OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors
by
Bogumil Hausman  Andrzej Ciepielewski  Seif Haridi
OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors

Bogumil Hausman, Andrzej Ciepielewski, Seif Haridi
Swedish Institute of Computer Science
Box 1263
S-163 13 Spånga
SWEDEN

Abstract

With the arrival of commercially available shared-memory multiprocessors, Prolog implementation efforts begin to shift from single processor architectures to the new ones. Among the main problems are efficient implementation of operations on variables and of task switching. Most of the solutions proposed so far suffer from expensive, non-constant time implementation of operations on variables. We propose a model (Versions-Vector Model) in which operations on all variables are constant time operations. The price we pay is a non-constant time of a task switch. As a remedy we propose two ways of decreasing that price. The first is promotion of variables on a task switch, from versions-vectors to the stack or heap, making subsequent task switches cheaper. The second is delayed installation of variables in versions-vectors, decreasing the cost of short branches. We believe that the increased memory consumption induced by our model can be accepted as it is traded for speed.
Contents

1. Introduction ................................................................................................................. 3
2. Simple WAM .................................................................................................................. 4
3. Scheduling of Goals ...................................................................................................... 7
4. Storage Model and the Basic Operations of the VV-WAM ............................................. 9
   4.1 Storage Model .......................................................................................................... 9
   4.2 VV-WAM ................................................................................................................. 10
5. Optimizations ................................................................................................................ 16
   5.1 Promotion ............................................................................................................... 17
   5.2 Delayed Installation .............................................................................................. 19
6. Restricting Parallelism ................................................................................................... 20
7. Low Level Considerations ............................................................................................ 23
8. Conclusions .................................................................................................................. 23
References ....................................................................................................................... 24
Appendix I. Versions-Vector WAM (Promotion) ................................................................. 26
Appendix II. Versions-Vector WAM (Promotion, Sequential Choice-Points) ...................... 30
Appendix III. Versions-Vector WAM (Promotion, Sequential Choice-Points, Delayed
             Installation) ......................................................................................................... 35
Appendix IV. Versions-Vector WAM (Promotion, Sequential Choice-Points, Delayed
             Installation, Favoured Bindings) .......................................................................... 38
1. Introduction

With the advent of commercially available shared-memory multiprocessors, Prolog implementation efforts begin to shift from the single processor architectures to the new ones. A typical shared-memory multiprocessor has a moderate number of processors between 32 to 64 processors. For example, Sequent Balance has up to 30 processors. The challenge is to utilize this class of multiprocessor systems in such a way that most programs will run much faster and no program will run much slower than on single processor systems [1].

Of all the different types of parallelism described in the literature [2, 3], OR-parallelism seems to be the most promising. OR-parallelism seems to offer a good potential for large scale, large-granularity parallelism across a wide range of applications such as querying a deductive database, parsing a natural language sentence, and compiling a set of objects [14]. Implementation of OR-parallel systems is closest to the standard sequential implementation, and, as such, can utilize the shared memory most efficiently, and also incorporate large parts of a sequential system, saving a lot of work and time. Besides, in contrast to AND-parallel systems based on explicit synchronization [4, 5, 6], parallelism in OR-parallel systems can easily be adjusted to be coarse-grained and thus suitable for the current multiprocessors.

The main problem of OR-parallel implementation is the management of simultaneous multiple bindings of variables. The number of different solutions (storage models) to the problem has proliferated during the last few years, starting from the abstract models by Pollard [7] and Clepielewski and Haridi [8], through more and more implementation oriented models: Borgward [9], D.S. Warren [10], Lindstrom [11], and recently Tinker and Lindstrom [12], and Disz et al. [13]. The last model (Argonne Model) has a prototype implementation runnurable on most of the recent multiprocessors. The implementation is slow compared to e.g. Quintus, but many times faster than any of the simulators of parallel machines running on single processor systems.

In a recent paper [14] D.H.D. Warren organizes the field and reconstructs some of the models, deriving them from the classical "Abstract Model" of resolution theory. Warren proposes also two most important measures for comparing the storage models: the cost of creating and accessing variable bindings and the cost of creating multiple tasks. In standard Prolog implementations the creation of and access to bindings are very fast, constant time operations for all variables. The backtracking operation, which corresponds to task switching, takes typically 15-20 machine instructions, plus the time for "untrailing" variable bindings. In the models quoted above the creation and access of bindings are not constant time operations for all variables. On the other hand, the time for process creation can be made constant in most of the above models, except for untrailing when applicable.

In the paper quoted above, Warren proposes another model (SRI Model) in which creation of and access to bindings are constant time operations for all variables, but task switching is not. The idea is to extend the conventional WAM [14] with a large binding table per processor and modify
the trail to contain address/value pairs instead of just addresses. Each table is used by just one processor to store and access bindings. On a task switch the table for the processor starting a new task must be partially reconstructed using the trail in the processor from which the task is taken ("stolen"). Warren shows also how the access time in the Argonne model can be made proportional to the number of processors in the system, and thus bounded, but still much longer than in sequential implementations.

The solution we present in this paper has time characteristics similar to that of the SRI model, but instead of having one large table per processor we propose using a vector of instances (versions-vector) per shared variable. Each vector has the number of components equal to the number of processors in the system. Each component in a vector is used by just one processor to store and access one version of the variable's value. Like Warren we assume that there are no more active branches than there are processors.

Our idea is related to the one of Pollard presented in [7] and [15]. The difference is, that Pollard proposal uses a tree of bindings per variable together with a fairly complex naming scheme for identifying which branch (not processor!) owns which binding. A similar proposal has been independently put forward by Yang [16]. Also Tinker and Lindstrom pursue a similar path [12].

The rest of this paper is organized as follows. In section 2 we shortly describe the standard WAM in order to create the frame of reference and be able to show the relative cost of the following modifications. In section 3 we give the scheduling algorithm which replaces backtracking. The algorithm is a mixture of breadth-first and depth-first scheduling and leads to coarse-grained parallelism approximating sequential execution on each of the processors. In section 4 we present our storage model and specify the basic operations of the extended machine. In section 5 we describe the optimizations reducing the cost of task switching. In section 6 we discuss restrictions to parallelism and their consequences. In section 7 we discuss some important low-level aspects like locking and storage allocation. We conclude in Section 8 by summarizing pros and cons of our model.

2. Simple WAM

In this section we present a stripped-off version of the WAM used by SICS. We shall call it S-WAM for conciseness. The description captures only the features of the machine relevant for comparing with the extended WAM (VV-WAM) presented in the following sections. The specification below is meant as a frame of reference and as a reminder of the simplicity of operations in sequential implementations.

Data Areas
Memory of the S-WAM consists of 4 areas: environment stack, term heap, trail, and control stack. In this paper we will treat the environment stack and the term heap together and call it stack. The S-WAM differs from the original WAM by having an explicit control stack for storing
choice-points. There are even other differences, but they are of no importance here.

Registers
Part of the current computation state is held in a number of registers. We list only the registers relevant for this paper.

P Program pointer
ST Top of stack pointer
B Top of control stack pointer
TR Top of trail pointer

Contents of a Choice-Point
A choice-point is used to store the part of the machine state to be used at backtracking. For simplicity we make explicit some information (open alternatives) which is not explicitly present in the choice-points of sequential implementations. We only show part of a choice-point contents.

P' Alternative clause pointer
ST' Alternative top of stack
TR' Alternative top of trail
OA' Number of alternative clauses left (open alternatives)

ST' divides the stack into the "private" (created later than the top-most choice-point), and the "shared" parts. We introduce the terminology private and shared for compatibility with the following sections. We will also use names "open" and "closed" choice-points for choice-points with OA' > 0 and OA' = 0, respectively.

Data Objects
A Prolog term is represented by a word containing a tag and a value. The tag distinguishes the type of term. We assume for simplicity that there are just two types of terms: variable terms (with the tag VAR) and non-variable terms (with the tag NON-VAR). An unbound variable is represented by a variable term bound to itself.

Conventions
In the description of operations we adopt mostly the C language-like conventions, with the following additions. Machine registers and components of the top-most choice-point are used as global variables. Local variables have names starting with a lower case letter. We use some help functions: tag(V) returns the tag part of the term V, value(V) returns the value part of the term V, tagged(T,V) returns a tagged word (term) with the tag T and the value V. Finally "<" and ">" are operators comparing pointers. For example: if U < V is true then U points to an object on the stack
(trail, control stack) that has been created earlier (is older) than the object pointed by V.

Operations
The relevant operations are: deref - dereference a variable, bind - bind a variable to a value and possibly trail the address, create_choice-point (corresponds to TRY instruction in WAM) - create a new choice-point and save the current state, backtrack (corresponds to RETRY and TRUST) - untrail variables, choose the next alternative in the search tree, and possibly remove the current choice-point. The operations are divided into operations on a single variable (deref and bind), and scheduling operations (create_choice-point and backtrack). Again we introduce a new name, scheduling, to put the reader in the right state of mind.

Operations on a single variable

deref(V) term *V
  {if tag(*V) == VAR and not (value(*V)) == V then %a bound variable
deref( *V)
  else %follow chain of variables
  V}

bind(U,V) term *U,V
  {if U << ST' then %U in shared section
    *(++TR) = U %trail U
    *U = V} %bind U to V

Scheduling operations

create_choice-point(NumAlt) integer NumAlt
  {allocate a choice-point and let B point to it;
    <P',ST',TR',OA'> = <P,ST,TR,NumAlt-1 >; %save state
    assign the address of the first alternative to P}

backtrack
  {if not (B == top_of_tree) then %more choice-points
    untrail-vars;
    <P,ST,TR> = <P',ST',TR'>; %restore state
    assign the address of the next alternative to P' and P;
    OA' = OA' - 1;
    if OA' == 0 then %last alternative
      {remove the last choice-point;
        let B point to the previous choice-point}
    else %no more choice-points
      terminate the computation}

untrail-vars
  {while TR >> TR' do %below the top choice-point
    {x0 = *(TR--); %untrail a variable
      *(x0) = tagged(VAR,x0)}} %unbind a variable

Note: top_of_tree is a constant indicating that there are no more choice-points.
3. Scheduling of Goals

In sequential implementations of Prolog the search tree of a program is traversed in a depth-first manner. The choice-point stack is used to store the open alternatives (Figure 1A). In a multiprocessor implementation there are several processors performing the search. The work could be divided among them in nearly any manner. We choose a scheduling scheme where an idle processor always takes the top-most open alternative in some subtree, and works on the subtree starting at that alternative until all branches are exhausted. At that point it becomes idle again and looks for more work (Figure 1B).

![Figure 1. A search tree traversed by one (A) and four processors (B).](image)

We have chosen the above heuristics believing that it will in the majority of cases minimize the number of task switches, because we expect that the subtrees closest to the top are the largest ones [3]. Letting each processor work on a large subtree will give a local execution similar to that of sequential Prolog, and thus efficient.

The processor should choose among top-most alternatives so that the cost of starting the new task is minimal. We define the cost to be the distance between the last leaf explored by the process looking for work and a node with an open alternative. We shall define the cost more precisely later on.
Figure 2. A search tree with open alternatives in nodes A and B.

We propose the following heuristics for choosing one of the top-most alternatives. The processor which becomes idle (no more alternatives on the path from the current leaf to the root) will search the tree upwards until it comes to the first node under which there is a subtree with open alternatives. It then takes the top-most alternative in that subtree. For example, processor 1 on Figure 2 has no more open choice-points on the way to the root. When it becomes idle, it will search the tree upwards until the closed choice-point X, then it will start going downwards until the open choice-point B. The argument is, that the part of the distance which must be covered on the way up will be shortest that way. The part of the distance to be covered on the way down can be of any length, but we cannot see any reason why, statistically, it should be longer than in any other path which could be chosen (consider paths for processor 1 to nodes A and B on Figure 2).

In order to implement the scheduling scheme outlined above we have to extend the S-WAM slightly. We add a dispatching pool (DP) consisting of queues of pointers (dispatching elements) to the choice-points containing open alternatives (open choice-points). There will be one queue per processor. The head of each queue points to the top-most open choice-point in some subtree. Choice-points are extended with three components: a pointer (DE') to the dispatching element pointing to this choice-point (used to remove the element from the dispatching pool), a component (AA') to keep the number of active paths emerging from this choice-point (used to decide when a choice-point can be removed and whether a processor passing this choice-point upon backtracking should look for work in the subtree emanating from that choice-point) and finally a component (AP') keeping the record of the processors working in the subtree below this choice-point (used to decide which queues in the dispatching pool should be looked up while looking for work). The last component can be implemented as a bit vector if the number of processors is moderate.
Figure 3 below illustrates the use of the new components of the machine.

![Diagram](image)

Figure 3. A snapshot of the information in the dispatching pool and the search tree of some program. Only part of the contents of choice-points is shown.

We assume until Section 6 that alternatives can be stolen from any choice-point, i.e. that the parallelism is unrestricted.

4. Storage Model and the Basic Operations of the VV-WAM

The scheduling scheme presented in the previous section assures that the number of branches searched in parallel does not exceed the number of processors in the system (cases when searching a branch must be suspended will be discussed in Section 6). In the storage model presented below we take advantage of the fact that the maximal number of active branches is fixed. Our data areas (the trail, and the stack) are now represented as stack-trees, but a single processor still sees its own areas as in S-WAM. In the rest of this section we present the storage model and specify the operations on variables and the scheduling operations.

4.1 Storage Model

We solve the problem of multiple, simultaneous bindings to variables bound in shared sections of the stack by introducing a version-vector per such variable. Each vector has a number of components equal to the number of processors in the system. Each component of a vector is used by just one processor to store and access the binding belonging to that processor (see Figure 4).

As in the S-WAM a Prolog term is represented by a tagged word. A variable having several versions of bindings is bound to a version vector with, possibly, different values in different
components. Such variable is represented by a word tagged with a new tag VV. An unbound variable, as seen by processor I, is represented either by a reference to itself (unbound in all processors having access to it), or by a pointer to a versions-vector with the component I with a reference to itself (see Figure 4).

The trail of the VV-WAM contains address/value pairs instead of just addresses. The values are used during a task switch to install variables in versions vectors of a processor starting a new task (see section 4.2).

![Diagram of VV-WAM](image)

Figure 4. A variable with several versions. Bindings by processors 3 and 4.

### 4.2 VV-WAM

The VV-WAM specified below is a version of the S-WAM with the modifications required for the new scheduling scheme and the storage model.

#### Registers

A register DP, pointing to a queue in the dispatching pool is added. There is a set of registers per processor. In contrast to the other registers, DP in each processor can be accessed by the other processors.

- **P** Program pointer
- **ST** Top of stack pointer
- **B** Top of choice-point stack pointer
- **TR** Top of trail pointer
- **DP** Dispatching pool queue pointer
Contents of a Choice-Point
Choice-points are extended with three components (DE', AA', and AP') whose functions have been explained in Section 3.

P' Alternative clause
ST' Alternative top of stack pointer
TR' Alternative top of trail pointer
OA' Number of alternative clauses left
DE' Associated dispatching element
AA' Number of active paths
AP' Active processors

Operations
All the operations are modified. The scheduling operations are extended with the procedure steal invoked when there is no work left on the local stack. We added one important help procedure, install-task, invoked before starting a new task in order to add bindings to versions-vectors and a processor identifier to choice-points.

Operations on a single variable
The deref procedure must consider the case of a variable bound to a versions-vector. In that case the address of the component is computed using the processor id (Pid). Notice that, if the variable is unbound, pointer to the term on the stack is returned.

deref(V,Pid) term *V, integer Pid
{if tag(*V) == VAR then
   if not value(*V) == V then
      deref(*V,Pid)
   else
      V
else if tag(*V) == NON-VAR then
   V
else
   {mv = value(*V) + Pid;
    if not value(*mv) == mv then
deref(*mv,Pid)
   else
      V }
%
unbound on stack
%tag(*V) == VV, variable bound to
%a versions-vector
%bound in version-vector
%unbound in versions-vector
%return pointer to stack

A variable is bound (by bind) on the stack in the usual way only if it is in the private section. Otherwise the address and the new value of the variable are trailed and the value is stored in the proper component of the versions-vector. If it is the first binding to that variable then a versions-vector is allocated. Notice that the vector need not be initialized on every allocation. All the vectors can be initialized prior to or at the first allocation and then are cleaned up during untrailing
(see untrail-vars).

\begin{verbatim}
bind(U,V,Pid) term *U,V, integer Pid
  {if U <<= ST' then
    *(TR++) = <U,V>;
    if value(*U) == U then
      {allocate a versions-vector and assign its address to vec;
        *U = tagged(VV, vec)}
    else
      vec = value(*U);
      vec(Pid) = V}
  else
    *U = V
\end{verbatim}

The operations on variables have become more complex, but the overhead is invoked only for variables being bound in the shared section (bind) or already bound (deref) in versions-vectors. Even in those cases the invoked overhead is constant and small.

Scheduling operations
The \texttt{create_choice-point} procedure initializes the components of a choice-point. As a side-effect an element is added to the dispatching pool. The calling processor becomes the first active one under that choice-point.

\begin{verbatim}
create_choice-point(Pid, NumAlt) integer Pid, NumAlt
  {allocate a choice-point and let B point to it;
    add element to the DP queue pointing to the choice-point and let DE' point to it;
    <P',ST',TR',OA'> = <P,ST,TR, NumAlt-1 >;
    AA' = 1;
    AP'(Pid) = true; %the first active processor
    assign the address of the first alternative to P}
\end{verbatim}

The \texttt{backtrack} procedure has two main cases, the first (\texttt{OA'}=0) is similar to the main case in the sequential backtracking, and the second (\texttt{OA'}=0) corresponds to the case in which the sequential execution terminates because the whole tree has been searched.

In the first main case there are alternatives to be taken from the choice-point. The basic actions are the same as during sequential backtracking (untrail variables, restore state and possibly remove the choice-point). There are two important subcases. In the first (\texttt{AA'}=1) there are no more active branches under that choice-point, and thus no more processors. In the second there are more active branches below. Only in the first case the backtracking processor can free some of the versions vectors and deallocate the choice-point if it takes the last alternative. It is very important to remove choice-points whenever possible because then the private section of the (binding) stack can be extended, and the need for versions-vectors decreases.

In the second main case there are no more alternatives to be taken from the choice-point, because they were taken by other processors. Variables are untrailed. If the backtracking processor is the last one using that choice-point (\texttt{AA'}=0) it has to continue backtracking higher up in the tree,
otherwise there are active branches below that choice-point, and the backtracking processor attempts to steal alternatives from processors working on them. If the stealing fails the processor will go on up the tree. If the processor continues backtracking above the choice-point, it is removed from the record of active processors (AP'). Notice that the active path counter (AA') and the argument Alone shown below are used together as a distributed counter of active branches.

```
backtrack(Pid, Alone) integer Pid, bool Alone
   {if OA' > 0 then
      {OA' = OA'-1;
       if AA' == 1 then
         {if OA' == 0 then
            untrail_vars(Pid, true)
         else
            untrail_vars(Pid, false);
            <P, ST, TR> = <P', ST', TR'>;
            assign the address of the next alternative to P and P';
            if OA' == 0 then
               {remove the element pointed by DE' from the DP queue;
                remove the last choice-point;
                let B point to the previous choice-point}\}
         else
            {untrail_vars(Pid, false);
            <P, ST, TR> = <P', ST', TR'>;
            assign the address of the next alternative to P and P';
            if OA' == 0 then
               remove the element pointed by DE' from the DP queue}}}
   else
      {if Alone then
         AA' = AA'-1;
         if AA' == 0 then
            {untrail_vars(Pid, true);
             remove the last choice-point;
             let B point to the previous choice-point;
             backtrack(Pid, true)}
         else
            {untrail_vars(Pid, false);
             steal(Pid);
             AP'(Pid) = false;
             let B point to the previous choice-point;
             backtrack(Pid, false))}\}
   %open choice-point
   %last processor
   %untrail and clean up
   %just untrail
   %last open alternative
   %not last processor
   %just untrail
   %last open alternative
   %closed choice-point
   %alone in its subtree
   %branch removed
   %no subtrees below
   %untrail and clean up
   %go on upwards, still alone
   %subtrees below
   %just untrail
   %look for work, return only if
   %nothing to steal
   %Pid no longer active below
   %go upwards, no longer alone
```

The steal procedure is responsible for finding an open alternative and, if found, setting up a new task. In case when there are no open alternatives and no other active processors, the computation terminates. If an open choice-point is found the new task is set up by loading the registers of the calling processor from the choice-point, adjusting contents of the choice point, and finally installing variables on the trail between the open choice-point and the choice-point on which the backtracking stopped (cross choice-point), and also adding the processor id to all choice-points on the way.
steal(Pid) integer Pid
\{pid' = find processors with non-empty DP(pid'), where pid' in AP', and choose the one
\text{whose \text{DP}(pid')} contains the highest open choice-point; \}
\text{if pid' == none then} \quad \% \text{not found}
\\{\text{if B == top_of_tree then}
\\quad \text{if all other processors waiting then}
\\quad \quad \text{terminate computation}
\\quad \text{else wait until a parallel choice point is created}
\text{else}
\quad \text{return}\} \quad \% \text{to continue backtracking}
\text{cross = B; \% save address of cross ch-p B}
\text{B = get the address of an open choice-point from DP queue in pid';}
\text{<P,ST,TR> = <P',ST',TR'>;}
\text{assign the address of the next alternative to P and P';}
\text{OA' = OA'-1;}
\text{if OA' == 0 then}
\\quad \{\text{remove choice point from DP(Pid');}
\\quad \text{DE' = none}\}
\text{AA' = AA'+1; \% new active path}
\text{install_task(B,cross,Pid)} \quad \% \text{add bindings and Pid}

The \text{untrail-vars} procedure removes variables from the trail, makes the proper component in
the versions-vectors unbound, and finally removes versions-vectors for variables in the section of
the stack which has become private or will be deallocated. A section becomes private when the last
open alternative is taken and the choice-point is removed by the processor taking the alternative
(OA'=1, AA'=1). A section is deallocated when the last active processor in a subtree passes a
closed choice-point looking for work (OA'=0, AA'=1).

\text{untrail-vars}(Pid,Last) integer Pid, bool Last
\{\text{if Last then}
\\quad \{\text{assign the address of the previous choice-point to chp;}
\\quad \text{new_limit = chp->ST'}\}
\text{while TR >> TR' do}
\\quad \{<x0._>_ = *(TR'-); \% get address from trail}
\\quad \text{vec = value('x0'); \% unbind a component}
\\quad \text{vec(Pid) = tagged(VAR,vec+Pid);}
\\quad \text{if Last and x0 >> new_limit then \% cleaning up}
\\quad \quad \text{deallocate the version vector pointed by vec (notice that all}
\\quad \quad \quad \text{components are unbound);}}\}

The \text{install_task} procedure copies values from the trail, between the choice-point from which
the alternative is stolen and the cross choice-point, to the proper component (Pid) of the
versions-vectors. Notice that if the values were not installed the processor starting the new task
would have to search for the values on the trail. In addition, information about the new active
processor is added to all the choice-points between the ones mentioned above.
install-task(Bsteal, Bcross, Pid) pointer Bsteal, Bcross, integer Pid

{b1 = Bsteal;
 t1 = b1->TR';
 loop
   {b2 = address to the choice-point preceding b1;
    t2 = b2 ->TR';
    while t1>> t2 do
      <x0, value0> = *(t1--);
      vec = value(*x0));
      vec(Pid) = value0};         %bind a component
    b1->AP(Pid) = true;
    exit if b2 == Bcross;
    b1 = b2;
    t1 = t2 }}

We can now define the cost of a task switch to be the sum of the time for untrailing between the top of the trail and a cross choice-point, the time for installation between the cross choice-point to the choice-point where an alternative is taken, and the time for looking up the dispatching pool. The cost of untrailing is there even in sequential implementations, the rest is overhead induced by parallelism.
We conclude this fairly technical section by an example illustrating backtracking, stealing and installation.

![Diagram](image)

Figure 5. Backtracking, stealing and installation.

Variables on the trails are shown on appropriate arcs. Processor P1 has terminated a task and found a new one. Variables X and W have been untrailed and Z' has been installed.

5. Optimizations

The cost of task switching is a major cost in the VV-WAM. We propose two optimizations decreasing that cost. The first, promotion, consists of moving bindings of some variables from versions-vectors and the trail back to the stack. It is done during installation phase of a task switch and saves installation effort for the subsequent processors starting tasks in the same subtree. The second, delayed installation, postpones installation until some variables are actually accessed. This decreases mainly the cost of starting short branches.
5.1 Promotion

When the last alternative is taken from a choice-point in the S-WAM, the choice-point is removed, and the part of the stack up to the previous choice-point becomes private (i.e. variables there no longer need to be trailed). In the VV-WAM that corresponds to two distinct situations: the first when a processor takes the last open alternative from a choice-point and the processor is the last one using that choice-point, and the second when a processor during backtracking passes a closed choice-point with just one active path (possibly shared by many branches) below besides the one being backtracked (OA'=0, AA'=1), we shall call such choice-point "dead" (see Figure 6). The first situation is already taken care off in a similar way as in the S-WAM (install-task). The second is dealt with below.

When a choice-point becomes dead, the section of the stack between it and the previous choice-point, call it "regained" section, may be modified as if the dead choice-point had never existed. Had the "dead" choice-point not existed, the process extending the branch below could bind the variables in the regained section as if they were private, until it created another choice-point. This means that some of the variables in the regained section can be moved from versions-vectors to the stack and removed from the trail, we call that process promotion.

Our aim is to save installation time, thus it is sufficient to do the promotion during installation of the first task fetched from a choice-point below the dead one. The process installing the task will promote values of the variables in the regained section for the trail segment between the dead choice-point and the choice-point from which the task is stolen. Subsequent processes fetching alternatives in this subtree will have less values to install.
Promotion is illustrated by Figure 6.

Figure 6. Processor 3 has terminated a task and passed the choice-point B on backtracking. B became dead. When an alternative is stolen from the choice-point C the variable Z' in the regained section is promoted.

We have tried in our design to avoid locking in operations on variables. During promotion there are, possibly, many processors having access to the promoted variables and their versions-vectors. To avoid locking we replace the trail by two separate trails: a trail for installation and promotion (installation trail), and a trail for cleaning up and removing versions-vectors (clean-up trail). An element of an installation trail is a pair <address, value>, and an element of a clean-up trail the address of a versions-vector. During promotion binding of each promoted variable is changed from a versions-vector pointer into the promoted value, and the corresponding element is removed from the installation trail, the contents of the versions-vector and the clean-up trail is not changed. In this way no processor can get a wrong value and no locking is needed. Cleaning up and removing of versions-vectors is delayed until backtracking, when the clean-up trail is used.
The use of the two trails is illustrated in Figure 7, below.

![Diagram of two trails: Before and After](image)

Figure 7. Data structures for promotion.

The only operation influenced by promotion is installation (some extra tests must be done in `install_task`). There is no overhead on any other operation.

### 5.2 Delayed Installation

Overhead of starting short branches is a trouble in any implementation. The problem becomes even more serious in our implementation because of the expense of setting up a new task. One means of avoiding short branches is indexing. Unfortunately even with indexing (especially on the first argument only) there will still exist branches that will fail shortly after creation. We propose delayed installation mainly to decrease the cost of such branches.

The delayed installation means that the bindings on the trail are not moved to the versions-vector when a task is set up, but first when some shared variable is accessed. More precisely, the installation procedure is invoked when a variable unbound in its version-vector is
dereferenced. The procedure will install all the variables on the relevant part of the trail up to the accessed variable. There is no overhead on other accesses.

The example below shows a case of delayed installation.

Figure 8. Processor P4 has terminated a task and started a new one from the choice-point n4. Installation of the variables Y and Z' was delayed. The figure shows a snapshot when Z' has already been accessed and installed. The pointer to the trail shows where the installation will proceed later on.

6. Restricting Parallelism

We have assumed so far, that all branches in a search tree are always explored and always (potentially) in parallel. It must be possible to prohibit or stop execution of some branches. That can be done using constructions like cut or commit. There are also several reasons (e.g. efficiency, algorithms, side-effects) for allowing a programmer to annotate programs so that only clauses in some predicates will be allowed to be executed in parallel.

Implementation of cut/commit in a parallel environment is an intricate problem. We believe to have a solution but will not go into it because it would take another article. It is enough to say here that the solution requires suspension of branches when cut/commit are considered together with global side-effects (assert, I/O). That violates our assumption about the number of branches explored simultaneously not exceeding the number of processors.

That problem can be solved in several ways. One solution is to let the processor executing the branch which is suspended to idle until the branch is activated again. It is simple, but could severely limit the number of available processors. Another solution is to limit the number of simultaneous
branches by some arbitrary number larger than the number of processors. The usefulness of that solution depends on the typical number of simultaneous suspensions. There is, finally, a general solution. When a branch is suspended the processor working on it can clean up the relevant versions-vectors and look for some other work (backtracking without cleaning up the trails). The suspended branch can be executed later on by some other processor, using the values left on the installation trail to set up the new task. If suspension is not a very frequent operation this solution is preferable.

Having parallel and sequential predicates can, in contrast to cut/commit, be used to increase efficiency of the VV-WAM. Having two types of predicates implies that there should be two types of choice-points. The sequential type, used by one processor at a time, and the parallel type shared by several processors. Alternatives can only be stolen from parallel choice-points. We utilize that fact to decrease the number of variables to be installed on a task switch (even dereferencing chains will be shortened, but it is only a marginal gain).

Observe that only the variables unbound when a parallel choice-point is created may get several simultaneous bindings. If there are one or more sequential choice-points below the last parallel choice-point, variables in the stack section below the parallel one might get several values but not simultaneously (just like variables in the shared section of a sequential implementation), and thus do not require versions-vectors and do not have to be installed (their values are stored directly on the stack). We shall call shared only the section of the stack above the latest parallel choice-point, the section between the latest parallel and the latest sequential choice-points will be called non-shared. The private section is below the last choice-point, whatever its type.

The following changes are done to the VV-WAM. Two types of choice-points are introduced, sequential - as in the S-WAM, and parallel as in the previous version of the VV-WAM. A register (PB) pointing to the last parallel choice-point is added. The B register, referring the top-most choice-point (sequential or parallel), is used now to find the border between the private and the non-shared sections, and the PB register is used to find the border between the non-shared and the shared sections. Notice that when the last choice point on the stack is a parallel one (B=PB) there is no non-shared section.

Only the procedures **bind** and **backtrack** are influenced by the changes. The first must handle variables in the non-shared section and the second must take care of sequential choice-points. When binding a variable in the non-shared section the value is stored directly on the stack (as in S-WAM) and is trailed only on the clean-up trail. There is a new case in scheduling: a processor can not take an alternative from a sequential choice-point if there are active branches below it, because doing that the processor could unbind variables on the shared section of the stack. The processor will instead go to the closest parallel choice-point and go on with the backtracking procedure. The sequential alternatives will be taken by the last processor in that subtree.
The example below shows the situation with a sequential choice-points temporary "trapped" above a parallel choice-point.

Figure 9. Sequential and parallel choice-points.

n3 is a sequential choice-point, all the others are parallel. Processor P3 has finished one task. It may not take the alternative from n3, because P2 is still working below. Instead it goes on to n2 to finally find an open alternative in n5. The open alternative in n3 will be executed by P2. The variable Z' was bound before the parallel choice-point was created (it was in the non-shared section), and thus does not have a versions-vector.

Studying Figure 9 one could notice that there are more optimizations to be done. The subtree under the n5 choice-point is only processed by P4, and there is a possibility that this processor will explore even the branches starting in the choice-points n6 and n7. These are parallel choice-points, meaning that P4 will allocate versions-vectors just to deallocate them again very soon. P4 will be the only user of those versions-vectors. It would be good to avoid using versions-vectors at all in situations like this one, saving a lot of space and some time.

It would help if n6 and n7 were sequential, but the need for making them sequential is dynamic (depends on the current processor usage), and thus difficult to discover (it is not so in many other cases where annotations are suggested by the algorithm).

We have another solution where the Versions-Vector Model is extended using the idea of "favoured" branch introduced by Overbeek [13]. In the example above part of the P4's branch below n5 could be considered favoured for P4. Variables in the favoured section would not need be bound to full versions-vectors.
7. Low Level Considerations

Any specification stops at some level hiding more or less details from the reader. Our specification hides two important issues, synchronization and storage management.

Synchronization is an issue not present in sequential implementations of Prolog. Synchronization does not only take time in form of instructions executed, but can also cripple parallelism. We have designed our algorithms taking great care to avoid synchronization. In the VV-WAM no synchronization is needed in management of variables, with one exception. When a variable in the shared section is about to be bound and it is not yet bound to a version-vector, the variable cell must be locked until the vector is allocated and the cell bound to it. As mentioned earlier the extra trail in implementation of promotion has been introduced in order to avoid locking in promotion. An alternative would be to deallocate versions-vectors immediately when the corresponding variables are promoted, but that would make locking necessary on all read accesses (dereferencing) to the shared section. We found that unacceptable. Our storage model could be simplified if we assumed cheap locking. We have nearly managed to avoid locking in operations on variables, but complex synchronization is still needed in scheduling operation, not least when cut/commit and side effects are considered, as described in [17].

The storage management is complicated by a property the VV-WAM has inherited from the S-WAM. In the S-WAM the physical addresses are used e.g. to decide if a variable is in the shared section, and to decide which of two variables has been created later. It is not a major problem because the VV-WAM is meant to be implemented on a shared-memory multiprocessor where the physical addresses can easily be compared, but it is a complication, because several branches are explored at the same time. If the physical addresses are to be used in the same way as in the S-WAM, addresses must grow down each path in the three, and that implies that the block of storage freed by one branch can only be reused by another branch if it has an address higher than addresses of all blocks in that branch. That means there could be plenty of free storage which could not be used. In order to avoid that, it is reasonable to consider a storage management scheme not relying on physical addresses as described in [18], even though part of the advantage of shared memory would be lost.

The issues described in this section must be considered in any parallel implementation.

8. Conclusions

Our goal is to execute a superset of Prolog on shared-memory multiprocessors so that no programs run much slower and most programs run much faster than on single processor system. The abstract machine we propose brings us closer to that goal. It can execute sequential programs with nearly no overhead. It allows simultaneous execution of several branches with very little overhead in operations on variables. It actually assures that operations on all variables are constant time
operations. The cost of starting a new task depends on the distance in a search tree to be covered during installation of bindings. By the use of promotion, delayed installation, and sequentialization we bring that cost down. The promotions prohibits the cost from growing when the tree grows, the delayed installation makes execution of short branches cheap, and sequentialization decreases the number of variables requiring versions-vectors. The main disadvantage of our proposal is the, possibly, high memory consumption. The memory consumption depends on the fraction of variables requiring versions-vectors. That fraction is program dependent [3]. All the above optimizations, and also the additional one mentioned at the end of Section 6, bring that fraction down.

The informal cooperation between Argonne National Laboratory, the University Manchester and SICS has resulted in proposals for several abstract machines having many common features but different storage models. The Argonne's prototype implementation [13] is currently used to evaluate the different proposals using a fairly rich set of non-toy programs.

Acknowledgements

This work would not be possible without the exciting cooperation, called Gigalips project, between Argonne National Laboratory, University of Manchester, and Swedish Institute of Computer Science. We are especially grateful to Ross Overbeek, David H. D. Warren, and Mats Carlsson for their constructive criticism.

References

[7] G. H. Pollard, Parallel Execution of Horn Clause Programs, PhD Thesis, Imperial College of
Science and Technology, University of London, 1981


Appendix I. Versions-Vector WAM (Promotion)

We show here the specification of the VV-WAM with the first optimization - the promotion, which is explained in section 5.1.

Registers
P Program pointer
ST Top of stack pointer
B Top of choice-point stack pointer
ITR Top of installation trail pointer
CTR Top of clean-up trail pointer
DP Dispatching pool queue pointer

Contents of a Choice-Point
P' Alternative clause
ST' Alternative top of stack pointer
ITR' Alternative top of installation trail pointer
CTR' Alternative top of clean-up trail pointer
OA' Number of alternative clauses left
AA' Number of active paths
DE' Associated dispatching element
AP' Active processors

Operations on a single variable

deref(V,Pid) term *V, integer Pid
{if tag(*V) == VAR then
 if not value(*V) == V then
deref(*V,Pid)
 else
 V
 else if tag(*V) == NON-VAR then
 V
 else
 {mv = value(*V) + Pid;
 if not value(*mv) == mv then
deref(*mv,Pid)
 else
 V
}}
bind(U,V,Pid) term *U,V, integer Pid
   {if U <= ST' then
    *(ITR++) = <U,V>;
    if value(*U) == U then
      {allocate a versions-vector and assign its address to vec;
       *U = tagged(VV,vec)}
    else
      vec = value(*U);
    *(CTR++) = vec;
    vec(Pid) = V
   else
    *U = V}
%
in shared section
%trail a pair
%totally unbound
%bind U to versions-vector
%trail an address of v-vector
%bind a component
%in private section
%bind directly

Scheduling operations

create choice-point(Pid, NumAlt) integer Pid, NumAlt
   {allocate a choice-point and let B point to it;
    <P',ST',ITR',CTR',OA',AA'> = <P,ST,ITR,CTR,NumAlt-1,1>;
    assign the address of the first alternative to P;
    add element to the DP queue pointing to the choice-point and let
    DE' point to it;
    AP'(Pid) = true
   }
%the first active processor

backtrack(Pid,Alone) integer Pid, bool Alone
   {if OA' > 0 then
    {OA' = OA'-1;
     if AA' == 1 then
      {if OA' == 0 then
       untrail_vars(Pid,true)
      else
       untrail_vars(Pid,false);
      <P,ST,ITR,CTR> = <P',ST',ITR',CTR'>;
      assign the address of the next alternative to P and P';
      if OA' == 0 then
       {remove the element pointed by DE' from the DP queue;
        remove the last choice-point;
        let B point to the previous choice-point}
      else
       {untrail_vars(Pid,false);
        <P,ST,ITR,CTR> = <P',ST',ITR',CTR'>;
        assign the address of the next alternative to P and P';
        if OA' == 0 then
         remove the element pointed by DE' from the DP queue}
      else
       {if Alone then
        AA' = AA'-1;
        if AA' == 0 then
         {untrail_vars(Pid,true);
          remove the last choice-point;
          let B point to the previous choice-point;
          backtrack(Pid,true)
         }%go on upwards, still alone
       %not last processor
       %just untrail
       %last open alternative
       %OA' == 0, closed choice-point
       %alone in its subtree
       %branch removed
       %no subtrees below
       %untrail and clean up
     else
     untrail_vars(Pid,false);
     <P,ST,ITR,CTR> = <P',ST',ITR',CTR'>;
     assign the address of the next alternative to P and P';
     if OA' == 0 then
      remove the element pointed by DE' from the DP queue}
    else
    backtrack(Pid,Alone)
}
else
    \{untrail_vars(Pid, false);\} %subtrees below
    steal(Pid); %just untrail
    AP'(Pid) = false; %look for work, return only if
    let B point to the previous choice-point; %nothing to steal
    backtrack(Pid, false)); %Pid no longer active below
%go on upwards, no longer alone

steal(Pid) integer Pid
\{pid' = find processors with non-empty DP(pid'), where pid' in AP', and choose the one
whose DP(pid') contains the highest open choice-point;\}
if pid' == none then %not found
    \{if B == top_of_tree then
        if all other processors waiting then
            terminate computation
        else wait until a parallel choice point is created\}
else
    return\} %to continue backtracking
    cross = B; %save address of cross ch-point B
    B = get the address of an open choice-point from DP queue in pid';
    <P,ST,ITR,CTR> = <P',ST',ITR',CTR'>;
    assign the address of the next alternative to P and P';
    OA' = OA'-1;
    if OA' == 0 then
        \{remove choice point from DP(Pid');
            DE' = none\}
    AA' = AA'+1; %new active path
    install_task(B, cross, Pid) %add bindings and Pid

Procedure untrail-vars unbinds all the variables which addresses are stored on the clean-up trail
between current CTR and CTR'.

untrail-vars(Pid, Last) integer Pid, bool Last
    \{if Last then
        \{assign the address of the previous choice-point to chp;
            new_limit = chp->ST'\}
    while CTR >> CTR' do
        \{x0 = *(CTR'-);
            vec = value(*(x0));\} %get address from trail
            vec(Pid) = tagged(VAR, vec+Pid); %unbind a component
            if Last and x0 >> new_limit then %cleaning up
                deallocate the version vector pointed by vec (notice that all
                components are unbound)))\}
For installation and promotion of variables processor uses the installation trail. Installation starts from the "stolen" parallel choice point (Bsteal) and proceeds until the "cross point" (Bcross). Finding a dead choice-point processor promotes variables in the "regained" section.

**install-task** (Bsteal, Bcross, Pid) pointer Bsteal, Bcross, integer Pid

```c
{b1 = Bsteal;
 t1 = b1->ITR';
 loop
   {b2 = address to the choice-point preceding b1;
    t2 = b2->ITR';
    if b2->OA' == 0 and b2->AA' == 1 then %dead choice-point
        {Promote = true;
         b3 = address to the choice-point preceding b2;
         limit = b3->ST'}
   else
     {t1 = t2}
   while t1>> t2 do
     {<x0, value0> = *(t1);
      if Promote and x0 >> limit then
        {/*(x0) = tagged(VAR,value0);
         ITR = ITR - <*(t1)>}
      else
        {vec = value("*(x0)");
         vec(Pid) = value0}
    t1--}
  b1->AP'(Pid) = true;
 exit if b2 = Bcross;
 b1 = b2;
 t1 = t2 }}
```

**Initially** (ProgramStart,Pid) pointer ProgramStart, integer Pid

```c
{B = top_of_tree;
 ST = stack_start;
 ITR = installation_trail_start;
 CTR = clean-up_trail_start;
 DP = all queues are initialized to empty;
 if Pid == chosen_processor_nr then
  P = ProgramStart
 else
  steal(Pid)}
```
Appendix II. Versions-Vector WAM (Promotion, Sequential Choice-Points)

We show only these parts of the specification which are influenced by the sequential choice-point strategy. Use of the sequential choice points is explained in section 6.

Registers
Register PB points to the last parallel choice-point.

P  Program pointer
ST  Top of stack pointer
B  Top of choice-point stack pointer
ITR  Top of installation trail pointer
CTR  Top of clean-up trail pointer
DP  Dispatching pool queue pointer
PB  Parallel choice-point pointer

Contents of a Choice-Point
There are two types of choice-points: parallel and sequential. Type of a choice point is calculated by a comparison of B and PB, if B == PB then parallel else sequential. Parallel choice points are linked by PB', the field in choice points.

OR-parallel
P'  Alternative clause
ST'  Alternative top of stack pointer
CTR'  Alternative top of clean-up trail pointer
ITR'  Alternative top of installation trail pointer
OA'  Number of alternative clauses left
AA'  Number of active paths
DE'  Associated dispatching element
AP'  Active processors
PB'  Parallel choice-point pointer

Sequential
P'  Alternative clause
ST'  Alternative top of stack pointer
CTR'  Alternative top of clean-up trail pointer
OA'  Number of alternative clauses left
Operations on a single variable

Binding a variable below a sequential choice-point we have to consider one more section of the stack (besides shared and private) - "non-shared" section, it is the part of the stack between the last sequential choice point and the last parallel choice-point. The bindings done to variables in that section are saved directly on the stack (as in WAM) and the address of a variable is trailed only on the clean-up trail.

bind(U,V,Pid) term *U,V, integer Pid
{if U << PB->ST then %in shared section
 *(ITR++) = <U,V>; %trail a pair
 if value("U") == U then %totally unbound
 {allocate a versions-vector and assign its address to vec;
  *U = tagged(VV,vec)} %bind U to versions-vector
 else
  vec = value("U"); %trail an address of v-vector
  *(CTR++) = vec;
  vec(Pid) = V} %bind a component
else if not(B == PB) and U << ST' then %in non-shared section
 {*(CTR++) = U;
  *U = V} %trail an address of variable %bind directly
else %in private section %bind directly

Scheduling operations

There is an additional argument "Parallel" in the create_choice-point procedure which defines the type of created choice-point.

create_choice-point(Pid, NumAlt, Parallel) integer Pid, NumAlt, bool Parallel
{allocate a choice-point and let B point to it;
 <P',ST',CTR',OA'> = <P,ST,CTR,NumAlt-1>;
 assign the address of the first alternative to P;
 if Parallel then
 {add element to the DP queue pointing to the choice-point and let
  DE' point to it;
  ITR' = ITR;
  AA' = 1;
  AP'(Pid) = true; %the first active processor
  PB' = PB; %update parallel ch-p pointer
  PB = B}

31
While backtracking only the last processor on the branch can execute an open alternative from a sequential choice-point. If processor is not the last one it proceeds to the next parallel choice-point.

```
backtrack(Pid, Alone) integer Pid, bool Alone
{if not(B == PB) then
  \%sequential choice-point
  untrail_vars(Pid,true);
  \%untrail and clean up
  \langle P,ST,CTR\rangle = \langle P',ST',CTR'\rangle;
  assign the address of the next alternative to P and P';
  OA' = OA'-1;
  if OA' == 0 then \%last open alternative
    remove the last choice-point;
    let B point to the previous choice-point}
else if OA' > 0 then \%B == PB, open parallel ch-point
  \langle OA' = OA'-1;\%
  if AA' == 1 then \%last processor
    \langle if OA' == 0 then \%untrail and clean up
      untrail_vars(Pid,true);
    else untrail_vars(Pid,false);
    \%just untrail
    \langle<P,ST,CTR,CTR,PB> = <P',ST',CTR',CTR',PB'>;
    assign the address of the next alternative to P and P';
    if OA' == 0 then \%last open alternative
      remove the element pointed by DE' from the DP queue;
      remove the last choice-point;
      let B point to the previous choice-point\}
else \%not last processor
  \langle untrail_vars(Pid,false);
  \%just untrail
  \langle<P,ST,CTR,CTR,PB> = <P',ST',CTR',CTR',PB'>;
  assign the address of the next alternative to P and P';
  if OA' == 0 then \%last open alternative
    remove the element pointed by DE' from the DP queue\}
else \%OA' == 0, closed parallel ch-point
  \langle if Alone then \%alone in its subtree
    AA' = AA'-1;
  if AA' == 0 then \%branch removed
    \langle untrail_vars(Pid,true);
    remove the last choice-point;
    let B point to the previous choice-point;
    backtrack(Pid,true))\}
else \%go on upwards, still alone
  \%subtrees below
  \langle untrail_vars(Pid,false);
  steal(Pid);
  \%look for work, return only if
  \%nothing to steal
  AP'(Pld) = false;
  let B point to the previous parallel choice-point;
  PB = PB->PB'; \%update parallel ch-point pointer
  backtrack(Pid,false))\})
```
steal(Pid) integer Pid

{pid' = find processors with non-empty DP(pid'), where pid' in AP', and choose the one
 whose DP(pid') contains the highest open choice-point;
 if pid' == none then
   {if B == top_of_tree then
     if all other processors waiting then
       terminate computation
   else wait until a parallel choice point is created
   return

   cross = B;

   B = get the address of an open choice-point from DP queue in pid';
   <P,ST,ITR,CTR,IBP> = <P',ST',ITR',CTR',IBP'>;

   assign the address of the next alternative to P and P';
   OA' = OA'-1;
   if OA' == 0 then
     {remove choice point from DP(Pid');
     DE' = none}

   AA' = AA'+1;

   install_task(B,cross,Pid)}

%to continue backtracking
%save address of cross ch-point B
%new active path
%add bindings and Pid

Procedure untrail-vars unbinds all the variables whose addresses are stored on the clean-up trail
between current CTR and CTR'. Considered values are versions vector's elements (variables in
shared section) or terms on the stack (variables in non-shared section).

untrail-vars(Pid,Last) integer Pid, bool Last

{if Last then
  {assign the address of the previous choice-point to chp;
   new_limit = chp->ST'

   while CTR >> CTR' do
     {x0 = *(CTR'--);
      if tag(*(x0)) == VV then
        {vec = value(*(x0));
         vec(Pid) = tagged(VAR,vec+Pid);
         if Last and x0 >> new_limit then
           deallocate the version vector pointed by vec (notice that all
           components are unbound)}
     else
       *(x0) = tagged(VAR,x0))}

%get address from trail
%versions vector
%unbind a component
%cleaning up
%value on the stack
%unbind

33
install-task(Bsteal, Bcross, Pid) pointer Bsteal, Bcross, integer Pid
{b1 = Bsteal;
 t1 = b1->ITR';
 loop
 {b2 = address to the parallel choice-point preceding b1;
  t2 = b2 ->TR';
  if b2->OA' == 0 and b2->AA' == 1 then %dead choice-point
   {Promote = true;
    b3 = address to the parallel choice-point preceding b2;
    limit = b3->ST'}
  while t1>> t2 do
   {<x0, value0> = *(t1);
    if Promote and x0 >> limit then
     {'(x0) = tagged(VAR,value0);
      ITR = ITR - <*(t1)>}
    else
     {vec = value('*(x0));
      vec(Pid) = value0}
    t1--}
  b1->AP'(Pid) = true;
  exit if b2 = Bcross;
  b1 = b2;
  t1 = t2 }}

initially(ProgramStart,Pid) pointer ProgramStart, integer Pid
{B = PB = top_of_tree;
  ST = stack_start;
  ITR = installation_trail_start;
  CTR = clean-up_trail_start;
  DP = all queues are initialized to empty;
  if Pid == chosen_processor_nr then
    P = ProgramStart
  else
    steal(Pid)
Appendix III. Versions-Vector WAM (Promotion, Sequential Choice-Points, Delayed Installation)

We show only these parts of the specification which are influenced by the delayed installation technique, which is explained in section 5.2.

 Registers
We add two more registers CP (cross point pointer) and NI (next installation pointer) which point to choice-points and are used during delayed installation. These two pointers are used to find part of the trail with not yet installed variables.

P Program pointer
ST Top of stack pointer
B Top of choice-point stack pointer
ITR Top of installation trail pointer
CTR Top of clean-up trail pointer
DP Dispatching pool queue pointer
PB Parallel choice-point pointer
CP Cross point pointer
NI Next installation pointer

Operations on a single variable
The installation procedure is invoked when a variable unbound in its version-vector is dereferenced and there are still variables to be installed.

deref(V,Pid) term *V, integer Pid
  {if tag(*V) == VAR then
    if not value(*V) == V then
      deref(*V,Pid)
    else
      V
  else if tag(*V) == NON-VAR then
    V
  else
    {mv = value(*V) + Pid;
     if not value(*mv) == mv then
       deref(*mv,Pid)
     else if not (NI == CP) then
       {install_task(NI,CP,Pid,V);
        deref(V,Pid);}
     else
      V }}

%unbound on stack
%tag(*V) == VV, variable bound
%to a versions-vector
%bound in version-vector
%unbound in versions-vector and
%there are vars to be installed
%install variables
%try again
%return pointer to stack
Scheduling operations

Procedure **steal** updates registers CP and NI without invoking install_task. CP is updated to a new "cross" point if CP is "younger" (was created later) than a new "cross" point, NI is updated to the current B. In that way we have the actual information about the part of the trail with not installed variables. There is a new procedure add_active adding the processor id to all choice-points on the way (done before by the procedure install_task).

**steal(Pid)** integer Pid

\( \text{pid}' = \text{find processors with non-empty DP(pid'), where pid' in AP', and choose the one whose DP(pid') contains the highest open choice-point; \}

if pid' == none then %not found

{if B == top_of_tree then

if all other processors waiting then

terminate computation

else wait until a parallel choice point is created

else

return} %to continue backtracking

cross = B; %save address of cross ch-point B

B = get the address of an open choice-point from DP queue in pid';
<P',ST',ITR',CTR',PB'> = <P',ST',ITR',CTR',PB'>;
assign the address of the next alternative to P and P';
OA' = OA'-1;
if OA' == 0 then

{remove choice point from DP(Pid');

DE' = none}

AA' = AA'+1; %new active path
add_active(B,cross,Pid); %add Pid to all AP' on the way
if CP >> cross then CP = cross; %updating cross point pointer
NI = B} %updating next installation pointer

**add_active**(Bsteal, Bcross, Pid) pointer Bsteal, Bcross, Integer Pid

{b1 = Bsteal;

loop

{b1->AP'(Pid) = true; %add processor

b1 = address to the parallel choice-point preceding b1;

exit if b1 = Bcross}
The procedure *install-task* will install all the variables on the relevant part of the trail up to the accessed variable *Var*, register *NI* will be updated then to the last passed choice-point.

*install-task*(Bsteal, Bcross, Pid, Var) pointer Bsteal, Bcross, integer Pid, term *Var

{b1 = Bsteal;
 t1 = b1->ITR';
 installed = false;
 loop
   {b2 = address to the parallel choice-point preceding b1;
    t2 = b2->TR';
    if b2->OA' == 0 and b2->AA' == 1 then %dead choice-point
      {Promote = true;
       b3 = address to the parallel choice-point preceding b2;
       limit = b3->ST'}
    limit of regained section
   while t1->>t2 do
    {<x0, value0> = *(t1);
     if Promote and x0 >> limit then
      {*(x0) = tagged(VAR,value0);
       ITR = ITR - <*>'(t1)>}
     %promotion
     else
      {vec = value*(x0));
       vec(Pid) = value0} %remove from installation trail
     %bind a component
     t1--; %accessed variable installed
     if x0 == Var then installed = true
    b1->AP'(Pid) = true;
    exit if b2 == Bcross or installed;
    b1 = b2;
    t1 = t2 }
    %add processor
}

NI = b2} %update next installation pointer

*initially*(ProgramStart,Pid)pointer ProgramStart, integer Pid

{B = PB = NI = CP = top_of_tree;
 ST = stack_start;
 ITR = installation_trail_start;
 CTR = clean-up_trail_start;
 DP = all queues are initialized to empty;
 if Pid == chosen_processor_nr then
  P = ProgramStart
 else
  steal(Pid)}
Appendix IV. Versions-Vector WAM (Promotion, Sequential Choice-Points, Delayed Installation, Favoured Bindings)

The possibility of using favoured bindings is just mentioned in section 6. In this section we explain and specify the technique. The optimization utilizes the idea of favoured bindings introduced in [3, 13, 14]. Shared section of the stack is divided now into favoured and non-favoured sections. Each favoured section belongs only to one "favoured" processor which created that section. Variables in favoured section are not bound to full versions-vectors, they are bound to a new data structure - reduced vector, with tag RV (Figure 10). Variables in non-favoured section are bound to the full versions-vector via the reduced vector (Figure 11). In that way we do not have to allocate full versions-vectors for the variables, which being potentially shared are not shared in reality. This optimization is very useful when processors after getting a big chunk of work to do, proceed independently without stealing tasks from each other.

Figure 10. Processor P3 binds a variable in it's favoured section and reduced vector is allocated.

Figure 11. Processor P4 binds variable in it's non-favoured section and full versions-vector is allocated.
Registers
We add a register FS (favoured section pointer) which points to border of a favoured section.

P    Program pointer
ST   Top of stack pointer
B    Top of choice-point stack pointer
ITR  Top of installation trail pointer
CTR  Top of clean-up trail pointer
DP   Dispatching pool queue pointer
PB   Parallel choice-point pointer
CP   Cross point pointer
NI   Next installation pointer
FS   Favoured section pointer

Contents of a Choice-Point
OR-parallel
P'    Alternative clause
ST'   Alternative top of stack pointer
CTR'  Alternative top of clean-up trail pointer
ITR'  Alternative top of installation trail pointer
OA'   Number of alternative clauses left
AA'   Number of active paths
DE'   Associated dispatching element
AP'   Active processors
PB'   Parallel choice-point pointer

Sequential
P'    Alternative clause
ST'   Alternative top of stack pointer
CTR'  Alternative top of clean-up trail pointer
OA'   Number of alternative clauses left
Operations on a single variable
There is one more data object - reduced vector. Dereferencing and binding we have to consider if a variable is in a favoured or a non-favoured section of the stack. Now there are four classes of variable bindings: private, non-shared, favoured and non-favoured (Figure 12). Notice that the favoured pointer is not adjusted when a choice point is removed. Adjusting the boundaries of the favoured section is not as critical in our model as in Argonne (hash window based model). Having the favoured section we save only the time and space during the allocation of the versions-vectors while in Argonne model the favoured section principle minimizes the variable access time (the main overhead of the model). In our model the variable access is always a constant time operation.

\[
\text{deref}(V, Pid) \text{ term } \ast V, \text{ integer Pid} \\
\{ \text{if tag}(\ast V) = \text{ VAR then} \\
\quad \text{if not value}(\ast V) = V \text{ then} \\
\qquad \text{deref}(\ast V, Pid) \\
\quad \text{else} \\
\qquad V \\
\text{else if tag}(\ast V) = \text{ NON-VAR then} \\
\qquad V \\
\text{else if V } << \text{ FS then} \\
\quad \{ \text{nv = value}(\ast V) + 2; \\
\quad \text{mv = value}(\ast \text{mv}) + \text{pid;} \\
\quad \text{if not value}(\ast \text{mv}) = \text{mv then} \\
\qquad \text{deref}(\ast \text{mv}, Pid) \\
\quad \text{else if not(NI } = \text{ CP) then} \\
\qquad \{ \text{install_task}(\text{NI, CP, Pid}, V); \\
\qquad \text{deref}(V, Pid) \\
\quad \text{else} \\
\qquad V\} \}
\]

Figure 12. Four sections of the stack of processor P2 and corresponding stack tree.

\text{deref}(V, Pid) \text{ term } \ast V, \text{ integer Pid} \\
\{ \text{if tag}(\ast V) = \text{ VAR then} \\
\quad \text{if not value}(\ast V) = V \text{ then} \\
\qquad \text{deref}(\ast V, Pid) \\
\quad \text{else} \\
\qquad V \\
\text{else if tag}(\ast V) = \text{ NON-VAR then} \\
\qquad V \\
\text{else if V } << \text{ FS then} \\
\quad \{ \text{nv = value}(\ast V) + 2; \\
\quad \text{mv = value}(\ast \text{mv}) + \text{pid;} \\
\quad \text{if not value}(\ast \text{mv}) = \text{mv then} \\
\qquad \text{deref}(\ast \text{mv}, Pid) \\
\quad \text{else if not(NI } = \text{ CP) then} \\
\qquad \{ \text{install_task}(\text{NI, CP, Pid}, V); \\
\qquad \text{deref}(V, Pid) \\
\quad \text{else} \\
\qquad V\} \}

\%bound or unbound on the stack \\
\%tag(\ast V) = RV, bound to reduced \\
\%vector in non-favoured section \\
\%bound in version-vector \\
\%unbound in versions-vector and \\
\%there are vars to be installed \\
\%install variables \\
\%try again \\
\%return pointer to stack
else

{nv = value("V") + 1;
 if not value("nv") == nv then
deref("nv,Pid")
else
  V}
%

While binding in the favoured section we put pointer to reduced vector of the variable on the clean-up trail.

\textbf{bind(U,V,Pid)} term *U,V, integer Pid
{if U << PB->ST' then
  \text{\%in shared section}
  \{*(ITR++) = <U,V>;
  \text{\%trail a pair}
  if value("U") == U then
    \text{\%totally unbound}
    \{allocate reduced vector and assign its address to rvec;
    *U = tagged(RV,rvec)\textbf{\%bind U to reduced vector}
  else
    rvec = value("U");
  if U << FS then
    \text{\%non-favoured section}
    \{if value(rvec(2)) == rvec+2 then
      \text{\%unbound}
      \{allocate versions-vector and assign its address to vvec;
      rvec(2) = tagged(VV,vvec)\textbf{\%bind second element of reduced
      \%vector to versions-vector}
    else
      vvec = value(rvec(2));
      vvec(Pid) = V\}
    else
      rvec(1) = V;
      *(CTR++) = rvec\}
  else if not(B == PB) and U << ST' then
    \{*(CTR++) = U;
    *U = V\}
  else
    \text{\%in private section
      *U = V\}

\textbf{Scheduling operations}

\textbf{create_choice-point}(Pid, NumAlt, Parallel) integer Pid, NumAlt, bool Parallel
{allocate a choice-point and let B point to it;
<P',ST',CTR',OA'> = <P,ST,CTR,NumAlt-1>;
assign the address of the first alternative to P;
if Parallel then
  \{add element to the DP queue pointing to the choice-point and let
  DE' point to it;
  ITR' = ITR;
  AA' = 1;
  AP'(Pid) = true;
  PB' = PB;
  PB = B\}
%the first active processor
%update parallel ch-p pointer
%

41
backtrack(Pid, Alone) integer Pid, bool Alone
    {if not(B == PB) then %sequential choice-point
        {untrail_vars(Pid, true);
            %untrail and clean up
            <P, ST, CTR> = <P', ST', CTR'>;
            assign the address of the next alternative to P and P';
            OA' = OA' - 1;
            if OA' == 0 then
                {remove the last choice-point;
                    let B point to the previous choice-point}
        }
    else if OA' > 0 then %B == PB, open parallel ch-point
        {OA' = OA' - 1;
            if AA' == 1 then %last processor
                {if OA' == 0 then
                    untrail_vars(Pid, true)
                    %untrail and clean up
                else
                    untrail_vars(Pid, false);
                    %just untrail
                    <P, ST, ITR, CTR, PB> = <P', ST', ITR', CTR', PB'>;
                    assign the address of the next alternative to P and P';
                    if OA' == 0 then
                        {remove the element pointed by DE' from the DP queue;
                            remove the last choice-point;
                            let B point to the previous choice-point}
                }
            else %not last processor
                {untrail_vars(Pid, false);
                    %just untrail
                    <P, ST, ITR, CTR, PB> = <P', ST', ITR', CTR', PB'>;
                    assign the address of the next alternative to P and P';
                    if OA' == 0 then
                        %last open alternative
                        remove the element pointed by DE' from the DP queue
                    }
            else %OA' == 0, closed parallel ch-point
                {if Alone then
                    AA' = AA' - 1;
                    if AA' == 0 then
                        {untrail_vars(Pid, true);
                            remove the last choice-point;
                            let B point to the previous choice-point;
                            backtrack(Pid, true)
                        } else
                            {untrail_vars(Pid, false);
                                steal(Pid);
                                AP'(Pid) = false;
                                let B point to the previous parallel choice-point;
                                PB = PB' -> PB';
                                backtrack(Pid, false))}
                %go on upwards, still alone %alone in its subtree %branch removed %no subtrees below %untrail and clean up %subtrees below %just untrail %look for work, return only if %nothing to steal %Pid no longer active below %update parallel ch-point pointer %go on upwards, no longer alone

42
The procedure **steal** updates register FS (favoured section pointer). A processor stealing a new task sets the beginning of its favoured section to the point from which a new task was allocated.

**steal**(Pid) integer Pid

{$pid' = $\text{find processors with non-empty } DP(pid'), \text{ where } pid' \text{ in } AP'$, \text{ and choose the one}}$

{\text{whose } DP(pid') \text{ contains the highest open choice-point;}}

{\text{if } pid' \text{ == none then}}

{\text{if } B \text{ == top_of_tree then}}

{\text{if all other processors waiting then}}

{\text{terminate computation}}

{\text{else wait until a parallel choice point is created}}

{\text{return}}

{\text{to continue backtracking}}

{B = get the address of an open choice-point from DP queue in pid';}

{$<P,ST,ITR,CTR,PB> = <P',ST',ITR',CTR',PB'>;}$

{FS = ST';}

{\text{assign the address of the next alternative to } P \text{ and } P';}

{OA' = OA'-1;}

{\text{if } OA' \text{ == 0 then}}

{\{\text{remove choice point from } DP(Pid');}}

{DE' = none\}}

{AA' = AA'+1;}

{\text{add_active}(B,cross,Pid);}

{\text{if } CP >> cross \text{ then } CP = cross;}

{\text{if } NI < B \text{ then } NI = B\}}

Elements of the clean-up trail point now to the reduced vectors instead of versions vectors as before. While cleaning up we deallocate both reduced vectors and versions vectors pointed from the clean-up trail.

**untrail-vars**(Pid,Last) integer Pid, bool Last

{\text{if } Last \text{ then}}

{\{\text{assign the address of the previous choice-point to } chp;}}

{\text{new_limit = chp->ST'}

{\text{while } CTR >> CTR' \text{ do}}

{\{x0 = "(CTR'--);}}

{\text{if } tag("(x0)) \text{ == RV then}}

{\{rvec = value("(x0));}}

{\text{vvec = value(rvec(2));}}

{\text{if } x0 < FS \text{ then}}

{\{vvec(Pid) = tagged(VAR,vvec+Pid) \}}

{\text{else}}

{\{rvec(1) = tagged(VAR,rvec+1); \}}

{\text{if } Last \text{ and } x0 >> \text{ new_limit then}}

{\{rvec(2) = rvec+2; \}}

{\text{deallocate the reduced vector pointed by } rvec \text{ and versions vector}}

{\text{pointed by } vvec(\text{notice that all components are unbound)}\}}}

{\text{else}}

{\{*(x0) = tagged(VAR,x0))\}}

{\text{\%value on the stack}}

{\\text{\%unbind}}
install-task(Bsteal, Bcross, Pid, Var) pointer Bsteal, Bcross, integer Pid, term *Var
{b1 = Bsteal;
t1 = b1->ITR;
installed = false;
loop
{b2 = address to the parallel choice-point preceding b1;
t2 = b2->ITR;
if b2->OA' == 0 and b2->AA' == 1 then
{Promote = true;
b3 = address to the parallel choice-point preceding b2;
limit = b3->ST'}
%dead choice-point
while t1>>t2 do
{<x0, value0> = *(t1);
if Promote and x0 >>= limit then
{*(x0) = tagged('VAR,value0);
ITR = ITR - <*(t1)>}
%promotion
else
{vec = value('*(x0));
vec(Pid) = value0}
%bind a component
t1--;
if x0 == Var then installed = true}
%accessed variable installed
b1->AP'(Pid) = true;
exit if b2 == Bcross or installed;
b1 = b2;
t1 = t2
}N1 = b2%
%update next installation pointer
}

initially(ProgramStart,Pid)pointer ProgramStart, integer Pid
{B = PB = N1 = CP = top_of_tree;
ST = FS = stack_start;
ITR = installation_trail_start;
CTR = clean-up_trail_start;
DP = all queues are initialized to empty;
if Pid == chosen_processor_nr then
P = ProgramStart
else
steal(Pid)