Working Paper Series
ISSN 1170-487X

The Design of an Optimistic AND-Parallel Prolog

by John G. Cleary & Ian Olthof

Working Paper 93/6

October, 1993

The Design of an Optimistic AND-Parallel Prolog

Ian Olthof*
John G. Cleary†

October 26, 1993

Abstract

A distributed AND-parallel Prolog implementation is described. The system can correctly handle all pure Prolog programs. In particular, it deals with the problem of distributed backtracking. Conflicts in variable bindings are resolved by assigning a time value to every unification. Bindings with smaller time values are given precedence over those with larger time values. The algorithm is based on the optimistic Time Warp system, with Prolog-specific optimizations. These optimizations include two new unification algorithms that permit unification and backtracking in any order. The result is a system which can fully exploit the parallelism available in both dependent and independent AND-parallelism.

1 Introduction

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 possible, or in attempting to extract that parallelism, give up the logical semantics of Prolog. The *Time Warp* model [Jefferson & Sowizral 1985] is used here as a basis for combining *independent* and *dependent* AND-parallelism with *backtracking*, thus achieving maximum parallelism while retaining Prolog's logical semantics.

The major hurdle in the successful exploitation of AND-parallelism is in handling backtracking in the presence of shared variables. The na ve 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 method is seen to be infeasible [Conery 1987, DeGroot 1984]; 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. Parallelism is thus restricted to goals known to be independent. Such knowledge may be gained statically, dynamically, or by a combination of both.

Kale's REDUCE-OR Process Model [Kale 1985] uses static analysis to produce annotations (specifically, *data join graphs*) to control parallel execution. Such analysis must be conservative to avoid pitfalls like aliased variables that could result in dependent goals running in parallel. Conery's AND-OR process model [Conery 1987] uses dynamic analysis and needs no annotations. Because goal independence is determined at run time, independent parallelism is maximized, though at the cost of a high run-time overhead. DeGroot's Restricted AND-Parallelism (RAP) model [DeGroot 1984] takes a hybrid approach, combining annotations generated at compile time with simple run-time checks for independence to handle difficult cases.

The *concurrent logic programming* (CLP) 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

*Author's address: Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, CANADA, T2N 1N4. *Email:* olthof@cpsc.ucalgary.ca

†Author's address: Department of Computer Science, University of Waikato, Hamilton, NEW ZEALAND. *Email:*cleary@waikato.ac.nz

channels between processes (goals). All goals—even those with variables in common—may execute concurrently. Languages in this class include PARLOG [Clark & Gregory 1986], Concurrent Prolog [Shapiro 1987] , and GHC [Ueda 1987] and their "flat" versions; newer languages like Strand [Foster & Taylor 1990] have recently come on the scene as well.

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 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. For several years, combining the two was deemed impractical. Recently, however, a number of algorithms have been published that combine stream AND-parallelism and backtracking [Cleary *et al* 1988, Pereira *et al* 1986, Somogyi *et al* 1988, Tebra 1987].

Central to all of these algorithms (either implicitly or explicitly) 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 *precedence* of a goal: the earlier it appears in the ordering, the higher its precedence. If two goals disagree on the value of a binding, the binding made by the higher-precedence goal is accepted; the lower-precedence goal must retract its binding and recompute. When a lower-precedence goal can find no solution, it may ask a higher-precedence goal to recompute its bindings.

Some systems, like those of [Pereira *et al* 1986] and [Tebra 1987], retain the depth-first ordering of sequential Prolog. Others, like that of [Somogyi *et al* 1988], base the ordering on producer/consumer relationships between goals. The least restrictive algorithm is that of [Cleary *et al* 1988]: any ordering will suffice.

Our algorithm is based on that given in [Cleary *et al* 1988] and [Olthof 1991], with optimizations suggested in those earlier works correctly incorporated into the main algorithm. The algorithm combines a number of desirable features. Like Tebra's design, execution is optimistic, allowing fully parallel execution of dependent goals. Like Somogyi's algorithm, ours is based on message passing, allowing its direct implementation on distributed-memory as well as shared-memory platforms. It is also a highly-optimized algorithm which minimizes wasted work by exploiting the regularity of Prolog to optimize Time Warp. Finally, because of its use of timestamped bindings, it is simple to add intelligent backtracking (based on the technique given in [Mannila & Ukkonen 1986]).

The next section provides an outline of virtual time and Time Warp, on which our AND-parallel Prolog system is based. Virtual times provide a total ordering on the goals in an execution, and Time Warp's rollback mechanism offers an undo operation. Section 3 introduces the distributed Prolog algorithm itself, describing the data structures and messages used, after which an execution model is given. Two specialized unification algorithms, designed to optimize the execution of the system, are described in Section 4. Section 5 illustrates the execution of an example contrived to demonstrate the behavior of all phases of execution. This is followed by a detailed description of the algorithm in Section 6. Section 7 outlines a number of further optimizations to the basic algorithm. Section 8 derives some simple results on the amount of parallelism that can be extracted by considering the limit when the overheads of the algorithm are negligible. Finally, Section 9 summarizes the work.

2 AND-Parallelism Using Virtual Time

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 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 can be used to construct a virtual time system.

One of two strategies may be used to ensure that the ordering within a distributed virtual time system is correct. The *conservative* strategy defers the processing of an event until it is certain that no other event with an earlier virtual time that can affect it is yet to be processed. Conversely, the *optimistic* strategy allows an event to be processed immediately, on the (optimistic) assumption that the real-time order of processing will agree with the virtual-time event ordering. If at some point in a computation this assumption is found to be false, some form of order repair must occur.

Optimistic distributed systems have been shown [Fujimoto 1990] to be significantly more efficient for distributed simulation than conservative ones. In the case where the optimistic assumption is correct and no order repair is necessary, optimism clearly provides more potential parallelism. Even in the case where order repair is necessary, a process in a conservative system would have to wait until the out-of-order message arrived anyway.

The *Time Warp* mechanism [Jefferson & Sowizral 1985] was used in the first optimistic implementation of a virtual time system. This implementation was designed with parallel simulation in mind, but the ideas behind it can be applied as well to parallel Prolog. In a Time Warp system, each simulation event is associated with a message. Receiving an incoming message corresponds to processing the associated event. An outgoing message schedules an event at some process.

The key component of the Time Warp mechanism is *rollback*, which is used to return the computation to an earlier state so that an incorrect ordering can be repaired. 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 negative message (or *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.

3 Distributed Prolog Algorithm

In examining our distributed Prolog algorithm, we begin with a high-level description of the basic algorithm. We describe the stack and ancillary data structures necessary to implement the algorithm. Next, the message types required by the algorithm and their behaviour in the message queues is considered. Finally, we give an execution model in the form of a state diagram. Sections 5 and 6 are closely related, the former giving an example of parallel execution and the latter providing a detailed description of the algorithm.

3.1 High-Level Description

Our algorithm is based on the Time Warp system. Each unification corresponds to a Time Warp event and each communication of binding information corresponds to a Time Warp message. Each solver (process) works on its own part of the whole computation, alternating local execution and message processing. As with basic Time Warp, each event and message is uniquely timestamped.

Local execution may process forward (goal selection, clause selection, and unification) or backward (backtracking). When a unification results in shared variables being bound, a BIND message is sent to each other solver sharing one or more of these variables. When backtracking causes bindings of shared variables to be retracted, an ANTI-BIND message is sent out to each solver that shares such a backtracked variable, so that the original BIND is annihilated. In a similar fashion, goals are allocated via GOAL messages and withdrawn via ANTI-GOAL messages. Finally, a distributed Prolog execution occasionally requires that a solver be able to force another to backtrack. This requirement is met by using a FAIL message, which is *not* a Time Warp message—it has no anti-message counterpart.

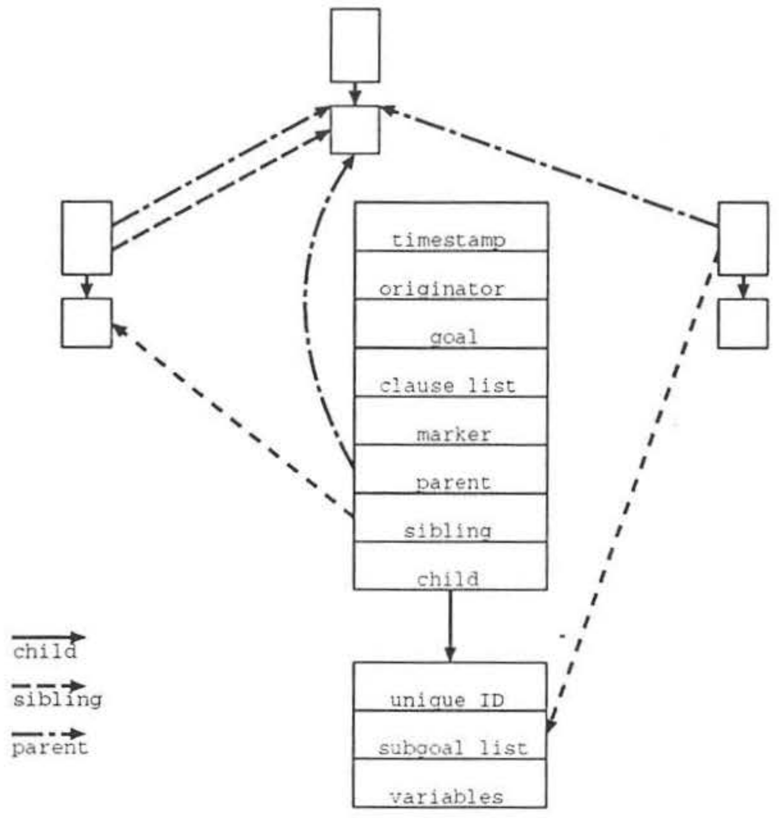

Figure 1: Stack structure and frame contents

This basic algorithm was first proposed by [Cleary *et al* 1988] and implemented in [Olthof 1991]. Test results from the latter exhibited good performance (as measured by the number of unifications performed) for deterministic test programs, but shallowly- and deeply-nondeterministic programs ran less successfully. These results motivated a careful examination of the optimizations first proposed in [Cleary *et al* 1988]. This examination culminated in the unification algorithms of Section 4 and in the full Prolog algorithm of Section 6.

3.2 Stack and Ancillary Structures

Like most sequential Prologs, the local execution of our algorithm is based around a stack. Specifically, each process in the distributed system uses its own stack, executing local goals sequentially and communicating with other processes in the system. The sequential interpreting engine is based on that given in [van Emden 1984]. Entries on the stack are *frames*. *Local* frames are created during the normal process of forward execution. *Remote* frames are created by the arrival of messages. The values in such a frame are inherited from some frame in another process. The main role of remote frames is to function as placekeepers for backtracking. When backtracking arrives at a remote frame, it is redirected to the remote originating frame.

Each frame is divided into two parts, one containing information related to its associated goal, and the other with that related to the currently-selected clause for that goal. These parts are referred to respectively as *goal frames* and *clause frames*. 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. 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.

The stack structure and the contents of each frame are illustrated in Figure 1. Each goal frame contains several components:

- The *timestamp* is the virtual time at which the frame's associated goal is executed.

- The *originator* is used in a remote frame to contain the ID of the process whose stack contains the originating frame. Taken together with the timestamp and unique ID, the originating frame can be unambiguously identified from this process ID. In a local frame, the originator is set to the local process ID.
- The *goal* is itself stored in the frame.
- The *clause list* identifies the clauses available to be unified with the goal.
- The *marker* contains a timestamp and is used to optimize the processing of ANTI-BIND messages. The timestamp refers to the time of a frame lower on the local stack. Each of the frames between (and including) the current frame and that referred to by the timestamp must be handled specially on backtracking. All unselected clauses must be revisited, rather than just those following the current selection.
- The *child, parent,* and *sibling* pointers denote various related clause frames. The child pointer refers to the goal frame's associated clause frame and provides access to clause variables on unification. The parent pointer refers to the clause frame for which the local goal is a body goal, and provides access to the parent clause's variables (i.e. the goal variables) on unification. The sibling pointer refers to the frame in the stack just above the current one, and thus to the next frame to backtrack on failure of the current goal.

Each clause frame also contains several components:

- The *unique ID* distinguishes the frame from any other frame with the same virtual time. This is necessary because timestamps 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 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.
- The *subgoal list* is a list of the goals in the body of the current clause. (For a remote frame, this list is empty.)
- The *variable space* contains allocated storage for each variable local to the clause.

This frame structure could be optimized considerably, particularly in the case of remote frames. We defer such optimization in order to concentrate on the distributed execution algorithm.

In addition to the stack structure, each process must maintain several other pieces of state information. These include the following:

- the network addresses of itself and processes with which it communicates
- the current local virtual time (LVT)
- an integer counter for numbering stack frames (see above)
- an input queue (IQ) to hold incoming messages (see below)
- a list of goals to be executed (`goal_list`)

3.3 Messages and Message Queues

We have chosen to describe the algorithm in terms of message passing rather than, say, a shared memory paradigm. This is done to ensure that the algorithm can be applied to a wide range of machines. It should also be straightforward to translate the algorithm to one which uses shared memory directly.

In a message passing system, variable bindings must be disseminated explicitly, and on backtracking, these bindings must be retracted explicitly. This hooks in neatly to the Time Warp concept of message/anti-message pairs. A binding is propagated via a BIND message; if that binding is later backtracked, it is withdrawn via an ANTI-BIND message that will annihilate the original BIND.

The same principle applies to the allocation of work. Parallel goals may be created throughout the execution and rescinded on backtracking, much as variable bindings are. A GOAL message is used to execute a goal at some process at the virtual time given in the message. Such a goal may be withdrawn by sending an ANTI-GOAL message.

A fifth type of message is necessary for a Prolog system, one that does not fit the Time Warp mould. 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.

Every message has certain basic information in common:

- a timestamp, whose value is given by the virtual time of the current stack frame of the originating process;
- an originator—the process ID of the sending process and the machine ID of the processor it is running on;
- a unique ID, again taken from the stack frame associated with the message.

Some message types have additional fields:

BIND one or more variable/term binding pairs, each indicating that the given variable is bound to the given term;

GOAL the goal to be executed;

FAIL a *context*, identifying the sibling frame (with a timestamp, originator, and a unique ID) of the originating frame. Such contexts are used to ensure that remote backtracking will search for all possible solutions.

ANTI-BIND and ANTI-GOAL messages contain no additional fields.

In order to support the transmission and receipt of all of these messages, the Prolog system needs to provide its own versions of the Time Warp input, output, and state queues (IQ, OQ, and SQ, respectively). 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; rolling back the stack to a certain virtual time amounts to backtracking all frames with greater timestamps. 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 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 five 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 message is not persistent, since it annihilates its corresponding BIND immediately on arrival. If no BIND corresponding to an arriving ANTI-BIND can be found, the BIND must have been rejected, and the ANTI-BIND may be discarded.[1] GOAL and ANTI-GOAL messages are treated exactly like BINDs and ANTI-BINDs, respectively. 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 and ANTI-GOAL 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; similarly for GOAL messages. FAIL messages are never stored in the OQ either. Once sent out, they are forgotten completely by the sender.

In standard Time Warp, global virtual time GVT is calculated regularly, so that *fossil collection* may be performed—entries in each of the queues whose timestamps are less than GVT are reclaimed. This is possible because no process will ever roll back to before the current GVT. In the Prolog algorithm, fossil collection is *not* done, because the history of the computation must be retained—GVT may decrease during backtracking. Thus, GVT is calculated only to determine termination.

[1] In this respect, we assume that the underlying message-passing system guarantees that messages from one process to another are delivered in the order they are sent. If this is not the case, an ANTI-BIND with no corresponding BIND in the input queue would have to be retained until that BIND arrived. The same assumption applies to GOAL and ANTI-GOAL messages.

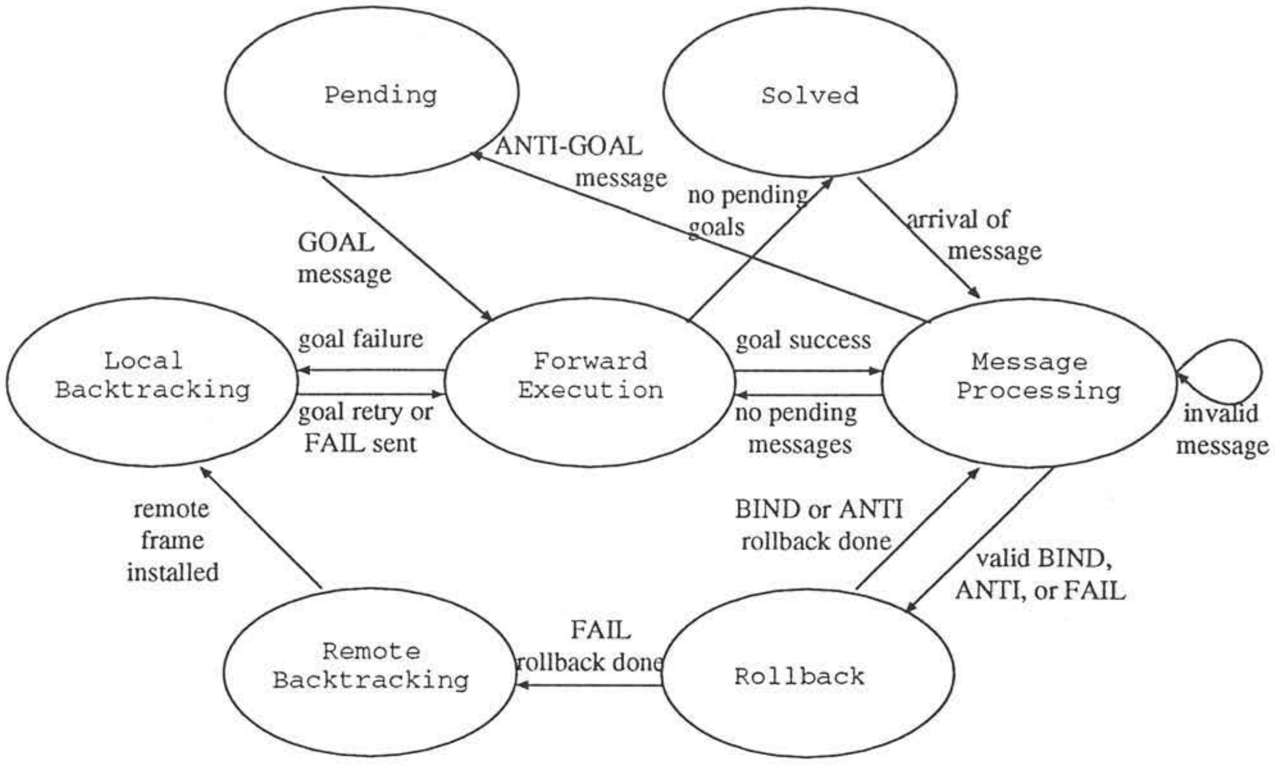

Figure 2: Execution model

3.4 Execution Model

Parallel execution is performed by *processes*. Each process attempts to solve exactly one top-level goal, which it receives in a GOAL message. (This restriction is easily and naturally relaxed, but is employed here for simplicity and fluency of exposition.)

A process may be *active* or *inactive*. A process is inactive either when it has not yet been given any work (the *pending* state), or when it has no work of its own left to do, and is merely waiting for messages to come in from other processes (the *solved* state). When all processes are inactive and no messages are yet to be received, the system has terminated, either with a solution or in failure. (An algorithm like that given in [Dijkstra *et al* 1983] suffices to detect termination.) Otherwise, a process is active, and may be in one of five states: *forward execution, message processing, rollback, local backtracking*, or *remote backtracking*.

These states and their interactions are illustrated in Figure 2. A process begins in the pending state. On receipt of a GOAL message, it begins actual Prolog execution. This is done in the *forward execution* state: a goal and clause are selected and unified. If unification succeeds, any pending messages are read. (Messages need not actually be *processed* at this time. They may arrive asynchronously and be stored in the input queue until they are processed.) If unification fails, local backtracking begins. This continues until either a local goal can be retried, in which case forward execution begins again, or a remote frame is encountered, in which case backtracking is forced on a remote process. Remote backtracking is forced by sending a FAIL message to the process that originated the remote frame. The process that sends the FAIL then restarts forward execution.

During message processing, a message will either be accepted or ignored. Examples of messages that will be ignored are a BIND whose variable bindings conflict with earlier local bindings, and an ANTI-BIND destined for a stack frame that has already been backtracked. If a message has a timestamp earlier than the local virtual time, execution must be rolled back to that timestamp before the message can be processed. A special case of this is the ANTI-GOAL message, which causes the entire stack to be rolled back, after which the process returns to the pending state, awaiting another GOAL message. Alternatively, if the message is a FAIL, remote backtracking occurs, after which forward execution resumes. Otherwise, further messages are processed (including those that may have been processed earlier but later rolled back). When no messages remain to be processed, forward execution resumes.

Execution continues to return to the forward execution phase until no goals remain to be executed, at which point the process becomes inactive. It remains inactive until another message arrives, whereupon it processes the message and (sooner or later) returns to forward execution.

4 Optimizing the Basic Algorithm

The simple form of Time Warp has a rollback occurring whenever any low-timestamped message arrives. When Time Warp is specialized for Prolog execution, these rollbacks often can be avoided, thus reducing overheads and improving total execution time. The results of [Olthof 1991] show that this is particularly important for programs that are nondeterministic. For example, when a program is shallowly nondeterministic, it is common for a BIND to be sent, followed immediately by its ANTI-BIND. The net effect on a process receiving these messages is to leave the binding environment unchanged, but in the meantime, the BIND and the ANTI-BIND will each have caused a rollback of execution which needs to be redone.

In a general Time Warp system, it is difficult to determine whether the contents of a message arriving in the past of a process will affect the computation "after" it. Thus, such computation *must* be rolled back and recomputed, though techniques like lazy cancellation [Gafni 1985] and lazy reevaluation [West 1988] may be employed to mitigate the effects of rollback.

In a Prolog system, the effect of a message on later computation is much easier to determine. Since BIND messages consist purely of variable binding information, and given the single-assignment property of Prolog, binding conflicts are easily detected, and rollback need only be done when such a conflict occurs.

If a BIND message arrives out of order but no conflict arises, it can be accepted without rolling back and recomputing. If a conflict does occur, a local binding may have higher precedence, so that the BIND may be rejected out of hand. Even if the incoming binding takes precedence, the solver need only roll back far enough to remove the offending local binding.

In a way, handling out-of-order ANTI-BIND messages is even simpler: the *removal* of a binding cannot possibly cause a conflict. Thus, rollback in this case is in principle never necessary. Still, two complications arise. The first of these occurs when a compatible but lower-precedence binding is possible for a variable unbound by an ANTI-BIND. Had the BIND corresponding to that ANTI-BIND never arrived, the compatible binding would have been recorded. That is, when a binding is withdrawn, the next-highest-precedence binding must be recorded to keep the computation correct.

The second complication is that when a binding at some virtual time is withdrawn, the constraint on later computation due to that binding is removed. That is, a clause that was rejected in forward execution because of conflict with an earlier remote binding must become a candidate clause again, and be reselected on backtracking.

In this section, we present two unification algorithms that support rollback avoidance. The first of these, the *order-independent unification* (OIU) algorithm, permits BIND messages to be processed in any order, requiring rollback only when an incoming binding conflicts with and has higher precedence than an existing binding. Normally, unification proceeds with only minor alterations to the stack and backtrack trail so that both appear to have been constructed in timestamp order.

Our second algorithm, the *order-independent backtracking* (OIB) algorithm, retains the functionality of the first, and also avoids rollback when processing ANTI-BIND messages—that is, it allows bindings to be withdrawn in any order. To solve the first of the complications noted above, it uses a more complex binding environment that maintains multiple (compatible) bindings for each shared variable. The second complication, that of reselecting candidate clauses, is external to the OIB algorithm, but is handled by the *marker algorithm* presented below.

For both algorithms, we discuss the data structures required, the interface with the overlying Prolog system, and the expected overhead.

4.1 Avoiding BIND rollbacks

As noted above, both the OIU and OIB algorithms permit BIND messages to be processed in any order and with any number of later frames on the stack. Each attempted unification is done at some *unification time*, given by the timestamp of the BIND that is being processed. Each returns a *failure time*, the time to which a rollback must occur for the binding to be accepted. Success (no binding conflicts whatsoever) results in a failure time of $+\infty$ (i.e. no rollback). Complete failure (conflict with a binding whose timestamp is lower than the unification time) gives a failure time equal to the unification time, and results in rejection of the message without recourse to rollback. Finally, partial failure (conflict with a binding later than the unification time) causes a time greater than the unification time but less than $+\infty$ to be returned. In this case, the binding conflict may be removed by rolling back to the failure time.

4.2 Avoiding ANTI-BIND rollbacks

The OIB algorithm maintains redundant bindings for each shared variable. Although this incurs some overhead, the following example demonstrates its necessity. Suppose that in some execution, the goal `p(X)` is to be executed, with `X` a shared variable and the clauses for `p/1` being:

```
p(3).
p(2).
p(7).
```

Say that at time 1, a BIND with `X = 2` is received. Later, at time 5, `p(2)` is executed. The clause `p(3).` does not unify, but `p(2).` does. The redundant binding `X = 2` is trailed at time 5, and forward execution continues. Now, if an ANTI-BIND arrives at time 1, we want to withdraw the binding for `X` without rolling back the stack. If the remote binding (at time 1) is discarded, then the local binding of `X = 2` at time 5 remains, keeping the binding environment as it would have been had the remote binding never occurred.

The overhead of maintaining multiple redundant bindings for variables can be substantial, but it is mitigated by two factors. At any given process, only bindings up to the first local binding need be maintained. Later bindings, whether local or remote, will always be withdrawn before that local binding is backtracked. Also, multiple remote bindings for a variable are unlikely, since it requires that two processors bind the same variable at almost the same real time (before either one hears of the other's binding). Thus, the most complex situation likely to occur is that of one remote binding and one local binding.

The second complication with optimizing ANTI-BIND processing is also illustrated in the example above. Normally, only the clause `p(7).` would be tried after backtracking `p(2).` However, the clause `p(3).` was only rejected because of the (now withdrawn) binding `X = 2` at time 1. Thus, both `p(3).` and `p(7).` need to be retried on backtracking.

A wide range of algorithms are possible to deal with this problem. These trade off execution overhead against accuracy in determining which clauses need to be retried. We believe that the marker algorithm described below is a good compromise. It requires only a small constant overhead on each ANTI-BIND and each time a frame is backtracked or rolled back.

The simplest case requiring clause reselection occurs when a withdrawn binding is the *only* binding for some shared variable. When such a binding is withdrawn, previously-rejected alternate clauses for goals with virtual times between that of the message and the current LVT may become candidates again. (Rollback accomplishes this automatically, but eagerly; a lazy method that retains the current solution path is preferable.) Our solution to this problem is to set a marker at the current top of stack, and continue executing forward. This marker is given the timestamp of the ANTI-BIND that caused it. (An unset or cleared marker is given the time $+\infty$.) When a set marker is encountered on backtracking some frame, *all* alternatives to the current (rejected) clause are retried, even those previously rejected. The marker is then passed down to the next lower stack frame, and the minimum taken with the marker on that frame. When the timestamp of the stack frame is less than that of the marker, the marker is cleared. Rejected clauses for goals with lower timestamps need not be retried, since they could not possibly have failed due to the presence of the withdrawn binding.

Setting a marker is necessary whenever the highest-precedence binding for some shared variable is withdrawn via an ANTI-BIND. If some lower-precedence binding exists, the marker is still given the timestamp of the ANTI-BIND, but it is set in the remote frame corresponding to the highest-precedence remaining binding rather than in the frame at the top of stack. For the frames above that of the remaining binding, no failed clauses need be reselected, since they would still fail.

4.3 Order-Independent Unification

We present here an algorithm that avoids rollback as much as possible when unifying incoming bindings: the order-independent unification (OIU) algorithm. With a traditional unification algorithm, all bindings must be processed in timestamp order. Thus, when a binding with a certain timestamp arrives, the local execution must be rolled back far enough to undo any bindings with higher timestamps. After the incoming binding is processed, the undone bindings can only be recreated through forward execution.

In contrast, the OIU algorithm attempts to insert each newly-processed binding into the stack according to its timestamp. If the variable being bound is unbound, the only effect is to add the new binding to the trail. If the variable is already bound, the situation becomes more complex, and two cases arise: either the bindings conflict, or they are compatible.

If the bindings conflict, then binding timestamps must be considered. If the existing binding has a lower timestamp, then the unification fails, and the incoming binding is rejected—but no rollback is necessary. If the incoming binding has a lower timestamp, a rollback is unavoidable, though it need only go back far enough to undo the existing binding, i.e. a *partial* rollback. The variable is then bound to the incoming binding, and the unification succeeds.

If the bindings are compatible, little need be done. When the incoming binding has the lower timestamp, the existing binding must be modified to reflect that earlier binding time. Otherwise, the existing binding stands. If the bindings are compound terms, each argument of the incoming binding is unified with its existing counterpart, and the success or failure of the entire unification rests on the success or failure of each sub-unification.

Since unification may require a partial rollback for success, some *rollback time* must be computed. Complete success results in a rollback time of $+\infty$, indicating that no rollback is necessary. Complete failure gives a rollback time equal to the timestamp of the incoming binding—that is, for the incoming binding to succeed, it must itself be rolled back, clearly a contradiction. A rollback time greater than the incoming binding time but less than $+\infty$ indicates the need for a partial rollback, after which the unification can succeed.

The only tricky part in calculating the rollback time comes in unifying compound terms. In this case, the rollback time of the entire unification is the minimum of the rollback times of each argument unification. The rollback time is computed incrementally, ending only when no more argument pairs remain to be unified (success) or when some argument pair fails to unify (failure).

Pseudocode for the OIU algorithm is given in Figure 3. The "@" notation used in this pseudocode deserves mention. The line

```
LD := RD @T
```

indicates that variable LD is being bound to term RD at virtual time T, and that the binding is recorded in the backtrack trail of the stack frame with time T.

In the sections that follow, the algorithm is described from several perspectives. First, we consider its interface to an overlying Prolog system. Next, we describe its dereferencing behaviour, and then we examine the unification algorithm in detail. At this point, we outline the data structures necessary to implement the algorithm. Finally, we discuss the time and space overheads that the algorithm incurs.

4.3.1 Prolog Interface

The unification algorithm is called from two places in a distributed-memory parallel Prolog interpreter. The first of these is in normal forward execution, in which case the algorithm behaves exactly like a standard Prolog unification algorithm. All bindings on the trail have lower timestamps (and thus higher precedence) than the bindings currently being created. Thus, the unification algorithm simply either succeeds or fails.

The second occasion for which unification is done is in receiving bindings from an interpreter on another machine. In this case, the incoming bindings may have higher precedence than some bindings already on the stack. Therefore, in addition to the the possibilities of success (no conflicts) and failure (conflict with a higher-precedence binding), there is also the possibility of conflict with a lower-precedence binding. If such a conflict occurs, it is necessary to roll back to the time of that lower-precedence binding.

The unification algorithm is given three parameters: T, the virtual time of the unification; L, the local binding term; and R, the remote (incoming) binding term. (In forward execution, L is the goal term and R is the clause term.) The algorithm returns a value indicating the success or failure of the unification.

```
unify(time T, term L, term R) returning rollback time RVAL

    (TL, LD) := deref(T, L)
    (TR, RD) := deref(T, R)

    if LD = RD
        return +inf
    else if LD is an unbound variable
        LD := RD @T
        return +inf
    else if RD is an unbound variable
        RD := LD @T
        return +inf
    else if LD is a bound variable
        if RD is a bound variable
            RVAL := unify(max(TL, TR), val(LD), val(RD))
            if RVAL > max(TL, TR) /* success */
                undo binding of LD
                LD := RD @T
            end if
            return RVAL
        else /* RD is a compound term */
            RVAL := unify(TL, val(LD), RD)
            if RVAL > TL
                undo binding of LD
                LD := RD @T
            end if
            return RVAL
        end if
    else if RD is a bound variable
        RVAL := unify(TL, LD, val(RD))
        if RVAL > TR
            undo binding of RD
            RD := LD @T
        end if
        return RVAL
    else if functors/arities match
        RVAL := +inf
        for each argument pair (LDi, RDi)
            RVAL := min(RVAL, unify(T, LDi, RDi))
            if RVAL = T
                return RVAL
            end if
        end for
        return RVAL
    else /* functors or arities don't match */
        return T
    end if
```

Figure 3: Pseudocode for order-independent unification

```
deref(time T, term VAR) returning pair (time T', term VAR')

    if VAR is bound
        Tb := binding time of VAR
        if T <= Tb
            return (Tb, VAR)
        else
            return deref(T, val(VAR))
        end if
    else /* VAR is unbound or a compound term */
        return (+inf, VAR)
    end if
```

Figure 4: Variable dereferencing

The value returned by the algorithm is in fact the time to which the caller must roll back to remove any inconsistencies. If the value is infinity, no conflicts were found and the unification succeeded. If the value is less than or equal to the original unification time, then the unification failed—the new bindings would themselves have to be rolled back. Finally, if the value is somewhere between the two extremes, then the caller must roll back to the time given by that value to remove any conflicts.

It should be noted that failure may occur partway through the unification of a group of incoming bindings. Because the unification algorithm alters existing bindings, care must be taken to ensure that the unification can be undone. Thus, any previous bindings must be retained until the whole unification has succeeded (with or without rollback). After this, the space consumed by the previous bindings may be reclaimed.

Because bindings may be altered, backtracking and rollback must also be performed carefully. A binding may have been trailed in some frame, and then had its binding time altered. Such a binding, after being untrailed from the removed frame (but *not* backtracked), must be appended to another trail: that of the stack frame whose virtual time is equal to the binding time. Only when this frame is removed will the binding actually be backtracked. In all other respects, backtracking and rollback are performed just as they are in systems using standard unification—e.g. if a binding with a certain timestamp is withdrawn, all bindings with later timestamps must be rolled back before the withdrawn binding is undone.

4.3.2 Dereferencing

During the course of execution, a variable may become bound to another unbound variable, and then to another, and another .... This gives rise to a reference chain that must be dereferenced when the variable is bound again to a new value (either a compound term, a constant, or another variable). Thus, the first step in the OIU algorithm, as in most unification algorithms, is to dereference both binding terms. (Remote or clause term dereferencing is unnecessary at the top level, but subterms may well require dereferencing.) In standard algorithms, a term is completely dereferenced—to an unbound variable or an integer, atom, or compound term. (For the balance of the paper, constants—integers and atoms—are treated as compound terms with arity zero.)

Under the order-independent algorithm, a term might not be completely dereferenced. Instead, it is dereferenced only until the binding time of the current alias is greater than the current unification time. This ensures that when a reference chain is formed or extended, the bindings of the variables in the chain are strictly increasing in timestamp from the beginning of the chain to the end. In this way, the chain appears to have been built up in timestamp order, and may be backtracked in the "opposite" order without destroying essential binding information.

Pseudocode for the dereferencing algorithm is shown in Figure 4. A (presumed) variable VAR is dereferenced with respect to some virtual time T. If VAR is an unbound variable or a compound term, it cannot be dereferenced, and is simply returned along with a "binding time" of $+\infty$. Otherwise, VAR is bound to some other term, so its binding time, Tb, must be examined. If T exceeds Tb, dereferencing continues with VAR's binding value. Finally, if Tb exceeds T, dereferencing stops, and VAR is returned along with its binding time.

To see how this works, consider the situation depicted in Figure 5. Variable P is bound (via a long reference chain) to the term f(X). The binding P = f(Y) arrives at virtual time 30. When P is dereferenced, the dereferencing algorithm returns the bound variable R and the time 40. The unification algorithm then calls itself recursively to unify the remainder of the reference chain (starting at S) with f(Y) at time 40.

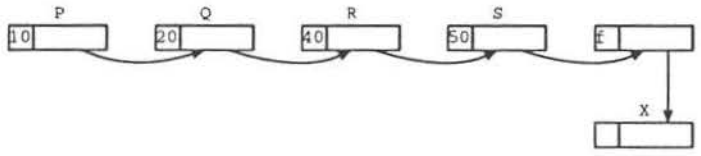

Figure 5: Binding structure before out-of-order unification

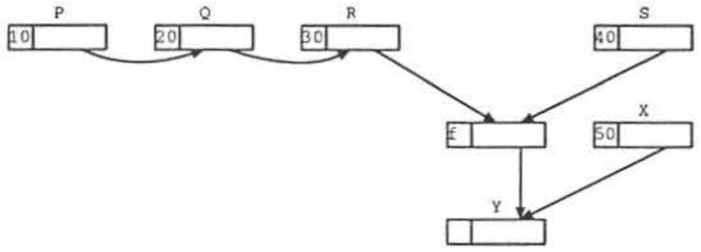

Figure 6: Binding structure after out-of-order unification

S is itself dereferenced, but not very far: the dereferencing algorithm returns S immediately, with binding time 50. Again, the remainder of the reference chain, f(X), is unified with f(Y), this time at time 50. This unification results in the binding of X to Y at time 50. Unwinding the recursion, S has its binding at time 50 to f(X) removed, and is rebound to f(Y) at time 40. R then receives similar treatment, losing its binding to S at time 40, but being bound to f(Y) at time 30. The resulting binding structure is shown in Figure 6; it is identical to the structure that would have resulted from undoing the bindings at times 40 and 50, and then performing unifications at times 30, 40, and 50.

4.3.3 Unification Algorithm in Depth

After dereferencing, unification begins in earnest. Each term may have been dereferenced to one of three types: an unbound variable, a bound variable, or a compound term. This gives rise to nine cases; the pseudocode of Figure 3 handles each of these cases.

The simplest case occurs when both dereferenced variables (LD and RD) are unbound. Either they are identical, or one must be bound to the other (by convention, LD is bound to RD) at time T. In either case, the unification succeeds, returning a rollback time of $+\infty$. When one term is unbound and the other is a compound term (two cases), the unbound variable is simply bound to the compound term with binding time T. Similarly, when one term is unbound and the other bound (also two cases), the unbound variable is bound to the bound variable with binding time T.

The remaining cases deal with unifying various combinations of bound variables and compound terms. The simplest of these comes in unifying two compound terms. No new bindings are made at the top level. If the terms differ in functor or arity, the unification fails, returning rollback time T. Otherwise, the rollback time is computed as the minimum of the return values from each pairwise argument sub-unification, and returned. (Atoms and integers are deemed to have an arity of zero; thus, no sub-unifications are done and the computed "minimum" is $+\infty$.)

The final three cases—two of which unify a bound variable with a compound term, and one of which unifies two bound variables—require one or more bindings to be altered. When a bound variable B is unified with a compound term C at a unification time T (as in the example of Figures 5 and 6), a sub-unification must first be done. C is unified with the remainder of B's reference chain, with a unification time equal to B's binding time. If this sub-unification succeeds, B's previous binding is undone, and replaced with a binding to C at time T. (The previous binding, though compatible, had a greater virtual time, and thus would not have been created had the unifications occurred in virtual time order.) If the sub-unification fails, this modification is not performed. In both cases, the rollback time received from the sub-unification is used as a return value.

Finally, when two bound variables are unified (assuming they are not identical), the remainders of their respective reference chains are unified first. If this succeeds, then one of the bound variables (by convention, LD) must have its old binding undone and be bound to the other (RD) at time T. If the sub-unification fails, no rebinding is done. Again, the return value is given by the rollback time returned by the sub-unification.

It should be noted here that, like most other unification algorithms, the OIU algorithm performs no occurs check. This also applies to the OIB algorithm, described in Section 4.4.

4.3.4 Data Structures

With respect to the data structures employed, the OIU algorithm differs from standard unification algorithms in three ways. First, a timestamp must be associated with each variable binding, so that binding conflicts can be resolved. Second, frames may be inserted at arbitrary points in the execution history (normally, the stack). Third, the backtrack trail associated with each stack frame must be extensible.

If the timestamp information is simple enough (say, representable by a single integer), it may be kept directly as part of each binding. This provides fast access to binding times during unification. If a timestamp consumes several machine words, it may be more reasonable (in terms of space overhead) to have each binding contain a pointer to the frame in which it was bound, and to store the timestamps only in the stack frames. Accessing a binding time on unification then requires a pointer dereference.

Because of the requirement for arbitrary insertion, a stack is no longer practical for maintaining the execution history. Inserting a frame at some time `t` would require that all frames with timestamps greater than `t` be popped, the new frame pushed, and the popped frames pushed back on the stack. More practical would be a linked list of frames, allowing arbitrary insertions (and deletions). Since most operations on this frame list remain "pushes" and "pops" from the end of the list, and in the interest of terminological nonproliferation, we shall continue to refer to it as a "stack," and trust that no confusion will result.

Finally, when a frame with some timestamp `T` is rolled back or backtracked, any variable binding in its trail with a binding time of less than `T` must not be undone. Rather, it must be trailed again, in the frame corresponding to its binding time. That is, if a variable is originally bound at virtual time `T` and later bound at time `T - k`, then when the frame with timestamp `T` is removed due to rollback or backtracking, the variable must be trailed in the frame with timestamp `T - k`.

4.3.5 Overhead

The OIU algorithm incurs very little overhead over that of the standard Prolog unification algorithm. With respect to space consumption, the only additional costs are in a timestamp for each binding, and in pointers to maintain the frame list. In forward execution, the only time cost is in setting the binding time of each binding produced. In processing incoming bindings, the comparison of binding times adds a small constant overhead for each bound variable unified with a compound term or another bound variable.

Otherwise, the only additional work is in adjusting bindings and binding times. Such adjustments occur when an incoming binding takes precedence over an existing binding, whether that binding is compatible or not. Unifying a variable with some binding may require several adjustments if that variable is already bound to a compound term or if it is part of a variable reference chain. Finally, unifications involving unbound variables incur neither the timestamp comparison cost nor the binding adjustment cost.

4.4 Order-Independent Backtracking

Although the OIU algorithm optimizes the addition of bindings to the trail in arbitrary order, it does nothing to mitigate the effects of *removing* such bindings. If a binding at some time T is withdrawn, all work done at later virtual times must be rolled back, and later redone. As an alternative, we present the OIB algorithm, which optimizes rollbacks due to binding withdrawal in addition to those due to binding arrival.

This further optimization comes at a price, however. In the case of the OIU algorithm, bindings can be added in any order, and the resulting state will be identical to the state that would be reached by unifying these bindings in virtual time order. Thus, only this "standard" state need be maintained. Because it allows bindings to be removed arbitrarily, the OIB algorithm must be able to go from a given state to a previous state. In fact, all *possible* previous states must be accessible, not just actual previous states.

```
unify(time T, term L, term R) returning (rollback time RVAL, set DEP)

    if L = R                              /* terms are identical, all done */
        return (+inf, ())
    else if L is a bound variable         /* L is bound, try to deref */
        LB := binding value of L
        TL := binding time of L
        if T >= TL                        /* new binding is later, so deref */
            return deref(T, L, LB, R)
        else if R is a bound variable     /* R is bound, try to deref it */
            RB := binding value of R
            TR := binding time of R
            if T >= TR                    /* new binding is later, so deref */
                return deref(T, R, L, RB)
            else if TL >= TR              /* L bound later than R, so rebind L */
                return rebind(T, L, R, TL, LB)
            else                          /* R bound later than L, so rebind R */
                return rebind(T, R, L, TR, RB)
            end if
        else                              /* R is a compound term, so rebind L */
            return rebind(T, L, R, TL, LB)
        end if
    else if R is a bound variable         /* R is bound, try to deref */
        RB := binding value of R
        TR := binding time of R
        if T >= TR                        /* new binding is later, so deref */
            return deref(T, R, L, RB)
        else                              /* new binding is earlier, so rebind */
            return rebind(T, R, L, TR, RB)
        end if
    else if L is an unbound variable      /* terms are distinct, L is unbound */
        L := R @T                         /* => bind L */
        return (+inf, ((T,L)))
    else if R is an unbound variable      /* terms are distinct, R is unbound */
        R := L @T                         /* => bind R */
        return (+inf, ((T,R)))
    else if functors/arities match        /* both L and R are compound terms */
        RVAL := +inf                      /* init return value */
        for each argument pair (Li, Ri) /* unify arguments pairwise */
            (RV, D) := unify(T, Li, Ri)
            if RV = T                     /* on failure, quit right away */
                return (RV, ())
            end if
            RVAL := min(RVAL, RV)         /* get lowest rollback time */
            DEP := union(DEP, D)          /* keep track of dependent bindings */
        end for
        return (RVAL, DEP)
    else                                  /* functors or arities don't match */
        return (T, ())                    /* return failure */
    end if
```

Figure 7: Main unification routine for order-independent backtracking

To accomplish this, we keep redundant binding information. Each bound variable has associated with it an *active* binding and a list of *inactive* bindings. The active binding for a variable is the lowest-timestamped binding for that variable, while the inactive bindings are all those with greater timestamps. The set of all active bindings corresponds exactly to the state computed by the OIU algorithm; the inactive bindings constitute redundant information.

Bindings must be convertible from active to inactive and back. Active bindings are explicitly recorded as substitutions, like normal Prolog bindings. Inactive bindings are recorded as pending unifications. When a new binding arrives whose timestamp is lower than any other binding for the same variable, it must become the new active binding, and the existing active binding must be converted to an inactive binding. When an active binding is removed, the lowest-timestamped inactive binding (if it exists) for the same variable must become the new active binding. These actions replace respectively the binding-time adjustment and rollback done by the OIU algorithm in the same situations.

Pseudocode for the OIB algorithm is given in Figures 7 through 10. The algorithm consists of four main routines: `unify` (Figure 7), `deref` (Figure 8), `rebind` (Figure 9), and `undo` (Figure 10). These routines

```
deref(time T, term L, term R, term B) returning (time RV, set D)

    (RV, D) := unify(T, R, B)               /* unify with the deferenced term */
    if RV > T                               /* successful unification */
        DEP(L) := union(DEP(L), D)          /* add in new dependent bindings */
        insert(INACT(L), (T, L, R))         /* new binding is inactive at top level */
        return (RV, ((T,L)))                /* return success and inactive binding */
    else
        undo(D)                             /* unification failed, undo */
        return (RV, ())                     /* return failure */
    end if
```

Figure 8: Dereferencing in OIB unification

```
rebind(time T, term L, term R, time TB, term B) returning (time RV, set D)

    unbind(L)                               /* remove old binding first */
    L := R @T                               /* install new binding */
    (RV, D) := unify(TB, B, R)              /* unify state further with new binding */
    DEP(L) := union(DEP(L), D)              /* add in new dependent bindings */
    insert(INACT(L), (T, L, B))             /* old binding becomes inactive */
    return (RV, ((T,L)))                    /* return success and new binding */
```

Figure 9: Variable rebinding in OIB unification

```
undo(dependent set D)

    for all (T,V) in D
        undo(DEP(V))                        /* undo all subsidiary bindings */
        unbind(T,V)                         /* remove a single binding */
    end for
```

Figure 10: Removal of dependent bindings in OIB unification

function as follows:

`unify(T,L,R)` unifies terms L and R at virtual time T. It decides what to do with each term depending on its type and (if bound) its timestamp.

`deref(T,L,R,B)` dereferences variable L by one step to term B. It then calls `unify` to unify term R with B.

`rebind(T,L,R,TB,B)` removes a binding B for variable L at virtual time TB, and replaces it with a new binding, R, with a lower timestamp, T. The old binding is then unified with the new via a call to `unify`, to ensure their compatibility.

`undo(D)` backtracks a set of bindings D, all of which depend on the existence of some other binding (and which must therefore be removed on the withdrawal of that binding).

4.4.1 Prolog Interface

The interface to a Prolog caller is very similar to that for the previous algorithm. Like the OIU algorithm, the OIB algorithm is also called in normal forward execution, and in processing incoming bindings. Both receive as parameters a unification time and two terms to unify. Both compute and return to the caller a rollback time, RVAL, indicating the success or failure of the unification.

Unlike the previous algorithm, however, the OIB algorithm is also called when an active binding is withdrawn. When this occurs, the lowest-timestamped inactive binding for the same variable becomes the new active binding—not via rollback, as in the OIU algorithm, but rather by undoing the withdrawn binding and then unifying the new active binding with the current state.

Also unlike the OIU algorithm, the OIB algorithm computes and returns a set of dependent bindings DEP. These are bindings which were made as a direct consequence of performing the top-level unification. When the top-level unification is undone, any bindings in the dependency set must be undone as well. (In the OIU algorithm, such dependent sets were unnecessary because any bindings created as a consequence of performing some unification would automatically be rolled back if that unification were undone.)

As in the OIU algorithm, failure may occur partway through a unification. In this case, however, no special care is needed to clean up any partial unifications, since the partially-computed set of dependent bindings, DEP, is also returned on failure. Thus, failure can be treated as binding withdrawal, and any bindings due to the failed unification can be accessed through the returned dependent set and undone.

Finally, when the OIB algorithm is used, the marker algorithm outlined in Section 4.1 must also be used, to ensure that no potential solutions are overlooked.

4.4.2 Dereferencing

As with most unification algorithms, the OIB algorithm requires some way of dereferencing variable reference chains, so that the last variable in a chain is the one bound, and accesses to any variable in the chain will return the proper binding. Most unification algorithms perform this dereferencing step explicitly, before performing the actual unification.

Under the OIB scheme, dereferencing is integrated with the unification process itself. This is because the OIB algorithm must record an inactive binding for each dereferencing step, rather than just searching down the chain. A simple example illustrates this behaviour.

Suppose that at virtual time 30 we perform the binding `P = Q`, at time 40 the binding `X = Y`, and at time 50 the binding `P = 7`. Under the OIU and standard schemes, the last unification would deference P to Q and cause Q to be bound to 7. The information that the last binding was originally made to P can be discarded safely, since the last binding will always be removed before the first.

Under the OIB algorithm, this is no longer the case. The idea is that the binding `P = Q` can be removed without having to undo all later bindings, only to redo all those bindings to restore the proper state. Instead, only the withdrawn binding—and each of the bindings that depend on it—is undone, and the first available inactive binding is recorded in its stead. Thus `P = 7` must be recorded as an inactive binding for P when it is first encountered, as well as causing `Q = 7` to become the active binding for Q. If `P = Q` is later removed, the

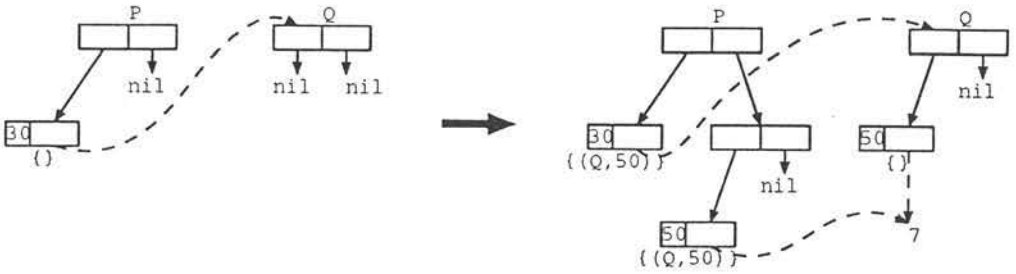

Figure 11: Calculation of dependent sets

binding for Q disappears, and P = 7 becomes the active binding for P, without rolling back and disturbing the binding X = Y.

In other respects, dereferencing behaviour is similar to that of the OIU algorithm. In particular, a reference chain may not be dereferenced completely: dereferencing stops when a binding is encountered in the chain whose binding time is greater than the unification time. Thus, a "fully-dereferenced" variable may still be bound.

4.4.3 Dependent Sets

Associated with each binding for a variable is a dependent set, which is initially empty. As a result of unifying each previous binding for the variable with the new binding, this dependent set is augmented to include subsidiary bindings that are directly due to the new binding—thus, if the new binding is removed, so will all the subsidiary bindings. If the new binding is to a nonground term, such as another variable or an incomplete compound term, the dependent set will also be augmented each time a dereferenced or subterm variable is bound due to a unification that references the nonground binding.

Consider the following execution fragment, illustrated in Figure 11 and based on the example of the previous section. Solid arrows in the diagrams denote pointers in the binding structure for each variable, while dashed arrows denote bindings. The left-hand diagram illustrates the initial state: at virtual time 30, P is bound to Q. Then, at time 50, P is bound to 7. As a result, Q is also bound to 7 at time 50, resulting in the state shown in the right-hand diagram.

This binding of Q is dependent on both bindings of P, and if either were withdrawn, it must be withdrawn itself. That is, if either of P = Q or P = 7 is undone, the binding Q = 7 must be undone. Thus, the binding for Q is put in the dependent sets of P's bindings at times 30 and 50. The dependent sets are used in backtracking, so that when a binding is undone, any bindings that depend on it—exactly the bindings in the dependent set—are also undone.

4.4.4 Unification Algorithm in Depth

The main unification routine is structured much like that for the OIU algorithm. The terms L and R may each be either an unbound variable, a bound variable, or a compound term. The algorithm thus decomposes into several cases, each of which returns two values: a rollback time as in the OIU algorithm, and a set of dependent bindings.

The simplest case occurs when L and R are identical. The unification succeeds immediately, with a rollback time of $+\infty$ and an empty dependent set. Also simple are the cases in which either or both of L and R are unbound. If L is unbound, it is bound to R at time T returning a rollback time of $+\infty$ and a dependent set consisting of the binding itself. Otherwise, if R is unbound, it is treated in a similar manner. Because dereferencing is integrated into the algorithm, both L and R must be checked first to see whether either is bound. Otherwise an unbound variable could be bound to some undereferenced bound variable, creating a reference chain that violates the increasing-timestamp condition described in Section 4.3.2.

The remaining cases are more complex. If either or both of L and R are bound variables, some dereferencing may be necessary. As with the OIU algorithm, dereferencing is required if a variable's binding time (TL or TR in the pseudocode) is less than the unification time T. If L is a bound variable, it is dereferenced as much as possible; then the same is done for R.

When a variable has been dereferenced as much as possible and the term finally reached is still a bound variable (*i.e.* the binding time of that final variable is greater than the unification time), then the final variable must be *rebound* to the new value. When this is done, the old value must be unified with the new value. If this unification succeeds, the old value is retained as an inactive binding (otherwise, it is simply discarded). When both dereferenced terms are still bound, one must be chosen to be rebound. This is done on the basis of binding age: the term with the greater binding time of the two is the one that is rebound.

Finally, when both terms are compound terms, functors and arities are compared. On success, corresponding arguments from each term are unified, with return values and dependent bindings being accumulated in the process.

As well, several primitives are used in the pseudocode. `Union(S1,S2)` gives the union of sets `S1` and `S2`; `insert(L,B)` inserts an inactive binding `B` into a list `L` of such bindings (sorted in timestamp order); `unbind(T,V)` backtracks the binding for `V` at time `T`. `DEP(L)` refers to the set of bindings dependent on `L`, while `INACT(L)` refers to the list of inactive bindings for `L`.

4.4.5 Data Structures

Like the order-independent algorithm for unification alone, the OIB algorithm needs a timestamp associated with each binding in order to determine the success or failure of a unification and to determine a rollback time if binding conflicts are detected. The "stack" must again be a list, allowing the addition and deletion of internal frames.

As well, each binding must maintain a list of other bindings that depend on it, so that these dependent bindings can be backtracked when the original binding is backtracked. A binding may depend on more than one other binding, particularly when variables are bound to other variables (aliasing) or to incomplete compound terms. In the pseudocode, the set of dependent bindings dependent on the binding of `R` is known as `DEP(R)`.

To maintain information about inactive bindings, each variable has associated with it a timestamp-ordered list, referred to in the pseudocode as `INACT(L)` for some variable `L`. Each element of the list constitutes an inactive binding.

4.4.6 Overhead

Overhead for the OIB algorithm is somewhat higher than for other unification algorithms, including the OIU algorithm. This is mostly due to the cost of installing and removing inactive bindings and dependent sets. The remaining overheads are identical to those faced by the OIU algorithm. The execution "stack" is actually a list, so that the arbitrary insertion and removal of stack frames is possible. To determine binding conflicts, each binding must include a timestamp; this timestamp must be examined each time the binding is accessed.

In any case, the cost of doing an OIB unification will be linear in the cost of a standard unification. Also, the cost of doing a binding removal can be arbitrarily lower than the corresponding backtrack and re-unification of standard Prolog. In the worst case, the OIB cost will be linear in the standard cost (presumably with a somewhat lower constant).

In cases where all bindings occur in timestamp order, OIB unification will behave almost identically to standard unification, although dependent sets will be created and variables will have multiple bindings recorded. Even this cost can be ameliorated in the usual case. Consider a variable bound as a result of a remote unification from another processor. Any local attempt to match this variable with a clause head will cause an inactive binding to be recorded. However, this local binding can only be removed by backtracking, so later-timestamped bindings—both local and remote—need not be recorded. This optimization means that the most common case will be a variable bound with a single (active) binding marked as being local. Next most likely is a variable with an active binding caused by a remote unification plus one local (inactive) binding. More complex binding situations require relatively unlikely timing coincidences between remote processors.

5 Example

To illustrate the Prolog algorithm further, we present an example execution that enters each possible state. In this example (and for the remainder of the paper), we assume that the OIB unification algorithm and the marker

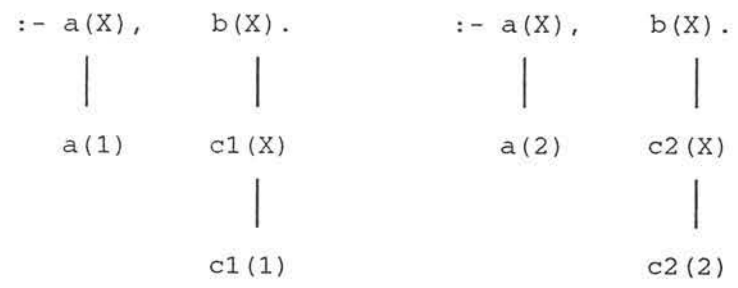

Figure 12: Proof trees for example

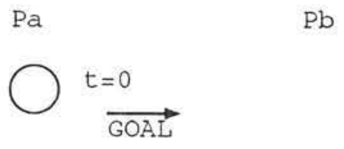

Figure 13: Initialization

algorithm are used. The example has been carefully contrived to exhibit as much different message passing behaviour as possible with a very simple program. In fact, the example given would most likely reach first solution in three messages rather than the eight shown below.

Consider the query

```
:- a(X),b(X)@Pb
```

run with respect to the program

```
a(1).        b(X) :- c2(X).    c2(2).
a(2).        b(X) :- c1(X).    c1(1).
```

The goals in this query indicate that `a(X)` is to be run locally by the top-level process and `b(X)` to be run in parallel on another processor. The top-level process will be denoted by P_a, while the cooperating solver process will be denoted by P_b. The query has two solutions, `X = 1` and `X = 2`, with proof trees as shown in Figure 12.

Figure 13 depicts process initialization. P_a begins by scanning the top-level query. It puts the goal `a(X)` on its own goal list, and allocates the goal `b(X)` to P_b by sending P_b a GOAL message at virtual time 0.[2] It then begins forward execution. P_b receives the GOAL message, puts `b(X)` on its own goal list, and begins forward execution itself.

P_b selects goal `b(X)` and clause `b(X) :- c2(X).`, and unifies them, all at virtual time 1. It then checks for incoming messages. Finding none, it continues executing forward, selecting goal `c2(X)` at time 6, unifying it with clause `c2(2).`. This unification produces a binding for the shared variable `X`; thus P_b sends a BIND message binding `X` to the value 2 to P_a with a timestamp of 6.

Meanwhile, P_a selects goal `a(X)` and clause `a(1).`, and unifies them at time 3. This unification also produces a binding for `X`, namely `X = 1`. P_a therefore sends a BIND to P_b at time 3. Note that at this point, two incompatible bindings have been generated (see Figure 14). This conflict is resolved quickly, as both P_a and P_b enter the message processing phase.

P_a receives the BIND at time 6 and discovers that the binding `X = 2` conflicts with its own binding of `X = 1`. Seeing that its own binding antedates the incoming message, P_a discards the message. (It could require P_b to backtrack by sending it a FAIL, but this is unnecessary since P_b will handle the conflict itself through rollback.) P_a, having resumed forward execution, finds that it has no further goals to execute. It becomes inactive, setting its virtual time to $+\infty$ and waiting for incoming messages.

[2]Times used in this example have been chosen arbitrarily, albeit increasing up the stack.

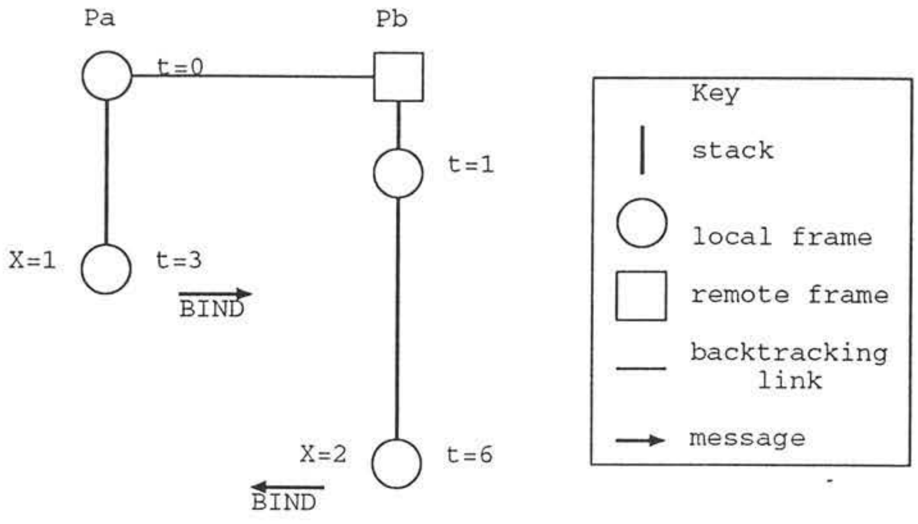

Figure 14: Forward execution

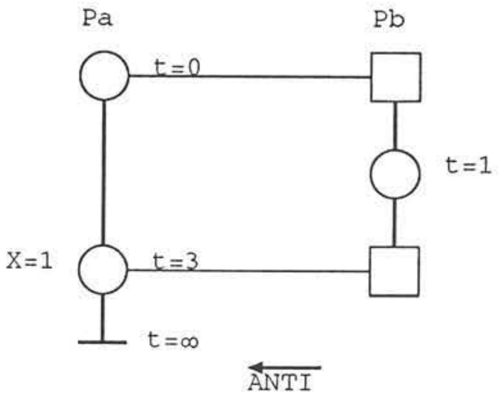

Figure 15: Message receipt and rollback

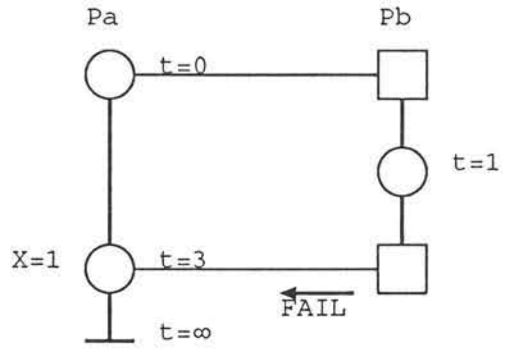

Figure 16: Forcing remote backtracking

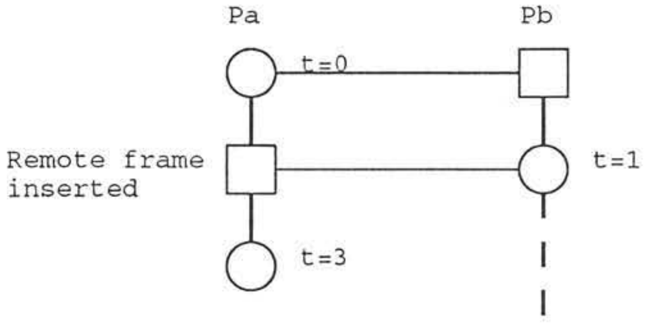

Figure 17: Remote backtracking

P_b receives the BIND at time 3 and finds that the binding within conflicts with its own binding at time 6. Thus, it rolls back to before time 6, undoing its local unification and sending P_a an ANTI-BIND message with timestamp 6. (This message could be optimized away, since every process receiving the BIND at time 6 would also receive that at time 3.) It then accepts the incoming binding, and installs a remote frame at time 3 to contain the binding. No further messages arrive, so it returns to forward execution. This message-handling activity is illustrated in Figure 15.

P_a now receives the ANTI-BIND from P_b but finds no corresponding BIND. P_a thus discards the message and returns to inactive status.

P_b selects goal `c2(1)` (note the variable substitution) and clause `c2(2).` When unification fails, it begins local backtracking, trying to find another clause to match `c2(1)`. Finding none, it backtracks a step further and encounters the remote frame at time 3. Remote backtracking must now occur at P_a: P_b sends a FAIL to P_a so that the binding `X = 1` will be retracted. (Included in this message is information about the sibling frame in P_b's stack—that is, the local frame at time 1. Remote backtracking must be able to return eventually to P_b, since its own bindings could have caused the original failure.) This situation is shown in Figure 16. P_b removes the remote frame (and the binding for `X`) and resumes forward execution, executing goal `c2(X)` again.

When P_a receives the FAIL from P_b, it returns to active status, rolling back to time 3. Before backtracking from that point, it inserts a remote frame in its stack at time 1 to direct remote backtracking back to P_b later in backtracking (see Figure 17). (This remote frame will be used later when the second global solution is sought.) It then backtracks, sending out an ANTI-BIND to annihilate its original BIND and removing its own binding of `X = 1`. Next, it retries goal `a(X)`, this time with clause `a(2).`, and unifies them to produce the binding `X = 2`. Once again, it sends a BIND to P_b to convey the binding information. With this forward execution step complete, P_a looks for incoming messages. Finding none, and finding no more goals to execute, it returns to the inactive state.

Meanwhile, P_b has selected clause `c2(2).` and made the binding `X = 2` again. Again, it sends out a binding at time 6. Figure 18 shows the status of the system at this point, with two messages in transit from P_a to

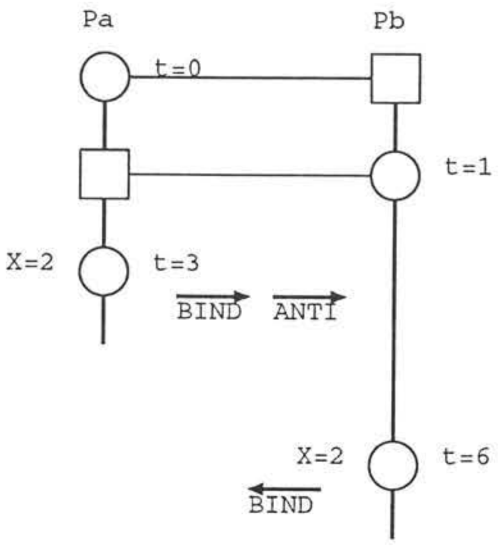

Figure 18: Return to forward execution, with messages exchanged

P_b and one back from P_b to P_a. P_a becomes active briefly to accept the BIND, and finds the binding within to be compatible. Since its local binding has higher precedence (timestamp 3) than the remote binding, the remote binding is ignored. Since no new bindings were made (not even redundant bindings), P_a does not add a remote frame to its stack. Instead, it ignores the message and returns to inactivity.

P_b processes its own incoming messages, finding an ANTI-BIND and a BIND at time 3. It finds no frame for the incoming ANTI-BIND (since it destroyed that frame when sending the FAIL), and ignores it. In processing the incoming BIND, it finds the binding within to be compatible with its own, but with a higher precedence. Thus, it installs a remote frame at time 3 *without rolling back* and trails the redundant binding. P_b then resumes forward execution, but finds no more work to do. It too becomes inactive, as shown in Figure 19. Since both processes are now inactive and no messages are pending, the system has terminated, in this case finding the solution `X = 2`.

When another solution is requested, the point of introducing a new remote frame on receipt of a FAIL becomes evident. P_a, as the top-level process, begins backtracking. P_a backtracks to time 3 and withdraws its binding of `X = 2`, sending an ANTI-BIND to P_b. P_b receives the ANTI-BIND from P_a at time 3 and removes the remote frame at that time. In its local frame at time 6, it sets the marker to time 3. (As it turns out, this marker is never used, but more complex executions would require it to guarantee completeness.) After it sets the marker, P_b returns to inactivity.

P_a, still backtracking locally, finds no further clauses to match goal `a(X)`. It backtracks a step further and encounters the remote frame at time 1, installed when it previously received the FAIL from P_b. P_a sends the failure back to P_b at time 1, forcing remote backtracking, and restarts forward execution. (The frame at time 0 is included in the FAIL as context.) P_a tries goal `a(X)` anew, selects clause `a(1).`, and sends a BIND at time 3. It then finds no messages and no further goals, and becomes inactive.

P_b receives the FAIL from P_a, wakes up, and rolls back to time 1 (in the process sending out an ANTI-BIND at time 6). The marker disappears as rollback continues to before the marker's time. P_b then begins backtracking, retrying `b(X)` with clause `b(X) :- c1(X)..` It checks for incoming messages, finds the BIND from P_a, and installs a remote frame at time 3 for the binding `X = 1`. Finally, it tries goal `c1(1)`, unifies with clause `c1(1).`, and sends out its own (compatible) BIND at time 6. It then becomes inactive. P_a wakes up briefly to accept the ANTI-BIND ands BIND messages, ignores them both, and returns to inactivity. A second solution has been found; the state of each process is illustrated in Figure 20.

If yet another solution is requested, the system finally concludes that no other solutions exist. This is determined after several messages are sent: an ANTI-BIND, and a BIND with a new value for `X` from P_a, next an ANTI-BIND and a FAIL from P_b, and then an ANTI-BIND followed by a successive FAIL from P_a. P_b sends a final FAIL to P_a at time 0, indicating that P_b's goal has failed completely.

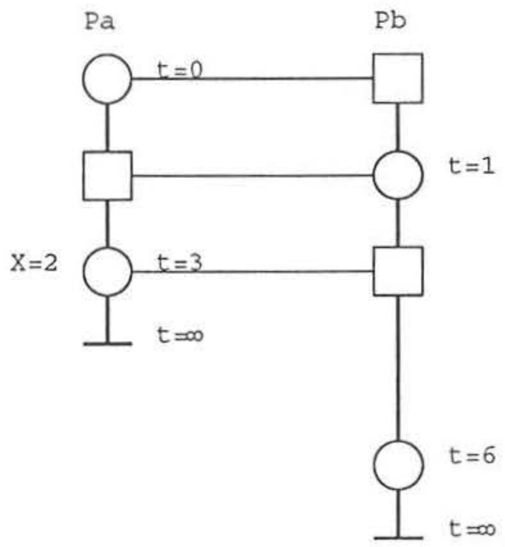

Figure 19: Termination with solution

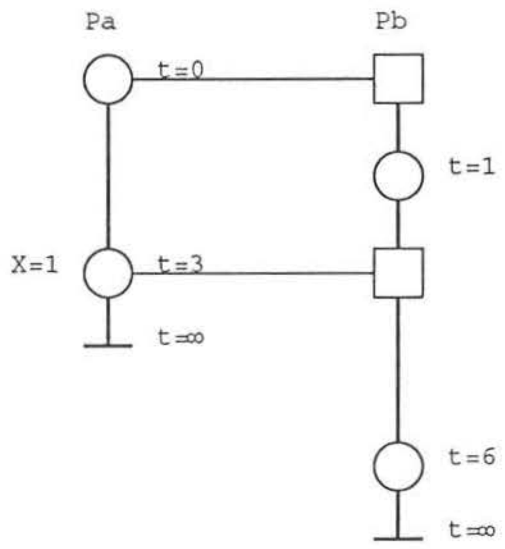

Figure 20: Termination with second solution

It may seem that this example is much to do about nothing. After all, eight interprocess messages have been used to reach the first solution of a problem that requires only three unifications and one shallow backtrack on a sequential system. However, the situation which is illustrated—the same variable being given incompatible bindings on different machines—should be relatively rare. If either P_a or P_b were to bind X early enough that the binding reached the other before it did its own binding, the first solution would be reached in only three interprocess messages (a GOAL and two BIND messages). The faster the underlying message passing system, the smaller the window in which such a conflicting binding can occur.

6 Detailed Algorithm

The example above has illustrated most aspects of the optimized algorithm using the OIB unification algorithm. In each of the sections that follow, pseudocode is given for one of the individual execution phases, together with a detailed description of that phase.

6.1 Pending State

```
wait for GOAL message to arrive
set LVT = timestamp of message
set up stack with remote frame at LVT
put goal from message on goal_list
continue with goal execution
```

A process is in the pending state when it has not yet received a goal to execute. The only way to leave this state is through the receipt of a GOAL message, which contains a goal for the process to solve.

When a GOAL message arrives, the receiving process begins by setting up its stack with a remote frame for the GOAL message—thus, failure of the goal results in a FAIL message being sent back to the goal's originator. LVT is set to the timestamp of the message.

6.2 Forward Execution

Goal Execution

```
if goals exist on goal_list
    select next goal to be executed
    increment LVT
    create goal stack frame
    continue with clause execution
else
    enter terminated state
end if
```

Clause Execution

```
select clause to unify with goal
create clause stack frame
increment ID counter and use value as unique ID for frame
unify selected goal with selected clause
if unification succeeds then
    find all shared variables bound by the unification
    send one BIND message to each process sharing a variable
      bound by the unification
    for each subgoal in current clause
        if subgoal is remote
```

```
                send a GOAL message with LVT as timestamp
            else
                put goal on goal_list
            end if
        end for
        continue with message processing
    else
        continue with clause failure
    end if
```

Forward execution is straightforward. Following from the goal frame/clause frame separation, it is divided into goal execution and clause execution components. Goal execution amounts to selecting the next goal to execute, incrementing the local virtual time, and creating a goal frame for that time containing the necessary information about that goal.

Clause execution begins with selecting a clause and creating a clause frame. The frame takes its unique ID from the value of a counter that is incremented each time a clause frame is created. Next, the current goal is unified with the head of the current clause. If the unification fails, local backtracking begins with clause failure. Otherwise, all variables that were bound and are shared with other processes are identified. One BIND message is sent to each process sharing one or more such shared variables. Subgoals of the clause may be sequential or parallel. Sequential goals are added to the local list of goals to be executed; parallel goals are scheduled[3] to be run by some other process.

Unification is done according to the OIB unification algorithm described in Section 4.4. For local execution, the effect is similar to that of standard unification: the attempt either succeeds with new bindings, or it fails.

6.3 Message Processing

Top-level Processing

```
while incoming messages remain in IQ
    select the next message from IQ
    if message is a BIND
        perform BIND processing
    else if message is an ANTI-BIND
        perform ANTI-BIND processing
    else if message is an ANTI-GOAL
        roll back entire stack
        return to pending state
    else /* message is a FAIL */
        perform FAIL processing
    end if
end while
```

BIND Processing

```
create remote frame to hold bindings
unification_time = message timestamp
faliure_time = +inf
for each binding pair (V,B) in message
    failure_time = min(failure_time, unify(unification_time, V, B))
    if failure_time <= unification_time /* unification fails */
        ignore message /* unif. cannot succeed, even after rollback */
```

[3]Scheduling may be done in many ways, from hard-wired goal-to-process allocation through full-blown load-balancing scheduling.

```
            undo unifications already done for message
            return to top-level processing
        end if
    end for
    if unification_time < failure_time < +inf /* partial rollback required */
        roll back to failure_time
    end if
    return to top-level processing
```

ANTI-BIND Processing

```
    if remote frame referred to in message exists
        end_time = start_time = min(marker_time of frame, time of ANTI-BIND)
        for each variable binding in frame
            if binding is first (earliest) for its variable
                if next binding for that variable exists
                    start_time = max(start_time, time of next binding)
                else
                    start_time = time at top of stack
                end if
            end if
            undo variable binding
        end for
        if start_time > end_time     /* some frame needs marker set */
            marker_time of frame at start_time = end_time
        end if
    else /* remote frame doesn't exist */
        ignore message
    end if
    return to top-level processing
```

FAIL Processing

```
    if frame referred to in message exists
        if frame is local
            roll back all frames later than FAILed frame
            continue with remote backtracking
        else /* frame is remote */
            install context frame in own stack
            send FAIL message to originator of remote frame,
              including sibling frame as context
            perform ANTI-BIND processing on FAILed frame
            return to top-level processing
        end if
    else /* frame doesn't exist */
        ignore message
        return to top-level processing
    end if
```

Messages may arrive at any phase of the execution, but they are not processed immediately. Message processing is performed after each successful unification (and *only* then—accepting messages at other times can cause incorrect execution). Messages that have arrived since the last message processing phase are processed in timestamp order to minimize rollback.

If the message is a BIND, the values it contains are unified with the local copies of the shared variables bound in the message. If the unification succeeds without conflicts occurring, nothing more need be done. If it fails, then some higher-precedence local binding conflicts with an incoming binding, and the message is rejected. If a conflict with a lower-precedence local binding is detected, a partial rollback to the time of the local binding is necessary. (If more than one local binding must be undone, the process must roll back to the time of the earliest such binding.) The time to which execution must be rolled back is given by the failure time, computed by the OIB unification algorithm described in Section 4.

If the message is an ANTI-BIND, the remote frame created for its corresponding BIND must be removed. When no such frame exists, the ANTI-BIND is ignored—either the original BIND was ignored, or the frame has already been backtracked. If the frame does exist, it is removed from the stack and all variable bindings due to it are undone. If the remote frame's marker is set, the time of that marker must be taken into account when calculating the new marker time (the virtual time at which the marker may be cleared). If any variable becomes completely unbound as a result of the ANTI-BIND, the frame at the top of the stack must be marked with the marker time given by the minimum of the marker time of the old frame and with the virtual time of the removed frame.

If the message is an ANTI-GOAL, the entire stack is rolled back, and the process becomes inactive, returning to the pending state. (Note that the message can never be a GOAL, given the single-goal assumption of the execution model given in Section 3.4). Finally, if the message is a FAIL, the frame it refers to must be backtracked. If this frame does not exist, the FAIL is ignored. If the frame is remote, the context of the FAIL is installed in the stack, the remote frame is removed, and a FAIL is sent to the originator of the remote frame. Otherwise, all frames later than the FAILed frame are rolled back, remote backtracking begins, and no further messages are processed until after the next successful unification.

6.4 Rollback

```
end_time = LVT
while LVT > rollback_time
    send any ANTI-BINDs stored in current frame
    undo bindings associated with frame
    for each subgoal of clause associated with frame
        if subgoal is remote
            send ANTI-GOAL message to process solving subgoal
        else
            remove subgoal from goal_list
        end if
    end for
    if current goal frame's marker_time < rollback_time
        end_time = min(end_time, marker_time)
    end if
    de-allocate goal and clause frames
    set LVT = time of sibling frame
end while
if end_time < LVT
    frame's marker_time = min(marker_time, end_time)
end if
```

Rollback is done in order to handle two types of incoming messages: a BIND whose acceptance would result in binding conflicts without the removal of "later" (lower-precedence) local bindings, and a FAIL directed at a local stack frame. In the first case, the stack is rolled back far enough to remove any conflict; in the second, rollback is done back to the timestamp of the FAIL. During rollback, marker times must be propagated down the stack so that frames not rolled back but requiring clause retrial will be handled correctly.

When a process rolls back from a virtual time t_2 to an earlier time t_1, all work between these times is undone. Any bindings made after t_1 are backtracked; BIND messages sent out after t_1 are cancelled by sending out their corresponding ANTI-BIND messages. In this, rollback appears much like backtracking. The salient distinction between the two is that rolling back a goal resets the list of clauses with which it may unify, while backtracking causes the current clause to be discarded and the next to be selected. As well, rollback causes all work past a specific time to be undone, while backtracking goes back only far enough to find an untried clause or remote frame.

6.5 Local Backtracking

Clause Failure

```
if marker_time of current frame <= LVT
    ensure all clauses other than the currently-selected one will
      eventually be selected on backtracking
    make rejected clauses available for retrial
    if marker_time <= timestamp of previous frame
        previous frame's marker_time =
          min(previous marker_time, marker_time)
    end if
end if
select next clause
if no alternative clauses for current goal
    replace goal on goal list
    delete goal frame
    set LVT = timestamp of sibling stack frame
    continue with goal failure for sibling stack frame
else
    continue with clause execution for selected clause
end if
```

Goal Failure

```
backtrack all bindings associated with clause frame
if current stack frame is a remote frame
    send FAIL message to originator (include timestamp and
      unique ID of sibling frame as context in stack)
    delete clause frame and parent goal frame
    continue with goal execution
else /* frame is local */
    send one ANTI-BIND message for each outgoing message of current frame
    for each subgoal of clause associated with frame
        if subgoal is remote
            send ANTI-GOAL message to process solving subgoal
        else
            remove subgoal from goal_list
        end if
    end for
    delete clause frame
    continue with clause failure
end if
```

Local backtracking begins as a result of a failed attempt at unification of a goal and a clause head. This is known as *clause failure*; the next available clause for the goal must be tried. If another clause is available, forward execution resumes. Otherwise, *goal failure* results, and the current goal must be backtracked. From this point, the previous clause frame on the stack is backtracked. If the previous frame is local, the clause frame is undone as for rollback and backtracking continues with clause failure. If the previous frame is remote, failure is passed on to the originator of the remote frame via a FAIL message.

In the process of backtracking, marker times are propagated down the stack. If a backtracked frame's marker time is less than its own time, all of its clauses are made available again. If the current LVT becomes less than the marker time, the marker is reset to $+\infty$. Also, if a marker time is propagated down to a frame that already has its own marker time, the lower marker time prevails.

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 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. To provide this context, it is sufficient to include the timestamp and unique ID of the sender's sibling stack frame with the FAIL message. (However, see Section 7.1, which optimizes this process.)

6.6 Remote Backtracking

```
if remote frame with timestamp and unique ID equal
  to those in context of message does not yet exist
    create a remote frame based on context given in FAIL
      (process that sent FAIL, timestamp, frame ID)
    insert in the local stack according to timestamp
end if
continue with goal failure from current frame
```

Remote backtracking begins at a process as a result of receiving a FAIL message from some other process. It is identical to local backtracking except that two preparatory steps must be taken. First, the FAILed process must roll back to the virtual time given in the FAIL. (Details are given in the sections on message receipt and rollback.)

Second, the FAILed process must ensure that it takes the context provided by the FAIL's originator into account. 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 sibling stack frame. (See Figures 16 and 17 for an example of this.) If it encounters this frame during later backtracking, it reacts as it would to any other remote frame. Once these steps have been taken, normal backtracking begins with goal failure of the FAILed frame.

6.7 Solved State

```
wait for some message to arrive
continue with message processing
```

Actions performed in the solved state are straightforward. A solved process simply waits until a message arrives, and resumes execution by processing it.

7 Further Optimizations

Although the most significant optimizations suggested in [Cleary *et al* 1988] have been incorporated into the current algorithm, further optimizations are still possible. FAIL messages may be optimized to send multiple contexts, making it more likely that a later FAIL will be sent directly to the originator of a rejected binding, rather than to some intermediate process. As well, the algorithm is amenable to the application of intelligent backtracking techniques, particularly the scheme proposed in [Mannila & Ukkonen 1986]. Related to this are two "intelligent retrial" techniques.

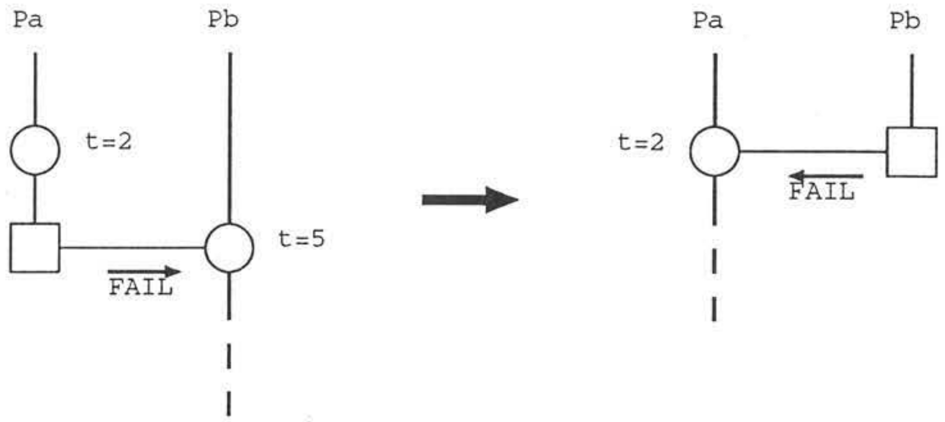

Figure 21: Process P_a initiates remote backtracking, causing P_b to fail. P_b backtracks to the time given in the FAIL's context without finding a new solution, and sends a FAIL back to P_a, which then backtracks again.

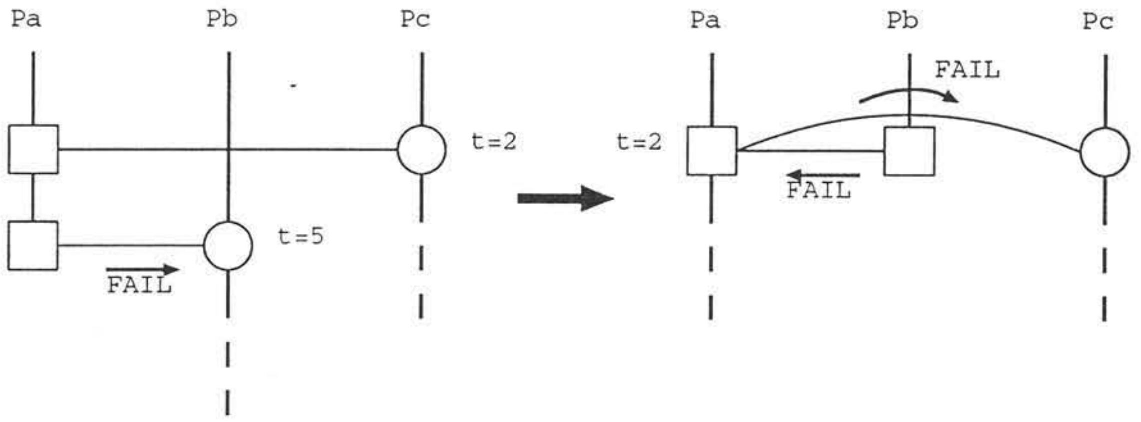

Figure 22: As for two-process backtracking, except that P_a passes failure on to P_c immediately on receiving the FAIL itself.

7.1 FAIL Optimization

The aim of trying to optimize FAIL messages is to reduce the number of FAIL messages sent, and consequently to minimize wasted message processing and stack examination. 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 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 21), 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 22. In this case, when a FAIL is directed back at the original sender, that process checks back through its stack only to discover that the FAILed frame is due to a third process. It must then install a new context frame in its stack, remove the FAILed frame, and send out its own FAIL message to the third process before returning to forward execution. This is wasteful: an extra FAIL message is sent, and processing is done to remove a remote frame that would be removed anyway on transmission of the appropriate ANTI-BIND, which happens as soon as the FAIL reaches its final destination.

An attractive idea is to direct failure 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 23. This saves process P_a from having to roll back, and

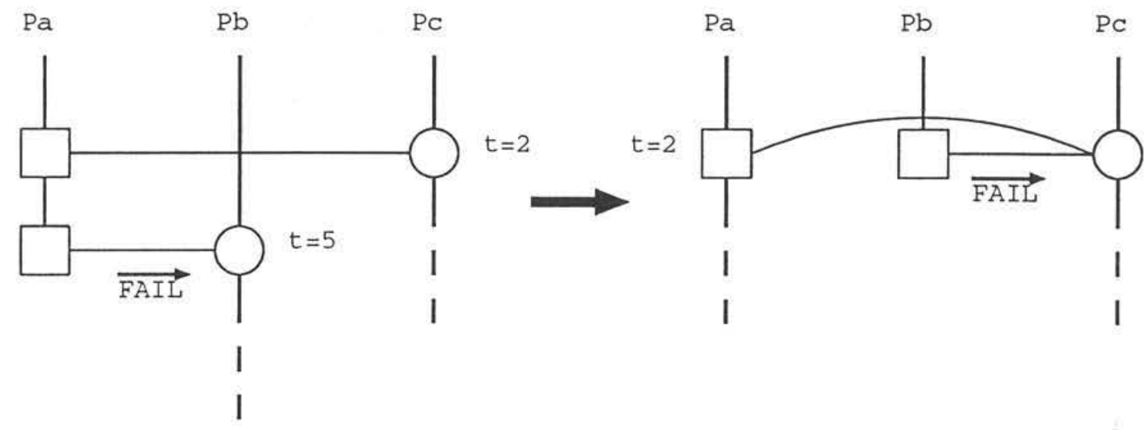

Figure 23: Rather than sending a FAIL back through P_a, P_b can send it directly to P_c, and avoid making P_a receive and process a FAIL.

results in one less FAIL message being sent out.

Though attractive, this optimization must be implemented carefully. It is important that the context provided in the FAIL message always make it possible to direct failure back to the process that sent the FAIL. If this is not done, situations can arise that cause possible solutions to be missed and thus incorrect execution [Olthof 1991].

The solution to this problem is to use a multi-place context, such that the i^{th} component of the context corresponds to the $(i+1)^{th}$ last frame in the sender's stack, be that frame local or remote. The exception is the last component of the stack, which *must* direct failure back to the sender, whether the corresponding frame on the sender's stack is local or remote.

Setting the number of contexts in a FAIL message to some small number seems appropriate. This might be determined by a natural message size in the implementation. We expect that sending three or four contexts will cover almost all available optimization.

7.2 Intelligent Backtracking

Through the use of FAIL messages, the distributed algorithm already exhibits a simple form of intelligent backtracking. When a process receives a FAIL for one of its bindings, it immediately rolls back to the virtual time of that binding, rather than uselessly retrying later goals. The cause of failure is obvious and localized to a specific group of bindings, and no amount of struggling with later bindings will remove the failure.

The challenge is to make this sort of backtracking possible for local execution as well. The intelligent backtracking algorithm proposed in [Mannila & Ukkonen 1986] seems ideal for this purpose. Like our algorithm, it keeps track of binding timestamps. Though their reason for maintaining such timestamps differs from ours, the method is so similar as to be nearly identical.

The central idea of their algorithm is that when all clauses for a goal have failed, backtracking should proceed to the latest-timestamped unification whose bindings could have contributed to the failure of any of the clauses. As each clause is tried and rejected, the bindings that caused it to be rejected—or rather their timestamps—are recorded. Adding such behaviour to our algorithm is straightforward and incurs little additional space overhead.

8 Extraction of Available Parallelism

While it is difficult to make accurate predictions about the execution time of a complex system such as this it is very desirable that some comparison be made with other similar parallel algorithms. We do this in this section by examining the amount of parallelism that can be extracted by the TimeWarp algorithm.

Estimating the parallelism available is essentially the same as predicting performance in the limit when overheads such as order independent unification, message passing and rollback are free. An example of such an analysis can be found in [Hermenegildo & Rossi 1990] where it is shown that a "Non-Strict Independent AND-Parallel" system will never run slower than a sequential execution. In the first sub-section below we show informally, that basic TimeWarp will never run slower than a sequential SLD execution.

In the second sub-section we show that full TimeWarp is able to extract as much parallelism as any of a wide class of *conservative* AND-parallel algorithms. (All proposed AND-parallel algorithms that we are aware of are conservative). Formal proofs of these results can be found in [Cleary 1993]. Similar results comparing zero-overhead TimeWarp (for distributed simulation) with conservative algorithms are cited in [Fujimoto 1990].

8.1 Basic Algorithm

In this sub-section I will sketch an informal proof that the basic TimeWarp algorithm will execute at least as fast as a sequential SLD execution. This result needs to be qualified in two ways. First, the comparison is with an SLD execution algorithm which selects goals for execution in the same order as the timestamps in the TimeWarp execution. Second, TimeWarp must execute on either one or an unbounded number of processors. This last condition is necessary and the result does not hold for intermediate numbers of processors. To see this, consider the case when multiple goals are mapped to the same processor. When a goal is executed optimistically an earlier timestamped goal (on the same processor) may become available for execution, for example, when an ANTI-BIND is received. The optimisticly executing goal will then block the rollback and re-execution of the goal with the earlier timestamp. Meanwhile the SLD execution will have executed the goals in the correct order without blocking.

We start the proof with some definitions. At each point in real time there will be some event in the TimeWarp system – either forward execution or backtracking – which has the smallest virtual time. We will refer to this as the *global event*. (Because other operations such as message passing and rollback are assumed to have zero execution time they can be ignored). We show by induction on the sequence of global events that each global event corresponds to some event in the sequential SLD execution and the real time of its SLD execution will always be later than the real time of its TimeWarp execution. This is trivially true at the start of execution. Consider the point when a new global event starts execution.

If the new global event is a forward execution, then the previous global event must have completed successfully (otherwise the new global event would be a backtrack operation). Also, because the new global event has the lowest virtual time, there are no pending goals with a lower virtual time, and so it must be a goal in the SLD tree. In the SLD execution the new global event will certainly be executed later then the old one (otherwise it would have been scheduled first) and they may be separated by any number of events (even an infinite number) which were optimisticly executed in the TimeWarp execution (or cut by a smart backtrack). The upshot of all this is that the new global event cannot complete later than the corresponding event in the SLD execution.

If the new global event is a backtrack (with its consequent ANTI-BIND, and ANTI-GOAL messages) then its virtual time may be earlier than the previous global event which may or may not have completed. Assuming the correctness of the TimeWarp algorithm, the SLD execution must eventually backtrack to the destination of the TimeWarp backtrack without generating any solutions in the meantime. (Because of the "smartness" inherent in the TimeWarp algorithm it may have immediately backtracked a number of events).

At the end of execution the last event executed will be a global event so the TimeWarp execution will terminate while the SLD execution is still running.

It is also notable that the basic TimeWarp algorithm has better termination properties than the sequential SLD algorithm – any SLD computation which finitely fails will also finitely fail under TimeWarp and there are computations which finitely fail under TimeWarp and do not finitely fail under sequential execution.

8.2 Full Algorithm

It is difficult to place a lower bound on the execution time of all possible AND-parallel algorithms. Instead we compare TimeWarp with a class of *conservative* algorithms defined as follows. An AND-parallel algorithm is conservative if there is a partial ordering between the goals in an execution such that the following rules are obeyed: if two goals can both bind the same variable they are ordered; a child goal lies after it's parent in the ordering; once a goal has been selected for execution its ordering is fixed; and if two goals are ordered, the execution of the earlier goal will complete before the later goal starts execution and all alternative clause selections for the later goal will complete before any alternative clause selections are explored for the earlier goal. All proposed non-optimistic AND-parallel algorithms that we are aware of are conservative. Of course, other classes of algorithms such as combined AND/OR-parallel algorithms and ones that dynamically re-organize the goal ordering might achieve lower execution times.

Given these definitions a strong result is available for the full TimeWarp algorithm (with both the BIND and ANTI-BIND optimizations). It is able to execute as fast as any conservative algorithm when both are executed on an unbounded number of processors. To achieve this the TimeWarp algorithm must use the unbounded number of processors by allocating each new goal its own processor.

The proof is essentially the same as above. We show by induction on the sequence of global events that each global event corresponds to some event in the conservative execution and that the real time of its conservative execution will always be later than the real time of its TimeWarp execution. This is trivially true at the start of execution. Consider the point when a new global event starts execution.

If the new global event is a forward execution, then the previous global event must have completed successfully (otherwise the new global event would be a backtrack operation). As each goal is on its own processor it is always immediately scheduled for execution, so the beginning of execution of the new global event must have been preceded either by the successful execution of its parent goal or by a rollback caused by an incompatible binding. Because of the ordering restrictions on the conservative algorithm the global event could not have executed any earlier within the conservative execution. (A goal is ordered after its parent and to be rolled-back by an incompatible binding it must share a variable with an earlier timestamped goal). The upshot of all this is that the new global event cannot complete later than the corresponding event in the conservative execution.

If the new global event is a backtrack (with its consequent ANTI-BIND, and ANTI-GOAL messages) then its virtual time may be earlier than the old global event which may or may not have completed. Assuming the correctness of the TimeWarp algorithm and the ordering restrictions on backtracking in a conservative execution, the conservative execution must eventually backtrack to the destination (of the TimeWarp backtrack). The earliest this can happen is at the same time or after some (potentially unbounded) number of conservative events.

It is also notable that the full TimeWarp algorithm has at least as good termination properties as any conservative algorithm – any conservative computation which finitely fails will also finitely fail under TimeWarp.

9 Summary and Future Work

In arriving at the fully-optimized algorithm, an interesting process has been followed. A well-understood sequential algorithm, namely backtracking for Prolog, has been transformed into a distributed algorithm by "applying" Time Warp to it. Time Warp was originally described as an algorithm for distributed simulation. However, every sequential algorithm has a virtual temporal coordinate system imposed by the order of operations during a sequential execution. By extracting this order and making it explicit, Time Warp can execute the algorithm in parallel. In this case, the resulting algorithm was sufficiently specialized that significant optimizations to the rollback processing of Time Warp were available. More "traditional" Prolog optimizations, like those for intelligent backtracking, may still be applied. We look forward to seeing other sequential algorithms, for which it is difficult to construct an efficient distributed version, transformed in the same fashion.

An implementation of the optimized algorithm is planned for the near future. Results are expected to compare favourably with concurrent logic languages for deterministic programs and with independent AND-parallel systems for nondeterministic programs, as well as with the unoptimized algorithm implemented in [Olthof 1991].

One area that we will be exploring with the implementation is the effect that different timestamp allocation algorithms have on performance. At one extreme, we can mimic the standard depth-first backtrack order by appropriate timestamp selections. However, we believe that some programs will respond well to other orderings that the flexibility of Time Warp allows.

Also, with care, it is possible to ensure that the timestamps assigned side-effecting operations such as `assert`, `retract`, and input/output operations will be in the same order as in a purely sequential execution. Coupled with database manipulation and I/O operations that can be rolled back, this would allow transparent distribution of programs containing such side effects.

10 Acknowledgements

This work was supported by the Natural Science and Engineering Research Council of Canada and Jade Simulations International Corporation.

References

[Clark & Gregory 1986] K.L. Clark and S. Gregory. PARLOG: parallel programming in logic, *ACM TOPLAS*, 8(1):1–49, 1986.

[Cleary 1993] J.G. Cleary. *Completeness of an Optimistic AND-parallel Algorithm*. Technical Report, Dept. Computer Science, University of Waikato, New Zealand, 1993.

[Cleary *et al* 1988] J.G. Cleary, B.W. Unger, and X. Li. A distributed AND-parallel backtracking algorithm using Virtual Time, In *Proceedings of the Distributed SImulation Conference*, pages 177–182, San Diego, February 1988.

[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.

[Fujimoto 1990] Richard M. Fujimoto. Parallel dicrete event simulation, *CACM*, 33(10):30–53, 1990.

[Gafni 1985] Anat Gafni. *Space Management and Cancellation Mechanism for Time Warp*. PhD thesis, University of Southern California, 1985.

[Hermenegildo & Rossi 1990] M. Hermenegildo and F. Rossi. Non-strict Independent And-Parallelism, In *Logic Programming: Proceedings of the Seventh International Conference*, Jerusalem, 1990.

[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 *Proccedings of the SCS Distributed Simulation Conference*, San Diego, CA, January 1985.

[Kale 1985] L.V. Kale. *Parallel Architectures for Problem Solving*. PhD thesis, Department of Computer Science, SUNY Stony Brook, 1985.

[Mannila & Ukkonen 1986] Heikki Mannila and Esko Ukkonen. Timestamped term reprsentation for implementing Prolog, In *Proceedings of the 1986 Symposium on Logic Programming*, pages 159–165, Salt Lake City, Utah, 1986.

[Olthof 1991] Ian Olthof. *An Optimistic AND-Parallel Prolog Implementation.* Master's thesis, Department of Computer Science, University of Calgary, 1991.

[Pereira *et al* 1986] L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Apar cio. 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 1987] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter, In *Concurrent Prolog—Collected Papers*, chapter 2, pages 27–83. MIT Press, 1987.

[Somogyi *et al* 1988] Z. Somogyi, K. Ramamohanarao, and J. Vaghani. A stream AND-parallel execution algorithm with backtracking, In *Fifth International Conference and Symposium on Logic Programming*, pages 386–403, Seattle, August 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.

[Ueda 1987] K. Ueda. Guarded Horn clauses, In *Concurrent Prolog—Collected Papers*, chapter 4, pages 140–156. MIT Press, 1987.

[van Emden 1984] M.H. van Emden. An interpreting algorithm for Prolog programs, In *Implementations of Prolog*, pages 93–110. Ellis Horwood, 1984.

[West 1988] Darrin West. *Lazy Re-evaluation in Time Warp.* Master's thesis, Department of Computer Science, University of Calgary, 1988.