations:

- it must “unreceive” already-received messages whose timestamp is greater than the given time;
- it must cancel outgoing messages whose timestamp is greater than the given time;
- it must restore its internal state to what it was at a time just before the given time.

Clearly, then, some form of state-saving is necessary. A Time Warp process uses three queues to do this: an input queue (IQ), an output queue (OQ), and a state queue (SQ). The IQ contains (in timestamp order) incoming messages for the process. The OQ holds *negative* copies of all messages sent out by the process; a message is cancelled simply by sending out its corresponding *anti-message*. The SQ contains “snapshots” of the process at various virtual times; the internal state can be reconstructed using these snapshots.

Receipt of an anti-message may also cause a rollback. If its corresponding positive message is on the IQ but not yet received, the two messages can just “annihilate” each other; if the positive message has been received, the system must perform a rollback to the time of that message before annihilation may occur.

## 4.2 Stacks and Frames

Like most sequential Prologs, the execution of the algorithm is based around a stack. Entries on the stack are *frames*. One frame is created for each resolution step; such frames are known as *local* frames. In the distributed case, *remote* frames are also created to record variable bindings from other processes.

Each frame contains several pieces of information, including the following:

- a unique timestamp
- the identity of the frame’s originator
- a unique identifier

Frames are kept on the stack in timestamp order: the frame with the earliest timestamp is at the bottom of the stack; that with the latest timestamp is on the top. This ordering is



based on the virtual time paradigm [Jefferson 1985] and replaces the depth-first backtrack order of the sequential algorithm. For local frames, this timestamp order is exactly the depth-first order; the distinction comes from the remote frames, which are interspersed among the local frames in the timestamp ordering.

A frame's originator must be known to allow for correct backtracking when a process can find no solutions compatible with the bindings of another process; the failing process must be able to cause the other process to backtrack.

A unique identifier for each frame is necessary because timestamps, although they are unique (in the sense that no two frames on any one stack may have the same timestamp), can be reused after a rollback or backtrack. For example, if a clause for some goal is backtracked and another clause chosen, the frame with the old clause and that with the new will have the same timestamp. Since messages may (and often do) arrive late, a message intended to affect an old frame may errantly affect a new one instead. Thus these frames must be disambiguated; a simple integer counter for each process suffices for this task.

### 4.3 Messages

Because of the distributed character of the algorithm, processes must communicate with each other via messages, rather than by merely binding or unbinding variables, as in the shared-memory approach. Thus, variable bindings must be disseminated explicitly; on backtracking, these bindings must be retracted explicitly. This hooks in neatly to the Time Warp concept of message/anti-message pairs. A binding can be propagated via a BIND message; if that binding is later backtracked, it can be withdrawn via an ANTI-BIND message that will annihilate the original BIND.

A third type of message is necessary for a Prolog system, one that is not present in the Time Warp scheme. (This is because Time Warp assumes a deterministic execution model, so that virtual time will always increase.) This is the FAIL message, through which a failing lower-precedence goal in one process may cause a higher-precedence goal in another to backtrack.

#### 4.4 Mixing Prolog and Time Warp

The Prolog system needs to provide its own versions of the Time Warp input, output, and state queues. The SQ has an immediate Prolog analogue: the stack. The contents of the stack up to a given virtual time exactly specify the state of a process at that time. The stack can also serve as an OQ: messages sent at a given virtual time can have their anti-messages stored within the stack frame of that time. The only queue that needs special treatment is the IQ, since it may contain unprocessed messages that are in the future of the receiving process; the stack can only record messages from the past.

The IQ holds messages of all three kinds. The BIND message is a classic Time Warp message; after being received it will remain in the input queue until its corresponding ANTI-BIND arrives and annihilates it. That is, it remains in the queue even after it has been processed, so that if a rollback causes it to be “unreceived,” it will be reprocessed when the receiving process begins forward execution again.

The ANTI-BIND messages are also persistent, since an ANTI-BIND may arrive before its corresponding BIND. Such an ANTI-BIND is not processed; it is merely enqueued until its BIND arrives, whereupon both are annihilated. The odd one out is the FAIL message. Since it is not a Time Warp message, a FAIL is removed from the input queue immediately

on being processed the first time, never to be replaced.

The OQ holds only ANTI-BIND messages, each to be sent off when a rollback causes the local virtual time to fall below that message's timestamp. No BIND messages are stored in the OQ; for each BIND sent out, the corresponding ANTI-BIND is enqueued in the OQ. FAIL messages are never stored in the OQ either; once sent out, they are forgotten completely by the sender.

## 4.5 Execution Model

The distributed algorithm can be broken into several phases. Those most similar to parts of sequential Prolog are examined first: forward execution and local backtracking. Next, rollback and remote backtracking, which both relate to interactions between processes, are discussed. Finally, an example that demonstrates all of these phases is presented, giving a unified view of the system.

Note that the execution model given below is very general; while it refers to processes, these are defined very loosely. They could be physical processes, distinguishable by the operating system; they could be logical, like those in a process-model view [Conery 1987]. This choice is left up to the implementor. Similarly, the algorithm does not specify whether parallelism is to be implicit or explicit (annotated); again, this is up to the implementor.

### 4.5.1 Forward Execution

Forward execution is straightforward. A process will select a goal to execute, next select a clause, and then attempt to unify the two. This is recorded in a local stack frame, along with any bindings made during unification; the timestamp of the frame is set to the current local

virtual time. Values for any variables bound by the unification are then sent (via a BIND message) to every other process that shares any of those variables. Next, processes may be spawned for the parallel execution of goals in the body of the selected clause. Finally, the process checks for any incoming messages. Local virtual time is then incremented, and the loop begins again with goal selection; this continues until no more goals are available to execute. When all goals on all processes have succeeded, a solution has been found and may be printed.

#### **4.5.2 Local Backtracking**

Local backtracking is also simple. This phase begins as a result of a local goal failure; the previous frame on the stack is then backtracked. If this frame is a local frame, then backtracking occurs locally. (If the frame is remote, then remote backtracking must occur.) When a local goal fails, the variable bindings resulting from the unification of the previous goal and its current clause are undone. For each BIND message sent out after that unification, an ANTI-BIND message is now sent out. Also, any new processes created after the unification are destroyed. If another clause is available for the backtracked goal, forward execution begins again with their unification; otherwise, backtracking continues down the stack.

#### **4.5.3 Rollback**

Rollback may occur at the end of a forward execution cycle, when checking for incoming messages. If a BIND or ANTI-BIND message arrives whose timestamp is earlier than a process's current LVT, the process must roll back its state to what it was at a virtual time just before that of the message; only then may it accept the message. That is, all

of the work—goal and clause selection, unification, output BIND messages, and process creation—that was done at a virtual time after the message arrival time must be undone.

If the message that caused the rollback was a BIND, a remote stack frame is created whose timestamp is equal to that of the received BIND. Binding values from the message are stored in this frame and compared with local values. If all bindings are compatible, the frame is retained; otherwise, it is discarded, and the originator of the BIND must backtrack. (There is no need to send a FAIL message, however; the originator will itself receive a BIND with the earlier values, thus rolling back on its own.) In either case, the receiving process then simply restarts its forward execution, possibly redoing some of its rolled-back work.

If the received message was an ANTI-BIND, it will annihilate its corresponding BIND, and the remote frame for that BIND will be removed; variable bindings due to the BIND will be retracted. As in the previous case, the receiving process then restarts forward execution. (If the ANTI-BIND should somehow arrive before its corresponding BIND, no rollback is necessary; the receiver need merely hold on to the ANTI-BIND until the proper BIND arrives, at which point they annihilate each other.)

#### 4.5.4 Remote Backtracking

Like local backtracking, remote backtracking is initiated by a local goal failure in some process. In this case, however, the backtracking mechanism finds that it can no longer backtrack locally: the stack frame it attempts to backtrack turns out to be a remote frame. At this point, the process sends a FAIL message to the originator of the remote frame, so that the bindings in that frame can be backtracked.

After sending a FAIL message, a process *restarts forward execution* from the virtual

time at which it sent the FAIL. It does *not* continue backtracking, since the recipient of the FAIL is now backtracking. For any one failure, only one process—whether or not it is the one that originally failed—may backtrack at a time; backtracking must be serialized in order that potential solutions are not missed. Of course, if a process fails on its own while backtracking is under way elsewhere, it may begin backtracking itself. If the two backtrack paths stay separate, both may continue; if both cause the same process to fail, backtracking continues with the earlier failure of the two.

The recipient of a FAIL message checks first to see that the message refers to a valid frame. If not, then the frame must already have been backtracked or rolled back, so the message is ignored and forward execution continues. If the referenced frame is a valid one, it must be backtracked. To accomplish this, the process rolls back to a time just after the timestamp of the frame in question; it then begins backtracking the frame.

It is important to note that simply being able to make another process backtrack is not sufficient for correct remote backtracking. The bindings rejected by the sender of the FAIL may not even be the cause of that sender's failure; they may merely have the latest timestamp of a large group of "suspects," each of which *could* have contributed to the failure. An earlier binding in that group may be the real culprit. Thus, a process that receives a FAIL message needs some context with that message, since it may eventually backtrack to the time of the next-latest suspect. If this occurs, the process must then stop backtracking and force another process to backtrack—specifically, the binder of that next-latest suspect.

This context may be maintained in the stack of a FAIL's recipient by inserting a remote frame whose originator is the FAIL's sender and whose timestamp is that of the sender's previous stack frame. If it encounters this frame during backtracking, it reacts as it would

to any other remote frame: it sends a FAIL message (including the timestamp of its own previous frame) back to the originator, and restarts forward execution. The originator will then take over backtracking again, inserting a new remote frame in *its* stack.

Remote backtracking actually provides a weak form of intelligent backtracking. In sequential Prolog, a goal that was executed before a failed goal but after the cause of the failure will be backtracked, even if it is independent of the failed goal. In the parallel system, such an independent goal will not be backtracked, since it will not appear in the context of the failing goal. Refinements to this feature are discussed in [Cleary *et al* 1987]; [Somogyi *et al* 1988] extends intelligent backtracking to certain types of dependent goals.

#### 4.5.5 Example

In order to understand the algorithm more clearly, an illustrated example demonstrating each phase of the execution may be of benefit.<sup>1</sup> Consider the query

$$:- a(X), b(X)$$

run in parallel with respect to the program

$$\begin{array}{llll} a(1) . & b(X) :- c2(X) . & c2(2) . \\ a(2) . & b(X) :- c1(X) . & c1(1) . \end{array}$$

The processes in this execution will be denoted  $P_a$  and  $P_b$  for top-level goals  $a(X)$  and  $b(X)$  respectively.

At first, both  $P_a$  and  $P_b$  proceed with forward execution (see Figure 4.1).  $P_b$  selects goal  $b(X)$  and clause  $b(X) :- c2(X) .$ , and unifies them, all at virtual time 1. It then continues executing forward, selecting goal  $c2(X)$  at time 6. Meanwhile,  $P_a$  selects goal

---

<sup>1</sup>Note that timestamps in this and later examples are quite arbitrary, and have no special meaning.

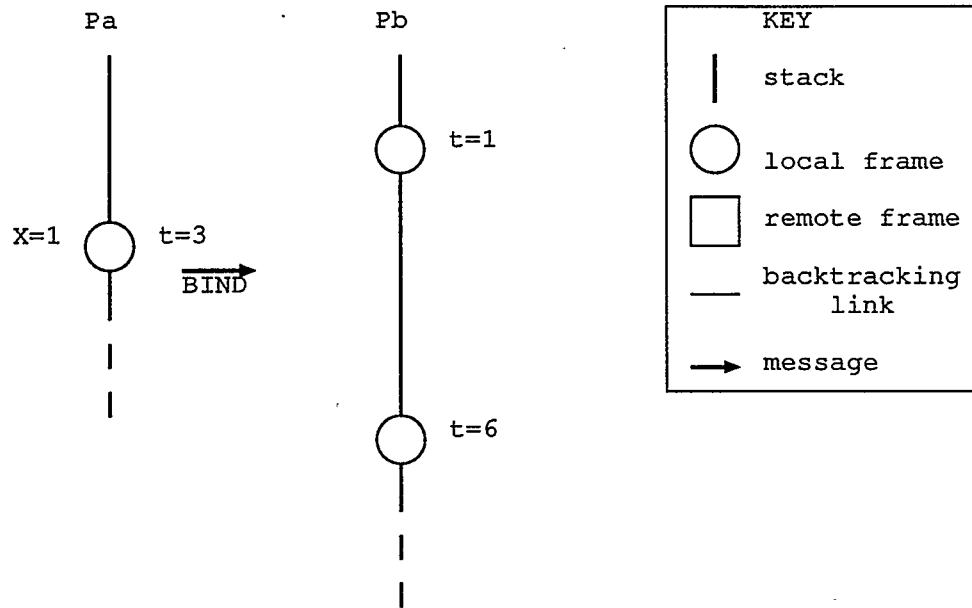


Figure 4.1: Forward execution

$a(X)$  and clause  $a(1)$  ., and unifies them at time 3. This unification produces a binding for the shared variable  $X$ ; thus  $P_a$  sends a **BIND** message to  $P_b$  with a timestamp of 3.

$P_b$ , now executing at time 6, receives the **BIND** message from  $P_a$  and discovers that it must roll back to process the **BIND** properly. It does so, rolling back to time 3 and accepting the binding  $X = 1$ .  $P_b$  then restarts forward execution, executing the goal  $c2(1)$  (note the variable substitution) at time 6. Meanwhile,  $P_a$  finds that it has no more goals to solve; thus, it sets its virtual time to  $+\infty$  and awaits termination. (This is the situation in Figure 4.2.)

After selecting clause  $c2(2)$  .,  $P_b$  finds that unification with goal  $c2(1)$  fails. Because of this failure, it begins backtracking locally, trying to find another clause. When this search fails, backtracking proceeds one step further back and encounters a remote frame originating at  $P_a$ .  $P_b$  then initiates remote backtracking by sending a **FAIL** message to  $P_a$ ,



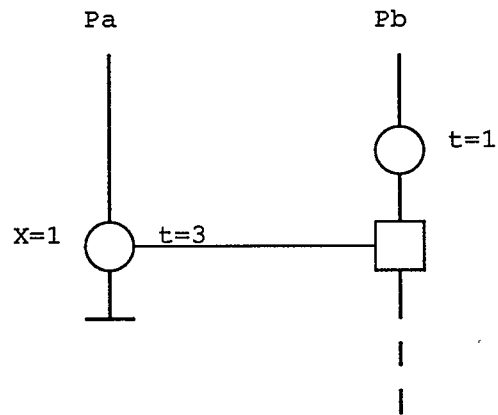


Figure 4.2: Rollback

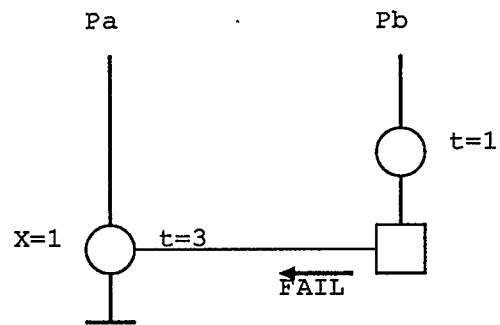


Figure 4.3: Remote backtracking

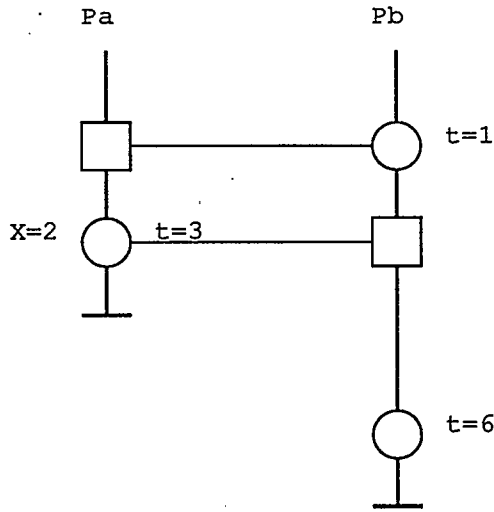


Figure 4.4: Termination with solution

including the information that it began the backtracking and that  $P_a$  should not backtrack to before time 1 (see Figure 4.3).

When  $P_a$  receives the FAIL message, it immediately backtracks to time 3, retracting the binding  $X = 1$  and sending out the corresponding ANTI-BIND message. It then tries to find another clause to match  $a(X)$ . Finding the clause  $a(2)$ , it begins forward execution again, unifies the two, and produces the binding  $X = 2$ , again to be sent to  $P_b$ .  $P_b$  receives the message from  $P_a$  and accepts the binding  $X = 2$ . It then selects goal  $c2(2)$  and clause  $c2(2)$ , and finds that they unify. After this, neither  $P_a$  nor  $P_b$  have any more goals to solve; both set their virtual times to  $+\infty$ , and the system terminates with the solution  $X = 2$  (see Figure 4.4).

While this example is quite straightforward, it demonstrates many of the mechanisms used by the system. Note the new remote frame in  $P_a$ 's stack; if another solution were requested,  $P_a$  would eventually force  $P_b$  to backtrack to time 1 and select its second clause for  $b(X)$ ; thus the context contained in the original FAIL message of Figure 4.3) allows

further backtracking without missing potential solutions.

## 4.6 Optimizations

One of the premises of Time Warp is that rollbacks will be infrequent enough and shallow enough that their cost will be low compared to the benefit gained from allowing processes to compute ahead without waiting for messages to arrive. This premise should also apply to a Time Warp-based Prolog system. Whether or not that premise holds, it is clearly beneficial to attempt to reduce both the frequency and severity of rollbacks.

Prolog has an advantage over other applications in that its messages are highly defined: variable bindings are either being asserted or retracted. Thus, it is easy to tell what effect a message will have on the state of the system, just by inspecting its contents. Two optimizations suggest themselves, one applying to BIND messages and one to ANTI-BIND messages. These optimizations are examined below.

The algorithm presented in [Cleary *et al* 1987] also attempted to minimize the number of FAIL messages sent, while the algorithm presented here does not. When the Prolog system was implemented, what appeared to be an obvious optimization for reducing FAIL messages turned out to be an *over*-optimization.

In the text that follows, each of these optimizations is described and given motivation. The BIND and ANTI-BIND optimizations were not implemented; unfortunately, the FAIL over-optimization was. Details on the proposed or actual implementation of each may be found in the following chapter.

#### 4.6.1 BIND Optimization

The idea behind the BIND optimization is that some, even many, BIND messages need not cause rollbacks. The only real criterion for requiring rollback is some incompatibility between the bindings carried in a BIND message and the other bindings known to the receiving process.

If the two sets of bindings are not inconsistent with each other, it should not be necessary to roll back, absorb the incoming message, and work ahead again. The very same state can be achieved simply by incorporating the BIND message as a remote frame in the stack and updating timestamps on local bindings, thus saving work in rollback and especially in recomputation.

If the two sets of bindings *are* mutually inconsistent, two possibilities exist. Among the bindings that are inconsistent, either at least one binding from the BIND message has a later timestamp than its corresponding locally-known binding, or none have a later timestamp. In the first case, the BIND message will be rejected whether or not a rollback is performed, so there is clearly no point in rolling back.

In the second case, a rollback is necessary, but even here there are potential gains to be had: rather than rolling back all the way to the time of the BIND message, the system need only roll back to the time of the earliest binding conflict. At this point, no bindings will be inconsistent; the BIND message can be integrated into the stack just as in the wholly-consistent case, and forward execution can begin again.

In the best case, a rollback becomes completely unnecessary. Even in the worst case, the severity of the rollback may be significantly reduced. Only when a rollback to the time of earliest inconsistency is equivalent to a full rollback are no gains realized.

Thus, implementing this optimization seems well worthwhile; a possible implementation is presented in the next chapter.

#### 4.6.2 ANTI-BIND Optimization

A similar notion of avoiding rollback and recomputation occurs for receiving ANTI-BIND messages. Compared to performing a rollback, removing the remote frame corresponding to the BIND to be annihilated, and executing forward again, it would be much quicker simply to remove that frame and the bindings associated with it.

Unfortunately, and unlike the case for the BIND optimization, adding the ANTI-BIND optimization makes the execution algorithm rather more complex [Cleary *et al* 1987]. The distinction here is that accepting a BIND message and the variable bindings it contains has the effect of constraining the solution space; receiving an ANTI-BIND message and undoing some variable bindings has the opposite effect. That is, once some bindings are removed, a previously-rejected search path may become acceptable again. If the algorithm is to have any chance at completeness, such potential solution paths must eventually be retried.

As for the BIND optimization, a proposed implementation of the ANTI-BIND optimization is given in the following chapter.

#### 4.6.3 FAIL Over-optimization

The aim of trying to optimize FAIL messages is to reduce the number of FAIL messages sent, and consequently to minimize wasted rollbacks. In the unoptimized algorithm, the context sent with a FAIL message refers to the timestamp of the previous frame on the sender's stack. If the FAIL's recipient cannot find an alternative without backtracking to

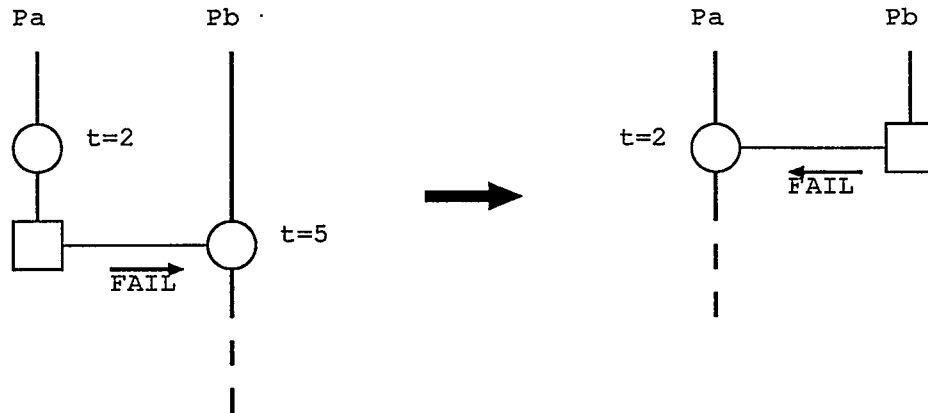


Figure 4.5: Process  $P_a$  initiates remote backtracking, causing  $P_b$  to fail.  $P_b$  backtracks to the time given in the FAIL's context without finding a new solution, and sends a FAIL back to  $P_a$ , which then backtracks again.

before that timestamp, it sends a FAIL message back to the sender (along with its own previous frame as context, of course).

In some cases (for example, in Figure 4.5), the previous frame sent as context is local to the sender; in this case, if the failure comes back to it, the original sender will backtrack locally. Often, however, the context refers to a remote frame, as in Figure 4.6. In this case, when a FAIL is directed back at the original sender, that process rolls back all of its later state, sends out a FAIL to a third process, and executes forward again. This is wasteful: work is rolled back and then immediately redone, and another FAIL message is sent.

The optimization proposed in [Cleary *et al* 1987] suggests that failure can be directed immediately to the third process, bypassing the originator completely. This may be accomplished by including the process ID of the previous frame's originator in the context, along with its timestamp; this method is illustrated in Figure 4.7. This saves process  $P_a$  from having to roll back, and results in one less FAIL message being sent out.

Though attractive, the optimization is buggy. In some cases, passing failure on to the

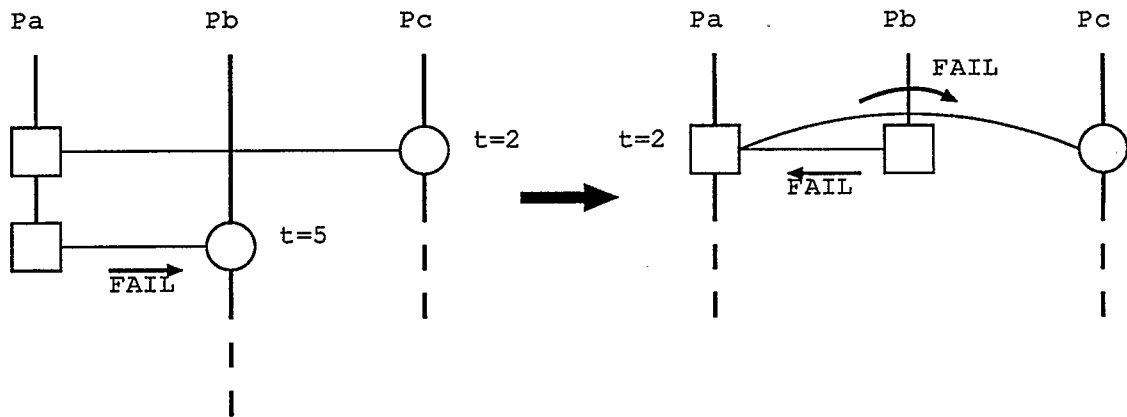


Figure 4.6: As for two-process backtracking, except that  $P_a$  passes failure on to  $P_c$  immediately on receiving the FAIL itself.

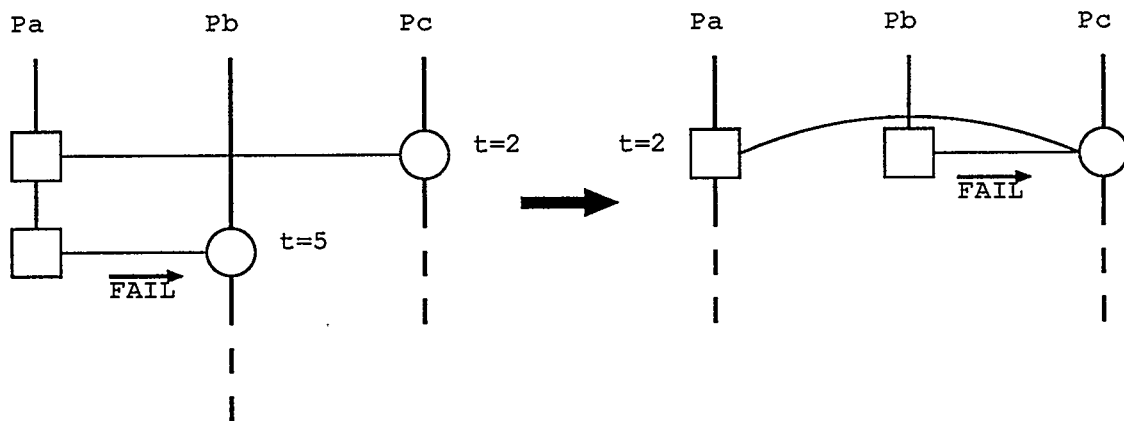


Figure 4.7: Rather than sending a FAIL back through  $P_a$ ,  $P_b$  can send it directly to  $P_c$ , and avoid making  $P_a$  roll back.

previous frame's originator rather than back to the FAIL's originator results in incorrect execution. Details are given in the next chapter.



# Chapter 5

## Implementation

Based on the algorithm presented in the previous chapter, I have designed and implemented an AND-parallel interpreter for pure Prolog. This interpreter runs Edinburgh-style Prolog programs explicitly annotated for parallel execution. The real work of the system is done by *solver* processes; these solvers are accompanied by *ear* processes, which aid in interprocess communication.

This chapter begins with an overview of the system; this is followed by a detailed look at solver and ear processes. Next, several features of the system are examined: the distributed variable binding environment, the use of delay annotations on user and builtin predicates, and the detection of system termination. Finally, the optimizations introduced in the previous chapter are discussed; though they were not implemented, the structures and strategies necessary to implement them are presented.

### 5.1 Overview of the System

As noted previously, the interpreter uses a message-passing approach for interprocess communication, rather than the shared-memory approach found in many parallel Prolog systems. Each approach has its own advantages and drawbacks, but these are beyond the scope of this thesis. For reasons of stability, reliability, and ease of use, the JIPC system [JADE 1985] was chosen as the message-passing subsystem. The use of JIPC does, however, constrain the Prolog system. Since JIPC runs under the UNIX operating system,

JIPC processes are UNIX processes; thus, the system was designed under the assumption that processes<sup>1</sup> are heavyweight entities.

Rather than having every goal become a process, only particular goals do so (specifically, those designated as parallel in the text of the input program). Because of this, the parallelism exploited is generally less, but the overhead of process creation and scheduling is drastically reduced. Goals may be run in parallel with only minor annotations: the clause

$$p(X) \text{ :- } q(X) @p1, r(X) .$$

indicates that when the head  $p(X)$  is unified,  $q(X)$  should be run as a separate process;  $r(X)$  is executed as part of the original process.

Although parallelism must be designated, *any* goal may be executed in parallel, whether it is completely independent from any other goal, part of a producer/consumer relationship, or one of a group of goals that could provide competing bindings. In this, the system differs from both the IAP and CLP systems. Independent AND-parallel schemes require that dependent goals be executed sequentially. Concurrent logic programming languages allow parallelism among goals that could provide conflicting bindings, but due to their committed-choice semantics, they will provide at most one possible solution even if many are available. Such systems may even report failure when a solution actually exists.

The system also uses *delay annotations* to prevent wasted execution. Such annotations allow a goal to be unified only if its arguments are sufficiently instantiated. For example, a goal  $q(X)$  may be delayed until its single argument is bound; to accomplish this, the predicate  $q/1$  would include the declaration

---

<sup>1</sup>For the remainder of this thesis, the term *process* refers to a physical process, distinguishable by the operating system, rather than a logical process, as would be implied in a process model view.

$$?- q(X) \text{ when } X.$$

More complex declarations are possible, and are expected to be in disjunctive normal form: a disjunction of conjunctions of individual arguments. The declaration

$$?- p(A, B, C) \text{ when } A \text{ and } B \text{ or } C.$$

indicates that for a goal  $p/3$ , either its first two arguments or its last argument must be bound to a nonvariable.

Although these annotations come in the guise of NU-Prolog *when* declarations, they are rather less sophisticated. In NU-Prolog [Thom & Zobel 1988], annotations may specify structure within arguments; here, they are implemented as *triggers* [Naish 1986], which are restricted to determining whether or not an argument is bound at all. (For this implementation, triggers are sufficient to demonstrate the execution, and were easy to implement. True *when* declarations are harder to implement, but allow much greater precision, and should be part of any future implementation of the algorithm.)

Just as the language is kept as close to sequential Prolog as possible, so too is the user interface kept similar. Beyond allowing annotations of top-level goals at the “?-” prompt, the distributed system behaves to the user exactly like a sequential system: prompting for a top-level goal, attempting to solve the goal, and printing top-level variable bindings to the screen. As in sequential Prolog, if another solution is desired, it can be requested by typing a semicolon immediately after receipt of the previous solution. This is accomplished by initiating backtracking from the last frame on the stack of the *master process*, which handles the user interface.

## 5.2 Solver Processes

At the core of the parallel Prolog system are the *solver* processes. The solver processes work together to find a solution to a top-level goal, each solving some annotated subgoal. Solver execution is generally sequential, following the goal-selection – clause-selection – unification model of standard Prolog.

Solution begins with the master process. It begins by creating a child solver process for every annotated subgoal in the top-level goal conjunction. As it executes the remaining sequential goals, it creates another solver every time it encounters an annotated subgoal in the latest clause body. For child processes, execution follows the same scheme; each annotated subgoal is forked off as a new solver as it appears in the body of a newly-unified clause.

Thus, parallelism may occur anywhere in the search tree. If *every* subgoal is annotated, the execution takes on a (horrendously inefficient) process-model character; conversely, if *no* subgoals are annotated, execution is purely sequential.

As they execute, the solvers communicate binding information with each other. Specifically, each solver sends messages to the other solvers with which it has variables in common. Since the master process either binds a variable itself or shares it with some child process that binds the variable, upon termination it will find bindings for all of its top-level variables, and print them to the user.

## 5.3 Ear Processes

Because of the nature of the message-passing subsystem, the solver processes cannot stand alone. The JIPC system is a synchronous protocol in which a process sending a message

is blocked until the receiver replies. Using such a protocol, solver processes that blithely sent messages to one another could easily cause deadlock.

In the usual case, message passing occurs as follows:

1. Process  $P_a$  sends a message to  $P_b$  and blocks.
2.  $P_b$  receives the message.
3.  $P_b$  acknowledges receipt to the sender,  $P_a$ , via a reply.
4.  $P_a$  gets the reply, and resumes execution.

If  $P_a$  and  $P_b$  sent messages to each other at the same (real) time, deadlock would occur: neither could receive or reply to the other's message, since both would be blocked.

One solution to this problem is to use interrupt-driven message receipt and acknowledgement, so that both processes could receive and reply to incoming messages even while blocked. Since JIPC provides no interrupt facilities, however, this approach is not feasible; thus, a nonblocking or asynchronous protocol is necessary. Some form of mediation is therefore required for message passing between solver processes.

In this implementation, such mediation is handled by *ear* processes—JIPC processes whose purpose is to make communications between solver processes appear asynchronous. One ear process runs on each processor in the network; its main task is to “listen” for input messages intended for interpreter processes running on the same processor.

Thus, solver processes do not communicate directly. When one solver wants to communicate with another, it actually sends its message to the other's associated ear process. This is illustrated in Figure 5.1. The ear process replies immediately to the

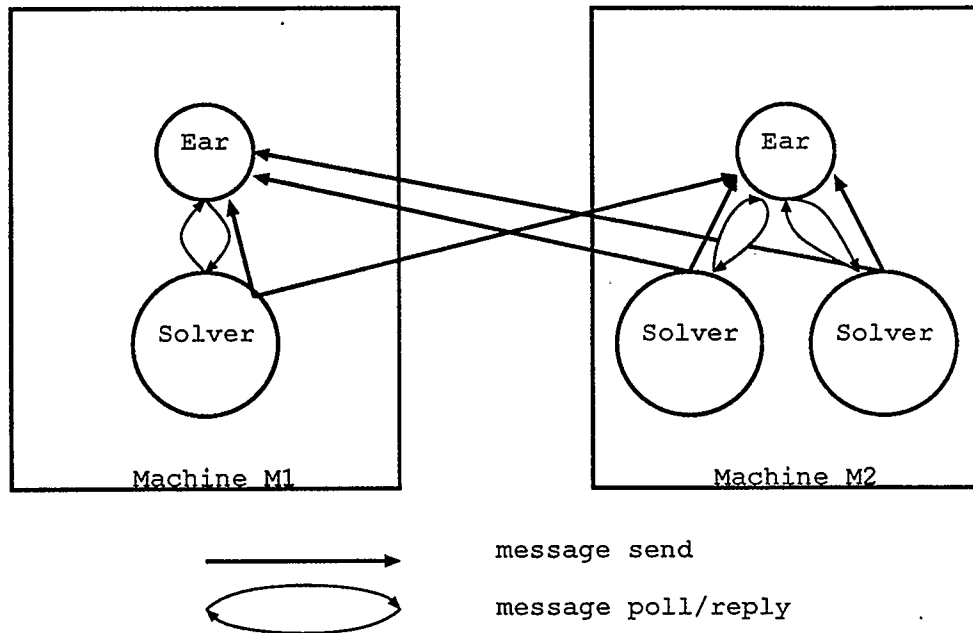


Figure 5.1: Relationship between ear and solver processes

sender, allowing that process to continue execution. Then, it queues the incoming message, waiting for the target process to poll it for messages.

Message receipt by a solver process is also nonblocking; if a solver polls its ear when no new messages have arrived, it receives a null message in return and continues executing normally. This agrees with the optimistic philosophy of Time Warp—that is, if no new messages have arrived, assume none will arrive.

There is one exception to the nonblocking approach. A solver process will choose to block until its next input message arrives if it has completed all of its own local computation. In this way, it avoids consuming the system resources that it would use if it performed a busy wait.

As the implementation evolved, responsibility for handling the input queue functions gradually migrated from the solver processes to the ear processes. Initially, an ear process

did no more than enqueue incoming messages in the order that they arrived. It was obvious that with only slightly greater effort, the messages could be enqueued in timestamp order. A small optimization was immediately apparent: annihilation of BIND/ANTI-BIND message pairs could take place within the ear process, rather than sending both messages on to an solver to be processed there.

Finally, it became evident that the ear processes should take over the input queue functions completely. Implementing queuing facilities in both ear and solver processes cannot help but be somewhat wasteful, and since the ear processes must do some queuing, why not let them do it all?

The solver processes are thus freed of responsibility, for example, of trying to “un-receive” messages on rollback. Instead, each ear process keeps track of which input messages have been received by each solver process it serves. If a solver rolls back, it need only inform its associated ear process of how far it rolled back by including a timestamp with its next polling message. The ear process uses this timestamp to adjust its notion of which messages have been received, returning messages that postdate the timestamp to unreceived status. (This also allows message annihilation to occur in cases where a BIND message was received and passed on, its corresponding ANTI-BIND arrived and was enqueued, and then the BIND was rolled back.)

Of course, the original reason for using ear processes must not be forgotten. The effect of using ears to mediate communication is that message passing is guaranteed to be deadlock-free (provided that memory is not exhausted). A message can be exchanged only between an solver process and an ear process, and not directly between two solver processes or between two ear processes. Solver processes initiate all communications and receive information only via replies to polling messages, rather than via explicit receives.

An ear process blocks until it receives a message, whereupon it replies appropriately to the sender, and blocks again to wait for the next message.

## 5.4 Binding Environment

One of the most important components of a distributed Prolog system is the binding environment. Variables from different processes must be distinguished by unique internal names<sup>2</sup>In the discussion that follows, “name” will refer to this internal name, as distinct from the “print name” visible to the user. so that they are not confused with one another. If a variable is shared among two or more processes, it must be readily distinguishable as such. (Note that “shared” is not used literally—each process that “shares” a variable maintains its own copy, and changes to that copy are made known via binding messages.) If a variable is not yet shared, it may be in the future through the creation of child processes, so it must be possible to upgrade it to shared status.

### 5.4.1 Naming

In a sequential system, the generation of unique variable names is accomplished by allocating on the stack: the memory address of a variable becomes its internal name. In a distributed system, this is not sufficient; different processes may produce variables with identical names, and if either or both of these variables is shared, confusion could result. A simple and effective solution to this problem is to incorporate information about the process ID of the solver that creates the variable, and about the processes on which that solver runs, into the variable’s internal name.

---

<sup>2</sup>,



### 5.4.2 Local and Shared Variables

Most variables begin their careers as local; the exceptions are the variables in the top-level goal of a new child process, which are always shared with at least the parent process. (For uniformity, the top-level variables of the master process are also considered shared.) Local variables are allocated on the stack, much as they are in sequential interpreters. When a process rolls back or backtracks over some stack frame, all variables in that frame are freed.

Shared variables are tagged for easy identification, since they must be treated specially. A shared variable must be directly accessible, since not every access to it will be through the stack. In particular, when a solver receives an external binding for one of its shared variables, that variable will generally not be found in the latest stack frame. If direct access is not available, the shared variable can only be found by searching back through the stack. On the other hand, a shared variable must also be accessible through the stack, to handle cases when it is referenced or bound locally.

A local variable becomes shared when the process that created it forks off a new process whose top-level goal contains that variable. When this occurs, a new variable is allocated, and the local variable on the stack is made to reference the shared variable. Indeed, every shared variable must have a “local” associated with it on the stack, to allow the Prolog component of the algorithm (particularly unification) to access it. The mechanism for creating a new shared variable is simple: whenever a variable is encountered in process creation, if that variable is still marked local, then a shared counterpart is allocated and linked to the local.

When a BIND message is sent, the information in it comes in pairs, each composed of

a variable name and its value. The variable name is fully qualified, including machine and process IDs. The value is an instantiated term—either an integer or a structure, which may contain variables (actually, fully qualified variable names) that are unbound. (If a bound variable is detected within a structure at the sending end, only the value is passed on.)

### 5.4.3 Incomplete Data Structures

Particularly in producer/consumer style programs, a shared variable may be only partially instantiated. For example, a list may have its head bound, while its tail remains unbound:

$$L = [27 | L1]$$

If  $L$  is a shared variable, so must  $L1$  be. Thus, whenever such a binding occurs, whether locally or through the receipt of a binding, if the outside variable is shared, the variable inside the data structure is allocated in the shared area, and linked to the stack.

## 5.5 Delayed Predicates

As noted above, user predicates may be given delay annotations. Such delays are discretionary and need not be present for correct execution; their only effect is to make execution more efficient. A number of builtin predicates also come with delays, but these delays are mandatory. Leaving them out would result in an incorrect execution.

### 5.5.1 User Predicates

The need for delays in user predicates arises from producer/consumer process pairs in which the consumer executes ahead and guesses wrong. When a binding from the producer it forces a rollback, but this rollback may not be deep enough to undo the incorrect guess.

The timestamp of the frame for the incorrect guess may well antedate the timestamp of the incoming binding. In this case, some amount of remote backtracking will have to occur before the consumer backtracks far enough and makes the correct choice. Work is lost in every participating process, not just in the process making an incorrect choice. This inefficiency is rooted in the way priorities are assigned to goals.

In Tebra's system [Tebra 1987], priorities are depth-first, so that given two distinct subtrees, each goal in the left subtree has higher priority than any in the right subtree. If in a producer/consumer situation, optimistic execution by the consumer works well as long as it is in the right subtree with respect to the producer, so that when a binding arrives the process is rolled back completely. If the consumer is in the left subtree, then its subgoals have higher priority than those of the producer; if it executes optimistically and makes an incorrect choice, remote backtracking occurs with a vengeance, first of every goal in the producer's subtree and then in the consumer's subtree. Unlike the Time Warp case, no goals are rolled back; every goal is backtracked. Thus, Tebra's system implicitly requires producers to precede consumers in order to extract good performance. Further, this producer-before-consumer ordering is purely textual and cannot take into account input and output modes determined by the top-level query. Still, for well-behaved programs, Tebra's system provides maximum optimism.

Conversely, in a goal-ordering system like that in [Somogyi *et al* 1988], a consumer is blocked until its corresponding producer sends it a binding, so no optimistic execution occurs. Producers and consumers are assigned by checking argument modes: a goal whose inputs are sufficiently instantiated becomes a producer for its outputs; a goal that still needs some inputs bound becomes a consumer of those inputs. Since consumers are blocked until their required inputs arrive, execution is conservative; no optimistic work will have

to be undone.

In an unadorned Time Warp system, timestamps may be interleaved between goal subtrees; thus, producers and consumers will have roughly comparable priorities, and the backtracking problems noted above will occur if optimistic execution is attempted. One solution would be to prioritize producers over consumers, while still allowing consumers to execute optimistically. This approach would be rather complex, since priorities are already given by Time Warp timestamps.

Another approach, and the one used in this implementation, is to use delays to mimic producer/consumer ordering with timestamp ordering, at the expense of optimistic execution. If a consumer lacks the input bindings it needs, it falls asleep. The consumer wakes up again after it receives the bindings it requires, so that the frames it creates have a greater timestamp (thus a lower priority) than the incoming bindings. Rejection of the Time Warp ideal is necessary, since allowing a consumer goal to execute ahead may give it a lower timestamp than its corresponding producer. Such a situation can cause a great deal of backtracking before a solution is found.

### 5.5.2 Builtin Predicates

In a distributed Prolog system, many builtin predicates require delays. The arithmetic predicates (e.g.  $</2$ ) and equality ( $=/2$ ) are among these, but for different reasons.

#### Arithmetic

In most sequential implementations, arithmetic predicates are expected to be fully instantiated before they are called; if such a predicate is called with one or more of its arguments not fully bound, an instantiation error is flagged and the computation aborts. Given the

depth-first execution order of sequential systems, instantiation errors are not hard to avoid: the programmer need merely order goals so that producers come before consumers.

The distributed case is more problematic: since execution is no longer strictly depth-first, an arithmetic goal may be executed before its arguments are fully instantiated, thus causing the computation to abort unexpectedly. Consider the parallel execution of the following code:

$$X = 2, X < 4.$$

If the goal  $X = 2$  executes first, the computation completes and gives the expected result (i.e. success). If  $X < 4$  executes first, however, an instantiation exception will be raised. Such behavior is clearly undesirable; the results of a computation should not be susceptible to the vagaries of execution order.

Of course, it may be argued (and has been: [Naish 1986]) that this behavior is also undesirable in the sequential case; parallel execution merely illustrates the problem more vividly. In both cases, the solution is the same: an arithmetic goal should be delayed until its arguments are sufficiently instantiated; then it can be woken and executed.

### Equality

Equality does not suffer from the same problem as the arithmetic predicates. Since its effect amounts to unifying its arguments, the equality predicate does not logically require either of its arguments to be bound. Thus, running goals such as

$$A = B, B = C, C = 5$$

in parallel produces the same result independent of execution order: A, B, and C are all bound to the value 5.

The problem that does arise is rather more subtle, and is related to the timestamping of bindings in the distributed Prolog system. Consider the following example:

- Process  $P_f$  executes the goal  $B = C$  at time  $t$ . No BIND message is produced, since no variable was bound to an actual value.
- At time  $t + c$ , process  $P_g$  executes the goal  $C = 5$  and sends a BIND message with this binding to  $P_f$ .
- $P_f$  receives the binding and immediately discovers that  $B$  is now also bound. Thus, it must send out a BIND message at time  $t + c$  with the binding  $B = 5$ .

What has happened here? Two messages with the exact same timestamp have been sent from two different processes. Any process that receives both bindings is in deep trouble if it ever backtracks to time  $t + c$ : it has no basis for deciding which binding to FAIL, since either (or both) may have contributed to the failure. If it chooses arbitrarily, the goal  $B = C$  may never be backtracked, and the search for a solution may fail because of it.

This problem could be remedied by forcing process  $P_f$  to send out a BIND message at time  $t$  for the binding  $B = C$ . In this case, when  $P_g$  binds  $C = 5$ , it also binds  $B = 5$ , and both bindings are sent out. This would ensure that backtracking could be carried out successfully; each message would be ensured a unique virtual time.

Unfortunately, this solution causes another problem, due to *aliasing*: two variable names become aliases for the same object. In a sequential system, when two variables are unified, one of the variables is made to reference the other. If either variable is later bound, the binding is actually applied to the variable at the end of the reference chain; the value for the other can later be found by *dereferencing* the reference chain. Consider running the goals

$$A = B, B = C$$

sequentially; this produces a reference chain like

$$A \rightarrow B \rightarrow C$$

If C is later bound to 5, any reference to A, B, or C will be dereferenced to the value 5.

In the distributed case, however, the bindings for A and B must be made explicit. If C is bound, there is no immediate way to tell that A and B are also bound, because the reference chain is unidirectional. To allow access from any variable to all other aliases for it, the chain must be closed into a loop. Any non-variable binding must then be referenced separately, leading to a messy, if workable design.

A much more elegant solution arises from avoiding aliasing altogether. Rather, the equality predicate  $=/2$  is given delay conditions such that at least one of the two arguments must be instantiated before execution is allowed:

$$?- X = Y \text{ when } X \text{ or } Y.$$

A goal like  $B = C$  is not allowed to execute immediately; rather, it is delayed until either B or C is bound to a nonvariable term, for example when the binding  $C = 5$  is produced. Consider the following execution:

- Process  $P_f$  attempts to execute the goal  $B = C$  at time  $t$ . Since neither B nor C is bound, the goal is delayed, and some other goal is executed.
- At time  $t + c$ , process  $P_g$  executes the goal  $C = 5$  and sends a BIND message with this binding to  $P_f$ .
- $P_f$  receives the binding and discovers that the delayed goal  $B = C$  should be woken.

- $P_f$  executes the goal  $B = C$  at time  $t + c + d$  and sends out the binding  $B = 5$ .

For local variables, it is not necessary to avoid aliasing, since if any of the aliases later becomes bound, only the variable at the end of the reference chain would actually be linked to a shared variable. For the sake of simplicity and uniformity, however, the unification of two local variables is also delayed.

This uniformity has a price, however. When a local variable is bound, this binding may trigger the wakeup of several delayed goals, each of which may wake up goals itself. Fortunately, this cost is mitigated by a beneficial effect. When aliasing is allowed, a goal sequence like

$$A = B, B = C, C = 5$$

results in a reference chain like

$$A \rightarrow B \rightarrow C \rightarrow 5$$

When aliasing is disallowed by delaying  $=/2$ , executing that goal sequence results in the following:

$$A \rightarrow 5; B \rightarrow 5; C \rightarrow 5$$

Since no variables are ever aliased, reference chains become unnecessary. Dereferencing a variable is accomplished by moving directly to its binding. (Of course, a smart aliasing system will dereference as much as possible at unification time, thus keeping reference chains fairly short.)



## 5.6 Termination Detection

One of the main problems of distributed execution lies in determining when the system has terminated. A naive approach would simply wait for every process to report its own completion and then declare the entire system to have completed execution.

If each process were completely independent, this would be quite sufficient. In general, however, processes must communicate with each other. In this case, messages that are still in transit cannot be ignored. Consider the following scenario:

- process  $P_a$  reports completion;
- process  $P_b$  sends a message to  $P_a$ , and then reports completion;
- process  $P_a$  discovers that it has not completed after all, and begins execution again by processing the incoming message;
- all other processes in the system have previously reported completion;
- since all processes have reported completion, the system is incorrectly considered to have terminated.

In a standard Time Warp system, termination is detected through the calculation of global virtual time: the minimum of the virtual times of all processes and of the timestamps of messages received but unprocessed or still in transit. When GVT reaches  $+\infty$ , the system has terminated.

GVT has other functions as well. Since the standard Time Warp definition assumes that all processes are completely deterministic, garbage collection of the input, output, and state queues becomes possible (and indeed necessary, for example in the case of a large

simulation system); GVT is computed regularly, and queue entries with timestamps less than GVT can be garbage collected. Termination detection is really a side effect of this garbage collection function, occurring when GVT reaches  $+\infty$ .

In a Prolog system, computing GVT is clearly overkill. Garbage collection cannot occur as in a standard Time Warp system; every entry in the IQ, OQ, and SQ may be needed for backtracking, so none should be removed. Thus, the main function of GVT calculation is wasted; termination detection remains as a side effect.

For this reason, an algorithm more specific to termination detection is indicated. [Dijkstra *et al* 1983] describe an algorithm for detecting termination in a ring of processes that can easily be adapted to the Prolog system. This algorithm uses a circulating token to gather information about the status of each process in the system. (The algorithm is actually very similar to the GVT algorithm, except that token propagation is delayed as long as possible, so that fewer passes through the system are made.)

Each process maintains information about its own state by coloring itself *white* or *black*, with black indicating that it is busy, and white indicating that it is idle. The token starts out white, but as it circulates through the system, it may be colored black, indicating that some process is still busy. For a ring of  $n$  processes  $P_0$  to  $P_{n-1}$ , the complete algorithm is as follows:

1. while it is active, a process keeps the token; when it becomes idle, it passes the token on to the next process
2. when a process sends out a message, it colors itself black
3. when a process propagates the token, it colors the token black if it is black itself; if the process is white, the token is passed on unchanged

4. if the token is black after completing the circuit, the system has not terminated; the initiating process sends the token out for another circuit
5. the initiating process starts a termination probe by coloring itself white and sending out a white token
6. after passing on the token, a process colors itself white

This algorithm must be adapted slightly to work for the Prolog system. First, rather than traveling around a ring, the token must traverse a hierarchy of processes. This is accomplished easily enough by a depth-first traversal of the hierarchy, with the token starting and finishing at the master process.

The second alteration is necessary to account for indefinitely-delayed goals. The execution of some programs may cause a goal to be delayed and never woken; this occurs when the bindings needed by the delayed goal are never produced. This may be due to incorrect delay conditions on the clauses for that goal, or it may simply mean that there are an infinite number of solutions for the goal [Naish 1986]. In either case, this condition, known as *floundering*, should be reported.

To handle this, a third token color is necessary: gray is used to represent the state in which all processes are idle, but one or more processes have indefinitely-delayed goals. Token coloring is modified to the following:

- if the process is black, it colors the token black
- if the process is gray and the token is not black, it colors the token gray
- if the process is gray and the token is black, it passes the token on unchanged

- if the process is white; it passes the token on unchanged

The system has terminated if the token is colored white or gray when it returns to the master process. If the token is gray, the system has floundered; otherwise, a complete solution has been found.

## 5.7 Optimizations

The BIND and ANTI-BIND optimizations, introduced in the previous chapter, have not been implemented. However, it is worth discussing how these optimizations *could* be implemented. Ironically, the FAIL over-optimization *was* implemented. An example is given to demonstrate that the optimization is incorrect; finally, a modification to correct it is proposed.

### 5.7.1 BIND Optimization

The BIND optimization allows the receiver of a BIND message to decide whether the contents of that message are compatible with its own bindings *before* it rolls back. Four cases are possible:

- The variable bindings in the message are consistent with the local bindings, so the BIND can be incorporated into the stack without rolling back.
- The bindings in the message would be inconsistent even after rolling back; the BIND can be ignored and forward execution can continue.
- Only a partial rollback is necessary to make the incoming bindings compatible; after this partial rollback, the BIND may be incorporated into the stack.

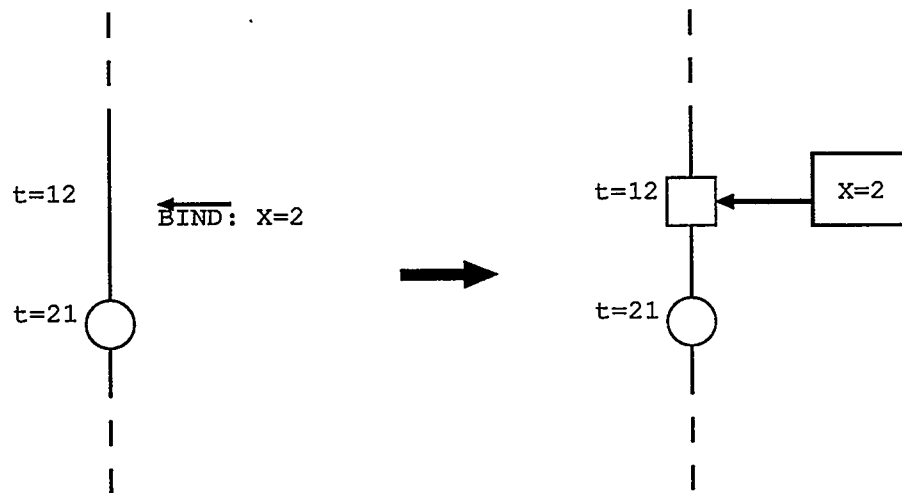


Figure 5.2:  $x$  is still unbound locally; the binding is simply incorporated into the stack.

- A full rollback is necessary and sufficient to make the incoming bindings compatible; this is the only case for which the optimization gives no benefit.

To distinguish these cases and determine the correct action to take, binding timestamps must be accessible to the shared variables (local variables cannot cause inconsistency with respect to another process). This can be accomplished by including in the shared variable structure a pointer back to the stack frame in which it was bound; the timestamp can then be found from the frame.

For each variable bound in a BIND message, the local copy of the variable must be examined. The actions that may be taken are illustrated in Figures 5.2 through 5.6. For simplicity, the incoming BIND is assumed to contain only one binding. If a partial rollback is necessary when several variables are bound in the incoming message, the rollback must be to the time of the earliest binding conflict.

The first three cases deal with compatible bindings. If the variable bound in the incoming message is still unbound locally (Figure 5.2), the binding need simply be linked

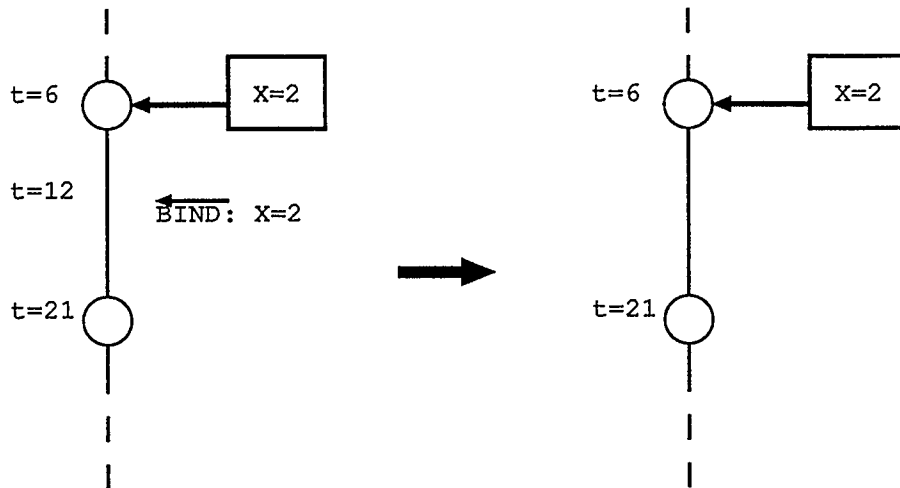


Figure 5.3:  $X$  is bound locally and compatible; the local timestamp is less, so it is retained.

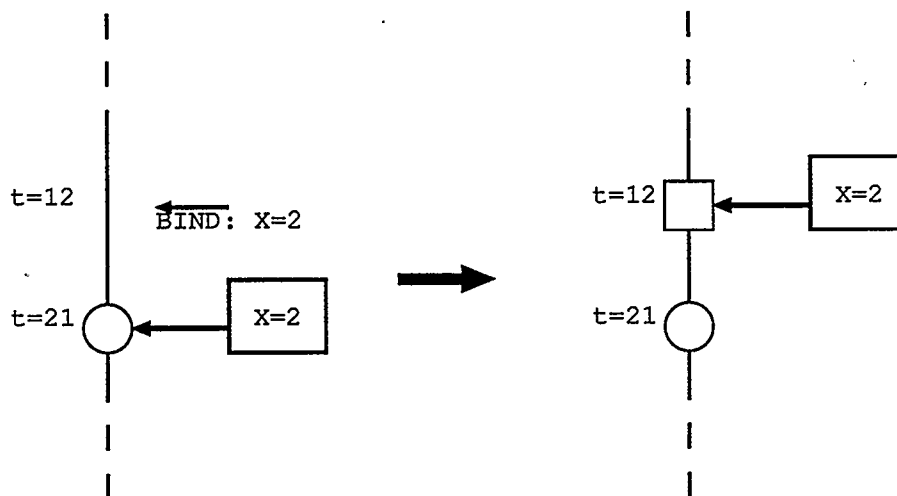


Figure 5.4:  $X$  is bound locally but is compatible; the message timestamp is less, so the binding pointer for  $X$  is adjusted.

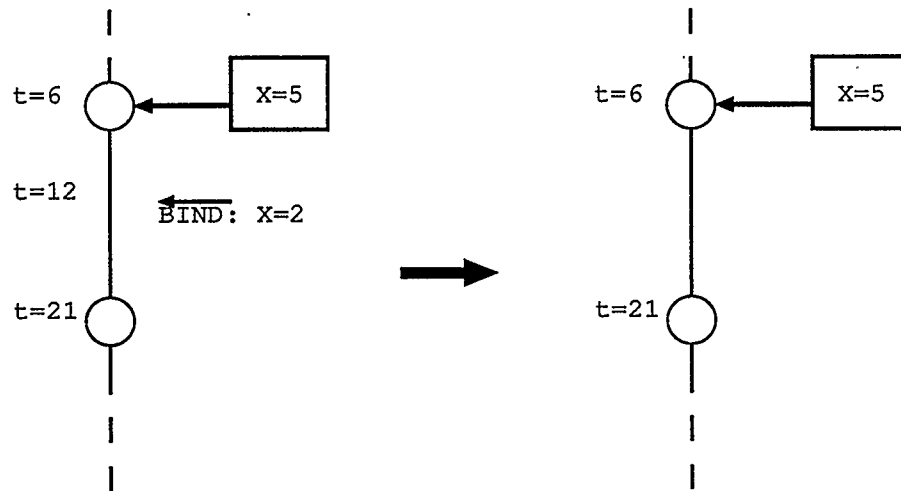


Figure 5.5:  $x$  is bound locally but incompatible; the incoming message has a higher timestamp, so it is ignored.

in to the stack; no rollback is necessary. If the variable in question is bound locally, two cases are possible. If the local binding antedates the incoming binding (Figure 5.3), it may safely be ignored. The sending process will eventually encounter the situation shown in Figure 5.4. In this case, the frame pointer for the bound variable must be adjusted to point to the earlier, remote frame.

In the final two cases, the local binding and that in the message are found to be inconsistent. If the local binding is earlier than the message timestamp (Figure 5.5), the message is simply ignored. At some point, the sending process will arrive at a state like that in Figure 5.6: the message timestamp will precede the local binding time. In this case, a partial rollback is necessary; if there are no stack frames with intermediate timestamps, this amounts to a full rollback.

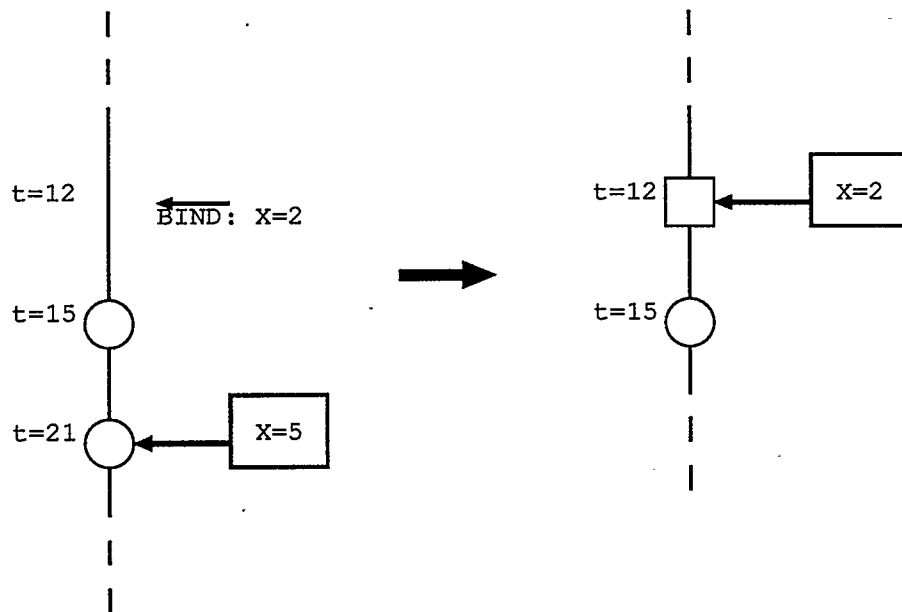


Figure 5.6:  $X$  is bound locally, but its value conflicts with the value in the message. The message has a lower timestamp, so the process must roll back to before the time of the local binding, and restart forward execution.

### 5.7.2 ANTI-BIND Optimization

The ANTI-BIND optimization is analogous to the BIND optimization in that it attempts to avoid rollback if possible. However, avoiding rollbacks in the ANTI-BIND case is more complex. It requires a modification to the basic execution algorithm; the BIND optimization needs only timestamp information to accomplish its attempted effect.

The necessary modification is in the clause selection component. When a rollback is performed, goals that are rolled back are eventually retried, and all clauses for each goal are available for selection. When a remote frame and its associated bindings are simply removed, this may not be the case. Those bindings may have constrained the execution; clauses rejected due to conflicts with these bindings may become candidates again once the bindings are gone.



If rollback is to be avoided, all goals with timestamps later than that of the incoming ANTI-BIND message must have their respective lists of candidate clauses adjusted so that previously-rejected clauses may be retried. The easiest way to do this is to maintain the clause list for each goal as a circular list; on receipt of an ANTI-BIND, the current clause becomes the “first” one. The clause list for a goal is exhausted when the *next* clause is also the first.

Suppose a goal  $g$  has three candidate clauses,  $c_1$ ,  $c_2$ , and  $c_3$ . Initially,  $c_1$  is the “first” available clause, and thus the first selected. After backtracking,  $c_2$  is selected. If an ANTI-BIND then arrives, the current clause,  $c_2$ , is made the first available clause and  $c_1$  the last available. Thus,  $c_1$  will eventually be retried—specifically, after  $c_3$  has been tried and backtracked.

There are two cases in which this sort of adjustment can be avoided. The first case involves the receipt of an ANTI-BIND for which the corresponding BIND was ignored. In this case, the ANTI-BIND may also be ignored. A remote frame may also be removed without recourse to rollback when a situation like that in Figure 5.4 exists. If a frame pointer is adjusted on receipt of a BIND, it can be adjusted back if that BIND is annihilated, as long as information about the previous frame pointer is retained. No adjustment of any clause list is necessary, since the binding values are no less constrained after the remote frame is removed.

### 5.7.3 FAIL Optimizations

As described in the previous chapter, the FAIL optimization gives a reasonable saving in terms of rollbacks performed and FAIL messages sent, but it comes at the expense of completeness: in a few pathological cases, this optimization causes solutions to be missed.

Consider the situation in Figure 5.7. Process  $P_f$  has been unable to succeed because the bindings from  $P_q$ ,  $P_r$ , and  $P_s$  have been incompatible with its own values. At this point, both  $P_r$  and  $P_s$  have been forced to backtrack and to come up with new bindings. These bindings are now compatible with the values in the first clause for  $f/3$ , yet the goal  $f(A, B, C)$  still fails. The problem is with  $P_q$ 's binding, but under the the small-context direct-backtracking model,  $P_q$  cannot be made to backtrack, as the following execution fragment shows:

- $P_f$  fails to unify goal  $f(1, 3, 4)$  with clause head  $f(2, 3, 4)$
- $P_f$  fails to unify goal  $f(1, 3, 4)$  with clause head  $f(1, 4, 5)$
- $P_f$  begins backtracking, sending a FAIL at time 8 to  $P_s$ , with context of  $P_r$  at time 4.
- $P_s$  begins backtracking and finds no further clauses, so sends a FAIL at time 4 directly to  $P_r$ , *with no additional context*
- $P_r$  backtracks and finds no further clauses, backtracks further and finds no previous stack frames, and fails completely.

In this case, the solution  $\{ A = 2, B = 3, C = 4 \}$  was missed, simply because it was impossible to make  $P_q$  backtrack; not enough context was available to allow further backtracking.

Under the unoptimized scheme, failure would have passed back to  $P_f$ , and thence to  $P_q$ . The problem with the optimized scheme is clear: not enough context is passed on in the FAIL message itself to allow correct backtracking. In the unoptimized case, limited context is sufficient because the remainder is implicit in the stack of the FAIL's sender; in the optimized case, that remainder may no longer be accessible.

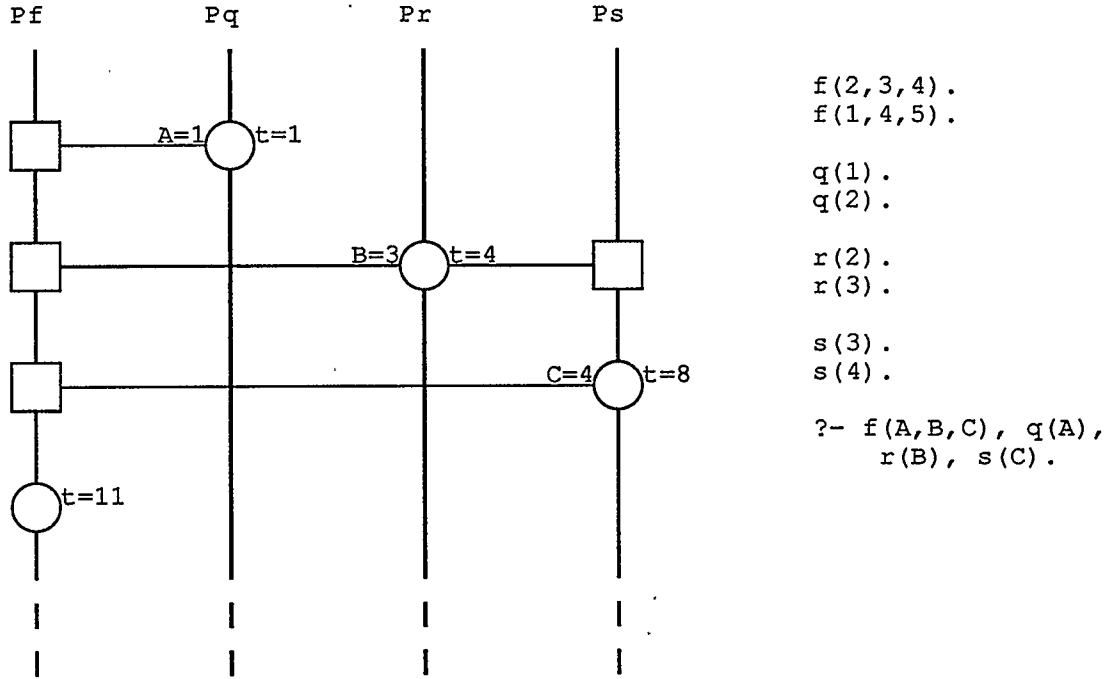


Figure 5.7: Pathological case for FAIL “optimization”

Extending the context by some fixed amount is of no avail. If the context of a FAIL message were extended to include the two previous stack frames, the example in Figure 5.7 could be extended by adding another process,  $P_t$ , contributing a binding to  $P_f$  at time 2, and process  $P_q$  would still never receive a failure. In general, for  $n$  frames of context, a counterexample with  $n + 3$  processes can be constructed. Thus, an indefinite amount of context in the FAIL message is necessary for the optimization to work.

Happily, this optimization is not entirely unsalvageable. The case described in Figure 5.7 is relatively uncommon, so a hybrid approach is possible: a FAIL message can contain a limited context, as long as the last context will force the originator to backtrack. Failures occurring in the remainder of the context may be optimized.

Easiest is a two-component context, with the first component corresponding to the first previous frame, and the second to the next previous frame. The first component directs backtracking to a stack frame's originator; if this is insufficient, the second component will eventually direct the failure back to the originating process. This scheme itself may be optimized: if both components direct the failure back to the same process, the second component need not be included, since backtracking will in any case return to the originator. (Naturally, this hybrid version of the FAIL optimization awaits experimental verification.)

## Chapter 6

### Testing and Results

#### 6.1 Testing Environment

The implementation was tested on a small distributed system: a network of five SUN3 workstations running UNIX. JIPC was used for interprocess communication. Ear processes were added to prevent deadlock and to provide input queue services for the solver processes.

Due to the limited number of processors, the algorithm for sending BIND and ANTI-BIND messages was simplified. Rather than requiring solvers to determine the recipients of BIND and ANTI-BIND messages and whether or not they should be propagated further, it was decided to broadcast such messages to all ear processes. Each ear would then determine which solvers on its processor were interested in the message. At the expense of a system-wide broadcast, the delay due to hierarchical message propagation (e.g. from a child process to its parent, and thence to its other children) is avoided.

This simplification assumes that several processes will be on each processor and that related processes are distributed throughout the system rather than highly localized. These assumptions are satisfied quite easily on a five-processor system, but not on a much larger one. Thus, the simplification is not scalable, especially since having every solver broadcast to every ear in a large, busy system could saturate the communication network. Still, it made implementing the system easier.

Because of the limited number of processors and the expense of process creation, parallelism was restrained by means of a depth bound. To implement this bound, the

master process is given level  $n$  for depth 0; its immediate children get level  $n - 1$  (depth 1), children of those solvers get level  $n - 2$ , and so on. A process at level 0 (depth  $n$ ) creates no children; goals annotated to run in parallel are instead run sequentially by that process. This also provides a simple control over the granularity of the parallel execution, even when many processors are available.

It should be noted that this depth bound is a very simple-minded technique. Using it to control parallelism may result in processes having significantly different amounts of work to do, leading to unbalanced execution and thus to reduced speedup. For this thesis, the point is moot, however: the aim is to present a working distributed Prolog system, not to find the best load-balancing algorithm.

Related to this is the issue of processor allocation: if all processes were allocated on just a few processors, parallel execution would again result in little speedup. Again, processor allocation is rather simple-minded: a process at level  $n$  creates a process on every  $2^{n-1}$ th processor (modulo the number of processors). In test runs, every processor showed activity; this was deemed sufficient for the purpose of the thesis.

## 6.2 Testing

A number of factors affect the parallelism achieved in a distributed execution. First, the type and degree of parallelism inherent in the test programs must be considered—for example, in an AND-parallel system, a searching program would be expected to perform worse than a divide-and-conquer program. As well, unfavorable delay characteristics (generally, executing before inputs are ready) can cause a reduction in performance. Finally, different execution strategies can affect the behavior exhibited by a test program. All of these

factors are examined below.

### 6.2.1 Determinism and Nondeterminism

As discussed in Chapter 2, deterministic programs are more amenable to AND-parallel execution, while nondeterministic programs are more suited to OR-parallel execution. The results presented in the next section bear this out.

The degree of nondeterminism may be characterized by the number of clauses unifiable with each goal as it is run. If every goal unifies with exactly one clause, the execution is fully deterministic, and no backtracking is necessary. An example of this is matrix multiplication.

Nondeterminism may be either *shallow* or *deep*.<sup>1</sup> Shallow nondeterminism occurs when a number of clauses may initially unify with a goal, but all clauses save one are quickly rejected by a test in the body, for example in the partitioning of a list in the quicksort algorithm. In deep nondeterminism, much more search is necessary to determine that a candidate clause must be rejected; search programs fall into this category.

Several example programs of each of these types were tested. For full determinism, `mmult` (matrix multiplication), `fib` (Fibonacci numbers), and `tak` (the *tak* benchmark) were run. Programs exhibiting shallow nondeterminism were `qsort` (quicksort), `union` (union of sets represented as trees), and `inter` (set intersection using trees). To test nondeterministic execution, two small programs, `x15` and `xy15` were composed. All of these programs and the queries with which they were tested are given in Appendix B.

---

<sup>1</sup>The notion of shallow vs. deep nondeterminism is similar to that of *don't-care* vs. *don't-know* nondeterminism.

### 6.2.2 Delay Characteristics

In Chapter 5 it was noted that the delay annotations used in the system were not the most precise available. Because of this, they may allow goals to execute prematurely, resulting in wasted execution as such goals are rolled back. The shallowly nondeterministic programs exhibit this behavior: when a clause is selected for some goal, their unification may wake up other goals; the clause may then be rejected because a test in the body fails, causing the woken goals to roll back and become delayed again.

Consider the following code fragment:

```
part([Head|Tail], Pivot, [Head|RestSm], Bigs) :-
    Head =< Pivot,
    partition(Tail, Pivot, RestSm, Bigs).
    :
?- qsort(List, _) when List.
```

When the goals

```
part([12,10,...],5,Smls,Bigs), qsort(Smls,SortSm)@p1
```

are run in parallel, the `qsort` goal is initially delayed, but as soon as `Smls` in the goal unifies with `[Head|RestSm]` in the clause head, it is woken. When the test `Head =< Pivot` fails, any work that the `qsort` call has done must be undone.

This problem can be alleviated only by ensuring that merely unifying a goal with some clause does not cause other goals to wake immediately. This can be done by shifting output unification into the clause body (after a test), rather than letting it occur in the head:

```
part([Head|Tail], Pivot, SmallS, Bigs) :-
```



```
X =< P,  
Smalls = [Head|Pivot],  
part(Tail, Pivot, RestSm, Bigs).
```

In this case, `qsort` does not begin executing until the output unification is made—that is, after the test has succeeded.

The problem may be even more severe for deeply nondeterministic programs. When a clause is chosen, it may be rejected only after much forward execution has transpired. The goals woken when the clause was bound will also have executed forward, and their work too must be undone. To make matters worse, all of this work might only be undone after exhaustive distributed backtracking.

On the other hand, allowing goals to execute immediately in a deeply nondeterministic program may allow failure to occur quickly, thus *saving* work. Further study would be desirable to indicate whether savings due to early failure can balance or even outweigh the cost of distributed backtracking to undo an incorrect choice.

### 6.2.3 Goal-ordering Strategies

Several different strategies may be applied to assigning goal priorities in a parallel system. As noted in Chapter 3, Tebra's system ([Tebra 1987]) bases the goal priorities on the depth-first ordering of a program's sequential search tree; [Somogyi *et al* 1988] use explicit producer/consumer relationships to assign priorities. Both of these strategies produce fixed, deterministic orderings.

In contrast, the algorithm of [Cleary *et al* 1987] does not assign goal priorities in advance; rather, the priority of each goal is assigned dynamically, based on the local

virtual time (LVT) of the process executing the goal. Beyond this, no goal ordering is specified—but in an implementation, some ordering must be chosen. No “best” ordering was immediately apparent, so several different orderings were examined. Three binary choices were available, giving eight different versions of the system.

The first parameter is concerned with the number of incoming messages a solver should process after it has completed a successful unification. On one side, a solver should use all the information available to it, including all pending messages. On the other hand, it may get so swamped with incoming messages that it accomplishes little useful work itself. Only the two extrema were tested: solvers would either accept just one message after each successful unification or process all pending messages.

The effect on goal priorities in this case is rather subtle. The salient point is that incoming messages cause LVT to advance; the more messages processed at some point, the greater the timestamp of the next unification is likely to be, and a higher timestamp corresponds to lower goal priority.

The second parameter is concerned with the size of the time window within which messages should be accepted. Certainly, all messages in the past of a solver’s current LVT must be processed, but the future is somewhat murkier. One option is for the solver to process incoming messages regardless of their timestamps. The solver’s LVT will then be adjusted to be later than the latest incoming message, and local work continues from that time. The idea here is based on the Time Warp philosophy: events in the future should be accepted on the assumption that no intervening events will occur. All pending external events (incoming messages) are therefore processed; intervening internal events (unifications) are avoided by adjusting the LVT.

A more deterministic approach to execution allows only those pending messages

	ONE	MANY
SMALL	earliest message within increment	all messages within increment
LARGE	earliest pending message	all pending messages

Table 6.1: Messages processed under combinations of goal-scheduling parameters.

whose timestamp is within a single increment of the current LVT to be processed. The next scheduled internal event (unification) is then performed at the incremented LVT. This approach, like the one-message alternative above, prefers internal work to external work; meanwhile, postponed future messages may be annihilated, or messages could arrive that precede the still-unprocessed messages, in both cases avoiding local rollback.

The final parameter alters the way in which a process's LVT is assigned. Under normal execution, LVT is based on the latest timestamp on the stack; if rollback occurs, then LVT is also rolled back. The alternative is one of *temporal inflation*: the LVT of a solver is based on the latest timestamp it has encountered, and grows throughout the execution. This is in fact an approximation to real time, as the LVT at each process is monotonically nondecreasing throughout the execution (except in backtracking and for brief rollbacks to handle messages in the past).

The idea here is to automatically synchronize processes so that producers will precede consumers in timestamp order, thus reducing wasteful execution. A producer will make a binding; recipients of that binding advance to the time of the binding. If the binding is discovered to be inconsistent, the consumers will not roll back their LVTs. Thus, they will not compete with an established producer by sending out bindings with low timestamps.

These parameters are not entirely orthogonal, particularly the first two—both affect the number of pending messages that are processed. The effect of combining these parameters is given in Table 6.1. The SMALL–ONE combination from this table appears rather

wasteful when compared with the SMALL-MANY combination: why accept only the first message preceding the next scheduled (internal) unification when any other pending messages with a timestamp before that unification will cause a rollback when it is processed? If message traffic is low, the effect may not be noticeable, but it could be pronounced if traffic is heavy.

The LARGE-MANY combination takes the opposite tack, favoring external work and postponing internal work. The SMALL-MANY and LARGE-ONE combinations take an intermediate course, trying to balance internal and external work.

The third parameter, concerned with inflationary vs. noninflationary execution (INFL vs. NON), is relatively independent of the other two. (Of course, the more messages accepted at once, the higher LVT is likely to go, so there is still a connection.) For deterministic programs, deliberately inflationary execution seems unlikely to have any effect, since timestamps will increase naturally, even for noninflationary execution. Only in nondeterministic programs might temporal inflation be expected to have an effect.

### 6.3 Results

In evaluating the parallel system, two measurements were used: the number of unifications done, and the number of messages sent and received. The first of these is a measure of the amount of work done in a parallel execution, and is readily comparable to the number of unifications done by a similar sequential interpreter. The second is used to gauge the message-passing overhead of the system.

### 6.3.1 Summary of Test Runs

Each of the programs listed above were run several times. Some, like the fully deterministic programs, needed very few runs (five times each) to give consistent results. The non-deterministic programs showed rather more variance and were run upwards of twenty times each. Each test was performed with the depth bound set at 2 to keep the five processors from being overloaded. Parallel execution results are considered from two perspectives: first, they are compared as a group against the results for sequential execution; then, they are compared with each other.

Averages were taken of all runs; extrema (largest and smallest values) have also been kept for the unification results to give some feel for the variance in the executions. For each program, parallel unification results were scaled to the sequential value, which was set at 1. Message counts were broken down into five parts:

- tokens: the number of times the termination-detection token was passed from one process to another
- BINDs: the number of BIND messages that reached another solver
- ANTIs: the number of ANTI-BIND messages that reached another solver
- FAILs: the number of FAIL messages sent
- annihilated messages: the number of BIND and ANTI-BIND messages that were annihilated at the ear processes, rather than being passed on to the solvers

In the graphs that follow, the different versions of the system are identified by three-letter abbreviations, as follows:

**ILO** Inflationary, Large window, One message

**ILM** Inflationary, Large window, Many messages

**ISO** Inflationary, Small window, One message

**ISM** Inflationary, Small window, Many messages

**NLO** Noninflationary, Large window, One message

**NLM** Noninflationary, Large window, Many messages

**NSO** Noninflationary, Small window, One message

**NSM** Noninflationary, Small window, Many messages

**SEQ** SEQuential

### 6.3.2 Deterministic Programs

`mmult`

Matrix multiplication is a classic “easy” program to run in parallel—even in imperative programming languages—since it is readily broken down by row and column into many independent and equal-sized subtasks. This offers high parallelism and balanced execution with minimal effort.

The results given in Figure 6.1 bear out the “easy parallelism” expectation. Only one version of the system took more than 10% more unification steps than the sequential interpreter took to arrive at the solution. This is mostly due to the independence of the subgoals: the top-level outputs are computed directly from the top-level inputs, without generating intermediate bindings that would require additional interprocess communication. Because

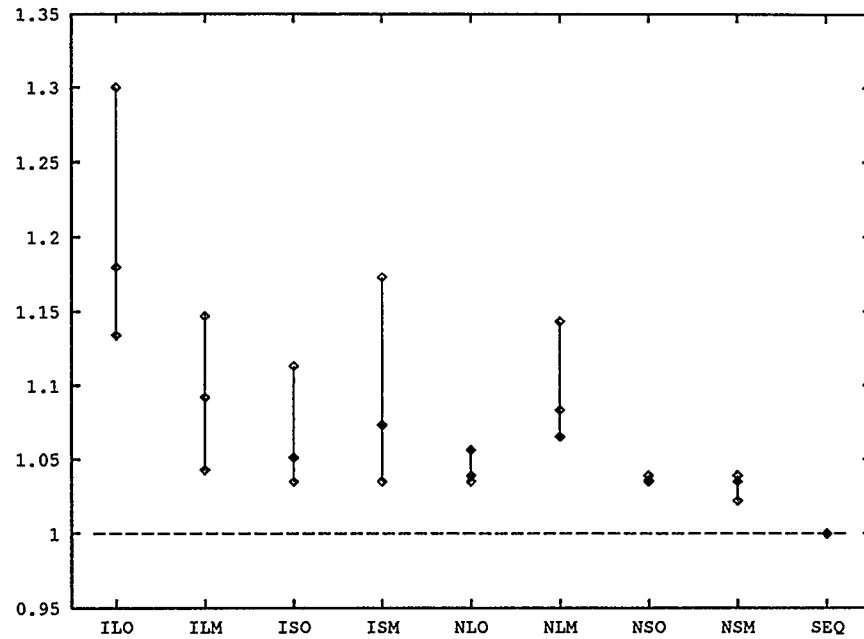


Figure 6.1: Average, minimum, and maximum unifications for mmult.

of the depth bound, process granularity remains fairly large—each process fully computes one element of the product matrix. This also keeps the message count low, and accounts for the small differences between system versions.

Results for different versions show a number of variations, both in unifications and in messages (Figure 6.2). The temporally-inflated versions were slightly busier and more erratic in terms of unifications than the noninflated versions. Overall message counts vary little, but when the individual components (BINDs, ANTI-BINDs, etc.) are considered, variations are apparent. The one-message versions had very few ANTI-BIND messages get through to the solvers; the multiple-message versions received significantly more. Large time-window versions received fewer BINDs than small-window versions.

There seems little correlation (or even a negative correlation) between unification totals

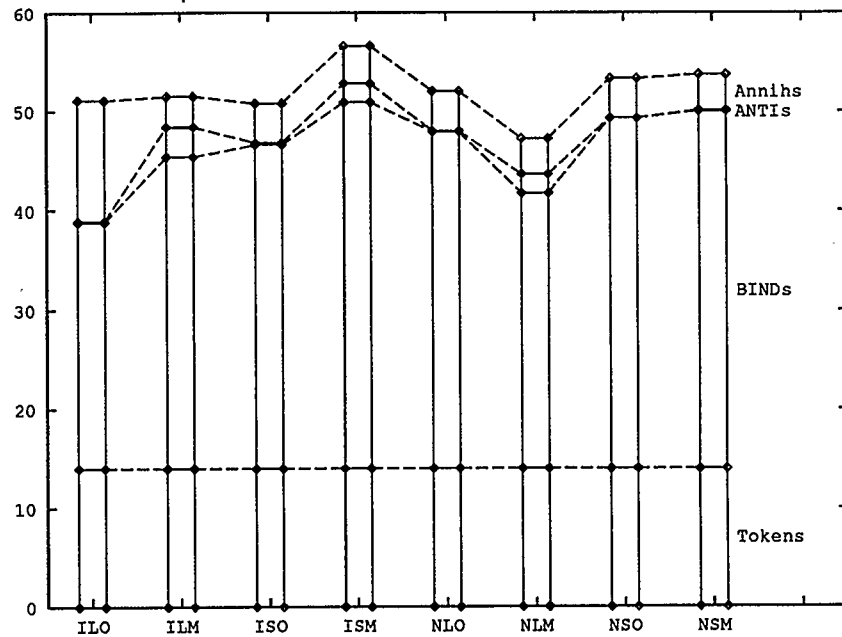


Figure 6.2: Tokens, BINDs, ANTI-BINDs, and message annihilations for `mmult.`

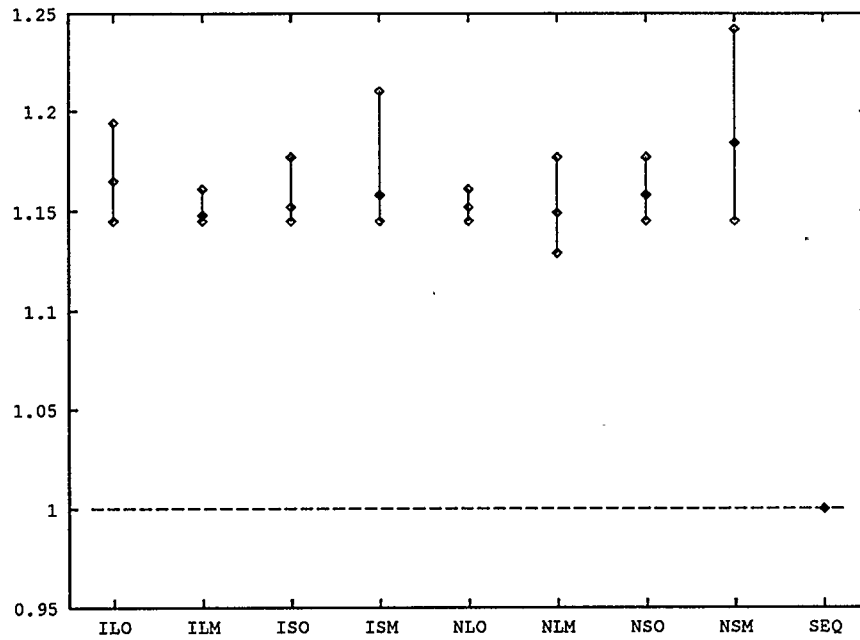
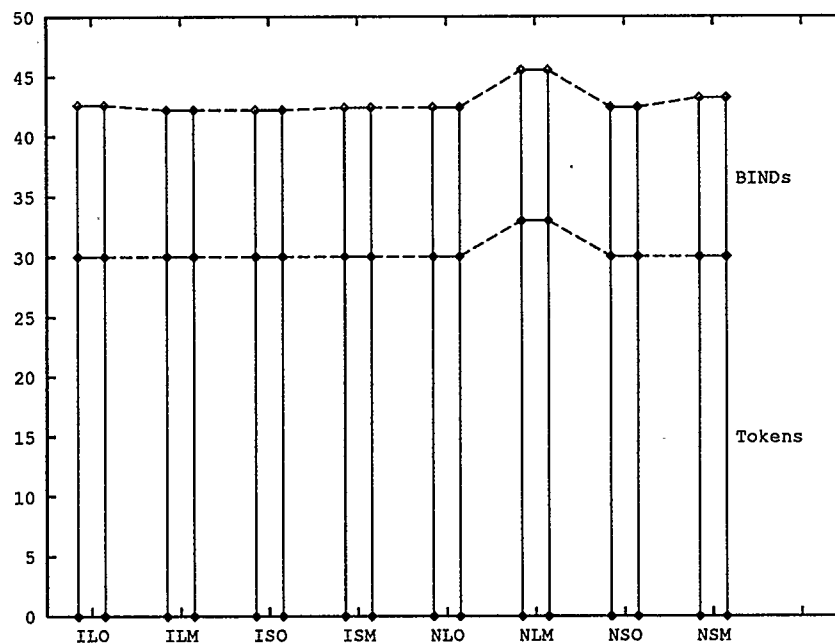
and message counts: note that the ILO, ILM, and NLM versions were high in unifications but low in message traffic; the converse holds for NLO, NSO, and NSM.

#### `fib`

The Fibonacci program uses a “divide-and-conquer” strategy, breaking a relatively complex problem into two independent, simpler problems. This approach is also well suited to parallel execution. Because of the depth bound, some processes do slightly more work than others, but the execution is still fairly well balanced.

As with matrix multiplication, the unification and message count results displayed in Figures 6.3 and 6.4 indicate that a divide-and-conquer program like `fib` can indeed execute well in parallel. Little difference between versions of the system is apparent; again, this is due to the low message traffic (itself again due to the nature of the problem).



Figure 6.3: Average, minimum, and maximum unifications for `fib`.Figure 6.4: Message counts for `fib`.

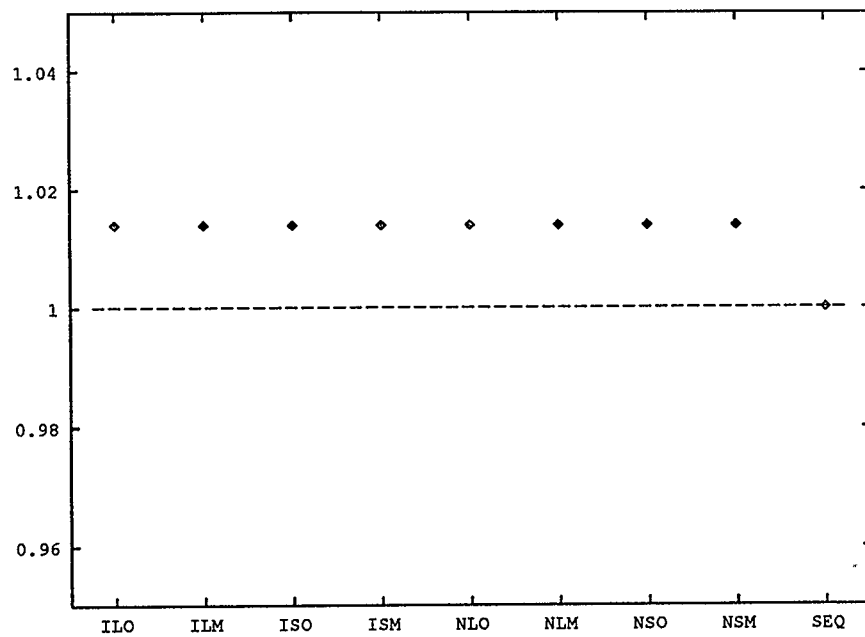
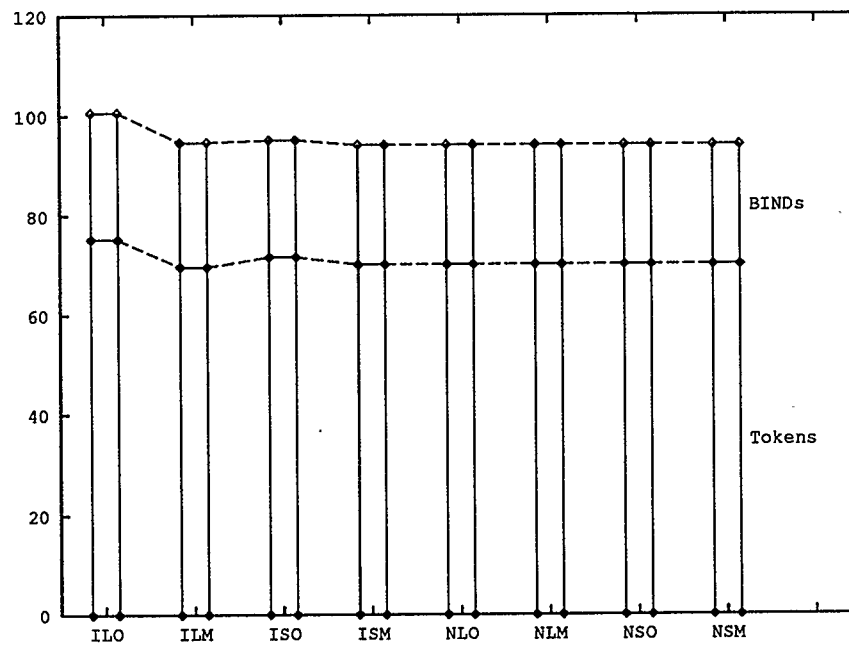
Comparing the results of the parallel versions with each other reveals little. The many-message versions appear more variable with respect to the number of unifications, but this effect is not universal (the ILO-ILM pair is an exception). As for message counts, almost no differences are visible (the small bump in the NLM result is due to a large number of termination-detection messages being sent in one of the runs of that version).

`tak`

The *tak* benchmark is multiply recursive, with each level of the recursion producing three mutually independent subgoals, and one subgoal that depends on the other three. Unlike the case for the other deterministic programs, these subgoals may differ greatly in the amount of work below them, so using the depth bound may result in unbalanced execution. Thus, parallelism is limited by two factors: goal dependence and the work disparity between goals at the same level (in particular, those at level 0).

The results for `tak` (see Figures 6.5 and 6.6) are consistent across *every* execution; each individual run took 1187 unifications, just 16 more than the sequential execution. Each of these 16 unifications corresponds to a goal that was tried once, delayed, and later woken again and run.

As with `fib`, almost no differences between versions can be detected when running `tak`. No variation occurred in unifications; in message counts, three of the temporally-inflated versions showed slightly more activity than the other versions.

Figure 6.5: Unification average, minimum, and maximum for  $t_{ak}$ .Figure 6.6: Message counts for  $t_{ak}$ .

### 6.3.3 Shallow-nondeterministic Programs

`qsort`

Quicksort, like the Fibonacci program, also uses a divide-and-conquer approach, partitioning its input list into two smaller lists and sorting each of those. The `qsort` program differs from `fib`, however, in that it takes advantage of some stream AND-parallelism: partitioning a large list can be done in parallel with sorting the two lists that result—as the partitioning process produces the sublists, the quicksort processes can consume them. This is a distinct gain over independent AND-parallel execution, which can only run the quicksort processes after the partitioning is complete.

The results shown in Figure 6.7 indicate that all parallel versions had to do much more work than the sequential interpreter—from 2.5 to 3 times as many unifications. This huge gap is directly attributable to the lack of precise delays, which causes consumer goals (in this case, the recursive quicksort calls) to awaken immediately a clause is unified with the producer goal (the partition call), rather than remaining delayed until the correct clause is determined. In cases when the partitioning process has made an incorrect choice, the quicksort processes must be rolled back and delayed again.

As for the parallel results alone, the temporally-inflated versions performed somewhat more unification work than their noninflated counterparts. For the message counts (see Figure 6.8), the situation is reversed, with the inflated versions sending fewer messages than the noninflated ones. Breaking the totals down by message type, the noninflated versions sent far more FAILs and somewhat more ANTI-BINDs, and experienced more annihilations. BINDS and termination token counts remained stable across all versions. This trend is reversed for message counts: the inflated versions sent fewer messages than

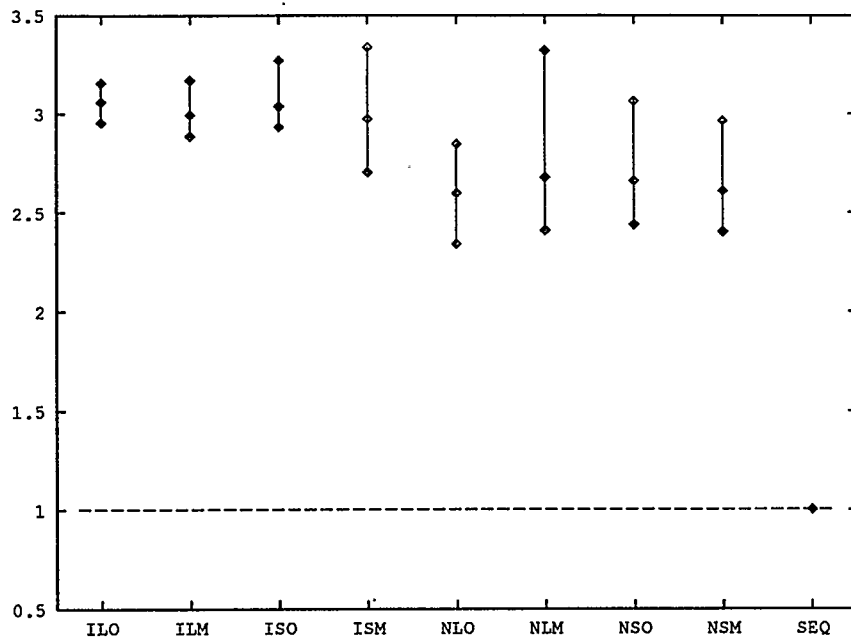


Figure 6.7: Unification average, minimum, and maximum for qsort.

the noninflated ones.

union

The set union program is similar in structure to quicksort, since the sets are represented as ordered trees. That is, the set union is performed by selecting the root of one of the trees as a pivot, and partitioning the other tree according to that pivot (the first tree is trivially partitioned). A divide-and-conquer approach may then be used to find the respective unions of the left subtrees and the right subtrees, thereby finding the union of the original two trees.

Unification results (Figure 6.9) demonstrate the same sort of behavior that quicksort did: all parallel versions take significantly more unifications than the sequential version to reach the solution. However, the penalty for allowing goals to awaken prematurely is not

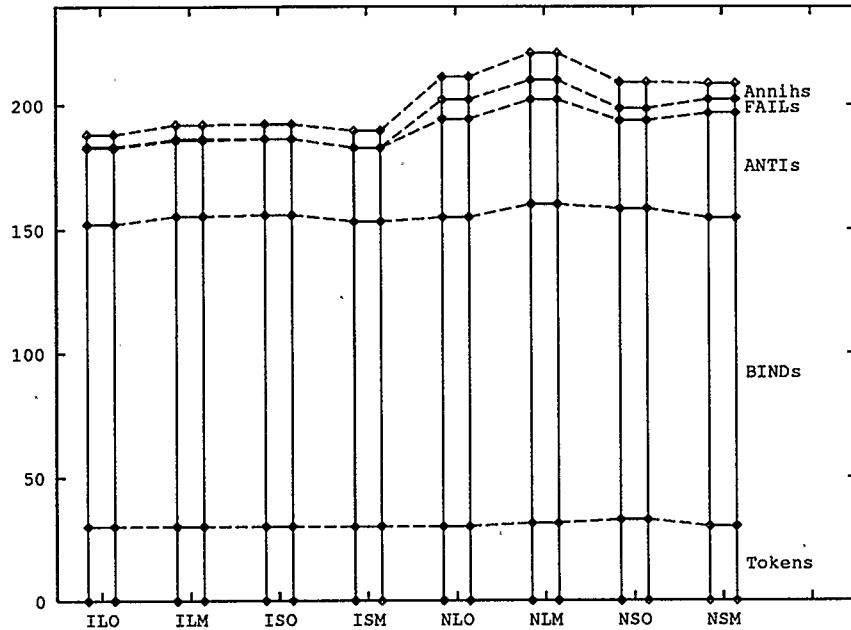


Figure 6.8: Message counts for qsort.

as severe here. The reason for this is simple: partitioning an ordered tree of size  $n$  does not require every node to be processed (and therefore,  $O(n)$  goals woken); on average, only  $O(\log_2 n)$  nodes need be accessed.

Comparing parallel results, the inflated versions did significantly more work than the noninflated ones, particularly in unifications done, but also in messages sent (see Figure 6.10). Changing the size of the time window also had a clear effect: small-window versions performed better than their large-window counterparts in both measures. The versions giving low message counts did so because they sent fewer ANTI-BIND messages and encountered fewer annihilations.

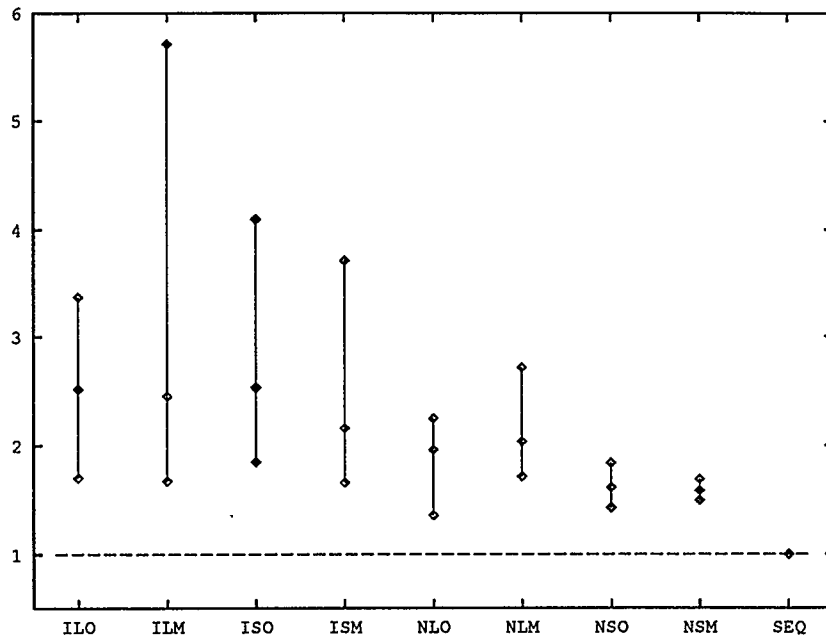


Figure 6.9: Unification average, minimum, and maximum for union.

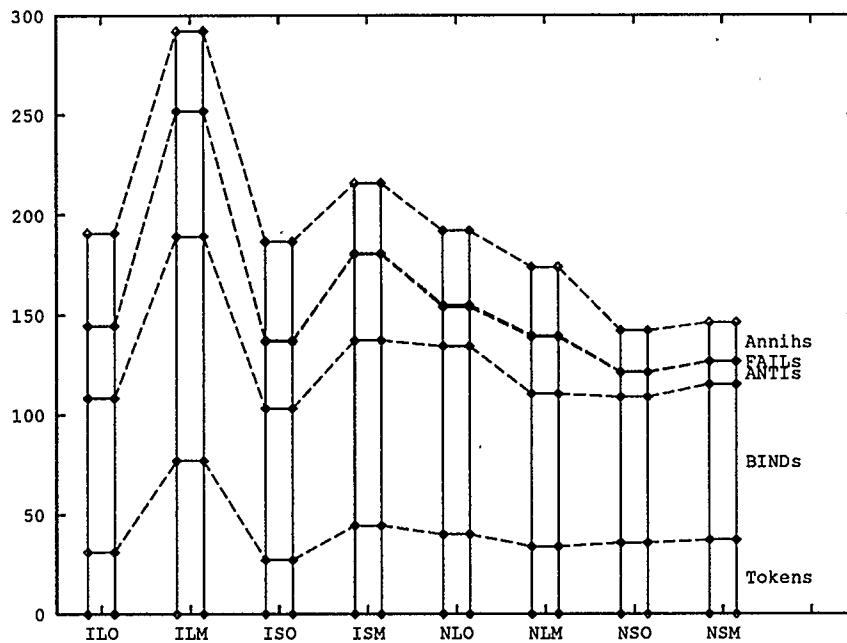


Figure 6.10: Message counts for union.

`inter`

The set intersection program is very similar to that for union. The main difference is that trees must be reorganized, since the root node of a (sub)tree must be deleted if it does not occur in the other tree; some other node must take its place. Beyond this, the divide-and-conquer approach is familiar: select the root node of one tree, partition the other according to that node, and perform the intersection on both of the resulting subtree pairs.

Unification results for the intersection program (Figure 6.11) are similar to those for the union program. Parallel execution took about twice as many unifications as sequential; this is again attributable to rollbacks and backtracking caused by having consumer goals awakened too soon.

Unlike many of the other test programs, the presence or lack of temporal inflation had little effect on the results for `inter`. In this case, more noticeable was that the versions accepting all pending messages outperformed those accepting only one incoming message per unification, both in the number of messages sent (see Figure 6.12) and in the number of unifications done. In breaking down the message count figures, the differences are mainly due to variations in the number of BIND and ANTI-BIND messages sent, either to be received or annihilated.

### 6.3.4 Deeply-nondeterministic Programs

`x15` and `xy15`

The `x15` program and its relative, `xy15`, were designed to display vigorous (if rather shallow) nondeterministic behavior. Because of this nondeterminism, performance was measured in two ways: once for finding the first solution, and again for finding all solutions.

The programs (and queries for each) are displayed in Figures 6.13 and 6.14: they



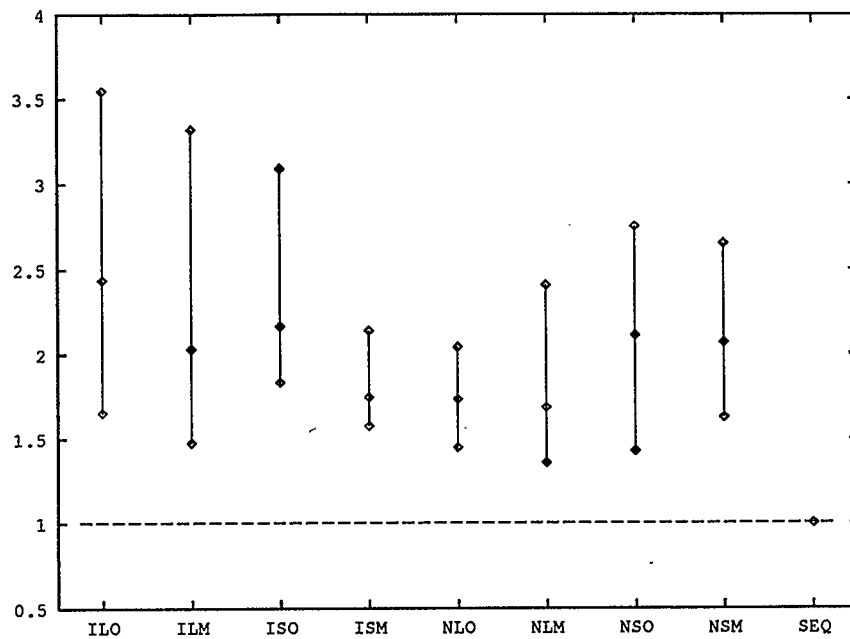


Figure 6.11: Unification average, minimum, and maximum for inter.

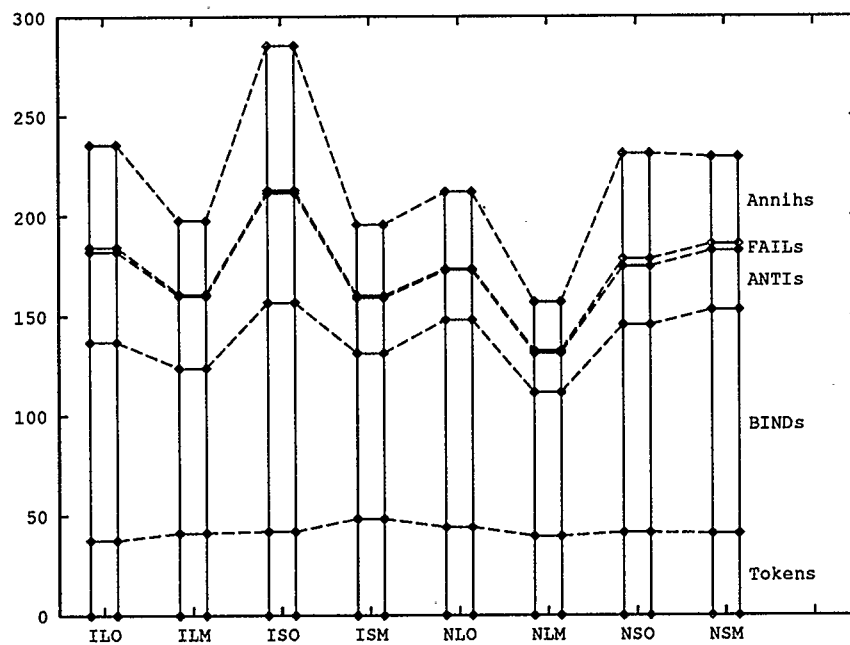


Figure 6.12: Message counts for inter.

```

x(1,2) .
x(2,3) .
x(3,4) .
x(4,5) .
x(5,1) .

:- x(A,B)@p1, x(B,C)@p2, x(C,D)@p3, x(D,E)@p4, x(E,A) .

```

Figure 6.13: Program and query for x15.

```

x(1,N) :- y(1,N) .    y(1,2) .
x(2,N) :- y(2,N) .    y(2,3) .
x(3,N) :- y(3,N) .    y(3,4) .
x(4,N) :- y(4,N) .    y(4,5) .
x(5,N) :- y(5,N) .    y(5,1) .

:- x(A,B)@p1, x(B,C)@p2, x(C,D)@p3, x(D,E)@p4, x(E,A) .

```

Figure 6.14: Program and query for xy15.

have the property that each process tries to bind two variables, and that each variable is shared by two processes. When x15 is run, three (or rarely, four) processes will contribute bindings; for xy15, each process may contribute a binding to the solution. As a result, execution can easily result in a situation like that in Figure 5.7, causing one or more solutions to be missed because of the over-optimization of FAIL messages. (In fact, this over-optimization, discussed in Chapters 4 and 5, was only discovered during all-solutions testing.)

Results for these tests are thus on rather shaky ground. In the all-solutions testing, test runs that missed solutions obviously have to be discarded—but then the successful runs whose results are retained no longer give a representative sample of the possible execution paths. The situation is even worse for the single-solution case: in every test run, a solution

was found, but there is no guarantee that that solution was “first.” Therefore, first-solution results must be deemed completely unreliable.

Only the results for successful all-solutions runs are presented (in Figures 6.15 through 6.18), and these permit only guarded comparison between different versions of the parallel system. (Some versions may succeed more often than others. For the `xy15` program, the inflated versions were less likely to miss a solution, given the synchronizing effect of temporal inflation and the fact that *any* process can be a producer.) All parallel versions do much more work than the sequential interpreter.

In unifications for `x15` (Figure 6.15), the noninflationary versions performed slightly better than the inflated ones. The same holds for the message counts (Figure 6.16). As well, the many-message versions sent and received fewer messages than the one-message versions, reporting fewer BINDs, ANTI-BINDs, and annihilations (but more FAILs).

For `xy15`, the results changed dramatically, both for unifications and for message counts (see Figures 6.17 and 6.18 respectively). The temporally-inflated versions performed much better than the noninflated versions. Also, the many-message versions were superior to the one-message versions, particularly in the noninflated cases. Breaking down the message counts by type reveals more detail. In every category but termination tokens, the noninflated versions reported higher counts than the inflated ones. One-message versions were higher in ANTI-BINDs, FAILs, and annihilations, but lower in BINDs, than their many-message counterparts.

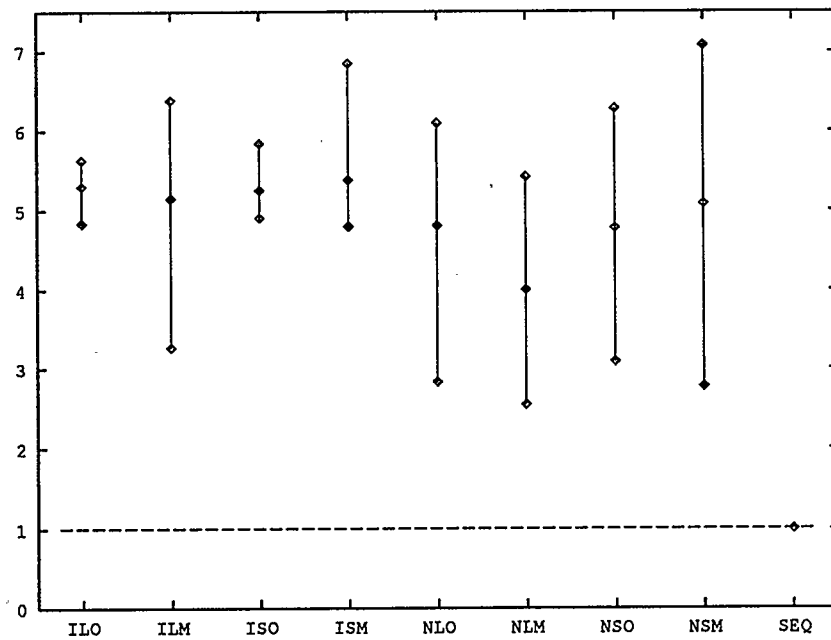


Figure 6.15: All-solutions averages, minima, and maxima for  $x_{15}$ .

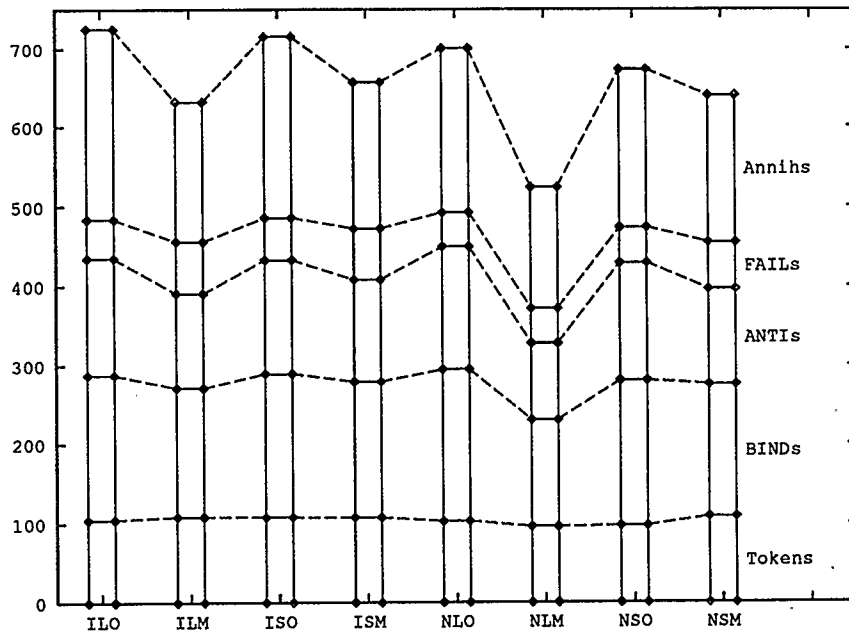


Figure 6.16: All-solutions message counts for x15.

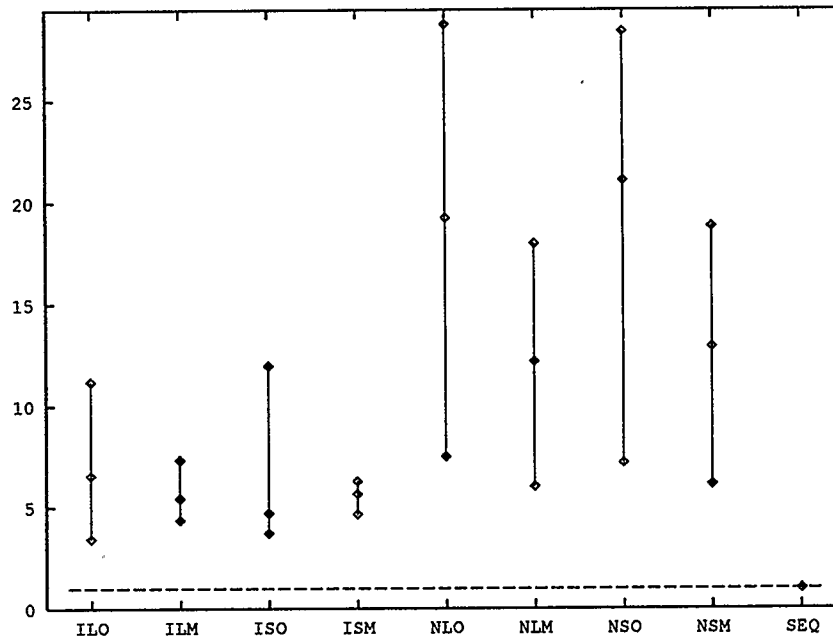


Figure 6.17: All-solutions averages, minima, and maxima for xy15.

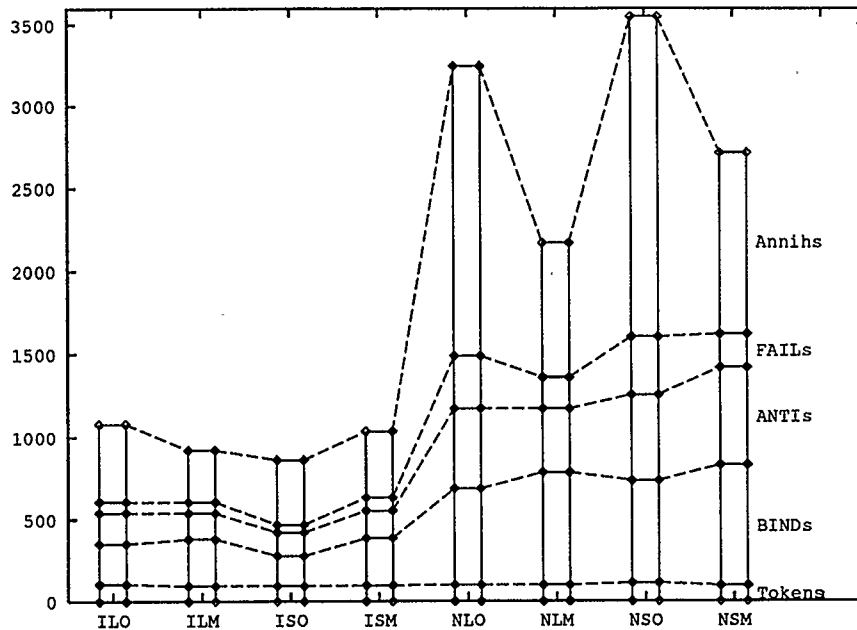


Figure 6.18: All-solutions message counts for xy15.

## 6.4 Analysis

### 6.4.1 Comparison with the Sequential Interpreter

In all of the fully-deterministic tests (`mmult`, `fib`, and `tak`), the parallel runs did little more work than the sequential interpreter did. The extra work done corresponds to rollbacks caused by late (but consistent) messages; it could be avoided if the BIND optimization were implemented as described in Chapter 5. Beyond this, achieving good speedups depends only on achieving a balanced execution—work distributed equally among all processors.

Programs that were shallowly nondeterministic (`qsort`, `union`, and `inter`) took significantly more work when executed in parallel than when executed sequentially. For the most part, this was because consumer goals were woken prematurely. A large improvement would be expected with the addition of more-precise delay declarations. Further (though

	mmult	fib	tak	qsort	union	inter	x15	xy15	Total
INFL/NON	0	0	0	+NON	+NON	0	+NON	+INFL	+2 NON
SML/LGE	0	0	0	0	+SML	0	0	0	+1 SML
ONE/MANY	0	0	0	+ONE	0	0	0	+MANY	0

Table 6.2: Summary of comparisons of unifications.

	mmult	fib	tak	qsort	union	inter	x15	xy15	Total
INFL/NON	0	0	0	+INFL	+NON	0	+NON	+INFL	0
SML/LGE	0	0	0	0	+SML	0	0	0	+1 SML
ONE/MANY	+ONE	0	0	0	0	0	+MANY	+MANY	+1 MANY

Table 6.3: Summary of comparisons of message counts.

less dramatic) improvement would come again from adding the BIND optimization.

The deeply-nondeterministic programs (x15 and xy15) were clear losers in parallel execution. Where there are clear producer/consumer relationships (for example, the generate-and-test strategy of the  $n$ -queens program), using delays to control stream AND-parallel execution may be of avail. Some programs, such as those tested, have no obvious producers or consumers; in this case, it must be sufficient merely to deal with the programs successfully, in the hope that the nondeterminism is isolated, so that the program as a whole still benefits from AND-parallel execution. Such programs could still benefit from both the BIND and ANTI-BIND optimizations, however.

#### 6.4.2 Comparison Between Parallel Versions

Differences between versions of the parallel system were not overwhelming. A summary of the better and worse results is presented in Tables 6.2 and 6.3. (As noted above, results for x15 and xy15 are somewhat suspect, but they are included for completeness.)

**Temporal inflation**

Allowing temporal inflation gave no benefit in the fully deterministic test runs, and seemed clearly detrimental in the shallowly nondeterministic executions. (This detrimental effect would likely be removed by avoiding the premature awakening of consumer goals and adding the BIND optimization.) Only in the `xy15` test was inflationary execution of any avail.

At present, it is too early to tell whether the inflationary option should be retained; the information presented above is inconclusive. Tests seemed to go as expected, with temporal inflation having little effect on deterministic programs, but affecting the execution of deeply nondeterministic programs.

Two factors must yet be examined: the effect of the BIND optimization on the results reported by the inflated versions, and the effect of inflationary execution on other deeply nondeterministic programs.

**Small vs. large time window**

The results comparing large and small time windows are even less conclusive than those for inflationary execution. Only one program exhibited behavior that depended on using a small or large message window. Testing of further examples is also indicated here; one line of attack would be to try “busier” programs, in order to test the system under heavier message traffic.

**One vs. many messages**

The results for accepting one or all pending messages are also inconclusive, although they at least suggest that the “many” version is superior for running deeply nondeterministic programs, while the “one” version may be better with deterministic and shallowly nonde-



terministic programs. Again, further testing is indicated, so that these suspicions can be corroborated or denied.

### 6.4.3 Conclusion

AND-parallel execution worked very well for the deterministic examples. For shallow nondeterminism, the results were less impressive, but there is hope that by applying the BIND optimization and delaying output unifications, parallel performance could improve dramatically. Finally, the results for executing deeply-nondeterministic programs were poor, as expected. Even in this case, there is hope: such programs would benefit from both BIND and ANTI-BIND optimizations, and inflationary execution may help sort priorities out so that parallel execution is not too expensive.

## Chapter 7

### Conclusion

This thesis describes the implementation and testing of a distributed AND-parallel interpreter for pure Prolog. The interpreter is based on an algorithm due to [Cleary *et al* 1987]. This is a stream AND-parallel backtracking algorithm, incorporating the advantages of both the concurrent logic programming languages (i.e. dependent parallelism) and the independent AND-parallel Prologs (i.e. backtracking). The combination offers more parallelism than is available in the independent AND-parallel systems, while retaining the Prolog semantics given up by the concurrent languages. The language accepted by the interpreter is kept as close to Edinburgh-style Prolog as possible.

In implementing the algorithm of [Cleary *et al* 1987], several problems arose, leading to minor corrections to the algorithm and a more exact specification. A prominent example came in dealing with aliased variables: if two variables were aliased to each other, then binding one would immediately require binding the other. On backtracking, the system would be unable to determine which binding occurred first, making correct execution impossible. To alleviate this problem, aliasing was prevented by using delays on  $=/2$ . Other refinements include specifying when a process may accept incoming messages safely: only during forward execution—that is, after a successful unification.

A number of optimizations were suggested in the original algorithm. A BIND message arriving in the past of a process need not cause that process to roll back, as long as the bindings in the BIND are compatible with those already known to the process. Similarly, an ANTI-BIND need not cause a rollback if search paths rejected because of the original

binding are retried. These optimizations were not implemented, but a proposed implementation was described, and testing suggests that adding them would be well worth the effort.

Testing demonstrated that the system worked well for fully deterministic test programs, with few runs requiring more than 10% more unifications than a comparable sequential interpreter. Results for shallowly-nondeterministic programs were less encouraging, as parallel runs took on average two to three times as many unifications as a sequential run. Still, this result is not as bad as it seems. Two alterations to the system would completely prevent this extra work from being done: first, implementing the BIND optimization; and second, preventing consumer goals from executing too soon by postponing output unifications in producer goals until the chosen clause is known to be the correct one.

Deeply-nondeterministic programs, as expected, performed poorly. Such programs would benefit from both the BIND and ANTI-BIND optimizations. Executing these programs also made it clear that the original algorithm contained an over-optimization that allowed potential solutions to be missed. This over-optimization, concerned with directing nonlocal backtracking via FAIL messages, was analyzed in depth and a new, safer optimization was proposed.

Eight different versions of the system were tested, corresponding to all possible combinations of the three binary execution parameters available. Variations between the versions were small, but visible. Further testing is required to determine the relative utility of each of the versions. This is particularly true of the deeply-nondeterministic programs, whose results were tainted by the effects of the FAIL over-optimization.

## 7.1 Future Work

Several opportunities exist for further work. Some affect the execution algorithm; others affect the speed or usability of the system. Improvements include the following:

- replacing the FAIL over-optimization with code that handles FAIL messages correctly;
- implementing the BIND and ANTI-BIND optimizations;
- altering the system to send BINDs and ANTI-BINDs to specific recipients, rather than broadcasting them;
- using a different, faster message-passing subsystem;
- implementing more powerful delays;
- adding more builtins (particularly negation).

As each of these improvements is made, the system should be tested and evaluated, using both the test programs used in this thesis and other ones.

The most important improvement is to remove the FAIL over-optimization. This alteration will result in correct execution in all cases, which is clearly desirable—and necessary. Adding the BIND optimization should improve the performance of all test programs. The ANTI-BIND optimization should help deeply-nondeterministic programs. (Deterministic and shallowly-nondeterministic programs do no nonlocal backtracking, and therefore will not gain from this optimization.)

Alterations to the message-passing system will improve the scalability and speed of the system. Scalability will be enhanced by eliminating message broadcasts, which work

well enough on a five-processor platform but could easily cause a larger system to be swamped with messages. Speed could be increased by using a faster message-passing system, whether ear processes were retained or an interrupt-driven scheme were used.

The remaining additions allow more functionality and flexibility to the user. As such, they appear somewhat tangential to the main goal of furthering research. However, they will eventually become necessary to allow a larger set of Prolog programs to be tested.

Many possible improvements have not been listed. For example, it would be nice to have a blindingly-fast compiler-based system. However, this and other additions must be left to the interested implementor.

## 7.2 Summary

The work presented in this thesis constitutes a significant contribution to research in the field of parallel Prolog systems. Contributions of the work include the following:

- A real, working distributed stream AND-parallel interpreter for pure Prolog, based on the algorithm of Cleary *et al*, has been implemented.
- Testing of the implementation demonstrates that a flexible goal-ordering method based on virtual time can achieve good performance, and that changing the goal ordering can affect that performance.
- Proposed implementations of optimizations to improve performance for shallowly- and deeply-nondeterministic programs were presented.
- The system provides a base for further research into parallel Prolog execution—for example, for experimenting further with different goal-ordering strategies.

Based on the results of this and other work [Tebra 1989, Somogyi *et al* 1988], it is clear that the benefits of stream parallelism need not be traded off against the ability to backtrack. Stream AND-parallelism and backtracking can successfully be combined in a single system, and at acceptable cost.

## Bibliography

- [Beer 1990] Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. PhD thesis, GMD Research Center for Innovative Computer Systems and Computer Technology, Technical University Berlin, 1990.
- [Clark & Gregory 1986] K.L. Clark and S. Gregory. PARLOG: parallel programming in logic, *ACM TOPLAS*, 8(1):1–49, 1986.
- [Cleary *et al* 1987] J.G. Cleary, B.W. Unger, and X. Li. A distributed AND-parallel backtracking algorithm using Virtual Time. Research Report 87/281/29, Department of Computer Science, University of Calgary, October 1987.
- [Conery 1987] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.
- [Conery & Kibler 1985] John S. Conery and Dennis F. Kibler. AND parallelism and nondeterminism in logic programs, *New Generation Computing*, 3(1):43–70, 1985.
- [DeGroot 1984] Doug DeGroot. Restricted AND-parallelism, In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, November 1984.
- [Dijkstra *et al* 1983] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations, *Information Processing Letters*, 16:217–219, 1983.
- [Foster & Taylor 1990] Ian Foster and Steve Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.
- [Hwang & Briggs 1984] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.
- [JADE 1985] SRDG, University of Calgary. *JADE User's Manual*, 1985.
- [Jefferson 1985] David R. Jefferson. Virtual Time, *ACM TOPLAS*, 7(3):404–425, July 1985.

- [Jefferson & Sowizral 1985] David R. Jefferson and Henry A. Sowizral. Fast concurrent simulation using the Time Warp mechanism, In *Proceedings of the SCS Distributed Simulation Conference*, San Diego, CA, January 1985.
- [Kalé 1985] L.V. Kalé. *Parallel Architectures for Problem Solving*. PhD thesis, Department of Computer Science, SUNY Stony Brook, 1985.
- [Li & Martin 1986] P. Li and A.J. Martin. The Sync model: a parallel execution method for logic programming, In *Proceedings of the 1986 Symposium on Logic Programming*, pages 223–234, Salt Lake City, Utah, 1986.
- [Lloyd 1984] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
- [Naish 1986] Lee Naish. *Negation and Control in Prolog*. PhD thesis, Department of Computer Science, University of Melbourne, 1986. Published as Lecture Notes in Computer Science 238 by Springer-Verlag.
- [Pereira et al 1986] L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparício. Delta Prolog: a distributed backtracking extension with events, In *Proceedings of the Third International Conference on Logic Programming*, pages 69–83, 1986. Published as Lecture Notes in Computer Science 225 by Springer-Verlag.
- [Shapiro 1983] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, February 1983.
- [Somogyi et al 1988] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A backtracking algorithm for the stream AND-parallel execution of logic programs. Technical Report 87/10, Department of Computer Science, University of Melbourne, May 1988.
- [Tebra 1987] Hans Tebra. Optimistic AND-parallelism in Prolog, In *Parallel Architectures and Languages Europe*, pages 420–431, 1987. Published as Lecture Notes in Computer Science 258 by Springer-Verlag.
- [Tebra 1989] Hans Tebra. *Optimistic AND-Parallelism in Prolog*. PhD thesis, Vrije Universiteit, Amsterdam, 1989.



- [Thom & Zobel 1988] J.A. Thom and J. Zobel. *NU-Prolog Reference Manual, Version 1.3*. Department of Computer Science, University of Melbourne, 1988.
- [Ueda 1985] K. Ueda. Guarded Horn clauses. Technical Report TR-103, ICOT, June 1985.

## Appendix A

### Pseudocode for AND-Parallel Prolog Algorithm

#### Local Goal Execution

```
handle incoming messages
if a woken goal is available then
    choose it for execution
else
    choose a pending goal
increment local clock
create goal stack frame
if goal satisfies delay conditions then
    continue with local clause execution for first clause of goal
else
    put goal on delayed list
    continue with local goal execution
```

#### Local Clause Execution

```
create clause stack frame
increment ID counter; use value as unique ID for frame
resolve current goal against head of chosen clause
if resolution fails then
    continue with local clause failure for this clause
else
    move any goals that now satisfy delays from delayed list to woken
    list
    determine all shared variables bound by the resolution
    send one BIND message to each processor
    for each outgoing BIND, store a corresponding ANTI-BIND
    if process limit not reached then
        create a process for each clause subgoal that specifies a
        new process
    continue with local goal execution
```

## Local Goal Failure

```

send one ANTI-BIND message for each outgoing message of current frame
if current stack frame is a remote frame then
    send FAIL message to originator
    include information from sibling stack frame, pretending that frame
    is local, even if it is actually a remote frame
    backtrack all bindings associated with this frame
    delete this clause frame and parent goal frame
    continue with local goal execution from sibling stack frame
else
    backtrack all bindings associated with this frame
    if child processes from this frame exist then
        kill those child processes
    delete subgoals of this frame from all lists
    return re-delayed goals from woken list to delayed list
    delete this clause frame
    continue with local clause failure from goal frame
    
```

## Local Clause Failure

```

select next clause
if no alternative clauses for current goal then
    delete goal frame
    if goal was woken then
        return goal to woken list
    else
        return goal to pending list
    continue with local goal failure of sibling
else
    continue with local clause execution for selected clause
    
```

## Receipt of Bind Message

```

look for goal frame with same timestamp as that of incoming message
if goal frame exists then
    if associated clause frame has a unique ID greater than message
    then
        ignore the incoming message and exit
    else
    
```

```

        roll back stack frames up to and including associated
        clause frame
        create clause frame with unique ID of message
    else
        roll back stack frames with timestamp greater than message
        create goal frame and clause frame for message
        attempt unification of bindings in message with local variables
    if unification fails then
        remove goal and clause frames
    else
        move any goals that now satisfy delays from delayed list to woken
        list

```

### Receipt of ANTI-BIND Message

```

    look for goal and clause frame with same timestamp and unique ID
    if frames exist then
        roll back all stack frames up to located goal frame
    else
        ignore message

```

### Receipt of FAIL message

```

    look for goal and clause frame with same timestamp and unique ID
    if frames exist then
        if prior stack frame does not already exist then
            create new goal and clause frames with sibling timestamp
            and ID supplied by FAIL message, and insert in stack
        if frames are remote then
            send FAIL message to originator
            include info from sibling, pretending that frame is local
            roll back all stack frames through to located goal frame
            continue with local goal execution from sibling stack
            frame
        else
            roll back all frames with timestamp greater than message
            continue with local clause failure of clause frame
    else
        ignore the message

```

## Appendix B

### Source Code of Test Programs

#### Matrix multiplication

```
mmult (M1,M2,MM) :-
    transpose (M2,M2T),
    matmult (M1,M2T,MM) .

transpose (M, []) :-
    nullrows (M) .
transpose (M1, [Row|M2]) :-
    makerow (M1, Row, M3),
    transpose (M3, M2) .

makerow ([], [], []).
makerow ([ [X|R1] | M1 ], [X|Row], [R1|M2]) :-
    makerow (M1, Row, M2) .

nullrows ([]).
nullrows ([[]|M]) :-
    nullrows (M) .

matmult ([], _, []).
matmult ([R1|M1], M2T, [MR1|RMM]) :-
    mult_row (R1, M2T, MR1) @ p1,
    matmult (M1, M2T, RMM) .

mult_row (_, [], []).
mult_row (R1, [C1|M2T], [D1|DR]) :-
    dot (R1, C1, D1),
    mult_row (R1, M2T, DR) .

dot ([], [], 0) .
dot ([H1|V1], [H2|V2], Dot) :-
    is (Part, * (H1, H2)),
    dot (V1, V2, RDot),
    is (Dot, + (Part, RDot)) .

?- mmult ([ [1,2,3,4], [6,7,8,9], [11,12,13,14] ],
    [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ], MM) .
```

**Fibonacci numbers**

```

fib(0,1).
fib(1,1).
fib(N,F) :-
    >(N,1),
    is(N1,-(N,1)),
    is(N2,-(N,2)),
    fib(N1,F1)@p1,
    fib(N2,F2)@p2,
    is(F,+(F1,F2)).

```

```

?- fib(5,N).

```

**The tak benchmark**

```

tak(X,Y,Z,A) :- =<(X,Y), =(Z,A).
tak(X,Y,Z,A) :-
    >(X,Y),
    is(X1,-(X,1)), tak(X1,Y,Z,A1)@p1,
    is(Y1,-(Y,1)), tak(Y1,Z,X,A2)@p2,
    is(Z1,-(Z,1)), tak(Z1,X,Y,A3)@p3,
    tak(A1,A2,A3,A).

```

```

?- tak(9,6,3,A).

```

**Quicksort using difference lists**

```

?- qsort(L1,L2) when L1.
qsort(L1,L2) :-
    qsort_dl(L1,L2,[]).

?- qsort_dl(L1,L2B,L2E) when L1.
qsort_dl([],L,L).
qsort_dl([X|L1],L2B,L2E) :-
    part(L1,X,Littles,Bigs),
    qsort_dl(Littles,L2B,[X|L2M])@p1,
    qsort_dl(Bigs,L2M,L2E)@p2.

```

```

?- part(A,P,L,B) when A and P.
part([X|Xs],Y,[X|Ls],Bs) :-
    =<(X,Y),
    part(Xs,Y,Ls,Bs).
part([X|Xs],Y,Ls,[X|Bs]) :-
    >(X,Y),
    part(Xs,Y,Ls,Bs).
part([],Y,[],[]).

?- qsort([27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,47,28,82,6,11],S).

```

### Set union using ordered trees

```

?- neq(N1, N2) when N1 and N2.
neq(N1, N2) :-
    >(N1, N2).
neq(N1, N2) :-
    <(N1, N2).

?- split(N, T, _, _) when N and T.
split(_, nil, nil, nil).
split(N, t(N,L,R), L, R).
split(N, t(X,L,R), LL, t(X,LR,R)) :-
    <(N, X),
    split(N, L, LL, LR).
split(N, t(X,L,R), t(X,L,RL), RR) :-
    >(N, X),
    split(N, R, RL, RR).

?- union(T1, T2, _) when T1 and T2.
union(nil, T, T).
union(t(X,L,R), nil, t(X,L,R)).
union(t(X,L1,R1), t(X,L2,R2), t(X,L3,R3)) :-
    union(L1, L2, L3)@p1,
    union(R1, R2, R3).
union(t(X1,L1,R1), t(X2,L2,R2), t(X1,L3,R3)) :-
    neq(X1, X2),
    split(X1, t(X2,L2,R2), TL, TR),
    union(L1, TL, L3)@p1,
    union(R1, TR, R3)@p2.

?- union(t(6,t(4,t(3,t(1,nil,nil),nil),t(5,nil,nil)),t(8,t(7,nil,nil),nil)),
    t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(7,t(6,nil,nil),nil)), T).

```

**Set intersection using ordered trees**

```

?- neq(N1, N2) when N1 and N2.
neq(N1, N2) :-
    >(N1, N2).
neq(N1, N2) :-
    <(N1, N2).

?- split(N, T, _, _) when N and T.
split(_, nil, nil, nil).
split(N, t(N,L,R), L, R).
split(N, t(X,L,R), LL, t(X,LR,R)) :-
    <(N, X),
    split(N, L, LL, LR).
split(N, t(X,L,R), t(X,L,RL), RR) :-
    >(N, X),
    split(N, R, RL, RR).

?- inter(T1, T2, T3) when T1 and T2.
inter(nil, _, nil).
inter(t(_,_,_), nil, nil).
inter(t(X1,L1,R1), t(X2,L2,R2), t(X1,LI,RI)) :-
    mem(X1, t(X2,L2,R2)),
    split(X1, t(X2,L2,R2), TL, TR),
    inter(L1, TL, LI)@p1,
    inter(R1, TR, RI)@p2.
inter(t(X1,L1,R1), t(X2,L2,R2), TI) :-
    non_mem(X1, t(X2,L2,R2)),
    split(X1, t(X2,L2,R2), TL, TR),
    inter(L1, TL, LI)@p1,
    inter(R1, TR, RI)@p2,
    make_tree(LI, RI, TI)@p3.

?- mem(N, T) when N and T.
mem(N, t(N,_,_)).
mem(N, t(X,L,_)) :-
    <(N, X),
    mem(N, L).
mem(N, t(X,_,R)) :-
    >(N, X),
    mem(N, R).

?- non_mem(N, T) when N and T.
non_mem(_, nil).
non_mem(N, t(X,L,_)) :-
    <(N, X),
    non_mem(N, L).
non_mem(N, t(X,_,R)) :-

```



```

>(N, X),
non_mem(N, R).

rightmost(t(X,L,nil), X, L).
rightmost(t(X,L,R), RM, t(X,L,RR)) :-
    rightmost(R, RM, RR).

?- make_tree(T1, T2, _) when T1 and T2.
make_tree(nil, T, T).
make_tree(t(X,L,R), nil, t(X,L,R)).
make_tree(t(X,L,R), t(X2,L2,R2), t(RM,LL,t(X2,L2,R2))) :-
    rightmost(t(X,L,L), RM, LL).

?- inter(t(6,t(4,t(3,t(1,nil,nil),nil),t(5,nil,nil)),t(8,t(7,nil,nil),nil)),
        t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(7,t(6,nil,nil),nil)), T).

```

### x15 program

```

x(1,2).
x(2,3).
x(3,4).
x(4,5).
x(5,1).

?- x(A,B)@p1, x(B,C)@p2, x(C,D)@p3, x(D,E)@p4, x(E,A).

```

### xy15 program

```

run(A,B,C,D,E) :- x(A,B), x(B,C)@p1, x(C,D)@p2, x(D,E)@p3, x(E,A)@p4.

x(1,N) :- y(1,N).
x(2,N) :- y(2,N).
x(3,N) :- y(3,N).
x(4,N) :- y(4,N).
x(5,N) :- y(5,N).

y(1,2).
y(2,3).
y(3,4).
y(4,5).
y(5,1).

?- run(A,B,C,D,E).

```