# The Muse Approach to Or-Parallel Prolog

Khayri A. M. Ali and Roland Karlsson
Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden

5 December 1994

### Abstract

*Muse* (*Mu*lti-*se*quential Prolog engines) is a simple and efficient approach to Or-parallel execution of Prolog programs. It is based on having several sequential Prolog engines, each with its local address space, and some shared memory space. It is currently implemented on a 7-processors machine with local/shared memory constructed at SICS, a 16-processors Sequent Symmetry, a 96-processors BBN Butterfly I, and a 45-processors BBN Butterfly II. The sequential SICStus Prolog system has been adapted to Or-parallel implementation. Extra overhead associated with this adaptation is very low in comparison with the other approaches. The speed-up factor is very close to the number of processors in the system for a large class of problems.

The goal of this paper is to present the Muse execution model, some of its implementation issues, a variant of Prolog suitable for multiprocessor implementations, and some experimental results obtained from two different multiprocessor systems.

**Key Words:** Or-Parallelism; Prolog; Multiprocessors; Experimental results; Scheduling.

# 1 Introduction

Logic programs offer many opportunities for the exploitation of parallelism, the main ones being Or-parallelism and And-parallelism(13). Or-parallelism allows the clauses of a predicate to be tried in parallel, while And-parallelism allows the goals of a clause body to be executed in parallel. There are two forms of And-parallelism: Independent And-parallelism, and Stream And-parallelism. In Independent And-parallelism goals that do not share any variable are executed in parallel(15,25,30). Stream And-parallelism allows goals that share variables to be executed in parallel with the value of the shared variable being communicated incrementally between the goals(18,37,42).

In Prolog, Or-parallelism and Independent And-parallelism are much easier to implement efficiently than Stream And-parallelism. This paper presents an approach that exploits Or-parallelism in Prolog programs. One of the main problems with implementing Or-parallelism is how to manage efficiently different bindings of the same variable corresponding to different branches of a Prolog search space. Several Or-parallel models have been proposed(3,6,8,12,22,27,38,44), incorporating different binding schemes. Binding schemes for combining Or-parallelism with And-parallelism have also been developed(7,14,28,45). Recent approaches attempt to exploit Stream And-parallelism with retaining the full search capability of Prolog(19,21,34,39,46).

Other important issues on efficient implementation of Or-parallel Prolog are scheduling work with minimizing speculative computations, and supporting the Prolog side-effects and cut(2,23,24,29).

## 1.1 History of BC-Machine Research

In 1986, a research project at SICS, called BC-machine, was started with the goal of finding a suitable approach to Or-parallel execution of Prolog programs on distributed memory multiprocessor machines. This class of machines is able to be scalable and to utilize faster processing units. Or-parallelism is easy to detect and exists in a large class of applications (e.g., natural language processing(26), expert systems(32), theorem proving(31), meta-interpreters(19), etc.).

Two ideas have been proposed by the BC-machine group; the Multi-sequential-machines approach(1) and the BC-machine approach(3)[1]. Both ideas are based on having multiple sequential Prolog engines. This allows keeping all advantages of the sequential Prolog technology, e.g., efficient implementation, garbage collection, etc. The main drawbacks of the former approach(1) are the overhead of 1) copying an engine state, and 2) load balancing. In the BC-machine approach(3), the load balancing problem has been solved assuming some shared memory in the system and copying overhead is reduced by using a special broadcast network for parallel copying.

[1] Notice that the Muse approach presented here is different from the one reported in Ref. 1. Actually, the one presented here is more closer to the approach reported in Ref. 3 than the one reported in Ref. 1 as it will be clear in Section 2.

### 1.2 Motivations to the Muse Approach

The initial thought was that copying overhead could be very high and hardware support, as broadcast network in the BC-machine, is essential for an efficient implementation. When we did our first experiment by implementing an Or-parallel Prolog system based on the BC-machine model on a simple prototype system without having any broadcast network, we obtained a very surprising result about copying overhead. Copying overhead is actually much lower than could be expected. It ranges from 0.01% to 25% of processors' time depending on programs. This percentage is small in comparison with the corresponding overhead (total task switching overhead) in other Or-parallel execution models. In Muse we reduce the copying overhead further to make the model suitable for a larger class of multiprocessor machines.

### 1.3 Current Implementations and Results

Currently, we have two Muse Or-parallel Prolog systems. The first Muse system supports a variant of Prolog, called Commit Prolog, with cavalier commit(9) instead of cut and asynchronous (parallel) side-effects instead of synchronous (sequential) ones. The standard Prolog semantics of cut and sequential side-effects can be obtained on Commit Prolog by following a few rules that restrict the degree of Or-parallelism. The first Muse system has been implemented on a 7-processors prototype system constructed at SICS, and on a 16-processors Sequent Symmetry machine. The prototype architecture is similar to ACE of IBM(17). It consists of a number of processing units, each provided with local memory, some shared memory, and a common bus. The sequential SICStus Prolog (version 0.6)(11), a fast, portable system, has been adapted to Or-parallel implementation. The extra overhead associated with this adaptation is low, around 5%.

The second Muse system is under development. It supports the full Prolog language. It currently runs on the constructed hardware prototype, Sequent Symmetry machine, a 96-processors BBN Butterfly I (GP1000), and a 45-processors BBN Butterfly II (TC2000)[2]. The two Muse systems are also being ported into uniprocessor Unix workstations. The preliminary results of the second Muse system on the four multiprocessor machines indicate that the extra overhead associated with adapting the sequential SICStus0.6 Prolog to Or-parallel implementation is still low in comparison with the other approaches. It is around 5% for the constructed prototype and Sequent Symmetry. The results also indicate that the speed-up factor is very close to the number of processors in the system for a large class of problems.

The major part of this paper is concerned with the first Muse system. Some preliminary performance results of the second Muse system on Sequent Symmetry and Butterfly machines will be presented in Section 11. Some parts of this paper has been presented in Ref. 4.

[2] Shyam Mudambi at the Brandeis University who has ported Muse system into the BBN Butterfly I and II machines.

### 1.4 Structure of the Paper

The paper is organized as follows. Section 2 describes the Muse execution model. Section 3 shows how the Muse model can be implemented as a minimal extension to the WAM. Section 4 discusses the characteristics of the Muse model. Section 5 discusses very briefly scheduling work principles of Muse. Section 6 specifies a version of Prolog that we call Commit Prolog and shows how to get full Prolog semantics using Commit Prolog. Section 7 discusses and presents very important optimizations of copying overhead. Section 8 discusses garbage collection in Muse. Section 9 presents some experimental results on two different multiprocessor systems. It also compares our results with the results obtained from related approaches. Section 10 describes briefly our graphical tool for tracing parallel execution and for debugging our implementations. Section 11 concludes the paper and discusses our plans for the continued development of Muse.

## 2 Muse Execution Model

In this section, we describe the Muse Or-parallel execution model in two steps: first the basic model, and then a method for reducing the main source of overhead in the basic model.

### 2.1 Multiprocessor System Assumptions

We assume a multiprocessor system having a number of processors or processes, called *workers*[3], with identical local address spaces, and some global address space shared by all workers. Operating systems like DYNIX (a parallel version of Unix) on Sequent machines and MACH on a number of multiprocessor machines support these requirements. We also assume that each worker is a sequential Prolog engine with its own local four stacks: *a choicepoint stack, an environment stack, a term stack, and a trail.* The first two correspond to the WAM local stack and the second two correspond to the WAM heap and trail respectively[(43)]. The stacks are not shared between workers. The Prolog program is either stored in the shared space or in each worker's space. It is assumed that the reader is familiar with the WAM[(43)].

### 2.2 Execution Model

1. When execution of a Prolog program starts, one worker P processes the top level query creating all data structures in its own stacks. The other workers are idle until P creates local nodes (choicepoints).

2. One of the idle workers Q interrupts P requesting work.

3. P allows Q to get a piece of work by sharing its local nodes with Q. P makes its local nodes shared with Q as follows.

[3] We use the name worker to represent process or processor exactly as it is used in Aurora[(32)].

   (a) For each local node, P creates a shareable frame in the shared space.
   (b) P moves information describing the unprocessed alternatives from the local nodes to the corresponding shareable frames.
   (c) Each local node acquires a pointer to the corresponding shareable frame.
   (d) P copies its state to Q. At this moment, P and Q have identical states and both share all of P's nodes (see Figure 1).

4. P and Q work together to finish processing the shared unprocessed alternatives (tasks). They process the shared nodes from the bottom-most up to the root node using the built-in backtracking mechanism of Prolog. P proceeds with its current task while Q simulates failure to release the next alternative (task) from the bottom-most shared node. Releasing an alternative from a shared node is done while locking the corresponding shared frame.

5. When a worker gets a task, it processes that task exactly as a standard sequential Prolog engine with the normal backtracking mechanism.

6. When a worker P creates local nodes and there is an idle worker Q, P makes its local nodes shareable with Q as in step (3).

7. When all work on a Prolog search tree is processed, all workers become idle and the execution terminates.

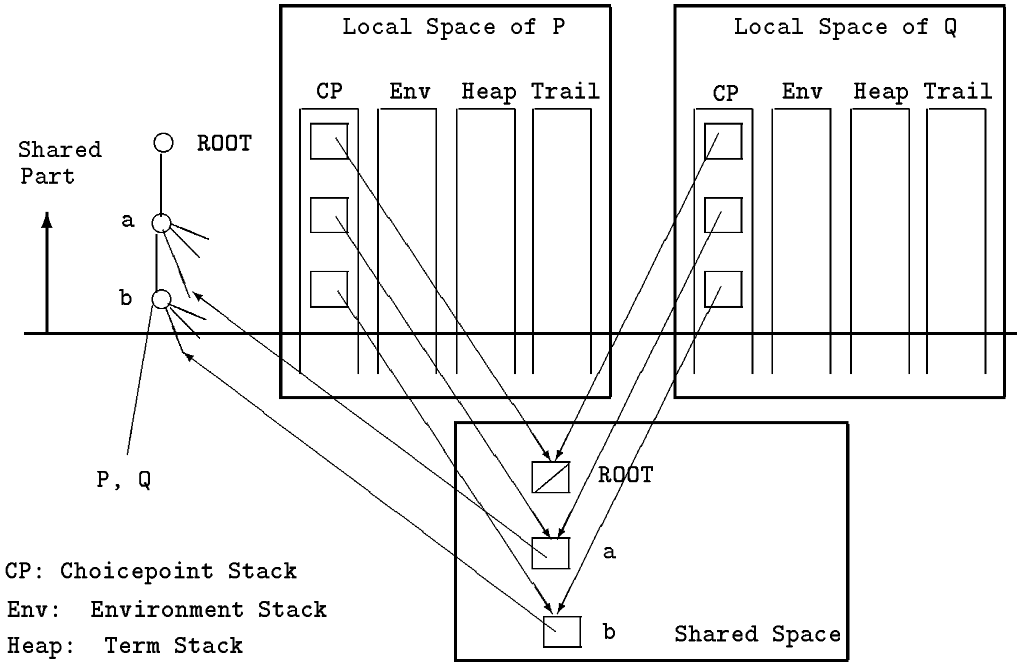

Figure 1: Sharing Work.

So far, this model is the same as the BC-machine model[(3)] with only one difference in step (3.d). In the BC-machine model, P copies its state not only to Q but to all idle workers in one operation assuming a broadcast copying network in the system.

Copying the whole worker state on every copy is expected to be an expensive operation. In the following section, we will illustrate how the copying overhead can be reduced substantially.

### 2.3 Incremental Copying

The main drawback of the above model is the overhead of copying a worker state in step (3.d). Actually, what we would like to achieve is simply that Q gets the same state as P. The idea of incremental copying is to make Q keep the part of its state which is consistent with P's state, and P copies only the difference between the two workers' states. Before showing how this can be supported efficiently, we would first like to briefly review the backtracking mechanism of Prolog.

On creation of a choicepoint $c$, the current state of computation is stored in $c$. The state of computation is represented by contents of the WAM registers. On processing an alternative which is not the last one from $c$, addresses of variables created before $c$, that get bound during execution of the alternative, are stored in the trail stack. On backtracking to $c$ for getting the next alternative, the state of computation is restored from $c$ and all modifications to variables created before $c$ are undone by going through the trail stack.

Now how to implement the incremental copying idea efficiently? Assume that there is no work available at any shared node on the current branch[4] of a worker Q and that there is another worker P with local nodes. Assume also that Q has selected P for sharing its local nodes. First, the worker Q backtracks to the youngest shared node $n$ with P (see Figure 2). Then P makes its local nodes shareable with Q and copies to Q only the differing parts between the two states. The differing parts are the top segments of stacks corresponding to nodes younger than $n$ and the local modifications that are made on the common parts after creating $n$. These modifications are known to P by accessing the top segment of its trail stack (see Figure 2). Now, P and Q have identical states and both share all of P's nodes exactly as in step (3.d) in the above section. Copying overhead is further reduced in Section 7.

## 3 Extending the WAM

In this section we describe the memory organization of the system and show how the Muse Or-parallel execution model has been implemented as a minimal extension to the WAM[(43)].

The Muse Or-parallel execution model is based on having multiple sequential Prolog engines, each with its own local address space, and some global address space shared by all engines. Each sequential Prolog engine is a worker with its own four stacks. Workers share a segment of memory for storing frames associated with shared nodes, global tables (e.g., symbol table, predicate table, etc), asserted rules, buffering instances of terms generated during parallel execution of *setof* and *bagof*, and some global registers. The shared memory segment is administrated as a general heap.

[4] The current branch is the path from the current node up to the root of the search tree.

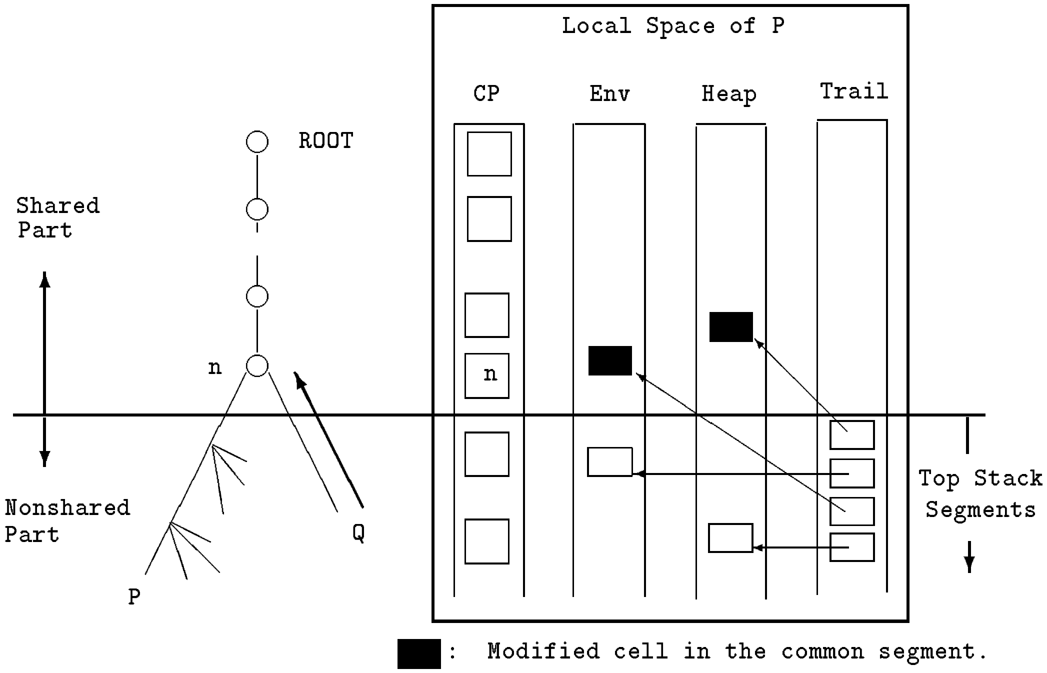

Figure 2: Incremental Copying Idea.

Part of the shared memory segment is maintained as a free list of fixed size blocks for frames associated with the shared nodes. Such frames are allocated by a worker who is going to make its local (nonshared) nodes shareable with another worker and are deallocated by the last worker backtracking from the node. There is one shared frame associated with every shared node. Each shared frame is referenced from the corresponding (local) choicepoint frames. Each shared frame basically contains a pointer to the next alternative of the associated shared node, a bit-map indicating workers which are at and below the node, and a lock for synchronizing accesses to the node.

The only modification to the WAM stacks is that one extra field (word) is added to each choicepoint frame. This word is used for two different purposes depending on whether the node is shared or not. For a shared node, the word contains a pointer to the next younger (child) choicepoint frame in order to find quickly the child node on cut/commit operations. (Notice that a shared node has logically many children but from a single worker perspective, there is only one child which corresponds to the current branch.) For a nonshared node $n$, the word contains the number of remaining alternatives of the nonshared nodes older than $n$. This number is used as a measure of the local load of each worker for guiding the scheduler to make good decisions on selecting a busy worker for sharing when there is an idle worker in the system. (There is a global register associated with each worker containing the current load of the corresponding worker.)

Another modification to the WAM is that on every procedure call, a worker checks for interrupt signals sent by other workers. The two possible interrupt signals are a request to make local nodes shareable, and abort the current task as part of cut/commit operation.

All the WAM stacks are located at a fixed address in local address space of every worker. This allows copying a segment of a stack from one worker to another without relocation of pointers.

A nice feature of the operating systems that are available to us (DYNIX on Sequent Symmetry, a mini Unix on our hardware prototype, and MACH on the Butterfly machines), is that the local address space of each process (worker) could be mapped into a separate part of the global address space of the system. This enables any worker to copy data directly into the local address space of another worker by using the global address of the latter. No extra storage is needed to buffer the copied information. For most bus based multiprocessors, this copying operation is supported efficiently.

## 4 Characteristics of the Execution Model

The Muse execution model provides a very high degree of locality of references. This property is crucial to any efficient execution model. There are two main reasons for the good locality of references in the Muse model. First, the WAM stacks, which represent the dominating part of accesses, are not shared among workers. Secondly, on incremental copying, top stack segments, which will be copied from a processor P to another processor Q, are always in P's cache before copying. After copying, each processor will work on its own copy of these segments in its own cache.

The Muse execution model is simple. It is very easy to adapt any sequential Prolog implementation to Or-parallel execution with very low extra overhead and with minimal effort. Since every worker is almost the same as the sequential Prolog engine, the speed of every worker is almost the same as the speed of the sequential Prolog engine. The Muse execution model keeps all advantages of the sequential Prolog technology, e.g., efficient compilation, indexing, unification with constant access time, stack based storage management, garbage collection, etc.

The Muse execution model is suitable for any multiprocessor system supporting local and global address spaces, and copying of memory blocks from one local space to another. Operating systems like DYNIX and MACH support these functions on a wide range of multiprocessor machines.

## 5 Scheduling Work

Nodes on the search tree are either shared or nonshared (local). They correspond to WAM choicepoints. These nodes divide the search tree into two parts: a shared part and a local part. Each shared node is accessible only to workers within the subtree rooted by the node. Local nodes are only accessible to the worker that created them. Each worker can be in either engine mode or in scheduler mode. The worker enters scheduler mode when it enters the shared part of the tree, or when it executes side-effects or *bagof*. In scheduler mode, the worker establishes the necessary coordination with other workers. The worker enters engine mode when it leaves scheduler mode. In engine mode, the worker works exactly as a sequential Prolog engine on local nodes, but is also able to respond to interrupt signals from other workers.

The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead. The sources of overhead in the Muse model include (1) copying a part of worker state, (2) making local nodes shareable, and (3) grabbing a piece of work (a task) from a shared node.

Our scheduling work strategies, that attempt to minimize the overhead, are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the number of shared work between the workers and allows each worker to release work from the bottom-most node on its branch (dispatching on the bottom-most) by using backtracking with almost no extra overhead.
- When a worker runs out of work from its branch it will try to share work with the nearest worker which has maximum load. The load is measured by the number of local unexplored alternatives, and nearness is determined from positions of workers on the search tree. This strategy attempts to maximize the shared work and minimize sharing overhead.
- Workers which cannot find any work in the system will try to distribute themselves over the tree and stay at positions where sharing of new work is expected to be with low overhead.
- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position on the tree before interrupting the busy worker for requesting sharing. This allows a busy worker to concentrate on its task and to response only to interrupts that have to be handled by it.

The basic algorithm of our scheduler for matching idle workers with available work is as follows. *When a worker finishes a task, it attempts to get the nearest piece of available work on the current branch. If none exists, it attempts to select a busy worker with excess local work for sharing. If none exists, it becomes idle and stays at a suitable position on the tree.* More detailed information about scheduling issues will be presented in another paper.

## 6 Commit Prolog

There are two main approaches concerning Prolog language for multiprocessor implementations:

1. Developing a new version of Prolog language suitable for multiprocessor systems as advocated by the ECRC group$^{(6)}$. Their main motivation is that the standard Prolog language is developed for uniprocessor systems.

2. Using the standard Prolog language as advocated by David H. D. Warren. His main motivation is that standard Prolog semantics is well accepted by the logic programming community.

Each approach has its own advantages and disadvantages. For instance, an advantage of the first approach is that simpler and more efficient implementations could be obtained. One disadvantage of the first approach is that not all existing Prolog programs can be executed with the standard Prolog semantics. In general, the user has to rewrite parts of his program in order to get the required semantics.

On the other hand, the second approach allows existing Prolog programs to be executed without modifications and getting the same standard Prolog semantics. It does not mean at all that all existing Prolog programs can get significant speed-ups when executed on multiprocessor systems. For Prolog programs that heavily use synchronous side-effects and cuts with large scopes, it is very difficult to get any speed-up. Supporting synchronous side-effects and cut with large scope efficiently always requires more complex implementations on multiprocessor systems.

In the first Muse system, a parallel version of Prolog, called *Commit Prolog*, is implemented. It is a Prolog language with cavalier commit[5] instead of cut, asynchronous (parallel) side-effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. Commit Prolog is simply an Or-parallel version of SICStus Prolog[(11)]. So, constructs like *setarg, freeze* and *undo*, that are supported by SICStus, are also supported in Commit Prolog with the same semantics.

The parallel annotation is used to annotate any predicate to allow clauses of the predicate to be executed in parallel whereas the sequential annotation is used to enforce clauses of a predicate to be processed sequentially. In our first system, by default all predicates are annotated sequential.

Cut semantics could be obtained on Commit Prolog by using commit and sequential annotations as it will be explained in Section 6.2. Sequential side-effect semantics could also be obtained on Commit Prolog by following the rules described in Section 6.1.

In Commit Prolog, the order of solutions collected by *bagof* (and *findall*) is arbitrary. That is,

```
:- bagof(X, p(X), L), bagof(Y, p(Y), L).
```

could fail sometimes, even when p(X) is solvable[(9)].

Commit Prolog is suitable for programs that do not use cuts with large scopes and that use sequential side-effects in the top level predicate. In many interesting benchmark programs that we have looked at, we found that most cut can be replaced by commit or cuts have small scopes. We found also that it is easy to get rid of side-effects or to use sequential side-effects in the top level predicate and parallel side-effects elsewhere. In general, it is possible to rewrite your program with the required style[(6,9)].

[5] Cavalier commit prunes branches both to the left and right of the committing branch, and is not guaranteed to prevent side-effects from occurring on the pruned branches[(9)].

### 6.1 Sequential Side Effects Semantics

When we cannot get rid of the sequential side-effects and internal database operations in a program, we could follow the following rules to get the sequential semantics of these operations. The basic idea here is to allow such operations when there is only one worker working on the tree.

The rules:

1. No sequential side-effects in a parallel predicate[6].

2. Parallel predicates do not call directly or indirectly any predicate with sequential side-effects.

3. No sequential side-effects follow parallel goals in a clause.

The next example shows a possible structure of the top level predicate. In the following examples, a *parallel-goal* is the goal that calls directly or indirectly a parallel predicate whereas a *sequential-goal* is the goal that does not call directly or indirectly a parallel predicate.

```
:- sequential p/0.
p :- parallel-goal1.                                     (1)
p :- side-effect1, sequential-goal1, side-effect2.       (2)
p :- sequential-goal2, side-effect3, parallel-goal2.     (3)
```

In this example, *p/0* clauses will be executed in the same textual order. First, all workers in the system could exploit Or-parallelism generated by execution of *parallel-goal1* in clause (1). Then, when all paths generated by clause (1) have been completely processed, one worker will process *side-effect1*, *sequential-goal1*, and *side-effect2* in clause (2). Then, one worker processes *sequential-goal2*, and *side-effect3* of clause (3). Last, Or-parallelism generated by execution of *parallel-goal2* in clause (3) could be exploited by all workers in the system.

From this example we find that side-effects are processed in the same order as in the standard Prolog. We find also that workers concentrate on the left-most subtree on the entire search tree reducing the possibility of doing speculative work. (Speculative work is the work which could turn out to be unnecessary.)

Here, the cost of getting sequential semantics of side-effects is to restrict parallelism which could result in reduced performance in some programs.

[6] Clauses of a parallel predicate could be processed in parallel while clauses in a sequential predicate must be processed sequentially.

### 6.2 Cut Semantics

When we cannot get rid of cut semantics in a program, we could follow the following rules to get cut semantics. The idea here is to turn off Or-parallelism within scope of cut.

The rules:

1. Every predicate p with cut semantics is annotated sequential.
2. No parallel goals precede the last cut in any clause of p.

The following example shows cut semantics by using sequential annotation and commit (!).

```
:- sequential p/0.
p :- parallel-goal1.                                                (1)
p :- sequential-goal1, !, sequential-goal2, !, parallel-goal2.     (2)
p :- ...                                                            (3)
p :- ...                                                            (4)
```

In this example also, clause (1) is processed first and all workers in the system could exploit Or-parallelism generated by *parallel-goal1*. Then, when all paths generated by clause (1) have been completely processed, processing of clause (2) starts by one worker that processes *sequential-goal1*, the first cut, *sequential-goal2*, and the **last** cut in clause (2). At this point, all choicepoints created after invoking *p* and also branches corresponding to clauses (3) and (4) are removed giving the correct semantics of cut. Now, workers in the system can exploit parallelism generated by *parallel-goal2* in clause (2).

Here also, the cost of getting cut semantics is to restrict parallelism. On the other hand speculative computations are avoided.

## 7 Reducing Copying Overhead

In this section we are going to reduce further the copying overhead which represents the main source of overheads associated with the Muse model. In Section 7.1, we show how to reduce copying overhead when the youngest local nodes are sequential nodes. In Section 7.2, we discuss a method that reduces copying overhead on a shared memory multiprocessor machine like Sequent Symmetry and Butterfly machines.

## 7.1 Reducing Overhead when Youngest Nodes are Sequential

A node on the search tree is either sequential or parallel. Parallel nodes have potential for Or-parallelism and could be processed in parallel while alternatives on a sequential node must be processed sequentially. That is, no alternative is utilized in the sequential node until all paths generated by an earlier alternative in the node have been completely processed.

A significant improvement of sharing and copying overhead is obtained when the youngest (local) nodes of the selected busy worker are sequential nodes. To illustrate the idea, assume that P is the selected busy worker which has the youngest nodes sequential and Q is an idle worker that has selected P for sharing. Assume also that P shares all its nodes with Q and copies the difference between the two workers states, and then P proceeds with its current task and Q simulates failure for getting a task from the nearest shared node. In this case, the worker Q will backtrack from all bottom-most sequential nodes and get an alternative from the bottom-most parallel node with available alternatives (the node *par* in Figure 3). So, sequential nodes, such as node *seq* in Figure 3, should not be shared and the corresponding state should (except the trail) not be copied from P to Q.

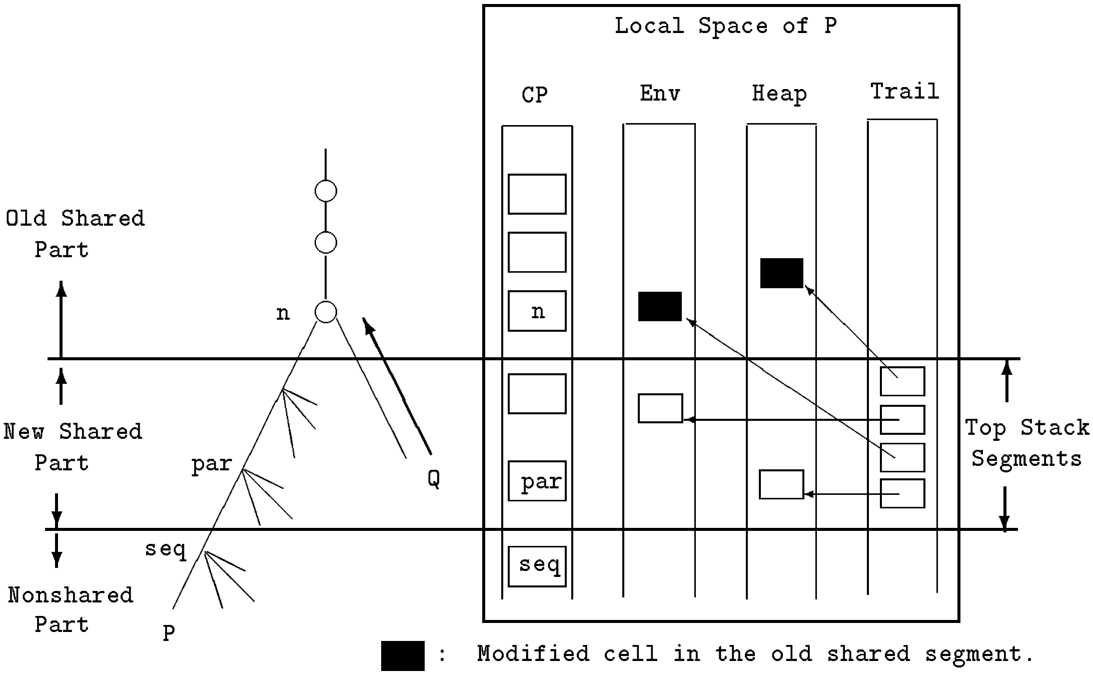

Figure 3: Reducing Copying Overhead.

This optimization is possible only when no *setarg* is invoked by P after taking the current alternative from the youngest parallel node. *setarg* is a backtrackable destructive assignment supported by SICStus Prolog$^{(11)}$. The *term stack* contains a goal for undoing *setarg* on backtracking. In this case, P copies the whole top segment of the *term stack*.

## 7.2 Parallel Copying

A nice feature of the DYNIX operating system on Sequent Symmetry (and MACH on the BBN Butterfly machines) is that the local address space of each process (worker) could be mapped into a separate part of the global address space of the system. We take advantage of this by making any worker copy directly into another worker's address space using the global address of the latter. This avoids using extra storage for buffering the copied information and reduces copying overhead as will be discussed below. In Section 2.3, when an idle worker Q requests sharing from a busy worker P with excess local load, P performs the following operations while Q is waiting until P completes these operations (see also Figure 3):

1. Makes nonshared nodes with Q shareable (sharing).
2. Copies the top segments of its own stacks to Q (copying).
3. Installs bindings from its local *environment stack* (Env) and *term stack* (Heap) associated with (old) shared nodes into Q's corresponding stacks (installation).

Copying overhead is represented by the time spent by P performing the last two operations and the time spent by Q waiting until P completes performing these operations. Our goal here is to minimize P's time spent on copying and to utilize Q's waiting time for copying as much as possible. Before describing the method of parallel copying, let us first investigate the three operations mentioned above (sharing, copying, and installation) and issues that arise when P's time on copying is minimized.

- Sharing is the only operation that must be performed by P.
- Sharing and copying (except the *choicepoint stack* (CP) segment) can be performed simultaneously. Sharing updates only the CP segment.
- Installation starts after copying the Env and Heap segments. Q must first get its own copy of the Env and Heap segments.
- Copying of the CP segment must be started after sharing has been completed.
- P cannot proceed with its current branch after sharing before the Env and Heap segments[7] (and the Trail segment, called Top-Trail, corresponding to the youngest sequential nodes discussed in Section 7.1[8]) are copied.
- P should not do either garbage collection or *setarg* before copying and installation are completed.
- Also, P cannot backtrack from any of the new shared nodes before copying and installation are completed. (Notice that P has to complete processing all available work on the local sequential nodes and the bottom-most parallel node, *par* in Figure 3, before backtracking from a shared node.)

[7] P should not modify its own copy of Env and Heap segments before getting another copy for Q.

[8] This part of Trail segment could be modified by P during processing the local sequential nodes and Q needs this part for undoing bindings in the copied Env and Heap segments.

The following method for parallel copying effectively utilizes the time of P and Q. Q performs copying while P is performing sharing. Q copies P's stack segments in the following order: Heap, Env, Top-Trail, remaining Trail segment, and CP. Q starts copying of the CP segment only when P has completed sharing. When copying is completed, Q performs installation. If Q has not completed copying of the Env, Heap, and Top-Trail segments when P has completed sharing, P will help Q to perform any of these operations. Then P proceeds with its current branch (i.e., executing Prolog). If P is going to backtrack from a shared node or to execute *setarg*, or to perform garbage collection, and Q has not completed copying the CP and Trail segments, or installation, P will help Q perform any of these operations.

## 8 Garbage Collection

One of the most important properties of the Muse approach is that no global garbage collection is needed. The reason is that none of the WAM stacks is shared between workers. So, any worker can do its local garbage collection asynchronously without involving another worker in its garbage collection operation.

Incremental copying presented in Section 2.3 is based on avoiding copying of a common state of the two workers involving on copying. If one of the two workers has performed garbage collection after the previous copying, finding a common part of the two workers' state is a problem. In this section we discuss two solutions to this problem.

The idea of the first solution is that when one of the two workers has performed a garbage collection since last copying, we copy the whole worker state. Otherwise, we copy the differing part. This simple solution is in general inefficient.

The second solution, which is much more efficient, is based on segmented garbage collection[(5)] as a local garbage collection scheme. Segmented garbage collection only collects garbage from segments that were not garbage collected before. The idea of the second solution could be illustrated by a simple system having two workers: W1 and W2. Assume that W1 has performed its first segmented garbage collection before copying its state to W2. After copying, W1 and W2 have segments that are garbage collected. Assume also that any of the two workers (or both) has performed a new segmented garbage collection. Then a new copy will be performed between them. The common part between the two workers which is garbage collected by W1 before the first copy will not be copied, because the already garbage collected segment will not be garbage collected on the new invocation of garbage collection. That is, no objects are moved in the already garbage collected segments (see below in Section 8.2 for updating pointers from old garbage collected segments into new garbage collected segment). So, based on Segmented garbage collection as a local garbage collection scheme, common parts corresponding to the same garbage collection invocation will not be copied. In the rest of this section we discuss the implementation of each solution.

### 8.1 First Solution

Every *g*arbage *c*ollection (*gc*) has a unique name. Every worker keeps the name of its last *gc*. When a worker P gets a signal from a worker Q for sharing and copying, P checks the *gc* name of Q with its *gc* name. If they are identical, P allows incremental copying. Otherwise, P allows total copying.

Unique names for every *gc* invocation can be implemented by using either a global counter with atomic increment or the worker identification number and a local counter.

### 8.2 Second Solution

Every *gc* has a unique name as before. The name of the current *gc* is stored in choicepoints residing in the garbage collected segment. When a worker Q requests sharing (and copying) from a busy worker P, and the *gc* names associated with the bottom-most common node are identical, sharing and incremental copying will be performed as usual. If the *gc* names are not identical, incremental copying will be done to the nearest common node in which P and Q have identical *gc* names. (The nearest common node with a different *gc* name is that node.) Along with incremental copying the pointers from the old garbage collected segments to the copied Heap stack segment should be updated on Q's old segment. (Segmented *gc* updates pointer cells in old garbage collected segments that point to objects in the new garbage collected segment(5).) The trail segment corresponding to the old garbage collected segment contains pointers to these cells. Such cells should also be installed for Q.

## 9 Experimental Results

In this section we present results of our experiments for a large group of benchmarks running on a shared memory multiprocessor machine (the Sequent Symmetry with 16 processors), and on a 7 processors machine with local/shared memory.

### 9.1 Benchmarks

The group of benchmarks used in this paper can be divided into two sets: the first set (*8-queens1, 8-queens2, tina, salt-mustard, parse2, parse4, parse5, db4, db5, house, parse1, parse3, farmer*) have relatively well understood granularity and ideal speed-up characteristics, and have been used by other researchers in previous studies(6,10,41). Thus, they allow measurement of basic overheads and comparison with previous results. This set contains benchmarks with coarse grain parallelism (*8-queens1, 8-queens2, tina, salt-mustard*), with medium grain parallelism (*parse2, parse4, parse5, db4, db5, house*), and with fine grain parallelism (*parse1, parse3, farmer*). *8-queens1* and *8-queens2* are two different 8 queens programs from ECRC. *tina* is a holiday planning program from ECRC. *salt-mustard* is the "salt and mustard" puzzle from Argonne. *parse1* – *parse5* are queries to the natural language parsing parts of Chat-80 by F. C. N. Pereira and D. H. D. Warren. *db4* and *db5* are the data base searching part of the fourth and fifth Chat-80 query. *house* is the "who owns the zebra" puzzle from ECRC. *farmer* is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC.

The second set of benchmarks (*pundit, semigroup, satchmo, andorra-interp*) contains large (except *satchmo*), real, programs with very interesting Or-parallelism. *pundit* is a natural language system from the Unisys Paoli Research Center$^{(26)}$. *semigroup* is a theorem proving program for studying the R-classes of a large semigroup from Argonne National Laboratory$^{(31)}$. *satchmo* is a little theorem prover for proving theorems in Predicate Logic. The original *satchmo* is written by Rainer Manthey and Francois Bry from ECRC, and modified by Lee Naish at the University of Melbourne. *andorra-interp* is an interpreter for the Andorra computational model written by Seif Haridi at SICS$^{(19)}$. All the benchmarks look for all solutions of the problem.

## 9.2 Performance Results: Timings and Speedups

### 9.2.1 On Sequent Symmetry

Table I presents the run times (in milliseconds) from the execution of the first set of benchmarks (described in Section 9.1) on a 16 processors Sequent Symmetry S81 with 32 Mbytes of memory. The run times given are the shortest obtained from several runs (as measured in Refs. 10 and 41). Times are shown for 1, 4, 8, 12, 15 workers with speed-ups (relative to the 1 worker case) given in parentheses.

| Goals | Muse Workers | | | | | |
|---|---|---|---|---|---|---|
| *repetitions | 1 | 4 | 8 | 12 | 15 | SICStus0.6 |
| 8-queens1 | 6910 | 1740(3.97) | 880(7.85) | 599(11.53) | 490(14.10) | 6770(1.02) |
| 8-queens2 | 17540 | 4419(3.97) | 2240(7.83) | 1510(11.61) | 1209(14.51) | 16450(1.07) |
| tina | 14580 | 3730(3.91) | 1920(7.59) | 1330(10.96) | 1099(13.27) | 13780(1.06) |
| salt-mustard | 2120 | 531(3.99) | 270(7.85) | 189(11.22) | 159(13.33) | 2020(1.05) |
| parse2 *20 | 5980 | 1780(3.36) | 1329(4.50) | 1160( 5.16) | 1149( 5.20) | 5870(1.02) |
| parse4 *5 | 5510 | 1500(3.67) | 920(6.00) | 770( 7.15) | 740( 7.45) | 5400(1.02) |
| parse5 | 3900 | 1049(3.72) | 589(6.62) | 459( 8.50) | 449( 8.69) | 3820(1.02) |
| db4 *10 | 2440 | 669(3.65) | 400(6.10) | 309( 7.90) | 289( 8.44) | 2240(1.09) |
| db5 *10 | 2970 | 819(3.63) | 480(6.19) | 370( 8.03) | 341( 8.71) | 2730(1.09) |
| house *20 | 4390 | 1331(3.30) | 840(5.23) | 721( 6.09) | 689( 6.37) | 4220(1.04) |
| parse1 *20 | 1579 | 620(2.55) | 579(2.73) | 610( 2.59) | 640( 2.47) | 1570(1.01) |
| parse3 *20 | 1360 | 581(2.34) | 519(2.62) | 540( 2.52) | 569( 2.39) | 1340(1.01) |
| farmer *100 | 3199 | 1399(2.29) | 1419(2.25) | 1419( 2.25) | 1429( 2.24) | 3060(1.05) |
| all-goals | 72478 | 20390(3.55) | 12599(5.75) | 10219( 7.09) | 9461( 7.66) | 69350(1.05) |

Table I: Run Times (in milliseconds) of Muse for the First Set of Benchmarks on Sequent Symmetry.

The last column shows running times on SICStus version 0.6 (SICStus0.6) with the ratio of running times on Muse for the 1 worker case and the SICStus0.6 times. For benchmarks with small run times the timings shown refer to repeated runs, the repetition factor being shown in the first column. *all-goals* in the last row corresponds to the goal: (*8-queens1, 8-queens2, tina, salt-mustard, parse2*20, parse4*5, parse5, db4*10, db5*10,* *house*20, parse1*20, parse3*20, farmer*100)*. That is, the timings shown in the last row correspond to running the whole first set of the benchmarks as one benchmark.

Figure 4: Exploited Parallelism by 15 Workers over Time for the *all-goals* Benchmark Measured by the Muse Graphical Tracing Facility.

Figure 4 shows the exploited parallelism by 15 workers over time for the *all-goals* benchmark, measured by *Mus*e graphical *t*racing facility (*Must*)(40). The whole area under the curve is divided into 13 parts corresponding to the 13 subgoals in the *all-goals* benchmark with the same order as they are listed above. Figure 4 shows also subareas corresponding to coarse, medium, and fine grain parallelism. (The difference between the execution time in Figure 4 and the corresponding time in Table I should be explained. The Muse version used in Figure 4 is slower because it generates trace information.)

Table II presents the run times (in seconds) for 1 and 15 workers from the execution of the second set of the benchmarks on Sequent Symmetry. The test sentences of the *pundit* program are listed in Table III.

| Benchmark | Muse Workers | |
|---|---|---|
| (or Goal) | 1 | 15 |
| pundit: | | |
| 1.1.1 | 64.31 | 4.54(14.17) |
| 4.1.1 | 406.98 | 27.88(14.60) |
| 4.1.3 | 49.81 | 3.73(13.35) |
| 5.1.2 | 35.56 | 2.62(13.57) |
| 6.1.2 | 130.02 | 9.16(14.21) |
| 9.1.1 | 180.25 | 12.42(14.51) |
| 9.1.4 | 368.89 | 25.43(14.00) |
| 22.1.1 | 61.78 | 4.41(14.01) |
| 25.1.3 | 35.64 | 2.87(12.42) |
| 28.1.1 | 145.22 | 10.92(13.30) |
| 31.1.3 | 63.16 | 4.49(14.07) |
| 31.1.4 | 67.30 | 4.87(13.82) |
| satchmo | 112.38 | 7.83(14.35) |
| semigroup | 4503.64 | 300.56(14.98) |
| andorra-interp | 96.24 | 6.93(13.95) |

Table II: Run Times (in seconds) of Muse for the Second Set of Benchmarks on Sequent Symmetry.

| Test Sentences | |
|---|---|
| 1.1.1 | starting air regulating valve failed. |
| 4.1.1 | while diesel was operating with sac disengaged, the sac lo alarm sounded. |
| 4.1.3 | pump will not turn when engine jacks over. |
| 5.1.2 | disengaged immediately after alarm. |
| 6.1.2 | inspection of lo filter revealed metal particles. |
| 9.1.1 | sac received high usage during two becce periods. |
| 9.1.4 | loud noises were coming from the drive end during coast down. |
| 22.1.1 | loss of lube oil pressure during operation. |
| 25.1.3 | suspect faulty high speed rotating assembly. |
| 28.1.1 | unit has excessive wear on inlet impellor assembly and shows high usage of oil. |
| 31.1.3 | erosion of impellor blade tip is evident. |
| 31.1.4 | compressor wheel inducer leading edge broken. |

Table III: Test Sentences of Pundit Benchmark.

### 9.2.2 On Constructed Hardware Prototype

Tables IV and V present the Muse run times from the execution of the benchmarks on a 7-processors hardware prototype constructed at SICS. The results shown in these tables correspond to a portable version of Muse on Sequent after porting to the hardware prototype without any significant modifications. That is, better results could be obtained than those in Tables IV and V when the Muse system takes advantages of the hardware prototype.

| Goals | Muse Workers | | |
|---|---|---|---|
| *repetitions | 1 | 4 | 6 |
| 8-queens1 | 11411 | 2873(3.97) | 1943(5.87) |
| 8-queens2 | 28955 | 7274(3.98) | 4890(5.92) |
| tina | 24039 | 6186(3.89) | 4267(5.63) |
| salt-mustard | 3520 | 1034(3.40) | 913(3.86) |
| parse2 *20 | 9094 | 2723(3.34) | 2470(3.68) |
| parse4 *5 | 8416 | 2310(3.64) | 1820(4.62) |
| parse5 | 5931 | 1599(3.71) | 1165(5.09) |
| db4 *10 | 3711 | 1033(3.56) | 820(4.53) |
| db5 *10 | 4524 | 1275(3.55) | 963(4.70) |
| house *20 | 7085 | 2196(3.23) | 1775(3.99) |
| parse1 *20 | 2421 | 950(2.55) | 1007(2.40) |
| parse3 *20 | 2073 | 883(2.35) | 931(2.23) |
| farmer *100 | 5226 | 2318(2.25) | 2683(1.96) |
| all-goals | 116757 | 33142(3.52) | 26438(4.42) |

Table IV: Run Times (in milliseconds) of Muse for the First Set of Benchmarks on the Hardware Prototype.

| Benchmark | Muse Workers | |
|---|---|---|
| (or Goal) | 1 | 6 |
| pundit: | | |
| 1.1.1 | 94.47 | 16.54(5.71) |
| 4.1.1 | 596.27 | 102.21(5.83) |
| 4.1.3 | 74.09 | 12.98(5.71) |
| 5.1.2 | 52.68 | 9.31(5.66) |
| 6.1.2 | 188.35 | 33.20(5.67) |
| 9.1.1 | 264.00 | 45.30(5.83) |
| 9.1.4 | 540.05 | 92.63(5.83) |
| 22.1.1 | 89.73 | 15.67(5.73) |
| 25.1.3 | 51.65 | 9.14(5.65) |
| 28.1.1 | 211.90 | 37.60(5.64) |
| 31.1.3 | 92.22 | 16.10(5.73) |
| 31.1.4 | 98.55 | 17.35(5.68) |
| satchmo | 157.53 | 27.36(5.76) |
| semigroup | – | – |
| andorra-interp | 140.71 | 23.97(5.87) |

Table V: Run Times (in seconds) of Muse for the Second Set of Benchmarks on the Hardware Prototype.

The hardware prototype consists of 7 68020 CPUs, 7 2.5 MBytes (local memory), 4 MBytes (global memory) and VME bus as a common bus in the system. We have a plan to build a copying network to study the percentage of performance improvements when such a network exists in the system. All our results reported in Tables IV and V are from the prototype without the copying network. The VME bus is used to access global memory and for copying.

### 9.2.3 Discussion of Performance Results

The performance results that Tables I, II, III and V illustrate are encouraging: almost ideal speed-ups for programs with coarse grain parallelism, reasonable speed-ups for programs with medium grain parallelism, and low speed-ups for programs with fine grain parallelism.

The main reason for having some better results in Tables I and II than those in Tables IV and V is that the capacity of the Sequent Symmetry bus is much higher than the capacity of the VME bus. The VME bus is slower than Sequent Symmetry bus by a factor around 20. It neither supports caching nor simultaneous locking (test-and-set) operations. That is, in our hardware prototype the ratio between communication speed to processor speed is much lower than in the Sequent Symmetry machine. *salt-mustard* in Table IV has lower speed-ups in comparison with that in Table I. The reason for this is that *salt-mustard* heavily uses meta-calls which require looking up the predicate table which resides in global memory, and that the VME bus does not support caching. Table VI shows better results for a version of *salt-mustard*, called *sm-mixtus*, without meta-calls. This version is a direct translation of *salt-mustard* generated by the Mixtus partial evaluator(36). We could not run the *semigroup* benchmark on the hardware prototype because the *semigroup* code size is larger than the available memory.

| | Muse Workers | | |
|---|---|---|---|
| Program | 1 | 4 | 6 |
| sm-mixtus | 735 | 194(3.79) | 144(5.10) |

Table VI: Run Times (in milliseconds) of Muse for a Version of *salt-mustard* without Meta-calls on the Hardware Prototype.

On one processor, Muse is about 5% slower than SICStus0.6 (shown in Table I), the sequential Prolog system from which it is derived. This overhead is smaller than in Aurora(32), 24% (see also below), and in PEPSys(35), 30%. The 5% extra overhead of Muse is divided into two parts. 2.5% is for maintaining the current value of the local load in each worker, and for checking arriving requests from the other workers in the system. We do not know yet the source of the other 2.5% overhead; it may be due to the C compiler or to the DYNIX operating system.

## 9.3 Basic Overheads of Or-parallel Execution

In this section, we present and discuss briefly the time spent in the basic activities of a Muse worker on Or-parallel execution. A worker time is distributed over the following activities:

1. *Prolog:* time spent in executing Prolog, checking arrival of interrupt signals, and maintaining value of the local load up to date.
2. *Idle:* time spent in looking for a worker with excess local work when there is no available work in the shared nodes.

3. *Sharing:* time spent in making local nodes shared with other workers.

4. *Grabbing Work:* time spent in grabbing available work from shared nodes.

5. *Copying:* time spent in copying.

6. *Installation:* time spent in installation.

7. *Waiting:* time spent in synchronization with other workers on sharing and copying activities.

8. *Backtracking:* time spent in moving up (towards the root node and without including the Prolog backtracking time) within the shared region (shared part of a search tree).

9. *Others:* time spent in other activities like spin lock, signalling other workers on either requesting sharing or performing commit/cut to a shared node, etc.

Table VII shows the total time spent in each activity, with the percent of that time relative to the total time, for the first set of benchmarks (*all-goals* benchmark) described in Section 9.1. Times shown in Table VII have been obtained from an instrumented system of Muse on Sequent Symmetry. Those times include the time spent in the measurements. The times obtained from an instrumented system are longer than those obtained from an uninstrumented system by around 7 – 9%. We believe that the percentage of time spent in each activity obtained from the instrumented system reflects what is happening in the uninstrumented system.

| | Muse Workers | | | |
|---|---|---|---|---|
| Activity | 4 | 8 | 12 | 15 |
| Prolog | 78041( 89.8) | 80813( 73.7) | 81906( 61.4) | 83265( 53.9) |
| Idle | 3863( 4.4) | 16627( 15.2) | 33915( 25.4) | 49544( 32.0) |
| Sharing | 632( 0.7) | 1579( 1.4) | 2346( 1.8) | 2884( 1.9) |
| Grabbing Work | 932( 1.1) | 1909( 1.7) | 2474( 1.9) | 2917( 1.9) |
| Copying | 1377( 1.6) | 3589( 3.3) | 5231( 3.9) | 6481( 4.2) |
| Installation | 956( 1.1) | 2234( 2.0) | 3166( 2.4) | 3786( 2.4) |
| Waiting | 733( 0.8) | 1765( 1.6) | 2508( 1.9) | 3017( 2.0) |
| Backtracking | 227( 0.3) | 534( 0.5) | 738( 0.6) | 918( 0.6) |
| Others | 174( 0.2) | 666( 0.6) | 1221( 0.9) | 1790( 1.2) |
| Total | 86935(100.0) | 109716(100.0) | 133505(100.0) | 154602(100.0) |

Table VII: Total Times (in milliseconds) Spent in Basic Activities of a Muse Worker for the First Set of Benchmarks on Sequent Symmetry.

As mentioned in Section 9.1, this set of benchmarks contains four benchmarks with coarse grain parallelism, six with medium grain parallelism, and three with fine grain parallelism. This set also represents benchmarks with lack of parallelism (see Figure 4). Lack of parallelism explains the reasons of decreasing the percentage of the *Prolog* time and increasing the percentage of the *Idle* time when increasing the number of workers in Table VII. The summation of these two percentages is almost constant (only 3% difference) from 8 workers to 15 workers. The other seven activities represent the real overheads of

Or-parallel execution. These overheads increase from the 4 workers case to the 8 workers case by 5.3%, from the 8 workers to the 12 workers by 2.1, and from the 12 workers to the 15 workers by 0.9%.

A possible explanation for the increase of overheads when increasing the number of workers is shown in Table VIII, which shows the effect of increasing the number of workers on the number of tasks and task sizes (expressed as a number of procedure calls per task) for the *all-goals* benchmark. A task is a continuous piece of work executed by a worker. In Table VIII granularity of parallelism is decreased from the 4 workers case to the 8 workers case by a factor around 2 whereas from the 8 workers to the 12 workers by a factor 1.23, and from the 12 workers to the 15 workers by a factor 1.13.

| | Muse Workers | | | |
|---|---|---|---|---|
| | 4 | 8 | 12 | 15 |
| Total Number of Tasks | 14131 | 27449 | 34307 | 38724 |
| Procedure Calls per Task | 63 | 32 | 26 | 23 |

Table VIII: Average Number of Tasks and Task Sizes for the First Set of Benchmarks.

Tables VII and VIII illustrate that the overhead increases when reducing the granularity of parallelism. The increase of overhead is distributed over the last seven activities as hown in Table VII. Table VII illustrates also that the cost of *Copying* overhead, which was feared to be high, is quite acceptable; 4.2% on 15 workers. Similarly, the cost of manipulating shared nodes (represented by *Backtracking* and *Grabbing Work* overheads) is low.

### 9.4 Comparison with Aurora

We have chosen the Aurora Or-parallel Prolog system(32) to compare with our performance results for the following reasons: Aurora is one of the best Or-parallel Prolog systems existing today for shared memory multiprocessor machines; Aurora runs on the same Sequent Symmetry which is available to us; and Aurora has been constructed by adapting the same sequential Prolog system (SICStus).

Aurora is based on the SRI model for Or-parallel execution of Prolog(44). The idea of the SRI model is to extend the conventional WAM with a large binding array per worker and modify the trail to contain address-value pairs instead of just addresses. Each array is used by just one worker to store and access conditional bindings, i.e. bindings to variables which are potentially shareable. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around the tree. When a worker finishes a task, it moves over the tree to take another task. The worker starting a new task must partially reconstruct its array using the trail of the worker from which the task is taken.

It is early to do a full comparison between the Muse and Aurora systems at this stage of development. The Aurora system supports parallel implementation of cut and sequential side-effects whereas the Muse version reported in this paper does not support them (see below for performance results of a new version of the Muse system that supports full Prolog). Additionally, the Muse system supports garbage collection but Aurora does not. However, a preliminary comparison between the performance results of Muse and Aurora would be possible. We compare our results with the Aurora results for the first set of benchmarks (described in Section 9.1). Aurora results based on SICStus0.3 are found in Refs. 10, 32 and 41. The Muse system is based on SICStus0.6 which is faster than SICStus0.3 by a factor around 1.5. (SICStus0.6 is only about 1.8 times slower than Quintus Prolog, one of the fastest commercial Prolog system.) The current version of Aurora is also based on SICStus0.6. Since no new results are published on the current version of Aurora, we have measured the run times given in Table IX (with the Manchester scheduler) on the same Sequent Symmetry and under the same conditions as the results in Table I.

| Goals | Aurora Workers | | | | | |
|---|---|---|---|---|---|---|
| *repetitions | 1 | 4 | 8 | 12 | 15 | SICStus0.6 |
| 8-queens1 | 7831 | 2000(3.92) | 1010(7.75) | 689(11.37) | 559(14.01) | 6770(1.16) |
| 8-queens2 | 20700 | 5179(4.00) | 2600(7.96) | 1750(11.83) | 1411(14.67) | 16450(1.26) |
| tina | 18270 | 4700(3.89) | 2400(7.61) | 1680(10.88) | 1370(13.34) | 13780(1.33) |
| salt-mustard | 2490 | 630(3.95) | 319(7.81) | 229(10.87) | 189(13.17) | 2020(1.23) |
| parse2 *20 | 7029 | 2310(3.04) | 1689(4.16) | 1601( 4.39) | 1669( 4.21) | 5870(1.20) |
| parse4 *5 | 6520 | 1770(3.68) | 1280(5.09) | 1090( 5.98) | 1020( 6.39) | 5400(1.21) |
| parse5 | 4599 | 1331(3.46) | 890(5.17) | 770( 5.97) | 669( 6.87) | 3820(1.20) |
| db4 *10 | 2880 | 800(3.60) | 460(6.26) | 380( 7.58) | 340( 8.47) | 2240(1.29) |
| db5 *10 | 3500 | 980(3.57) | 570(6.14) | 469( 7.46) | 419( 8.35) | 2730(1.28) |
| house *20 | 5021 | 1480(3.39) | 940(5.34) | 809( 6.21) | 769( 6.53) | 4220(1.19) |
| parse1 *20 | 1851 | 740(2.50) | 710(2.61) | 770( 2.40) | 829( 2.23) | 1570(1.18) |
| parse3 *20 | 1590 | 699(2.27) | 699(2.27) | 740( 2.15) | 790( 2.01) | 1340(1.19) |
| farmer *100 | 3620 | 2110(1.72) | 2110(1.72) | 2260( 1.60) | 2390( 1.51) | 3060(1.18) |
| all-goals | 86211 | 25020(3.45) | 16020(5.38) | 13490( 6.39) | 12740( 6.77) | 69350(1.24) |

Table IX: Run Times (in milliseconds) of Aurora for the First Set of Benchmarks on Sequent Symmetry.

It can be seen from Tables I and IX that Muse is faster than Aurora in all benchmarks. Table X shows in the last row the ratio of the running times on Aurora to the running times on Muse for *”all-goals”* benchmark. Aurora timings are longer than Muse timings by 19% to 35% between 1 to 15 workers. (The extra overhead associated with supporting full Prolog in a new Muse system for this set of benchmarks is 0% for the 1 worker case, around 2% for the 4 workers, 3% for 8 workers, 4% for 12 workers, and 5% for 15 workers.) We believe that on shared memory multiprocessor machines like Sequent Symmetry Aurora could be faster than Muse for other classes of benchmarks in which sharing of the WAM stacks is more efficient than copying.

| | Workers | | | | |
|---|---|---|---|---|---|
| System | 1 | 4 | 8 | 12 | 15 |
| Aurora | 86211 | 25020 | 16020 | 13490 | 12740 |
| Muse | 72478 | 20390 | 12599 | 10219 | 9461 |
| Aurora / Muse | 1.19 | 1.23 | 1.27 | 1.32 | 1.35 |

Table X: Run Times (in milliseconds) of Aurora and Muse, and the Ratio between them for the *"all-goals"* Benchmark on Sequent Symmetry.

Since the Muse model does not require sharing of the WAM stacks, it is suitable for a larger class of multiprocessor machines. That is, the Muse model is also suitable for multiprocessor machines that do not support caching like BBN Butterfly machine, and for machines with local/global memory like ACE-IBM machine$^{(17)}$. Shyam Mudambi at the Brandeis University has ported a Muse system that supports full Prolog on Sequent Symmetry into the BBN Butterfly I and II machines. His preliminary results are very promising$^{(33)}$ (see also Section 11).

## 10 *MUST:* The *MUS*E Graphical *T*racing Facility

A graphical tracing tool for understanding the behavior of the Muse system has been developed. On parallel execution of a Prolog program, important events are generated, then a graphical tracing facility called *Must* replays those events on Unix and the X Window system to show the dynamic behavior of the Muse system during execution of the Prolog program$^{(40)}$. This tool is similar to the Argonne tool$^{(16)}$ but *Must* has other facilities: recording real time with each event, showing utilization of processors (workers) over time, showing the structure of the whole search tree, and tracing on a query level (and not for the entire session).

In Figure 5 we show a typical snapshot of parallel execution of the satchmo program by 8 workers on Muse. The *Must* user interface contains three views; *sideview* to the left showing the whole structure of the search tree with positions of workers as black dots, *timeview* near top and to the right showing utilization of workers on executing Prolog over time, and *mainview* showing a selected part of the search tree in some details. The *Must* user interface contains also **file name** in the top, **passed time/total time** in the next row to the left, and **obtained speedup so far/the number of used workers** in the same row to the right. Figure 5 shows a search tree when 17407 milliseconds have been passed from 20014 milliseconds and obtained speedup so far is 7.98 on a system with 8 workers. The *timeview* shows utilization of workers on executing Prolog after passing 17407 milliseconds. The *sideview* and *mainview* show the shared part of the search tree at that time.

Figure 5: Snapshot of the Muse Graphical Tracing Facility.

In the *mainview*, a shared node is represented by circle with either a vertical line inside for a sequential node, with a number inside for a parallel node indicating the number of unexplored branches, or filled black indicating no available work. Predicate name/arity appears to the right of each node. A local tree (task) processed by a worker X is represented by a triangle with X inside. Small black rectangle below a triangle of worker X indicates that the worker X has excess local work (untried branches). When a worker moves on a tree the worker number will indicate its position on the tree. For example, worker 3 in Figure 5 has finished its task and is backtracking to position itself in a suitable node on the tree before requesting sharing from a worker with excess local work. Figure 5 illustrates also that all workers except worker 3 are executing Prolog and workers 2, 4 and 5 have excess local work.

*Must* allows also printing out the traced events with relevant information on the standard output simultaneously with showing the search tree. Many performance and implementation bugs have been discovered very early by investigating these events, search tree, and time diagram in the *timeview*. It seems that without such a tool it is very difficult to develop a good scheduler.

# 11 Conclusions

We have presented a simple and efficient execution model, named Muse, to support Or-parallel Prolog on different machines. The Muse model is based on having a number of sequential Prolog engines, each with its local address space, some shared memory space, and copying memory blocks from one local address space to another. Operating systems like DYNIX and MACH support efficiently these functions on a wide range of multiprocessor machines.

The Muse execution model provides a very high degree of locality of references. This property is crucial to any efficient execution model. It also keeps all advantages of the sequential Prolog technology, e.g., efficient compilation, indexing, unification with constant access time, stack based storage management, garbage collection, etc. So, it is very easy to adapt any sequential Prolog implementation to Or-parallel execution with very low extra overhead and with minimal effort. (The Melbourne group is using the Muse model for adapting the NU-Prolog.)

The Muse model has been implemented on a 16-processors Sequent Symmetry, a 7-processors machine with local/shared memory constructed at SICS, a 96-processors BBN Butterfly I (GP1000), and a 45-processors BBN Butterfly II (TC2000). The sequential SICStus Prolog system has been adapted to Or-parallel implementations. Extra overhead associated with this adaptation is very low in comparison with the other approaches. It is around 5% for the constructed prototype and Sequent Symmetry. The preliminary results on the Butterfly machines indicate also low extra overhead(33). The speed-up factor is very close to the number of processors in the system for a large class of problems. We have also ported the Muse systems into uniprocessor workstations (Sun3 and Sun4). One advantage of that is that timing bugs could be discovered very early.

The performance results on Sequent Symmetry machine are compared with the results of the related approaches. Muse has the lowest extra overhead per worker and has similar or better relative speed-ups for a large group of benchmarks.

Several small and large Prolog programs have been ported to the Muse system that supports Commit Prolog with annotating parts of the programs that contain Or-parallelism as parallel. The standard Prolog semantics of cut and sequential side-effects has been obtained on Commit Prolog by annotating some predicates sequential. In a new version of the Muse system, cuts and sequential side-effects semantics are obtained without user annotations. Clauses of a predicate with cuts and sequential side-effects can be executed in parallel, and the correct semantics is maintained by the system. The extra overhead associated with supporting full Prolog in the new Muse system for the first set of benchmarks (described in Section 9.1) is 0% for the 1 worker case, around 2% for the 4 workers, 3% for 8 workers, 4% for 12 workers, and 5% for 15 workers. The existing Muse systems do not have any special treatment for speculative work(24). The amount of unnecessary speculative work is very small for this set of benchmarks, basically because it was chosen not to contain major cuts(41). Shyam Mudambi at the Brandeis University who has ported the new Muse system into Butterfly I and II. He got the following preliminary speedups on an N-queens benchmark: a factor 63.3 on a 70-processors BBN Butterfly I and a factor 29.7 on a 32-processors BBN Butterfly II(33).

Two important issues are not discussed in depth in this paper. The first issue is the scheduler which is responsible for coordinating the activities of multiple workers looking for work in a Prolog search tree and for supporting directly full Prolog. The second issue is an elaborated performance evaluation study of all sources of overhead associated with the Muse model. Those issues will be discussed in more detail in another paper.

The existing Muse systems allow researchers to experiment with Or-parallel Prolog programs on a number of multiprocessor architectures. We are working on improving the capabilities and performance of Muse. We have a plan to port Muse to other machines like the Encore machine and the Data Diffusion Machine prototype at SICS(20). An interesting issue would be to investigate the integration of the Muse Or-parallel model with an And-parallel model in order to exploit more parallelism in Prolog programs.

## 12 Acknowledgments

We are grateful to Mats Carlsson and Seif Haridi for many useful discussions. We would like also to thank Torbjörn Granlund for his help in the implementation of Muse on Sequent Symmetry, Shyam Mudambi at the Brandeis University for porting Muse system into the Butterfly machines, Jan Sundberg and Claes Svensson for constructing the graphical tracing facility for Muse.

## References

[1] Khayri A. M. Ali. Or-parallel Execution of Prolog on a Multi-Sequential Machine. International Journal of Parallel Programming, Vol. 15, No. 3, pages 189 – 214, June 1986.

[2] Khayri A. M. Ali. A Method for Implementing Cut in Parallel Execution of Prolog. In Proceedings of the 1987 Symposium on Logic Programming, pages 449 – 456, 1987.

[3] Khayri A. M. Ali. Or-parallel Execution of Prolog on BC-machine. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages 1531 – 1545, 1988.

[4] Khayri A. M. Ali and Roland Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In Proceedings of the 1990 North American Conference on Logic Programming, MIT Press, October 1990.

[5] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahin. Garbage Collection for Prolog Based on WAM. In Communications of the ACM, June 1988.

[6] Uri Baron, Jacques Chassin de Kergommeaux, Max Hailperin, Michael Ratcliffe, Philippe Ropert, Jean-Claude Syre, and Harald Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pages 841 – 850, ICOT, November 1988.

[7] Prasenjit Biswas, Shyh-Chang Su, and David Y. Y. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND parallelism (RAP) in Logic Programs. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages pages 1160 – 1179, MIT Press, August 1988.

[8] Ralph Butler, Ewing Lusk, Robert Olson, and Ross Overbeek. ANLWAM – A Parallel Implementation of the Warren Abstract Machine. Internal Report, Argonne National Laboratory, 1986.

[9] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages pages 1590 – 1605, MIT Press, August 1988.

[10] Alan Calderwood and Péter Szeredi. Scheduling Or-parallelism in Aurora – the Manchester scheduler. In Proceedings of the sixth International Conference on Logic Programming, pages 419 – 435, MIT Press, June 1989.

[11] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual. SICS Research Report R88007B, October 1988.

[12] William Clocksin. Principles of the DelPhi Parallel Inference Machine. Computer Journal, 30(5):386 – 392, 1987.

[13] John S. Conery. The And/Or Process Model for Parallel Interpretation of Logic Programs. PhD thesis, The University of California at Irvine, 1983. Technical Report 204.

[14] John S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors, International Journal of Parallel Programming 17(2), pages 125 – 152, April 1988.

[15] Doug DeGroot. Restricted And-parallelism. In Proceedings of the International Conference on Fifth Generation Computer Systems 1984. pages 471 – 478, Tokyo, November 1984.

[16] Terrenc Disz and Ewing Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In Proceedings of the 1987 Symposium on Logic Programming, pages 46 – 53, 1987.

[17] Armando Garcia and Richard F. Freitas. The ACE Multiprocessor Workstation. IBM Research Division, Thomas J. Watson Research Center, Hawthorne, NY 10598, 1988.

[18] Steven Gregory. Parallel Logic Programming in Parlog. Addison-Wesley, 1987.

[19] Seif Haridi and Per Brand. Andorra Prolog, an Integration of Prolog and Committed Choice Languages. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pages 745 – 754, ICOT, November 1988.

[20] David H. D. Warren and Seif Haridi. Data Diffusion Machine – a Scalable Shared Virtual Memory Multiprocessor. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pages 943 – 952, ICOT, November 1988.

[21] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In Proceedings of the seventh International Conference on Logic Programming, pages 31 – 46, MIT Press, June 1990.

[22] Bogumil Hausman, Andrzej Ciepielewski, and Seif Haridi. OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors. In Proceedings of the 1987 Symposium on Logic Programming, pages 69 – 79, 1987.

[23] Bogumil Hausman, Andrzej Ciepielewski, and Alan Calderwood. Cut and Side-Effects in OR-parallel Prolog. In Proceedings of the International Conference on Fifth Generation Computer Systems 1988, pages 831 – 840, ICOT, November 1988.

[24] Bogumil Hausman. Pruning and Speculative Work in OR-parallel Prolog. PhD thesis, Swedish Institute of Computer Science, Report No. TRITA-CS-9002, March 1990.

[25] Manuel Hermenegildo. An Abstract Machine for Restricted And-parallel Execution of Logic Programs. In Ehud Shapiro, Editor, Third International Conference on Logic Programming, London, pages 25 – 39, Springer–Verlag, 1986.

[26] Lynette Hirschman, William Hopkins, and Robert Smith. OR-parallel Speed-up in Natural Language Processing: A Case Study. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages pages 263 – 279, MIT Press, August 1988.

[27] Laxmikant V. Kalé. The Reduce-OR Process Model for Parallel Evaluation of Logic Programs. In Proceedings of the Fourth International Conference on Logic Programming, pages 616 – 632, MIT Press, May 1987.

[28] Laxmikant V. Kalé, B. Ramkumar, and W. Shu. A Memory Organization Independent Binding Environment for And and Or Parallel Execution of Logic Programs. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages 1223 – 1240, 1988.

[29] Laxmikant V. Kalé, David A. Padua, and David C. Sehr. Or Parallel Execution of Prolog Programs with Side Effects. The Journal of Supercomputing, 2(2): 209 – 223, October 1988.

[30] Yow-Jian Lin and Vipin Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages 1123 – 1141, 1988.

[31] Ewing Lusk and Robert McFadden. Using Automated Reasoning Tools: A Study of the Semigroup F2B2. Semigroup Forum, 36(1): 75 – 88, 1987.

[32] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. New Generation Computing, 7(2,3): 243 – 271, 1990.

[33] Shyam Mudambi. Personal communication, September 1990.

[34] Lee Naish. Parallelizing NU-Prolog. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages 1546 – 1564, MIT Press, August 1988.

[35] Michael Ratcliffe. A progress report on PEPSys. July 1988. Presentation at the Gigalips Workshop, Manchester.

[36] Dan Sahlin. The Mixtus Approach to Automatic Partial Evaluator of Full Prolog. In Proceedings of the 1990 North American Conference on Logic Programming, MIT Press, October 1990.

[37] Ehud Shapiro, editor. Concurrent Prolog-Collected Papers. MIT Press, 1987.

[38] Yukio Sohma, Ken Satoh, Koichi Kumon, Hideo Masuzawa, and Akihiro Itashiki. A New Parallel Inference Mechanism Based on Sequential Processing. In Proc. of Working Conference on Fifth Generation Computer Architecture, Manchester, July 1985.

[39] Zoltan Somogyi, Kotagiri Ramamohanarao, and Jayen Vaghani. A Stream AND-parallel Execution Algorithm with Backtracking. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, pages 1143 – 1159, MIT Press, August 1988.

[40] Jan Sundberg and Claes Svensson. MuseTrace: A graphic Tracer for Or-parallel Prolog. In preparation as SICS Research Report.

[41] Péter Szeredi. Performance analysis of the Aurora Or-parallel Prolog System. In Proceedings of the 1989 North American Conference on Logic Programming, pages 713 – 732, MIT Press, March 1989.

[42] Kazunori Ueda. Guarded Horn Clauses. PhD thesis, Graduate School, University of Tokyo, March 1986.

[43] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 290, SRI International, 1983.

[44] David H. D. Warren. The SRI Model for Or-parallel Execution of Prolog – Abstract Design and Implementation Issues, In Proceedings of the 1987 Symposium on Logic Programming, pages 92 – 102, 1987.

[45] Harald Westphal, Philippe Ropert, Jacques Chassin de Kergommeaux, and Jean-Claude Syre. The PEPSys Model: Combining Backtracking, And- and Or-parallelism. In Proceedings of the 1987 Symposium on Logic Programming, pages 436 – 448, 1987.

[46] Rong Yang and Vitor Santos Costa. Andorra-I: A System Integrating Dependent And-parallelism and Or-parallelism. TR-90-03, University of Bristol, March 1990.