ISBN 91-7170-083-8 ISRN SICS/D--7--SE

A High Performance OR-parallel Prolog System

Roland Karlsson

A Dissertation submitted
for the Degree of Doctor of Technology
Department of Telecommunication and Computer Systems
The Royal Institute of Technology
Stockholm, Sweden

March 1992

The Royal Institute of Technology (KTH)
Department of Telecommunication
and Computer Systems
S-100 44 Stockholm, Sweden

SICS
Swedish Institute
of Computer Science
Box 1263
S-164 28 Kista, Sweden

TRITA-TCS-LPS-9202
ISSN 0284-4397

SICS Dissertation Series 07
ISSN 1101-1355

A High Performance OR-parallel Prolog System

Roland Karlsson

Abstract

PROLOG, the most popular logic programming language, has been developed for single-processor computers. The implementations of sequential Prolog became efficient with the development of the Warren Abstract Machine (WAM) and are still improving. Today many parallel computers are commercially available and the question is how to utilize this opportunity to speed up the execution of Prolog programs.

Our approach has been to study and develop efficient techniques for implementing OR-parallel systems for the full Prolog language on UMA and NUMA computers. Based on those techniques a high performance OR-parallel Prolog system has been designed and implemented on six different parallel computers. The system has a number of processes, called *workers*, consisting of two components: the *engine*, which does the actual Prolog work, and the *scheduler*. The schedulers, working together, divide the available work between the engines and support the sequential semantics of Prolog.

We have extended the WAM, resulting in a Prolog engine with almost the same speed as the original WAM, i.e. the OR-parallel execution model preserves the high efficiency of the sequential implementation. We have designed and implemented a scheduler supporting dynamic load balancing and low overhead. We have also designed and implemented a clean interface between the engine and the scheduler, improving the modularity of the system and preserving its high efficiency.

We have developed tools for debugging and evaluating the system. The evaluation of the system on Sequent Symmetry and on BBN Butterfly machines I and II shows very promising results in absolute speed and also in comparison with results of the other OR-parallel systems.

A layered approach for building a simple and efficient OR-parallel Prolog system by extending any sequential Prolog implementation based on the WAM has been designed.

Descriptors

Logic Programming, Prolog, Abstract Machines, OR-parallel Execution, Scheduling, Multiprocessors, Implementation Techniques, Performance Evaluation, Graphical Tools.

Acknowledgments

To make this kind of work possible you have to rely on help from lots of people. People at work, in your home, and elsewhere. I am very grateful to all of them. There are some to whom I would like to give special thanks.

The invaluable support given by my advisor Khayri A. M. Ali has made this work possible. The proof reading was mainly done by Khayri, Torkel Franzén, and Mats Carlsson. My wife Lena and my children Erik, Andreas, and Sara have been an important support.

The work presented has been carried out at LPSLAB (the Logic Programming and Parallel Systems Laboratory) of the Swedish Institute of Computer Science (SICS).

SICS is sponsored by Asea Brown Boveri AB, NobelTech Systems AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defence Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

Table of Contents

List of Figures

List of Tables

Chapter 1

Prolog

SOME years ago I was working at the Royal Institute of Technology in Stockholm at the department for Telecommunication Systems. I was then doing research and studies in the field of high speed switching using integrated optical devices. A very natural choice as I originally studied physics at Stockholm University. I managed to get half a doctorate, or more precisely the Swedish degree Licentiate of Technology, before something interrupted my work. The department I was working at was closely connected with the department for Computer Systems and a certain dose of my education was supposed to be in that field. A very interesting and expanding field indeed. The event that ended my research in optics for telecommunication was a combination of two things: I took a Prolog course (given by Dan Sahlin) and SICS (the Swedish Institute of Computer Science) was founded. I was roped in by Seif Haridi (manager for the Logic Programming Systems Laboratory at SICS). The intention was that I should build a piece of special hardware for executing Prolog programs in parallel based on ideas by Khayri A. M. Ali. The hardware should support very efficient copying with broadcast functions for a set of processors. The machine (called the BC-machine) was built, the SICStus Prolog system was ported to the machine and parallelized, but the copying network was never built. The ideas and algorithms that emerged from the parallelization made the system so efficient that it was very difficult to motivate the hard work of constructing the special copying hardware. Moreover, a very successful porting of the system to a commercial shared memory machine, the Sequent Symmetry, even made the BC-machine more or less obsolete. Now my conversion from doing research and studies in the mixed field of physics and telecommunication to the field of computer systems was completed.

⋆ ⋆ ★ ⋆ ⋆

For you novices in the field of OR-parallel Prolog, myself, and maybe others who are interested, I will in this prolog explain the execution of sequential Prolog and OR-parallel Prolog. The example program used is very simple but it explains some of the more fundamental principles and concepts of the execution.

1.1 Sequential Execution

This section explains the execution of a pure Prolog program, using a simple program. Pure Prolog is a subset of Prolog without cut or side effects. The program is found in Figure 1.1. For a more complete description of Prolog and its execution, see for example [81]. In considering the example all optimizations, such as indexing and early fail, are ignored.

```
color(sky,blue).                % Database
color(sea,blue).
color(grass,green).

state(sky,gaseous).
state(sea,liquid).
state(grass,solid).

thing(Thing,Color,State) :-     % Program
    color(Thing,Color),
    state(Thing,State).

?- thing(X,blue,liquid).        % Query
```

Figure 1.1: A pure Prolog example.

The program consists of three predicates: color/2, state/2, and thing/3. The two first predicates form a small database describing the known world. A "thing" can have two properties: color and state. The predicate thing/3 uses the database to find valid triplets $\langle Thing, Color, State\rangle$. Executing the given query, the Prolog system tries to find a (known) "thing" X that is blue and in a liquid state.

The initial goal is "thing(X,blue,liquid)". This goal matches the head of the predicate thing/3. After head unification (replacing the variable Color with "blue", and the variable State with "liquid") the goal can be replaced with the body of the predicate. The Prolog system now tries to find a solution for the goal "color(X,blue), state(X,liquid)". The task is now to first find solutions for "color(X,blue)" and then (for each X) try to find solutions for "state(X,liquid)". There are three alternative clauses (facts) for the predicate color/2. On the Prolog depth-first search strategy only the first alternative is considered before going on to the subgoal state/2. Later on it may be necessary to examine the second and third alternatives. To make it possible to return to examine the second alternative a choicepoint, storing the computation state and the next alternative, is created. We call the choicepoint CP_{color}. Unification of the goal "color(X,blue)" with the head of the first clause of the predicate color/2 binds the variable X to "sky" (this information is stored in what is called the *binding environment*). The goal is then reduced to "state(sky,liquid)". There are three clauses that might match this goal so the Prolog system creates a new choicepoint (called CP_{state}). The goal "state(sky,liquid)" does not match the first clause so the system, after trying this clause, resets (backtracks) to the state stored in the choicepoint CP_{state}. Neither the second nor the third clause matches (the sky is not liquid) so the goal “state(sky,liquid)” eventually fails. The choicepoint CP_{state} is removed and then the system backtracks to the choicepoint CP_{color} and tries the second alternative of the predicate color. The goal to prove is then “state(sea,liquid)” where X is bound to “sea”. A new choicepoint (called CP_{state2}) is then created. At the second try the execution succeeds in reducing the goal to the empty goal, binding the variable X to the value “sea”.

The search for the solution can graphically be represented as a *tree* searched depth-first left to right from the *nodes* 1 to 9. This tree is shown in Figure 1.2. Eventually the search through this tree ends up with a solution where the variable X is bound to the value “sea” at node 9. There are some untried branches in the tree that don't have to be explored for a solution to be found. The search for more solutions can be restarted from node 9 if the Prolog system is forced to fail. In this example there are no more solutions.

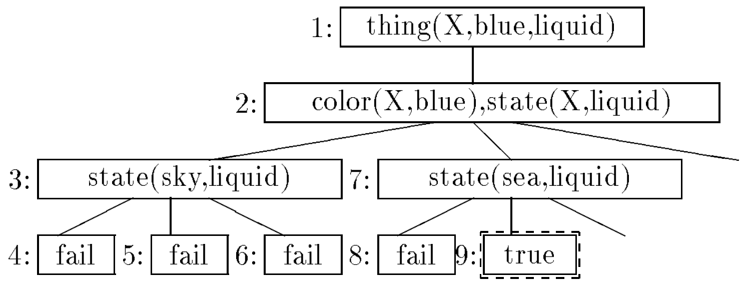


Figure 1.2: Search tree for the example in Figure 1.1.

1.2 Parallel Execution

As we have seen from this example, the branches of the search tree can be explored in parallel; at nodes 2, 3, and 7 there is potential for parallelism. Each predicate has a number of clauses and those clauses are independent of each other. This kind of parallelism is known as OR-parallelism.

The following text describes one scenario in an OR-parallel execution. As an example an extension of the search tree of the pure Prolog example above is shown in Figure 1.3. The nodes 10 and 11 have to be added because they might be reached in a parallel execution.

Let us assume that there are three processing units (*workers*) w1, w2, and w3. Let us also assume that all manipulating of information by a worker in a node is done atomically. One of the workers (say w1) is assigned the original query, and then it starts forward execution. The other two workers cannot find any work to do so they stay idle. After some time w1 reaches node 2. In this node there exist three

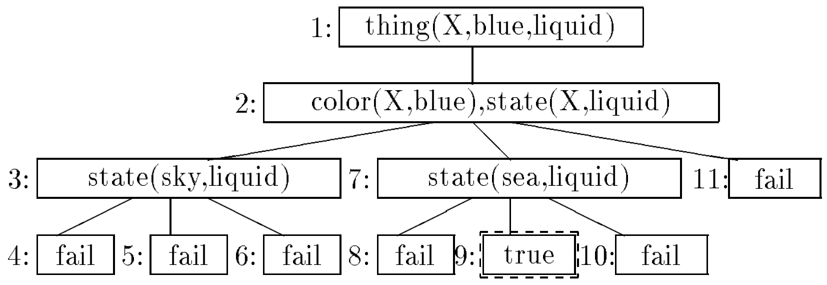


Figure 1.3: Maximal parallel search tree for the example in Figure 1.1.

possible alternative continuations. The worker w1 can then continue (as in the sequential execution) with the leftmost alternative. The other two workers could grab one alternative each. To be able to do so they have to get a computation state corresponding to node 2. Say that w2 grabs the second alternative and that w3 gets the last one. The worker w3 fails immediately at node 11. At the same time the other two workers reach the nodes 3 and 7. The worker w3 can now, using the same principle as above, grab an alternative from any of those two nodes. Eventually a worker will reach node 9 and find the solution.

Notice that not all nodes in the search tree must be visited for the solution to be found. If there is more than one solution to be found in a search tree, then the system finds the solutions in an indeterminate order. All solutions found by a sequential execution are also found in the parallel execution. If the program contains infinite loops then the parallel execution may find more solutions.

From the example it seems that OR-parallel execution of Prolog is very simple and does not face any problems. Actually there are problems, related both to efficiency and functionality, e.g.:

- How does a worker get a computation state?
- How does each worker maintain its binding environment?
- How can a worker find new work?
- How is the work distributed among the workers?
- How does one give the user the impression of using a sequential system?

This thesis will try to answer all those questions (and maybe others) through a simple and efficient OR-parallel system (called Muse).

Chapter 2

Introduction

THE most outstanding features of logic programming are its expressive power and its declarative semantics. The former leads to compact code that is easy to program and understand. The latter makes transparent parallelism possible and facilitates program transformations.

Prolog has become the most popular logic programming language due to its efficient implementations. It is a simple theorem prover based on first order logic. Given a program (also called theory) and a query, Prolog tries to satisfy the query using the program. At success the bindings of the query variables are reported. Prolog has the following features: (1) variables are *logical variables* which can be instantiated only once, (2) variables are *untyped* until instantiated, (3) variables are instantiated via *unification*, a pattern matching operation finding the most general common instance of two data objects, and (4) at unification failure the execution *backtracks* and tries to find another way to satisfy the original query. At backtracking Prolog investigates the different alternatives in the order in which they are listed in the program.

The first implementation of Prolog was an interpreter written by Robert Kowalski and Alain Colmerauer [59] at the beginning of the 70s as a part of research regarding natural language understanding. In 1977 David H. Warren made Prolog a viable language by developing the first compiler [90]. Current Prolog systems are based on the WAM (Warren Abstract Machine), an efficient execution model developed by Warren in 1983 [91]. Examples of commercial WAM based systems are Quintus [31] and "ProLog by BIM" [18], and examples of non-commercial systems are SICStus [26] and Sepia [66].

Prolog is highly suitable for symbolic processing applications. It has been used successfully in expert systems, natural language understanding, theorem proving, deductive databases, and compiler writing.

2.1 Parallel Prolog Implementations

With the availability of multiprocessors, many researchers have looked to parallel implementation of the Prolog language, developing a number of research prototypes. We are going to briefly review those prototype systems and mention some of the recent proposals.

The two main sources of parallelism in Prolog programs are known as *OR-parallelism* and *AND-parallelism* [29]. OR-parallelism allows different clauses of a predicate to be tried in parallel, whereas AND-parallelism allows goals in a clause body to be executed in parallel. AND-parallelism can be divided into *independent AND-parallelism*, where goals which do not share variables are allowed to run in parallel, and *stream AND-parallelism*, where one goal binding a variable acts as a producer, and another goal using the binding acts as a consumer.

The existing efficient parallel Prolog systems exploit only one form of parallelism, namely either independent AND-parallelism (as in [50]) or OR-parallelism (as in [65] and Chapter 3). This thesis is concerned with the latter.

The independent AND-parallelism approach was first taken by DeGroot [32], and then by Hermenegildo, who designed RAP-WAM, an extension of the WAM for independent AND-parallelism [50]. An alternative independent AND-parallel system has been developed by Lin and Kumar [61]. Data dependency analysis and compilation for parallel execution are important themes for AND-parallelism [69]. Abstract interpretation techniques are typically used for extracting data dependency information [70].

Exploiting stream AND-parallelism in Prolog-like languages has been recently considered. Naish extended the WAM for stream AND-parallel execution to support a variant of Prolog [71]. Conery's AND/OR process model was modified for stream AND-parallel execution in [80].

Stream AND-parallelism and OR-parallelism have been exploited in a parallel Prolog system, called Andorra I [16], developed at Bristol university. The IDIOM model [37] is a proposal for exploiting the three forms of parallelism: independent AND-parallelism, stream AND-parallelism, and OR-parallelism. Andorra I and IDIOM are based on the Andorra computational model originally proposed by David H. Warren and discussed in [42].

To our knowledge, there are only two OR-parallel Prolog systems that support the full Prolog language. The first one is Aurora [65], which has been developed as a collaborative effort between groups at Argonne National Laboratory, the University of Bristol, and SICS. The other one is Muse, which has been developed at SICS and is the subject of this thesis. The two systems are based on different approaches, but both are constructed by extending the SICStus Prolog system [26].

Combining OR-parallelism with independent AND-parallelism has been considered by a number of researchers. For example, the ROLOG system [73] based on the Reduce-OR model by Kalé [54] exploits independent AND-parallelism and OR-parallelism. The PEPSys system [14] developed at ECRC also exploits the two forms of parallelism. Neither the ROLOG system nor the PEPSys system supports the full Prolog language. Gupta and Jayaraman have proposed a model [39], which extends the binding arrays approach to handle independent AND-parallelism along with the OR-parallelism. Another proposal called ACE [38] extends the stack-copying model used in Muse to exploit the two forms of parallelism.

2.2 Problems in OR-parallelism

The two main problems associated with OR-parallelism are the efficient management of multiple bindings of a Prolog variable and the design of efficient schedulers.

2.2.1 Multiple Bindings

This problem is due to the fact that Prolog programs are non-determinate, i.e. a Prolog program may produce several alternative solutions corresponding to different ways of solving a problem. This implies that a variable can assume several different values depending on the branch of the execution tree in which it becomes bound. In a sequential execution the branches are traversed one at a time, and thus there can be no conflicts when binding a variable. This is not the case when an execution utilizes OR-parallelism. During OR-parallel execution several branches can be active at the same time, and thus a variable may have multiple, coexisting bindings. Maintaining multiple bindings plays a key role in efficient OR-parallel execution.

A recent paper by Gupta and Jayaraman [40] summarizes and gives a taxonomy for the proposed solutions of the problem of how to maintain different bindings in different branches. They defined three criteria for an ideal multiple bindings (or OR-parallel execution) scheme: (1) constant-time access to variables, (2) constant-time task creation, and (3) constant-time task switching. Their result is that all the three criteria cannot be simultaneously satisfied by any OR-parallel scheme based on a finite number of processors. Even a sequential WAM-like execution scheme is not ideal, because it provides constant-time environment creation and constant-time variable access and binding, but not constant-time task switching. The latter operation, known as backtracking in sequential implementation, requires the untrailing of variables, and the number of such variables is not constant in general. In this section we review briefly some of the proposed schemes.

The *copying* schemes allow each branch to have its physical copy of all bindings. *Environment closing* [30] is an example of a copying scheme, where variables are organized in such a way that all information needed for a resolution step is located in a single stack frame. This allows selectively copying a stack frame on every call. This means that task-creation is a non-constant time operation whereas variable-access and task-switching are constant-time operations

The other types of copying schemes are based on either copying the whole state when a process runs out of work (as in the *Kabu-Wake model* [60] and the *BC-machine model* [4]), or else copying only the difference between two processes' states (as in the *Muse model* described in Chapter 3).

Recomputation schemes [28, 2, 77] are related to the copying approaches but they are based on recomputing all data instead of copying it. The recomputation approach has problems with implementing side effects.

In the last class of copying schemes and in the recomputation schemes, task-switching is a non-constant time operation whereas variable-access and task-creation are constant-time operations. None of the schemes above has any problems with garbage collection. The schemes were first proposed for non-shared memory multi-processors, and some of them (e.g. the incremental copying of the WAM stacks used in Muse) have been implemented on shared memory machines with very interesting results.

Binding Arrays [89, 92] and *Version Vectors* [47] schemes are based on each process having a local copy of the conditional bindings in its branch. A binding is conditional if a variable can get several bindings. Unconditional bindings can be shared. For those schemes the trail stack stores address-value pairs instead of only addresses. Processes use the trail stacks for constructing their local copies of conditional bindings in their local binding arrays. In Binding Arrays and Version Vectors schemes, task-switching is a non-constant time operation whereas variable-access and task-creation are constant-time operations.

The *hash-window* schemes are based on sharing as much as possible to avoid either copying or reconstructing conditional bindings from the trail stacks. Each branch maintains a trail containing address-value pairs of conditional variables. To reduce the access time for those variables, a trail can be implemented as a hash table or a list of hash tables (hash-windows) [20, 14, 34]. On those schemes, variable-access is a non-constant time operation whereas task-creation and task-switching are constant-time operations.

The last two classes of schemes have been developed mainly for shared memory multiprocessors. Garbage collection is always difficult in those schemes.

2.2.2 Scheduling OR-parallelism

The second problem in OR-parallelism concerns efficiently maintaining the sequential semantics of Prolog and scheduling work between processes. This includes (1) efficient handling of cuts and side effects, (2) efficient support for all-solutions predicates (findall, setof, and bagof), and (3) reducing task-switching overheads (for systems based on a binding scheme with non-constant time task-switching).

Efficient handling of cuts and side effects is, in general, a difficult problem in OR-parallel Prolog systems due to the sequential semantics imposed by the depth-first left-to-right search strategy of Prolog. A cut prunes all branches to the right of a branch executing the cut that are generated by a predicate containing the cut. The sequential semantics of side effects restricts the parallelism and entails considerable overhead in implementations. A side effect operation could be suspended and resumed later on. It is not easy to implement suspension and resumption efficiently for systems based on a binding scheme with a non-constant time task-switching. In the thesis by Hausman [45], the problem of cut and side effects in OR-parallel Prolog has been investigated.

All-solutions predicates (e.g. *findall*) collect all solutions of a goal in a list. Solutions in the list should be identical to and occur in the same order as the ones generated by a sequential Prolog system. Again preserving the sequential order of solutions restricts the parallelism and entails overhead. Some parallel systems, like Aurora [25], have chosen not to preserve the sequential order for efficiency reasons, while the Muse system does preserve the sequential order. A novel and efficient approach has been developed and implemented in the Muse system.

Task-switching overheads depend very much on the chosen solution of the multiple bindings problem as mentioned in Section 2.2.1. For example, in hash-window based systems, like PEPSys [14], task-switching overheads are always very low whereas variable access is expensive. In Aurora, which is based on the binding arrays solution, and in Muse, which is based on incremental copying of the WAM stacks, the task-switching is not cheap whereas variable access in general is. In the latter two systems, task-switching overheads can be reduced by supporting dynamic load balancing and controlling the granularity of parallelism at run-time.

Techniques developed for the efficient scheduling of OR-parallelism and for maintaining the sequential semantics of Prolog are presented in Chapters 4 and 7.

2.2.3 Motivations and Goals

The three main motivations for investigating OR-parallelism in Prolog are (1) the easy to find transparent parallelism, (2) our belief in the possibility of an efficient implementation, and (3) the existence of OR-parallelism in a large class of programs.

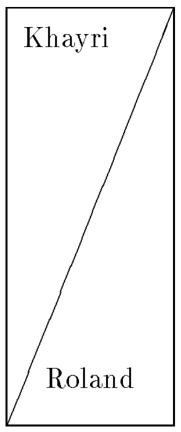
Our main goal was to make an efficient OR-parallel system that preserved the (full) Prolog semantics. The system should be able to execute ordinary Prolog programs without any user annotations faster than a sequential implementation. Our criteria have been simplicity and efficiency at all stages of the development, from the model to the actual implementation details.

2.3 Contribution of the Thesis

This thesis describes the results of joint work by me and Khayri A. M. Ali since 1986 at SICS. We have been working so closely together developing and implementing the Muse model that it is not easy to tell our contributions apart. I shall nevertheless try. First I have to give Khayri credit for being the one who originally believed in the copying model for OR-parallel Prolog. It was he who started the now very successful work against the “better” judgment of most other people in the field. During the development of the model and its implementation we became a team. One of us had an idea which was tested on the other one. After some minutes, hours, days, or weeks the idea was rejected or accepted. Accepted ideas were sometimes implemented. After an idea had been accepted it is easy to tell our work apart.

I tried to find an efficient implementation and Khayri tried to find a good way of describing the idea in words. Almost all Muse code was written by me with Khayri as co-programmer and the published papers written by Khayri with me as co-author.

The objective of this section is to try to separate out my contribution to this thesis. I have chosen to divide the work into several tasks listed in the figure below. The list of tasks is (more or less) sorted to indicate increasing involvement by me. Then I explain what my contributions are and where to find them in the thesis.



- The BC-machine model ([4]).
- The Muse model.
- Scheduling techniques.
- Performance evaluation.
- Tuning of the system.
- Design of the Muse system.
- Design of tools.
- Implementation of the system.
- Implementation of the tools.
- Non-Prolog features.
- Further thoughts.

My main contribution to **The Muse model** is the idea of reducing copying overhead (Section 3.7). My main contributions to the **Scheduling techniques** are improving the scheduler (4.5.3) and supporting cut and side effects (4.6). Most of the documented work concerning **Performance evaluation** was done by Khayri. I analyzed the performance of programs with cuts (4.7.4, 5.4) and programs with findall (7.8.5). My contributions to the **Tuning of the system** concerned the hardware architecture of each platform (7.8, 7.12, 7.13). The **Design of the Muse system** consists of three parts: the Muse engine, the scheduler, and the engine-scheduler interface. I designed most of the first and third parts (7 and 8). I have been deeply involved in the **Design of tools** for the Muse system. The benchmarking and the statistics tools were in the main my contribution (7.14). I carried out the design of the graphical tool Must (3.10) together with Jan Sundberg and Claes Svensson. The tools were used all the time in the development of the Muse system and also in the performance analysis presented in (3.9, 4.7, 5.4, 5.5, 6.5, 6.6). The **Implementation of the system** (7) is mainly due to me. Shyam Mudambi made a good job with the initial port of the Muse system to the Butterfly machines and Torbjörn Granlund has been consulted for low level machine and compiler dependent issues. The **Implementation of the tools** is mainly my work. The benchmarking and the statistical tools were implemented by me. The first version of the graphical tool Must was implemented by Jan and Claes under my supervision, and I developed it further. Extending the Muse system to generate the trace information was done by me. The integration of Muse and (the Madrid tool) VisAndOr into ViMust (7.14) was mainly done by me. The **Non-Prolog features** presented in (7.10, 7.11) were almost all developed by me. **Further thoughts** discussed in (7.16) are all mine.

2.4 Structure of the Thesis

The thesis is structured as follows. The Chapters *Prolog*, *Introduction*, *Conclusion*, and *Epilog* constitute the framing of the thesis. The Chapters 3 to 6 consist of revised versions of a series of four papers written by Khayri A. M. Ali with me as co-author. Those four chapters are sufficient to present and evaluate the Muse model for OR-parallel Prolog. The Chapters 7 and 8 consist of two papers. The first one was written by me and the second one by me with Khayri as co-author. The aim of those two chapters is to describe the actual implementation of the Muse system in such detail that it is possible to duplicate the work. The six papers included in this thesis were carefully selected from a set of papers [6, 7, 5, 9, 8, 11, 12, 10, 57, 58, 53] in order to give a comprehensive description of the Muse system. In what follows I give a very brief summary of the subject and main results of each of the six papers.

⋆ ⋆ ★ ⋆ ⋆

Ch 3 **The Muse Approach to OR-parallel Prolog:** A revised version of a published paper [6] presenting the Muse model. It also describes and analyzes the implementation of Commit Prolog, a version of OR-parallel Prolog using asynchronous side effects and cavalier commit instead of cut. (Implementing Commit Prolog was the first step towards supporting full Prolog, described in the next chapter.) The analysis consists in interpreting a large set of benchmark runs on both a conventional shared memory machine (Sequent Symmetry) and on a NUMA (Non Uniform Memory Access) machine. It also compares the Muse results with the Aurora results. The main conclusion is that the performance of the Muse system is very good.

Ch 4 **Full Prolog and Scheduling OR-parallelism in Muse:** A revised version of a published paper [5] presenting how to implement full Prolog in Muse and also presenting the scheduling principles used in the current version of Muse. The implementation is very thoroughly analyzed using a large set of benchmarks as in the previous paper. The main conclusion is that the good performance of the system is preserved when the full Prolog language is supported.

Ch 5 **Performance of Muse on Switch-Based Multiprocessor Machines:** A longer version [11] of a published paper [12] discussing implementation issues and analyzing the performance of Muse on two NUMA machines, the Butterfly I (GP1000) and the Butterfly II (TC2000). The main conclusion is that the good locality of memory references in the Muse model makes it suitable for both NUMA and UMA machines.

Ch 6 **OR-parallel Speedups in a Knowledge Based System: on Muse and Aurora:** A revised version of a published paper [10] analyzing the performance of a large practical Prolog program on a shared memory machine (Sequent Symmetry) and on two NUMA machines (Butterfly I and Butterfly II). The two main conclusions are (1) there exist real programs that show very high speedups even on "difficult" architectures and (2) the incremental copying approach used in Muse outperforms the SRI approach used in Aurora on NUMA machines.

Ch 7 **How to Build Your Own OR-parallel Prolog System:** A research report [57] aimed at showing how to construct an OR-parallel Prolog system. A suitable methodology is presented, an overview of the current implementation of the Muse system is given, and several implementation details are presented. Some issues related to features not implemented are also discussed.

Ch 8 **The Scheduler-Engine Interface used in Muse:** A research report [58] describing the interface used between a Prolog engine based on SICStus and the Muse scheduler. This interface has improved the modularity of the system, while preserving the high efficiency.

⋆ ⋆ ★ ⋆ ⋆

It is natural that there are some repetitions in the thesis, since it partly consists of a set of published papers. Some parts of the first paper (found in Chapter 3) are also somewhat out of date. But my feeling is that the chapters in the thesis form a natural sequence, from describing the Muse execution model to making it possible for the reader to build his or her own OR-parallel system.

Chapter 3

The Muse Approach to OR-parallel Prolog

Khayri A. M. Ali   *Roland Karlsson*

Published in IJPP April 1990 [6]
Revised March 1992

MUSE (Multi-Sequential 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-processor machine with local/shared memory constructed at SICS, a 16-processor Sequent Symmetry, a 96-processor BBN Butterfly I, and a 45-processor BBN Butterfly II. The sequential SICStus Prolog system has been adapted to OR-parallel implementation. The 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 chapter is to present the Muse execution model, some of its implementation issues, a variant of Prolog called Commit Prolog, and some experimental results obtained from two different multiprocessor systems.

3.1 Introduction

Logic programs offer many opportunities for the exploitation of parallelism, the main ones being OR-parallelism and AND-parallelism [29]. 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 [32, 50]. 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 [36, 78, 86].

In Prolog, OR-parallelism and independent AND-parallelism are much easier to implement efficiently than stream AND-parallelism. This chapter 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 [4, 14, 23, 28, 47, 54, 79, 92], incorporating different binding schemes. Binding schemes for combining OR-parallelism with AND-parallelism have also been developed [19, 30, 56, 94]. Recent approaches attempt to exploit stream AND-parallelism while retaining the full search capability of Prolog [42, 43, 71, 80, 76].

Other important issues in the efficient implementation of OR-parallel Prolog are scheduling work so as to minimize speculative computations, and supporting the Prolog side effects and cut [3, 46, 45, 55].

3.1.1 History of the BC-Machine Research

In 1986, a research project at SICS, called the BC-machine project, 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 are potentially scalable and can utilize faster processing units as they become available. OR-parallelism is easy to detect and exists in a large class of applications (e.g. natural language processing [51], expert systems [65], theorem proving [64], meta-interpreters [42], etc.).

Two ideas have been proposed by the BC-machine group; the Multi-sequential-machines approach [2] and the BC-machine approach [4]. Both ideas are based on having multiple sequential Prolog engines. This makes it possible to preserve all the advantages of the sequential Prolog technology, e.g. efficient implementation, garbage collection, etc. The main drawbacks of the approach [2] are the overhead of (1) copying an engine state, and (2) load balancing. In the BC-machine approach [4], the load balancing problem has been solved through the use of some shared memory in the system and copying overhead is reduced by using a special broadcast network for parallel copying.

3.1.2 Motivations for the Muse Approach

It was initially thought that copying overhead can be very high and that hardware support, like the broadcast network in the BC-machine, is essential for an efficient implementation. When we made 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 unexpected result regarding the copying overhead. The copying overhead is actually much lower than might be expected. It ranges from 0.01% to 25% of the processor time depending on the program. This percentage is small in comparison with the corresponding overhead (total task switching overhead) in other OR-parallel execution models. In

Muse we have reduced the copying overhead further to make the model suitable for a larger class of multiprocessor machines.

3.1.3 Current Implementations and Results

Currently, we have two OR-parallel Prolog systems. The first Muse system supports a variant of Prolog, called Commit Prolog, with cavalier commit [22] 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 in Commit Prolog by following a few rules that restrict the degree of OR-parallelism. The first Muse system has been implemented on a 7-processor prototype system constructed at SICS, and on a 16-processor Sequent Symmetry machine. The prototype architecture is similar to IBM ACE [35]. 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) [26], a fast, portable system, has been adapted to OR-parallel implementation. The 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, the Sequent Symmetry machine, a 96-processor BBN Butterfly I (GP1000), and a 45-processor BBN Butterfly II (TC2000). (Shyam Mudambi at Brandeis University has ported Muse to the Butterfly machines.) The two Muse systems are also being ported to uniprocessor UNIX workstations. The preliminary results of the second Muse system on the four multiprocessor machines indicate that the 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 chapter 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 3.11 and complete benchmark results can be found in the following three chapter. Some parts of this chapter have been presented in [7].

3.1.4 Structure of the Chapter

The chapter is organized as follows. Section 3.2 describes the Muse execution model. Section 3.3 shows how the Muse model can be implemented as a minimal extension of the WAM. Section 3.4 discusses the characteristics of the Muse model. Section 3.5 discusses very briefly the principles for scheduling work in Muse. Section 3.6 specifies a version of Prolog that we call Commit Prolog and shows how to get full Prolog semantics using Commit Prolog. Section 3.7 discusses and presents very important optimizations designed to reduce the copying overhead. Section 3.8 discusses garbage collection in Muse. Section 3.9 presents some experimental results on two different multiprocessor systems. It also compares our results with the results obtained from related approaches. Section 3.10 describes briefly our graphical tool for tracing parallel execution and for debugging our implementations. Section 3.11 concludes the chapter and discusses our plans for the continued development of Muse.

3.2 The 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.

3.2.1 Multiprocessor System Assumptions

We assume a multiprocessor system with a number of processors or processes, called *workers*, with identical local address spaces, and some global address space shared by all workers. We use the term worker to represent a process or processor, exactly as in Aurora [65]. 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 [91]. 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 [91].

3.2.2 Execution Model

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

2. One of the idle workers Q requests work from P.

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

   (a) For each local choicepoint, P creates a shareable frame in the shared space.

   (b) P moves information describing the unprocessed alternatives from the local choicepoints to the corresponding shareable frames.

(c) Each local choicepoint is given 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 choicepoints (Figure 3.1).

4. P and Q work together to finish processing the shared unprocessed alternatives (tasks). They process the shared choicepoints from the bottommost up to the root choicepoint 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 bottommost shared choicepoint. Releasing an alternative from a shared choicepoint is done while the corresponding shared frame is locked.

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

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

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

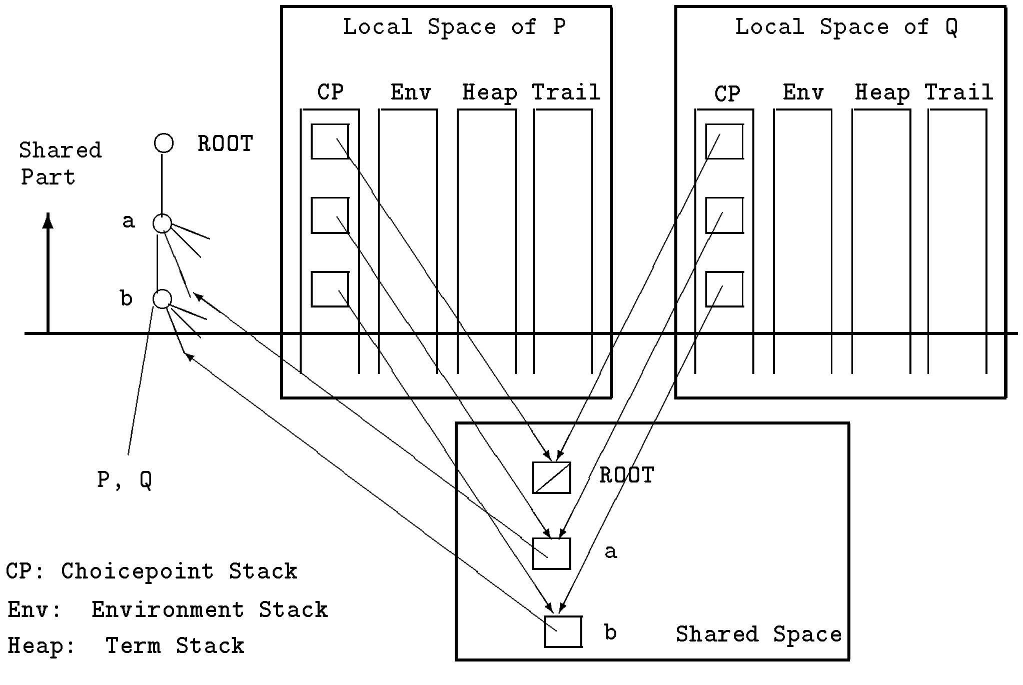


Figure 3.1: Sharing work.

So far, this model is the same as the BC-machine model [4] 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 in every copying operation can be expected to be expensive. In the following section, we will illustrate how the copying overhead can be reduced substantially.

3.2.3 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 have P copy only the difference between the two states. Before showing how this can be efficiently supported, we would first like to briefly review the backtracking mechanism of Prolog.

On creation of a choicepoint c, the current computation state is stored in c. The computation state 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 to get the next alternative, the computation state is restored from c and all modifications of variables created before c are undone by going through the trail stack.

Now how can the incremental copying idea be implemented efficiently? Assume that there is no work available at any shared choicepoint in the current branch[1] of a worker Q and that there is another worker P with local choicepoints. Assume also that Q is to share the local choicepoints of P. First the worker Q backtracks to the youngest choicepoint n shared with P (Figure 3.2). Then P makes its local choicepoints 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 choicepoints younger than n and the local modifications that have been made in the common parts after creating n. These modifications are identified by P by accessing the top segment of its trail stack (Figure 3.2). Now P and Q have identical states and both share all of P's choicepoints exactly as in step (3.d) in the above section. Copying overhead is further reduced in Section 3.7.

3.3 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 of the WAM [91].

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.

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

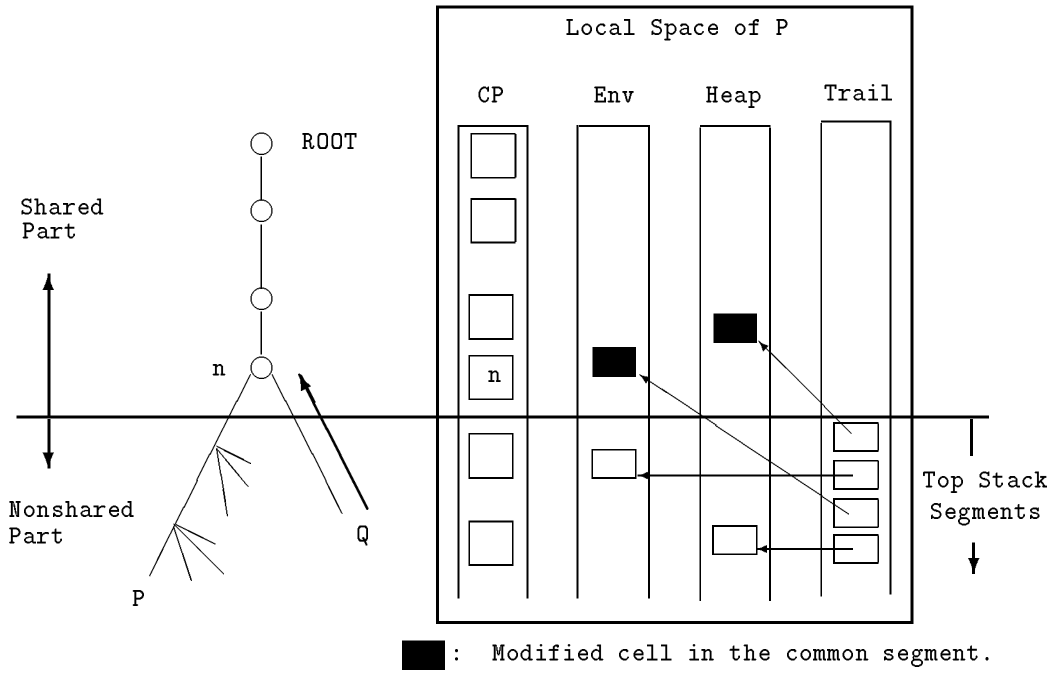


Figure 3.2: Incremental copying idea.

Workers share a segment of memory for storing frames associated with shared choicepoints, global tables (the atom table, the 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.

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

The only modification of 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 choicepoint is shared or not. For a shared choicepoint, the word contains a pointer to the next younger (child) choicepoint frame in order to find quickly the child choicepoint in cut/commit operations. (Notice that a shared choicepoint logically has many children but from a single worker perspective, there is only one child which corresponds to the current branch.) For a nonshared choicepoint n, the word contains the number of remaining alternatives of the nonshared choicepoints older than n. This number is used as a measure of the local load of each worker, guiding the scheduler in 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 that worker.)

Another modification of the WAM is that on every procedure call, a worker checks for requests sent by other workers. The two possible requests concern making local choicepoints shareable or aborting the current task as part of cut/commit operation.

All the WAM stacks are located at a fixed address in the 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) can 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.

3.4 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 account for the majority of accesses, are not shared among workers. Secondly, in 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. Since every worker is almost the same as the sequential Prolog engine, the speed of every worker is almost that of the sequential Prolog engine. The Muse execution model preserves 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.

3.5 Scheduling Work

Nodes in 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 at the node. Local nodes are only accessible to the worker that created them. Each worker can be in either engine mode or 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 like a sequential Prolog engine on local nodes, but is also able to respond to requests 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 strategies to minimize the overhead are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the amount of work shared between the workers and allows each worker to release work from the bottommost node in its branch by using backtracking with almost no overhead.
- When a worker runs out of work 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 by the positions of workers in 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 have low overhead.
- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position in the tree before requesting sharing from the busy worker. This allows a busy worker to concentrate on its task and to respond only to requests 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 in 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 in the tree.* More detailed information about scheduling issues is presented in Chapter 4.

3.6 Commit Prolog[2]

There are two main approaches to implementing Prolog on multiprocessor systems:

1. Developing a new version of the Prolog language suitable for multiprocessor systems, as advocated by the ECRC group [94]. Their main motivation is that the standard Prolog language was 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 established in the logic programming community.

Each approach has its own advantages and disadvantages. For instance, an advantage of the first approach is that 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 sequential Prolog semantics.

On the other hand, the second approach allows existing Prolog programs to be executed without modifications with standard Prolog semantics. This does not mean that all existing Prolog programs will execute significantly faster on multiprocessor systems. For Prolog programs that make heavy use of side effects and cuts with large scopes it is very difficult to get any interesting speed-up. Supporting side effects (with sequential semantics) and cut 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[3] instead of cut, asynchronous (parallel) side effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. 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 the sequential processing of a predicate.

Cut semantics can be obtained in Commit Prolog by using commit and sequential annotations, as will be explained in Section 3.6.2. Sequential side effect semantics can also be obtained in Commit Prolog by following the rules described in Section 3.6.1.

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

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

may fail sometimes, even when p(X) is solvable [23].

[2]Muse no longer (1992) supports Commit Prolog, but this section should be of interest in connection with systems that cannot support (full) Prolog.

[3]Cavalier commit prunes branches both to the left and the right of the committing branch, and is not guaranteed to prevent side effects from occurring in the pruned branches [23]

Commit Prolog is suitable for programs that do not use cuts with large scopes and use sequential side effects only in the top level predicate. In general, it is possible to rewrite your program in the required style [14, 23].

3.6.1 Sequential Side Effects Semantics

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

The rules:

1. No sequential side effects in a parallel predicate.
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 a goal that calls directly or indirectly a parallel predicate whereas a *sequential-goal* is a goal that does not call directly or indirectly any 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). Finally, 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 standard Prolog. We also find that workers concentrate on the leftmost subtree of the entire search tree, thus reducing speculative work. (Speculative work is work which could turn out to be unnecessary.)

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

3.6.2 Cut Semantics

When cut semantics is needed in a Commit Prolog program, we can apply the following rules to get cut semantics. The idea here is to turn off OR-parallelism within the scope of cut.

The rules:

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

In the following example cut semantics is enforced by using sequential annotations in a program using 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 having cut semantics is to restrict parallelism. On the other hand speculative computations are avoided.

3.7 Reducing the Copying Overhead

In this section we are going to reduce further the copying overhead which is the main source of overhead associated with the Muse model. In Section 3.7.1, we show how to reduce copying overhead when the youngest local choicepoints are sequential choicepoints. In Section 3.7.2, we discuss a method that reduces copying overhead on a shared memory multiprocessor machine like the Sequent Symmetry and Butterfly machines.

3.7.1 Reducing the Overhead when the Youngest Choicepoints are Sequential

A node in the search tree is either sequential or parallel. Parallel nodes have potential for OR-parallelism and can 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 reduction of sharing and copying overhead is obtained when the youngest (local) choicepoints of the selected busy worker are sequential choicepoints. To illustrate the idea, assume that P is the selected busy worker with the youngest choicepoints sequential and Q is an idle worker that has selected P for sharing. Assume also that P shares all its choicepoints with Q and copies the difference between the two states, and P then proceeds with its current task and Q simulates failure to get a task from the nearest shared choicepoint. In this case, the worker Q will backtrack from all bottommost sequential choicepoints and get an alternative from the bottommost parallel choicepoint with available alternatives (the node *par* in Figure 3.3). So, sequential choicepoints, such as the node *seq* in Figure 3.3, should not be shared and the corresponding state should (except for the trail) not be copied from P to Q.

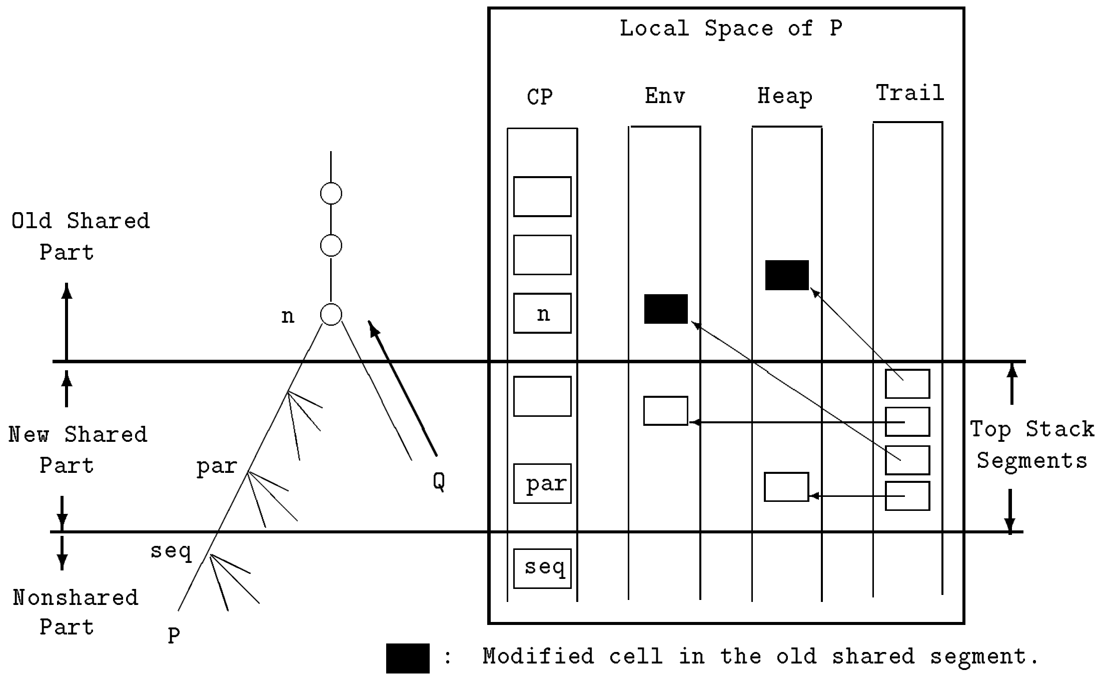


Figure 3.3: Reducing the copying overhead.

This optimization is possible only when no *undo* or *setarg* is invoked by P after taking the current alternative from the youngest parallel choicepoint. *setarg* is a backtrackable destructive assignment supported by SICStus Prolog [26]. The term stack contains a *undo* goal or a goal for undoing *setarg* on backtracking. In those cases, P copies the whole top segment of the term stack.

3.7.2 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) can be mapped into a separate part of the global address space of the system. We take advantage of this by making workers 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. When an idle worker Q requests sharing from a busy worker P with excess local load, Q could wait for P to complete the following operations. (see also Figure 3.3):

1. Share choicepoints with Q (sharing).
2. Copy the top segments of its own stacks to Q (copying).
3. Install bindings from its local *environment stack* (Env) and *term stack* (Heap) associated with (old) shared choicepoints into Q's corresponding stacks (installation).

The sharing session can be parallelized though. Copying overhead is represented by the time spent by P in the last two operations and the time spent by Q waiting until P completes these operations. Our goal here is to minimize the time P and Q spends in the sharing session. Before describing the method of parallel copying, let us first investigate the three operations mentioned above (sharing, copying, and installation) and issues that arise in minimizing P's time in copying.

- 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[4] (and the Trail segment, called Top-Trail, corresponding to

[4]P should not modify its copy of Env and Heap segments before Q gets another copy.

the youngest sequential choicepoints discussed in Section 3.7.1[5]) have been copied.

- P should not do either garbage collection or *setarg* before copying and installation are completed.
- Also, P cannot backtrack from any shared choicepoint before copying and installation are completed. (Notice that P has to complete processing all available work on the local sequential choicepoints and the bottommost parallel choicepoint, *par* in Figure 3.3, before backtracking from a shared choicepoint.)

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 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 choicepoint or to execute *setarg,* or to perform garbage collection, and Q has not completed copying the CP and Trail segments, or the installation, P will help Q perform any of these operations.

3.8 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 carry out its local garbage collection asynchronously without involving another worker in its garbage collection operation.

Incremental copying presented in Section 3.2.3 is based on avoiding copying of a common state of the two workers involving in copying. If one of the two workers has performed garbage collection after the previous copying, finding the 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.

The second solution tries to minimize data to be copied. It is based on segmented garbage collection [13] 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 can be illustrated by a simple system having two

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

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 either of the two workers (or both) has performed a new segmented garbage collection. Then a new copying is to be carried out. The part common to the two workers that was 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 3.8.2 about 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.

3.8.1 The First Solution

Every garbage collection (*gc*) has a unique name. Every worker stores the name of its last *gc*. When a worker P gets a request from a worker Q for sharing and copying, P checks the current *gc* name of Q against its own *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.

3.8.2 The 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 bottommost common choicepoint are identical, sharing and incremental copying will be performed as usual. If the *gc* names are not identical, incremental copying is performed as far as the nearest common choicepoint in which P and Q have identical *gc* names. Along with incremental copying the pointers from the old garbage collected segments to the copied Heap stack segment should be updated in Q's old segment. (Segmented *gc* updates pointer cells in old garbage collected segments that point to objects in the new garbage collected segment [13]) The trail segment corresponding to the old garbage collected segment contains pointers to these cells. Such cells should also be installed for Q.

3.9 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 processor machine with local/shared memory.

3.9.1 Benchmarks

The group of benchmarks used in this chapter 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 [14, 24, 83]. 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 [51]. *semigroup* is a theorem proving program for studying the R-classes of a large semigroup from Argonne National Laboratory [64]. *satchmo* is a little theorem prover for proving theorems in predicate logic. The original *satchmo* was written by Rainer Manthey and Francois Bry from ECRC, and modified by Lee Naish of the University of Melbourne. *andorra-interp* is an interpreter for the Andorra computation model written by Seif Haridi at SICS [42]. All the benchmarks look for all solutions of the problem.

3.9.2 Performance Results: Timings and Speedups

3.9.2.1 Results On Sequent Symmetry

Table 3.1 presents the execution times (in milliseconds) from the execution of the first set of benchmarks (described in Section 3.9.1) on a 16 processors Sequent Symmetry S81 with 32 MBytes of memory. The execution times given are the shortest obtained from several runs (as measured in [24, 83]). 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 3.1: Muse execution times (in milliseconds) for the first set of benchmarks on Sequent Symmetry.

The last column shows execution times on SICStus version 0.6 (SICStus0.6) with the ratio of execution times on Muse for the 1 worker case to the SICStus0.6 times. For benchmarks with small execution 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 3.4 shows the amount of parallelism exploited as a function of time by 15 workers executing the *all-goals* benchmark, measured by Muse graphical tracing facility (*Must*) [82]. The whole area under the curve is divided into 13 parts corresponding to the 13 subgoals in the *all-goals* benchmark in the same order as they are listed above. Figure 3.4 shows also subareas corresponding to coarse, medium, and fine grain parallelism. (The execution time in Figure 3.4 is longer than in Table 3.1 because in the former trace information is generated.)

Table 3.2 presents the execution 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 3.3.

3.9.2.2 Results On the Constructed Hardware Prototype

Tables 3.4 and 3.5 present the Muse execution times from the execution of the benchmarks on a 7-processor hardware prototype constructed at SICS. The results

Figure 3.4: The amount of parallelism exploited as a function of time by 15 workers executing the *all-goals* benchmark measured by the Muse graphical tracing facility Must.

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 3.2: Muse execution times (in seconds) 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 3.3: Test sentences of Pundit benchmark.

shown in these tables are those of a portable version of Muse on Sequent ported to the hardware prototype without any significant modifications. That is, better results could be obtained than those in Tables 3.4 and 3.5 if the Muse system took advantage of the features 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 3.4: Muse execution times (in milliseconds) for the first 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 a VME bus as a common bus in the system. All our results reported in Tables 3.4 and 3.5 refer to the prototype without a copying network. The VME bus is used to access global memory and for copying.

3.9.2.3 Discussion of Performance Results

The performance results that Tables 3.1 to 3.5 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 why some results in Tables 3.1 and 3.2 are better than those in the Tables 3.4 and 3.5 is that the capacity of the Sequent Symmetry bus is much higher than that of the VME bus. The VME bus is slower than Sequent Symmetry bus by a factor of around 20. It neither supports caching nor simultaneous locking (test-and-set) operations. That is, in our hardware prototype the ratio of communication speed to processor speed is much lower than in the Sequent Symmetry machine. *salt-mustard* in Table 3.4 has lower speed-ups in comparison with that in Table 3.1. 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

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 3.5: Muse execution times (in seconds) for the second set of benchmarks on the hardware prototype.

support caching. Table 3.6 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 [75]. 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 3.6: Muse execution times (in milliseconds) 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 3.1), the sequential Prolog system from which it is derived. This overhead is smaller than in Aurora [65], 24% (see also below), and in PEPSys [74], 30%. The 5% overhead of Muse can be divided into two parts. 2.5% is for maintaining the current value of the local load in each worker, and for checking for the arrival of 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.

3.9.3 Basic Overheads of OR-parallel Execution

In this section, we present and discuss briefly the time spent in the basic activities by a Muse worker in OR-parallel execution. The time is distributed over the following activities:

1. *Prolog:* time spent executing Prolog, checking arrival of requests, and keeping the value of the local load up to date.
2. *Idle:* time spent looking for a worker with excess local work when there is no available work in the shared choicepoints.
3. *Sharing:* time spent making local choicepoints shared with other workers.
4. *Grabbing Work:* time spent grabbing available work from shared choicepoints.
5. *Copying:* time spent copying.
6. *Installation:* time spent making an installation.
7. *Waiting:* time spent waiting for synchronization purposes in sharing and copying activities.
8. *Backtracking:* time spent moving up (towards the root) within the shared region (shared part of a search tree). This does not include engine activities such as backtracking.
9. *Others:* time spent in other activities, like trying to acquire a spin lock, requesting either sharing or performing commit/cut to a shared choicepoint from other workers, etc.

Table 3.7 shows the total time spent in each activity, and the corresponding percentage of the total time, for the first set of benchmarks (*all-goals* benchmark) described in Section 3.9.1. Times shown in Table 3.7 have been obtained from an instrumented system of Muse on Sequent Symmetry. Those times include the time spent on 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 by the instrumented system reflects what is happening in the uninstrumented system.

As mentioned in Section 3.9.1, 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 contains benchmarks that lack parallelism (Figure 3.4). Lack of parallelism explains why the *Prolog* percentage decreases and *Idle* percentage increases with increasing number of workers in Table 3.7. The sum 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.

	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 3.7: Total times (in milliseconds) spent in the basic activities of a Muse worker for the first set of benchmarks on Sequent Symmetry.

These overheads increase from the 4 workers case to the 8 workers case by 5.3% of the total execution time, from 8 workers to 12 workers by 2.1%, and from 12 workers to 15 workers by 0.9%.

A possible explanation for the increasing overhead is shown in Table 3.8, which shows the effect of increasing the number of workers on the number of tasks and the task sizes (expressed as the 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 3.8 the granularity of parallelism is decreased from the 4 workers case to the 8 workers case by a factor of around 2, whereas from 8 workers to 12 workers it decreases by a factor 1.23, and from 12 workers to 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 3.8: Average number of tasks and task sizes for the first set of benchmarks.

Tables 3.7 and 3.8 illustrate that the overhead increases when the granularity of the parallelism is reduced. The increase is distributed over the last seven activities as shown in Table 3.7. Table 3.7 also illustrates that the cost of the *Copying* overhead, which was feared to be high, is quite acceptable: 4.2% on 15 workers. Similarly, the cost of manipulating shared choicepoints (represented by *Backtracking* and *Grabbing Work* overheads) is low.

3.9.4 Comparison with Aurora

We have chosen the Aurora OR-parallel Prolog system [65] for a comparison 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 [92]. The idea of the SRI model is to extend the conventional WAM with a 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. A binding is conditional if a variable can get several bindings. 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 has finished 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.

A full comparison between Aurora and Muse is not carried out in this chapter. The Aurora system supports parallel implementation of cut and sequential side effects whereas the Muse version reported in this chapter does not support them (see below and the following three chapters 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 is possible. We compare our results with the Aurora results for the first set of benchmarks (described in Section 3.9.1). Aurora results based on SICStus0.3 are found in [65, 24, 83]. 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 have been published regarding the current version of Aurora, we have measured the run times given in Table 3.9 (with the Manchester scheduler) on the same Sequent Symmetry and under the same conditions as the results in Table 3.1 (page 30).

It can be seen from Tables 3.1 and 3.9 that Muse is faster than Aurora in all benchmarks. Table 3.10 shows in the last row the ratio of the execution times on Aurora to the execution times on Muse for the *"all-goals"* benchmark. Aurora timings are longer than Muse timings by 19% to 35% for 1 to 15 workers. (The 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.)

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 the BBN Butterfly machine, and for machines with local/global memory like the ACE-IBM

Goals *repetitions	Aurora Workers 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 3.9: Aurora execution times (in milliseconds) for the first set of benchmarks on Sequent Symmetry.

System	Workers 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 3.10: Aurora and Muse execution times (in milliseconds) and the ratio between them for the *"all-goals"* benchmark on Sequent Symmetry.

machine [35]. Shyam Mudambi at the Brandeis University has ported a Muse system that supports full Prolog on Sequent Symmetry to the BBN Butterfly I and II machines. His preliminary results are very promising [67] (see also Section 3.11). Complete benchmark results can be found in Sections 5.5 and 6.5.

3.10 *Must:* The Muse Graphical Tracing Facility

A graphical tracing tool used for understanding the behavior of the Muse system, called *Must,* has been developed. Various events during parallel execution of a Prolog program are recorded and then replayed under UNIX and the X Window system to show the dynamic behavior of the Muse system [82]. This tool is similar to the Argonne tool [33] but *Must* has other facilities: recording real time for each event, showing the utilization of processors (workers) over time, showing the structure of the whole search tree, and tracing on a query level (and not only for the entire session).

In Figure 3.5 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 the positions of workers as black dots, *timeview* near the top and to the right, showing the utilization of workers in executing Prolog over time, and *mainview* showing a selected part of the search tree in some detail. The *Must* user interface also contains **file name** at the top, **elapsed 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 3.5 shows a search tree when 17407 milliseconds have elapsed (out of a total of 20014 milliseconds). The obtained speedup so far is 7.98 on a system with 8 workers. The *timeview* shows utilization of workers in executing Prolog after 17407 milliseconds. The *sideview* and *mainview* show the shared part of the search tree at that time.

Figure 3.5: A snapshot of the Muse graphical tracing facility Must.

In the *mainview,* a shared node is represented by circle with either a vertical line inside for a sequential node, or a number inside for a parallel node indicating the number of unexplored branches, or solid 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. A small black rectangle below a triangle of worker X indicates that the worker X has excess local work (untried branches). When a worker moves in a tree the worker number will indicate its position in the tree. For example, worker 3 in Figure 3.5 has finished its task and is backtracking to position itself at a suitable node in the tree before requesting sharing from a worker with excess local work. Figure 3.5 illustrates also that all workers except worker 3 are executing Prolog and workers 2, 4 and 5 have excess local work.

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

3.11 Conclusions

We have presented a simple and efficient execution model, named Muse, which supports 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 and some shared memory space, and the copying of memory blocks from one local address space to another. Operating systems like DYNIX and Mach efficiently support 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 preserves 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 overhead.

The Muse model has been implemented on a 16-processor Sequent Symmetry, a 7-processor machine with local/shared memory constructed at SICS, a 96-processor BBN Butterfly I (GP1000), and a 45-processor BBN Butterfly II (TC2000). The sequential SICStus Prolog system has been adapted to OR-parallel implementations. The 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 also indicate low overhead [67]. 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 system to uniprocessor workstations (Sun3 and Sun4). One advantage of this was that timing bugs could be discovered very early.

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

The overhead associated with supporting full Prolog in the new Muse system for the first set of benchmarks (described in Section 3.9.1) 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 of speculative work [45]. The amount of unnecessary speculative work is very small for this set of benchmarks, basically because it was chosen not to contain major cuts [83]. Shyam Mudambi at Brandeis University, who has ported the new Muse system into Butterfly I and II, got the following preliminary speedups for 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 [67].

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. 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.

3.12 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 Brandeis University for porting the Muse system to the Butterfly machines, and Jan Sundberg and Claes Svensson for constructing the graphical tracing facility for Muse.

Chapter 4

Full Prolog and Scheduling OR-parallelism in Muse

Khayri A. M. Ali *Roland Karlsson*
Published in IJPP December 1990 [5]
Revised March 1992

MUSE is a simple and efficient approach to OR-parallel implementation of the full Prolog language. It is based on having multiple sequential Prolog engines, each with its local address space, and some shared memory space. It is currently implemented on a number of bus-based and switch-based multiprocessors. The sequential SICStus Prolog system has been adapted to OR-parallel implementation with very low overhead in comparison with other approaches. The Muse performance results are very encouraging in absolute and relative terms.

The Muse execution model and its performance results on two different multiprocessor machines for a parallel version of Prolog, named Commit Prolog, have been presented in the previous chapter. This chapter discusses supporting the full Prolog language and describes mechanisms being developed for scheduling OR-parallelism in Muse. It also presents performance results of the Muse implementation on Sequent Symmetry executing full Prolog. The results show that the overhead associated with supporting the full Prolog language is negligible.

4.1 Introduction

A variety of approaches to the exploitation of parallelism in Prolog programs are under current investigation. Many of these deal with the implementation of Prolog on multiprocessor machines by exploiting either OR-parallelism [6, 14, 23, 28, 54, 60, 65, 95] or independent AND-parallelism [32, 50, 61] or a combination of both [19, 30, 56, 94]. The Muse approach exploits only OR-parallelism (see Chapter 3). In the execution of Prolog a tree of alternatives is examined. OR-parallel execution of a Prolog program means exploring the branches of this *search tree* in parallel. In the Muse approach (as in other OR-parallel Prolog approaches, e.g. Aurora [65] and

PEPSys [14]), OR-parallelism in a Prolog search tree is achieved through a number of *workers* (processes or processors)[1]. This chapter describes the basic mechanisms used for exploring branches of a Prolog search tree by the Muse workers. It also describes mechanisms for maintaining the sequential semantics of cut, findall and side effect constructs.

The Muse approach 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 bus-based shared memory machine TP881V, from Tadpole Technology, with 4 (88100) processors, a bus-based machine with local/shared memory with 7 (68020) processors constructed at SICS, a bus-based shared memory S81, Sequent Symmetry, with 16 (i386) processors, and switch-based shared memory machines, BBN Butterfly I (GP1000) and II (TC2000), with 96 (68020) and 45 (88100) processors respectively. The sequential SICStus Prolog [26], a fast, portable system, has been adapted to OR-parallel implementation. The overhead associated with this adaptation is very low in comparison with the other approaches. It is around 3% for TP881V, and 5% for the constructed prototype and Sequent Symmetry. The performance results of Muse on the BBN Butterfly are reported in Sections 5.5 and 6.5. The results are very promising. The Muse execution model and its performance results on the constructed prototype and Sequent Symmetry machines for a parallel version of Prolog, Commit Prolog, have been presented in Chapter 3. Commit Prolog is a Prolog language with cavalier commit[2] instead of cut, asynchronous (parallel) side effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. Cut and sequential side effect semantics can be obtained in Commit Prolog by annotating Prolog programs according to some rules as described in Section 3.6. In this chapter, we discuss supporting the full Prolog language without such annotations. Some parts of this chapter have been presented in [9].

The chapter is organized as follows. Section 4.2 briefly describes the Muse execution model, to aid the understanding of the principles for scheduling work in Muse. Section 4.3 discusses principles for scheduling work in Muse. Section 4.4 discusses principles for scheduling work in related approaches. Section 4.5 presents and discusses the basic mechanisms for supporting scheduling work in Muse. Section 4.6 discusses the implementation of cut, findall, and sequential side effect constructs. Section 4.7 presents some performance results of the Muse system. Section 4.8 discusses our plans for the continued development of Muse. Section 4.9 concludes the chapter.

[1]In this chapter we try to be consistent with the Aurora terminology [65].

[2]Cavalier commit prunes branches both to the left and right of the committing branch, and is not guaranteed to prevent side effects from occurring in the pruned branches [22].

4.2 The Muse Execution Model – An Overview

This section briefly describes the Muse execution model presented in Section 3.2. We assume that the reader is familiar with the Warren Abstract Machine (WAM) [91].

A node in a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared(private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each shared node is accessible only to workers within the subtree rooted at the node. Private nodes are only accessible to the worker that created them. Another distinction is that a node can be either *parallel* or *sequential*. Alternatives from a parallel node can be executed in parallel whereas alternatives from a sequential node can only be executed one at time, from left to right. A node is either *live* (has unexplored alternatives) or *dead* (no alternative to explore).

A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. The Muse execution model is based on having a number of 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 stacks[3]. The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks.

This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the difference between the two states. This reduces copying overhead. Nodes are shared between workers to allow dynamic load balancing, which reduces the frequency of copying.

To illustrate how workers share nodes in Muse, let us take a simple system having two workers P and Q. A worker Q runs out of work in its branch and a worker P has excess load (i.e. live nodes). Assume also that Q is positioned at a node, N, which is common to P and Q (see Figure 4.1 (a)). P allows Q to share its nodes by creating a data structure called a *shared frame* in a shared-memory space for each private node, pointing to each shared frame from the corresponding choicepoint frame[4], and then copying to Q all choicepoint frames corresponding to nodes younger than N. Each shared frame basically contains information indicating unexplored work at the node, workers which are at and below the node, and the node lock. At this moment, the choicepoint stacks of P and Q are identical and each node is associated with a shared frame which is referenced from the corresponding choicepoint frames.

[3]The assumed worker's stacks are: 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 [91].

[4]In the standard implementation of Prolog, a choicepoint frame is created in a choicepoint stack when a nondeterministic predicate is invoked.

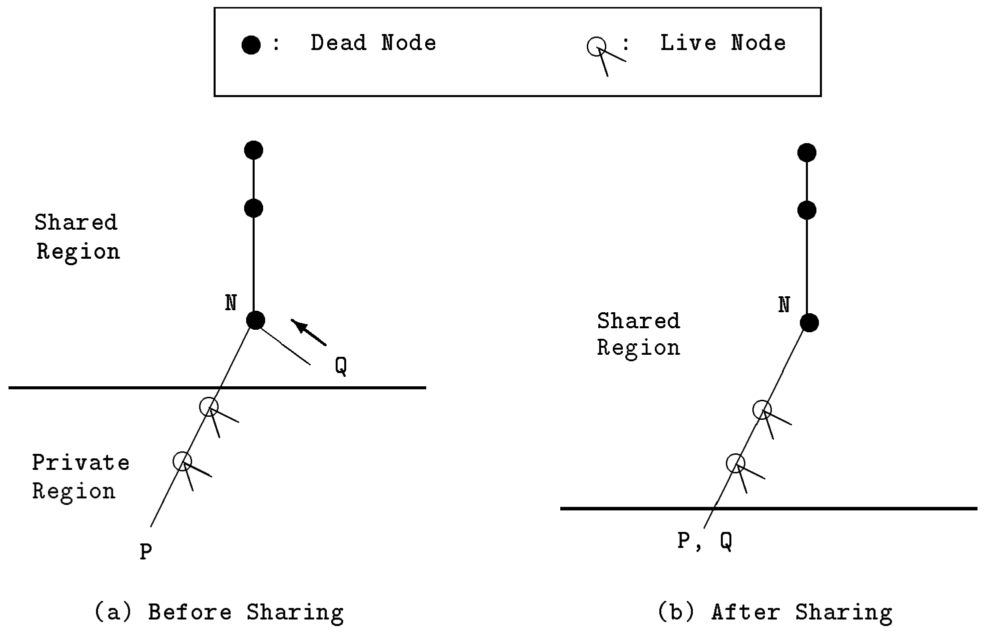


Figure 4.1: Sharing nodes.

In order to allow Q to execute alternatives from these shared nodes, Q has to get a copy of P's current state. Q gets only the parts of the WAM stacks that correspond to nodes younger than N, along with P's modifications of the uncopied parts. These modifications are bindings made by P after creating the node N of variables created before N. These modifications are known from P's trail part created after N. Notice that after copying and sharing, P and Q have identical states and both share all P's nodes (see Figure 1 (b)). Now they can work together, exploring alternatives in the shared nodes by using the normal backtracking mechanism of Prolog. Reducing sharing and copying overheads in the Muse model is described in Section 3.7.

4.3 Principles for Scheduling Work in Muse

Each worker can be in either engine mode or scheduler mode. The worker enters scheduler mode when it enters the shared part of the tree, or when it executes side effects or *findall*. In scheduler mode, the worker establishes the necessary coordination with other workers. The worker enters engine mode when it leaves the scheduler mode. In engine mode, the worker works exactly like a sequential Prolog engine on private nodes, but is also able to respond to requests 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 main sources of overhead in the Muse model are (1) copying a part of a worker state, (2) making local nodes shareable, and (3) grabbing a piece of work from a shared node. Our scheduling work strategies to minimize the overhead are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the amount of shared work between the workers and allows each worker to release work from the bottommost node in its branch (dispatching on the bottommost) by using backtracking with almost no overhead. Dispatching on the bottommost also entails less speculative work than dispatching on the topmost, i.e. when work is taken from the topmost live node in a branch [45]. (Speculative work is defined as work which is within the scope of a cut and therefore may never be done in a sequential execution.)
- When a worker runs out of work from its branch it will try to share work with the nearest worker that has maximum load. The load is measured by the number of local unexplored alternatives, and nearness is determined by the positions of workers in the search tree. This strategy attempts to maximize the shared work and minimize sharing overhead.
- Workers that 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 have low overhead.
- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position in the tree before requesting sharing from the busy worker. This allows a busy worker to concentrate on its task[5] and to respond only to requests that have to be handled by it.

4.4 Scheduling Work in Related Approaches

In this section we discuss strategies for scheduling work developed for the related OR-parallel Prolog approaches that are based on several sequential engines with constant access time for variables and nonconstant task-switching time [40]. Aurora [65], Kabu Wake [60], and ORBIT [95], are examples of these approaches.

Four separate schedulers are being developed for Aurora: the Argonne scheduler [22], the Manchester scheduler [24], the Wavefront scheduler [21], and the Bristol scheduler [17]. The basic strategy of the first three schedulers is that of dispatching on topmost and sharing one parallel node at a time in each branch. That is, all the three schedulers attempt to maintain at most one, live, shareable node in their current branch. The advantage of this strategy is that the size of the shared region is minimized and the size of tasks is kept as large as possible. One disadvantage is that finding a task always involves a general search in the tree, leading to relatively high task switching costs for fine (and medium) granularity programs [83]. Another disadvantage is that more speculative work is done.

[5]A task is a continuous piece of work executed by a worker.

The Argonne scheduler uses local information maintained in each node to indicate whether there is work available below the node. Workers use this local information to migrate towards parts of the tree where work is available. The advantage of this scheduler is the simplicity of its design. Its performance is best for coarse granularity programs.

The Manchester scheduler tries to match workers with the nearest available task, where *nearness* is measured by the number of bindings to be updated between the worker's current position and the available work. By using this strategy it is hoped to keep the task switching overheads to a minimum. The Manchester scheduler tries also to distribute idle workers evenly over the tree. The performance results of this scheduler shows improvements over the Argonne scheduler for fine and medium granularity programs.

The Wavefront scheduler maintains a data structure known as the wavefront which links all topmost live nodes together. Workers traverse and extend the wavefront when they are looking for work. The available experimental results indicate a small performance improvement for fine granularity programs over the Manchester scheduler, and slightly poorer performance for coarse granularity programs in comparison with the first two schedulers.

The Bristol scheduler is based on the Muse (or BC-machine [4]) principles; dispatching on bottommost and sharing several nodes in each branch at a time. The Bristol scheduler shows some performance improvements over the other three schedulers for fine and medium granularity programs. The overall performance results of the Bristol scheduler are similar to those of the Manchester scheduler.

Although the scheduling work principles of the Bristol scheduler are based on the Muse principles, there are differences between the Muse scheduler and the Bristol scheduler. In the Bristol scheduler, a busy worker with maximum load will be matched with any idle worker and not with the nearest idle worker as in Muse. Another difference is that idle workers in the Bristol scheduler stay at the bottommost detected dead node in their branches, whereas in Muse idle workers distribute themselves over the tree (in the current branches) and stay at positions that are expected to be better ones. A third difference is the measure used for estimating a worker's load. In the Bristol scheduler, a worker's load is measured by the number of live parallel nodes in its branch. It is impractical, for efficiency reasons, to keep this load exact. The estimated load in the Bristol scheduler is an overestimate of the real load. This leads to busy workers that do not have excess load being asked to share, exactly as in Muse (see Section 4.5.3.2). The estimate of a worker's load used in Muse is the number of unexplored alternatives of private live parallel nodes. The Muse measure of the load gives a better estimate than the one used by the Bristol scheduler, because the number of alternatives of each live node is not always equal for most programs. We believe that a better estimate of the load allows better decisions to be made. Finally, all mechanisms presented in this chapter are completely different from those used by the Bristol scheduler.

In the Kabu-Wake approach [60] and in the ORBIT approach [95], nodes cannot be shared among workers (i.e. processors), because these approaches are intended for nonshared-memory multiprocessors. Work at the topmost node is split into two parts and each of the two workers involved in copying takes a part of the work. This strategy does not allow dynamic load balancing between workers. Good results have been obtained for coarse granularity programs only.

4.5 The Scheduling Work Algorithm

The basic algorithm of the Muse scheduler for matching idle workers with available work is as follows. *(1) When a worker finishes a task, it attempts to get the nearest piece of available work in the current branch. (2) If none exists, it attempts to select a busy worker with excess work for sharing. (3) If none exists, it becomes idle and stays at a suitable position in the current branch.*

In the next three subsections, we are going to present and discuss the efficient implementation of the above three parts of the basic algorithm. Data structures used will be presented in context. In order to simplify the presentation, locking is not covered in detail.

4.5.1 Finding the Nearest Available Task

The nearest piece of available work is found in the bottommost live node in the current branch. In order to support this operation efficiently, an efficient mechanism for checking whether there is a live node in the current branch and finding such a node is required. In the current Muse implementation, we have a simple representation of the tree (see Section 4.6.1) which allows a worker to access only nodes in its own branch from its current position to the root of the tree. That is, a worker can only move over this part of the tree. A worker should leave its position in the tree (i.e. backtrack) only when the new position is better than the old one, i.e. closer to a live node or to a busy worker with excess load.

Our mechanism assumes an extra field, *nearest-livenode*, in each shared node (i.e. in each shared frame associated with each shared node). This field contains either a reference to the nearest upper live node in the current branch or a *dummy value*. The latter means all upper nodes are dead.

When a worker finishes a task, it first checks the bottommost shared node in the current branch. If the node is live, it just takes work from that node. If the node is dead, the worker checks the nearest-livenode field of the node. If nearest-livenode is not the dummy value, the worker checks a chain of shared nodes referred to by the nearest-livenode of the current node to determine the location of the nearest live node, keeping its position in the tree. (No locking is used in this operation.) If there is no such node, all nodes in the current branch get the dummy value in their nearest-livenode field. Then the worker backtracks to the nearest node with other workers and tries to select a busy worker with excess load as described in Section 4.5.2.

If work is found at node N, all shared nodes below N will get a reference to N, and the worker will attempt to position itself at N, as fast as possible, in order to take a piece of that work. The worker stops backtracking in any of the following situations:

1. N is reached.
2. The available work at node N is taken by the other workers. In this case, the worker repeats the procedure of determining the nearest live node described above by looking for nodes higher up than N in the current branch.
3. The worker is alone in a sequential node with available work. In this case, it takes a piece of work from the sequential node.
4. There is a pending cut that must be performed by that worker (see Section 4.6.3).

4.5.2 Matching Workers

The second part of the scheduling work algorithm matches idle workers with busy workers having excess load. Matching each idle worker with the nearest busy worker with maximum load is expected to be a good heuristic. Such matching minimizes copying overhead by reducing the amount and frequency of copying. Efficiently supporting this heuristic is not an easy task, but we now present a mechanism to support a version of this heuristic.

The goal here is to match idle workers with busy workers having excess load in such a way that a worker with maximum load within a subtree will be assigned to the closest idle worker. The basic idea of the matching mechanism is as follows. When a worker Q becomes idle, it first determines a set of busy workers, within its subtree[6], which are not closer to any other idle worker. Then, Q selects for sharing the worker P which has maximum load in this set.

If Q cannot find any worker with excess load in its subtree, it will determine a set of busy workers outside its subtree that are not closer to any other idle worker. Then Q backtracks to the nearest node, in its branch, with any worker in this set. After backtracking to that node, Q will have busy workers in its new subtree, and the same idea described above for selecting P within Q's subtree can be used.

If Q cannot find any busy worker with excess load in the system, it will try to position itself at a suitable node in its branch as will be described in Section 4.5.3.

[6]When we say a worker subtree, we mean the subtree rooted at the current node of that worker.

We divide the matching mechanism based on the idea described above into two parts. The first part is described in Section 4.5.2.1, and it concerns selecting P within Q's subtree. The second part is described in Section 4.5.2.2, and it concerns selecting P outside Q's subtree. The matching mechanisms described below use the following global information:

1. A counter, *load*, is associated with each worker containing the current load of the corresponding worker. The measure of load used in Muse is the number of private unexplored alternatives.
2. A register, *currentnode*, is associated with each worker containing a reference to its current position (shared node) in the tree.
3. A bitmap, *idlemap*, contains the current idle workers in the system.

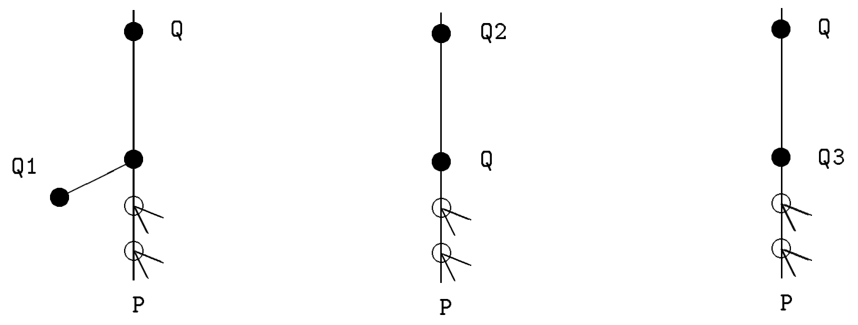
As mentioned in Section 4.2, each shared node is associated with a shared frame which contains a bitmap, *workersbitmap*, representing workers within the subtree rooted at the node.

4.5.2.1 Matching Workers within the Current Subtree

The idle worker Q selects P within its subtree as follows. Q checks whether there are busy workers in its subtree. If there are, it will determine the set of those busy workers that are not closer to any other idle worker. Then it will select for sharing the worker P which has maximum load in this set.

The worker Q can determine which workers are busy (*busy-set*) and which are idle (*idle-set*), in its subtree, from the workersbitmap of its current node and from the idlemap. (busy-set and idle-set are two local bitmaps.) Q can determine a subset of busy workers, in its subtree, which are not closer to any other idle worker, as follows. Q removes from its busy-set busy workers that are in the idle-set subtrees. Positions of idle-set workers are known from their currentnode registers. Finally, Q determines the one with maximum load in the remaining subset of those busy workers by investigating their load counters.

This mechanism allows Q to request sharing from P in the situations shown in Figure 4.2 (a) and (b), but not in (c). (Here we always refer to the current idle worker as Q and the other idle workers as $Q1 \ldots Qn$.) In Figure 4.2 (a), P is not in $Q1$'s subtree. We allow Q to request sharing work from P, because $Q1$ could take a long time to backtrack to a node with P. $Q1$ could, for instance, reach a sequential node that should be processed. In Figure 4.2 (b), there are no idle workers between P and Q. In Figure 4.2 (c), there is an idle worker ($Q3$) between P and Q, i.e. $Q3$ is the nearest idle worker to P.



(a) Q will request sharing from P.

(b) Q will request sharing from P.

(c) Q will not request sharing from P.

Figure 4.2: Matching workers within Q's subtree.

4.5.2.2 Matching Workers outside the Current Subtree

When Q cannot find any busy worker with excess load in its subtree, it will try to find one outside its subtree as follows. It first determines the set of busy workers outside its subtree that are not closer to any other idle worker. Then, it determines the nearest node N of those busy workers by investigating nodes in its branch that have any P with excess load. After that, Q performs backtracking as long as none of the following situations occurs:

1. Q reaches the node N.
2. Q reaches a sequential node that should be processed.
3. Q reaches a node with a pending cut that must be performed by Q.
4. P's private work is exhausted.
5. P shares its private work with another idle worker.

After backtracking to the node N, P will be in Q's subtree. Then Q uses the mechanism described in Section 4.5.2.1 to select a new P that currently has maximum load within its new subtree.

In order to allow other idle workers in the same situation as Q to select other busy workers simultaneously, we make P and Q invisible to those idle workers. The mechanism used for supporting invisible workers is as follows. There is an additional global bitmap, *invisibleworkersmap*. When an idle worker Q wants to reserve a busy worker P while backtracking, Q sets the two bits corresponding to Q and P in the invisibleworkersmap. The bits are resets when Q stops backtracking (for any of the reasons mentioned above).

The invisibleworkersmap is used only for matching workers outside the current subtree, so P could be requested for sharing by another idle worker.

The idle worker Q determines busy workers that are outside its current subtree and are not closer to any other idle worker as follows. It first determines which workers are visible and busy (*visible-busy-set*), and which are visible and idle (*visible-idle-set*) outside its subtree. It determines those workers from the workersbitmap of its current node, the invisibleworkersmap, and the idlemap. Then Q removes from its visible-busy-set workers that are in visible-idle-set subtrees. The positions of those idle workers are known from their currentnode registers. Notice that if Q itself is in any idle worker's subtree, Q should not backtrack since that idle worker is in a better position in the tree for requesting sharing. Therefore, before Q backtracks it has to check whether it is in the subtree of any idle worker of the visible-idle-set. One possible way to perform this check efficiently is for Q to add itself to the visible-busy-set before removing any worker, and check whether it has been removed from this set when some workers are removed. If removing workers from the visible-busy-set terminates and Q is still in this set, Q could backtrack and not otherwise.

The mechanisms described in this section allow Q to stop backtracking in the situations shown in Figure 4.3 (a) and (b), but not in (c). In Figure 4.3 (a), P is in $Q1$'s subtree. In Figure 4.3 (b), Q is in $Q2$'s subtree, i.e. $Q2$ is closer to P. In Figure 4.3 (c), P is not in $Q3$'s subtree, and Q finds P visible before $Q3$ does.

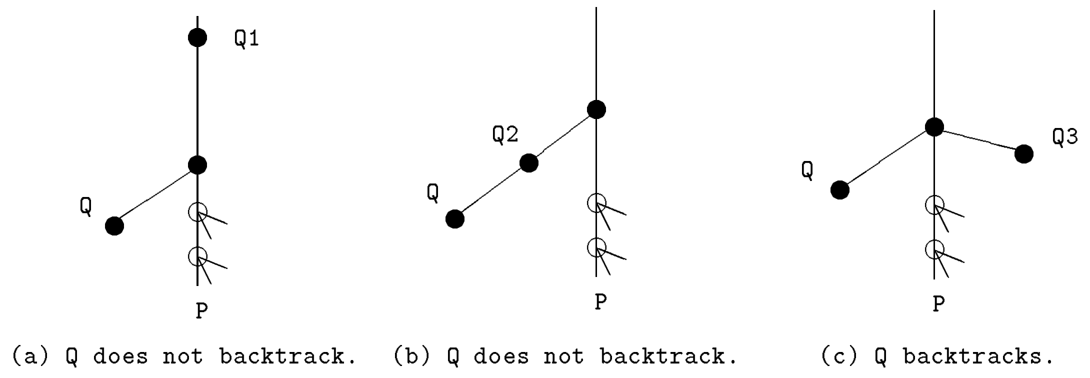


Figure 4.3: Matching workers outside Q's subtree.

4.5.3 Idle Workers

The third part of the basic algorithm concerns the role of idle workers when there are neither any accessible live nodes nor busy workers with excess private load. In this case, idle workers will stay at nodes that allow newly generated work to be shared with low overhead. In Section 4.5.3.1 we discuss heuristics that attempt to achieve that.

The measure of excess load used in Muse is the number of private unexplored alternatives. Using this measure, there can be a situation in which there is work at some shared nodes which is not visible to idle workers, but only to busy workers that will not generate any new work. In this situation, idle workers will be idle forever, although there is excess work in the system. In Section 4.5.3.2 we will discuss a mechanism that detects the above situation and allows the idle workers to share that work.

4.5.3.1 Distributing Idle Workers

The following heuristics attempt to distribute idle workers over the tree in such a way that when a busy worker generates work, an idle worker will share this work with low overhead. An idle worker Q leaves (backtracks from) its current node only if one of the following two situations occurs:

1. there are busy workers outside Q's subtree and Q is not in the subtree of any other idle worker, or
2. all workers in the current subtree are idle.

In the first situation, each idle worker will block all other idle workers in its subtree from backtracking. Only topmost idle workers in nonoverlapping subtrees will backtrack until one of them reaches a node common to all workers in the system. Moving some idle workers to nodes common to many busy workers allows sharing of work to be started earlier when a worker generates work. If the new position of an idle worker is closer to a worker that generates work, sharing overhead decreases. But if the old position was closer to the worker that generates work, sharing overhead increases. Blocking idle workers from backtracking when they are in any idle worker's subtree ensures that none of those idle workers loses its position. Experimental results have indicated performance improvements for this heuristic in comparison to different other heuristics for distributing idle workers over the tree.

In the second situation, we allow idle workers to backtrack to either the nearest sequential node or a node with a busy worker below. This is the right position for an idle worker to stay at for the following reasons. The last worker backtracking to a sequential node will take a branch from that node for processing and it might generate work. Also, for a node that has busy workers below, any of those workers might generate work.

4.5.3.2 Finding Invisible Work

The mechanism for finding work that is invisible to idle workers and sharing that work is based on the following idea. When an idle worker Q cannot find work in the system and there are busy workers in its subtree, Q asks each of them to check if it has work in its branch. Q will try to ask workers that have been busy for a long time and have never been asked before by any worker in the system. If any of them finds work, such work will be shared with Q.

To support this idea, there is a global bitmap, *mayaccessworkmap*, which contains workers that may access shared work. It is initially empty (all bits are reset). On every sharing the two bits corresponding to the two workers involved on sharing will be set. When a busy worker is asked for sharing and it turns out that the worker does not access shared work, the worker bit is reset.

There is also a local bitmap for every worker, *stablebusy*, which contains workers that were busy for a certain amount of time within the current subtree. When a worker becomes idle, it sets its own stablebusy bitmap to the busy workers within its subtree. While there are busy workers within its current subtree, it updates its stablebusy bitmap by resetting bits corresponding to workers that become idle. After k iterations of investigating the busy workers within its current subtree, it finds suitable workers to ask for invisible work by investigating its stablebusy bitmap and the mayaccessworkmap. (The selected value of k for the Muse implementation on bus-based machines is 10.) Bits that are set in both bitmaps correspond to these workers. The idle worker could ask these workers for sharing, but it asks only those that are not closer to any other idle worker as described in Section 4.5.2. If any of these workers accesses shared work, that work will be shared by the idle worker. Otherwise, the busy worker will refuse the sharing request, reset its bit in the mayaccessworkmap, and updates the nearest-livenode field in shared nodes to the dummy value (see Section 4.5.1).

This mechanism allows a busy worker that does not generate new work after sharing and does not access a live node to be asked for sharing at most once. It also allows workers to update the nearest-livenode fields, to avoid doing that later on. Thus, this mechanism entails almost no overhead.

4.6 Cut and Sequential Side Effects

The current implementation of Muse supports the the full Prolog language with its standard semantics. It also supports asynchronous (parallel) side effects and internal database predicates. In this implementation we have simple mechanisms for supporting cut and all standard Prolog side effects predicates (e.g. read, write, assert, retract, etc).

A simple way to guarantee the correct semantics of sequential side effects is to allow execution of such side effects only in the leftmost branch of the whole search tree. The current Muse implementation does not support suspension of branches. That is, a worker that executes a sequential side effect predicate in a branch that is not leftmost of the whole tree waits until that branch is leftmost of the tree. Similarly, to support the *findall* predicate a worker that generates a findall solution will wait until its branch is leftmost in a proper subtree. (A novel method has been implemented in the current (1992) version of Muse to remove the need to wait until becoming leftmost for assert and findall. This method is described in Section 7.8.5.)

The effect of *cut N* in a branch is to prune all branches to the right of the branch in the subtree rooted at the node N. In the current implementation of cut, when a worker executing cut is not leftmost in the relevant subtree, it will prune as much as it can and leave the pruning of the remaining branches to a worker to its left and then proceed with executing operations following the cut. When a worker detects that all left branches have failed, it will try to prune as much as it can until all branches within the scope of the cut have been pruned. The idea of this cut algorithm was proposed in [24].

In general, supporting cut, findall, and side effects requires efficient mechanisms for checking whether or not the current branch is leftmost in a tree. Also, efficient mechanisms for identifying workers working in branches to the right are required for supporting cut. In this section we discuss such mechanisms.

In the current Muse implementation, we have a very simple representation of a Prolog search tree. A worker can move only in its branch, i.e. there are no extra pointers in the nodes to the sibling nodes. The advantage of this simple representation is that adding and removing nodes are very efficient operations — neither locking overheads nor overheads for maintaining the topology of the tree are needed. The disadvantage is that a worker cannot traverse nodes in the other branches. This complicates the task of the Muse scheduler to perform a leftmost check, and to identify workers in branches to the right. To illustrate how the Muse scheduler carries out these operations, we first describe the representation of a Prolog search tree in the current Muse implementation.

4.6.1 The Tree Representation

A Prolog search tree is represented in the current implementation of Muse as follows. Each shared node is associated with a shared frame containing a *workersbitmap* identifying workers accessing the node (and other information; see Section 4.2). Each worker sharing a node has a choicepoint frame pointing to the shared frame associated with the node. Each alternative of a node is associated with its number. Each worker exploring an alternative of a node keeps the alternative number, *alt-number*, in its choicepoint frame associated with the node (or in a separate stack). Figure 4.4 shows the representation of a tree of two nodes a and b, and three workers X, Y, and Z. Each worker keeps two choicepoint frames in its choicepoint stack corresponding to the nodes a and b. The alt-number field in each choicepoint frame contains the corresponding alternative number of the node. For worker X, the alt-number is 1 in each of its choicepoint frames.

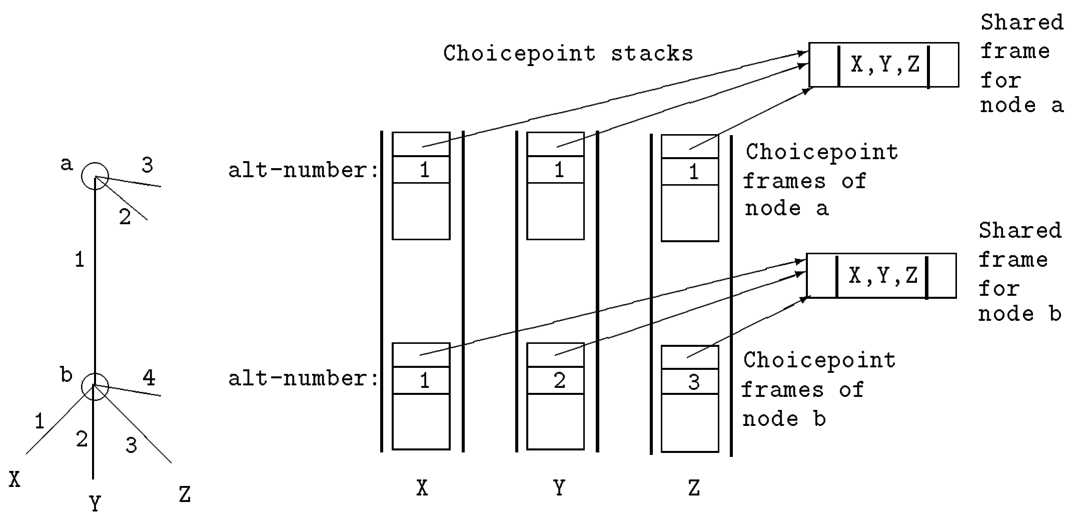


Figure 4.4: Representation of the tree.

4.6.2 The Leftmost Check

Let us suppose that the worker Y in Figure 4.4 wants to check if its branch is leftmost. It starts from the bottommost shared node, b, determines the other workers accessing b (from the shared frame associated with b), and then investigates the alt-number in the choicepoint frames of these workers to determine if any of them has alt-number less than 2. It finds X with alt-number 1. So, Y is not leftmost. But if X wants to perform the same check, it will not find any other worker that has alt-number less than its alt-number in the choicepoint frames associated with the nodes b and a. Thus, X is leftmost.

A leftmost algorithm based on this idea is as follows. When a worker P wants to check whether it is leftmost in a tree rooted at a shared node N, it starts from the bottommost shared node in its branch, reads the workersbitmap associated with the node to find all other workers sharing the node, and investigates the alt-number corresponding to that node for the other workers in the workersbitmap. The worker P finds the alt-number field of each other worker by calculating the remote address from the offset of its alt-number field in its choicepoint stack and the base address of each choicepoint stack of the other workers. If P finds any such alt-number less than its alt-number, P is not leftmost. But if P does not find any alt-number less than its alt-number and the current node is not N, P will repeat the same operation at the next upper node and so on until reaching N.

In this algorithm, the worker X in Figure 4.4 investigates four remote choicepoint frames (two at the node b, and then two at the node a). For a branch with n shared parallel nodes and a system of W workers, the number of remote choicepoint frames to be investigated will be $O(n * W)$. Also, all the n nodes will be investigated. This means that using this algorithm without substantial improvements is impractical for efficiency reasons.

We present improvements of the above algorithm that make it practical and reasonably efficient. The first improvement reduces the number of nodes in a branch to be investigated. The second improvement reduces the number of remote choicepoint frames to be investigated from $O(n * W)$ to only $W - 1$. The third improvement reduces the number of remote choicepoint frames to be investigated to zero.

4.6.2.1 Reducing Investigated Nodes

The first improvement reduces the number of nodes to be investigated by a worker in checking if its branch is leftmost. It is based on associating with each shared node a pointer, *nearest-leftnode*, pointing to the nearest upper node with a branch to its left. When performing the leftmost test the nearest-leftmost pointer is used to bypass nodes where no branches to the left exists. Whenever a leftmost test terminates with failure in a node N all pointers in the current branch, rooted at N, is updated to point to N. At termination with success all pointers are updated to the value of nearest-leftmost in N.

Each Muse worker shares several private nodes at a time. The nearest upper node that could have a branch to its left is the bottommost shared node. So, when a worker makes its private nodes shareable, it sets the nearest-leftnode field in each private node to the bottommost shared node. An optimization is to allow a leftmost worker at a bottommost shared node to set the nearest-leftnode field of its private nodes to the nearest upper node with a branch to its left. This improvement is also useful to all the Aurora schedulers, specially the Bristol scheduler.

4.6.2.2 Reducing Remote Access

The idea of the second improvement is that when a worker in an alternative L at node N performs a leftmost check, it will investigate only workers in the other alternatives at N. It determines these workers from the difference between workersbitmaps at N and its child node in the alternative L. (Each choicepoint frame associated with a shared node contains a *child* field that refers to the choicepoint frame associated with its child node.) If N is the bottommost shared node in a branch, all other workers in N will be investigated. For instance in Figure 4.4, the worker X will investigate the choicepoint frames of Y and Z at the node b, but none at the node a, since there are no new workers at a.

By this improvement only one choicepoint frame of each of the other workers will be investigated in a complete leftmost check in the whole tree.

4.6.2.3 Avoiding Remote Access

The third improvement reduces the number of remote choicepoint frames to be investigated to zero on performing a leftmost check. It is based on associating with each shared node a bitmap, *activealternativesmap*, for indicating active alternatives at the node. When an alternative is taken from a shared node the corresponding bit is set, and when an alternative fails (i.e. when all workers on that alternative have backtracked) the corresponding bit is reset. (The size of the activealternativesmap is equal to the maximum number of alternatives in a parallel node; i.e. number of clauses in a parallel predicate.) A worker on an alternative L of a shared node is leftmost at that node only when all bits less than L are reset.

This improvement reduces the number of remote choicepoint frames to be investigated on each leftmost check to zero. Efficient implementation of this improvement requires limiting the maximum number of clauses in each parallel predicate to the size of one machine word. (For instance, 32 clauses can be represented by a 32-bit word). This can be done by program transformation. This improvement has not been implemented.

4.6.3 Cut

The effect of *cut* N in a branch is to prune all branches to the right of the branch in the subtree rooted at the node N. It is implemented in Muse as follows. N can be either a private node or a shared node. In the former case, cut is processed by a worker exactly as in the sequential Prolog implementation. In the latter case, we have two different situations: (1) the cutting branch is leftmost, or (2) the cutting branch is not leftmost. In situation (1), the worker executing cut requests all other workers in the node N to backtrack to the parent of N, removes itself from N and all shared nodes below N in its branch, deallocates its choicepoint frames associated with those nodes, and then proceeds with processing the operations following the cut. (Deallocation of a shared frame is done by the last worker backtracking from the corresponding node.)

In situation (2), the worker executing cut finds a shared node M below N that has a branch to its left. It removes unexplored alternatives at M and requests all workers in branches to its right to backtrack to the parent of M. If there exist more branches to be pruned, the worker saves at M information indicating the alternative number in the current branch at M, say L, and the scope of the cut (i.e. N). A simple way of identifying the workers to the right is to investigate the choicepoint frames of all the other workers at M and send a pruning request to each worker that has its alt-number equal to or greater than L. Another efficient way of identifying these workers (based on the idea presented in Section 4.6.2.2) is to first find all the other workers on the child node of M in the alternative L, and then investigate only the choicepoint frames of the new workers at M. The new workers at M are obtained from the difference between workersbitmaps of M and its child node in the alternative L.

In order to avoid overhead through every worker backtracking to M to perform a leftmost check, we also save at M the name of one of the workers that are in branches to the left of L. Only that worker will perform a leftmost check for the alternative

L at M. If the leftmost check finds another worker Q that has its alt-number less than L and there exist branches above to be pruned, Q will be saved in the node M and the backtracking worker will try to find work in the tree. Otherwise, the backtracking worker will restart the pruning of branches by examining every node in its branch starting from M's parent until it reaches the node N. The worker will perform the following at each node: (1) it determines the alternative number $L1$ of the node in the current branch from its choicepoint frame associated with that node, (2) finds the difference between the workersbitmaps associated with the node and its child node in the current branch, (3) requests pruning from workers on alternatives greater than $L1$, (4) removes the remaining unexplored branches, and (5) checks if there is any worker on alternatives less than $L1$. If there exists such a worker, the name of that worker and $L1$ will be saved at the node. But if there is no such a worker and N is not reached, the same procedure will be performed on the next upper node in the current branch.

4.7 Performance Results

In this section we discuss the overhead for maintaining information when supporting cut, findall, and sequential side effect constructs. We also show the performance of the Muse scheduler for different classes of benchmarks. The timing results of Muse will be compared with the corresponding results for Aurora [65] with the Manchester scheduler [24]. This gives some ideas about how the Muse scheduler performs in comparison with another good scheduler for a similar system. Both Aurora and Muse are based on the same sequential Prolog, SICStus version 0.6. Neither Muse nor Aurora with the Manchester scheduler[7] has implemented any special treatment of speculative work. The main difference between Muse and Aurora in the implementation of cut, findall, and sequential side effect constructs is that Aurora supports suspension of branches whereas Muse does not. For instance, an Aurora worker executing cut in a nonleftmost branch of the cut node will suspend the branch and try to find work outside the cut subtree. In Muse, the pruning operation suspends while the worker proceeds with the next operation following the cut as described in Section 4.6.3. Other differences between Aurora and Muse are that Aurora uses a more general representation of the Prolog search tree than the one used in Muse, and Aurora is based on another model for OR-parallel execution of Prolog [92]. Finally, both run on the same Sequent Symmetry S81, with 16 processors and 32 MBytes of memory, which is available to us at SICS. The total scheduling overhead in Muse will also be discussed.

[7]A new Aurora scheduler which handles speculative work better is under development at the University of Bristol.

4.7.1 Benchmarks

The group of benchmarks used in this chapter 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*) has relatively well understood granularity, and has been used by several researchers in [14, 24, 83] and in Chapter 3. It is described in Section 3.9.1. This set of benchmarks does not contain major cuts and look for all solutions of the problem.

The second set of benchmarks (*mm1, mm2, mm3, mm4, num1, num2, num3, num4*) contains major cuts and has been used for studying different cut schemes [45]. *mm* is a master mind program with four different secret codes. The numbers program (*num*) generates the two largest numbers consisting of given digits and fulfilling specified requirements. The *num* program was run with four different queries. This set is divided into two groups known in the following sections as *mm* and *num*. All the benchmarks of the second set look for the first solution of the problem.

4.7.2 Results of Benchmarks with no Major Cuts

To evaluate the overhead for maintaining information that supports cut, findall, and sequential side effect constructs, we compare the execution times of the first set of benchmarks for a version of Muse supporting full Prolog with another those for version that supports a parallel version of Prolog named Commit Prolog described in Section 3.6. Commit Prolog is a Prolog language with cavalier commit [23] instead of cut, asynchronous (parallel) side effects and internal database predicates instead of the synchronous (sequential) counterparts, and sequential and parallel annotations. The standard Prolog semantics of cut and sequential side effects was obtained in Commit Prolog by following a few rules that restrict the degree of OR-parallelism as described in Section 3.6.

Table 4.1 presents the execution times (in seconds) from the execution of the first set of benchmarks by the Muse version of Commit Prolog. The execution times given are the shortest obtained from eight runs. Times are shown for 1, 4, 8, 12, 15 workers with speedups given in parentheses. These speedups are relative to execution times of SICStus0.6 on one Sequent processor shown in the last column. For benchmarks with small execution times the timings shown refer to repeated runs, the repetition factor being shown in the first column. $\sum$ 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 benchmarks as one benchmark. In the following tables, the last row for each group of a set of benchmarks represents the whole group as one benchmark.

For all programs in Table 4.1, except *parse1* – *parse3* and *farmer*, increasing the number of workers results in shorter execution times. For *parse1* – *parse3* and *farmer*, increasing the number of workers beyond a certain limit results in slightly

	Workers					
Benchmarks	1	4	8	12	15	SICStus
8-queens1	6.83(0.92)	1.72(3.65)	0.87(7.22)	0.59(10.6)	0.48(13.1)	6.28
8-queens2	17.38(0.95)	4.36(3.77)	2.21(7.43)	1.49(11.0)	1.20(13.7)	16.43
tina	14.44(0.95)	3.67(3.74)	1.90(7.22)	1.31(10.5)	1.08(12.7)	13.71
salt-mustard	2.10(0.96)	0.54(3.74)	0.28(7.21)	0.19(10.6)	0.16(12.6)	2.02
• High Σ	40.75(0.94)	10.29(3.74)	5.26(7.31)	3.59(10.7)	2.93(13.1)	38.44
parse2*20	5.99(0.95)	1.78(3.20)	1.28(4.45)	1.10(5.17)	1.11(5.13)	5.69
parse4*5	5.53(0.95)	1.50(3.51)	0.93(5.66)	0.76(6.92)	0.73(7.21)	5.26
parse5	3.92(0.95)	1.02(3.66)	0.57(6.54)	0.46(8.11)	0.45(8.29)	3.73
db4*10	2.39(0.95)	0.65(3.49)	0.39(5.82)	0.30(7.57)	0.28(8.11)	2.27
db5*10	2.91(0.95)	0.80(3.45)	0.47(5.87)	0.36(7.67)	0.33(8.36)	2.76
house*20	4.41(0.96)	1.33(3.18)	0.83(5.10)	0.66(6.41)	0.62(6.82)	4.23
⋆ Med Σ	25.15(0.95)	7.14(3.35)	4.53(5.28)	3.69(6.49)	3.58(6.69)	23.94
parse1*20	1.59(0.94)	0.60(2.50)	0.56(2.68)	0.59(2.54)	0.63(2.38)	1.50
parse3*20	1.36(0.96)	0.56(2.32)	0.50(2.60)	0.52(2.50)	0.54(2.41)	1.30
farmer*100	3.19(0.96)	1.38(2.22)	1.39(2.21)	1.38(2.22)	1.40(2.19)	3.07
∗ Low Σ	6.14(0.96)	2.54(2.31)	2.46(2.39)	2.49(2.36)	2.57(2.28)	5.87
Σ	72.04(0.95)	19.98(3.42)	12.25(5.57)	9.79(6.97)	9.11(7.49)	68.25

Table 4.1: Muse (Commit Prolog) execution times (in seconds) for the first set of benchmarks.

longer execution times. This degradation is due to the extra execution time scheduling overhead for programs with fine granularity. The scheduling overhead will be discussed in Section 4.7.4.

Table 4.2 presents the execution times (in seconds) from the execution of the first set of benchmarks for Aurora and the Muse version of full Prolog. The execution times given refer to each group of the first set of benchmarks and to the whole first set of benchmarks. If we compare the execution times for Muse in Tables 4.1 and 4.2, we find that the Muse version of full Prolog is slower than the Muse version of Commit Prolog by around 0% for the 1 worker case, 2% for 4 workers, 2% for 8 workers, 3% for 12 workers, and 2% for 15 workers. This means that the overhead of maintaining information that supports cut, findall, and sequential side effect constructs in Muse is very low. Table 4.2 shows in the last row the ratio of the execution times of Aurora to the execution times of Muse for the first set of benchmarks. Aurora timings are longer than Muse timings by 24% to 40% for 1 to 15 workers.

Figure 4.5 shows the speedup curves of Muse and Aurora for the three groups of the first set of benchmarks: High, Medium, and Low. Notice that all speedups in this chapter are relative to SICStus0.6. The results shown in Tables 4.2 and 4.5 show that the Muse scheduler performs well on each group of the first set of benchmarks.

Benchmarks	Workers 1	4	8	12	15	SICStus
Aurora						
○ High	51.97(0.74)	13.28(2.89)	6.76(5.69)	4.58(8.39)	3.74(10.3)	38.44
◇ Med	30.20(0.79)	9.09(2.63)	6.25(3.83)	5.37(4.46)	5.09(4.70)	23.94
o Low	7.31(0.80)	3.76(1.56)	3.70(1.59)	3.94(1.49)	4.14(1.42)	5.87
Σ	89.48(0.76)	26.16(2.61)	16.75(4.07)	13.94(4.90)	13.00(5.25)	68.25
Muse						
● High	41.00(0.94)	10.61(3.62)	5.32(7.23)	3.64(10.6)	2.98(12.9)	38.44
⋆ Med	25.24(0.95)	7.23(3.31)	4.64(5.16)	3.85(6.22)	3.70(6.47)	23.94
∗ Low	6.16(0.95)	2.59(2.27)	2.52(2.33)	2.58(2.28)	2.63(2.23)	5.87
Σ	72.40(0.94)	20.46(3.34)	12.50(5.46)	10.09(6.76)	9.31(7.33)	68.25
Aurora/Muse	1.24	1.28	1.34	1.38	1.40	–

Table 4.2: Aurora and Muse (full Prolog) execution times (in seconds) and the ratio between them for the first set of benchmarks.

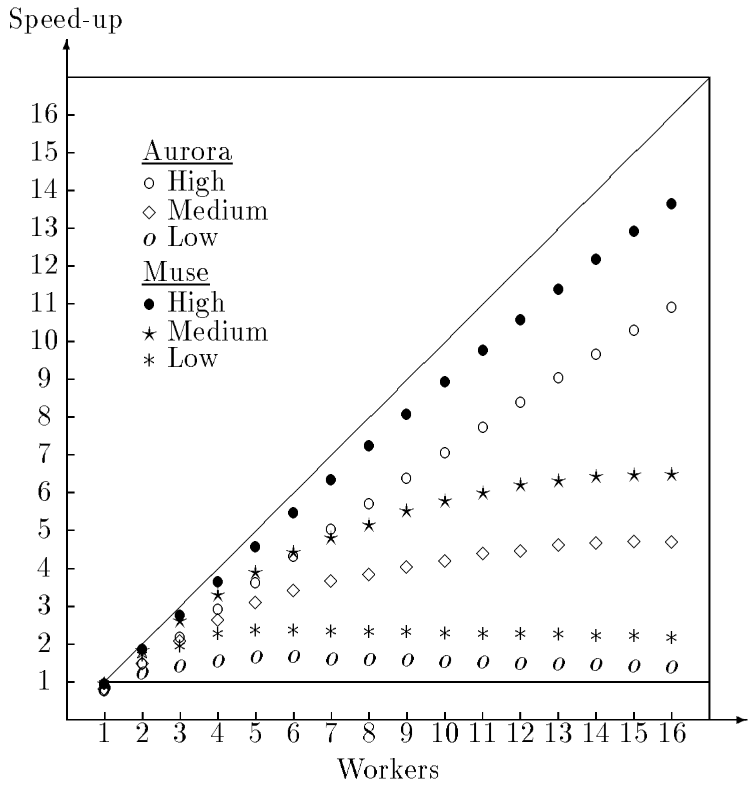


Figure 4.5: Speedups of Muse and Aurora for the first set of benchmarks.

4.7.3 Results of Benchmarks with Major Cuts

Here we show timing results for programs with major cuts. Table 4.3 presents the execution times (in seconds) from the execution of the second set of benchmarks for the Muse version of full Prolog. The execution times given are the mean values obtained from eight runs. For programs with (major) cuts, mean values are more reliable than best values because scheduling of speculative work changes from one run to another, causing larger variations in timing results.

	Workers					
Benchmarks	1	4	8	12	15	`SICStus`
`mm1`	4.10(0.96)	2.05(1.93)	1.16(3.41)	0.99(3.99)	0.93(4.25)	3.95
`mm2`	3.23(0.98)	1.09(2.90)	0.81(3.90)	0.59(5.36)	0.51(6.20)	3.16
`mm3`	9.26(0.97)	3.17(2.83)	2.12(4.24)	1.55(5.79)	1.39(6.46)	8.98
`mm4`	15.80(0.96)	5.06(3.00)	2.75(5.52)	1.79(8.47)	1.56(9.72)	15.17
• `mm` Σ	32.39(0.97)	11.37(2.75)	6.84(4.57)	4.93(6.34)	4.39(7.12)	31.26
`num1`	1.63(0.99)	0.94(1.72)	0.52(3.12)	0.31(5.23)	0.28(5.79)	1.62
`num2`	2.67(0.99)	0.91(2.91)	0.47(5.64)	0.34(7.79)	0.27(9.81)	2.65
`num3`	2.86(0.99)	0.83(3.41)	0.44(6.43)	0.31(9.13)	0.27(10.5)	2.83
`num4`	3.69(0.99)	0.95(3.84)	0.50(7.30)	0.33(11.1)	0.28(13.0)	3.65
⋆ `num` Σ	10.85(0.99)	3.63(2.96)	1.93(5.57)	1.29(8.33)	1.10(9.77)	10.75
Σ	43.24(0.97)	15.00(2.80)	8.77(4.79)	6.22(6.75)	5.49(7.65)	42.01

Table 4.3: Muse execution times (in seconds) for the second set of benchmarks.

Table 4.4 presents the execution times (in seconds) from the execution of the second set of benchmarks for Aurora. The execution times given refer to each group of the second set of the benchmarks and to the whole second set of benchmarks. It also shows in the last row the ratio of the execution times on Aurora to the execution times on Muse for the second set of benchmarks as one benchmark. Aurora timings are longer than Muse timings by 19% to 101% for 1 to 15 workers. There are two possible explanations for this difference in performance results between Muse and Aurora for this set of benchmarks. The first one is that the dispatching on the bottommost used in Muse entails less speculative work than that of dispatching on the topmost used in Aurora (by the Manchester scheduler). The second reason is that for those programs suspension of branches as used in Aurora is not the best choice.

Figure 4.6 shows the speedup curves of Muse and Aurora for the two groups of the second set of benchmarks: mm and num. These curves correspond to speedups obtained from Tables 4.3 and 4.4.

Benchmarks	Workers					
	1	4	8	12	15	SICStus
◦ mm Aurora	39.39(0.79)	22.73(1.38)	13.28(2.35)	10.75(2.91)	9.38(3.33)	31.26
◇ num Aurora	12.09(0.89)	4.54(2.37)	2.56(4.20)	1.95(5.51)	1.65(6.52)	10.75
Σ	51.48(0.82)	27.27(1.54)	15.84(2.65)	12.70(3.31)	11.03(3.81)	42.01
Aurora/Muse	1.19	1.82	1.81	2.04	2.01	–

Table 4.4: Aurora execution times (in seconds) and the ratio between Aurora and Muse timing for the second set of benchmarks.

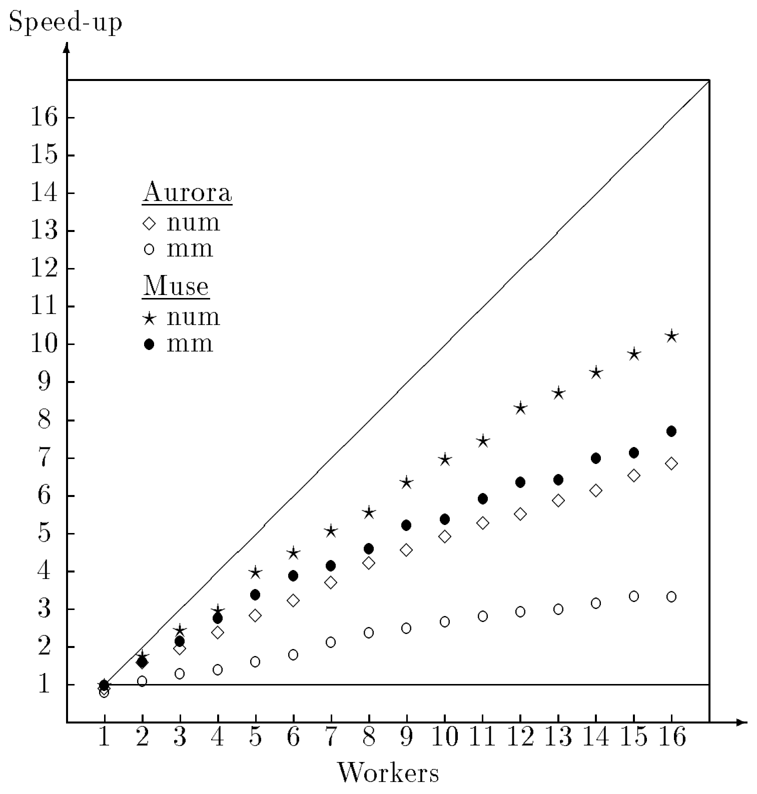


Figure 4.6: Speedups of Muse and Aurora for the second set of benchmarks.

4.7.4 Scheduling Overheads

In this section, we present and discuss briefly the time spent in scheduling activities in the Muse version of full Prolog. A Muse worker's time is distributed over the following three basic activities:

1. *Prolog:* time spent executing Prolog, checking for the arrival of requests, and maintaining the value of the local load.
2. *Idle:* time spent when there is no available work in the system: looking for a worker with excess local work or positioning itself in the tree.
3. *Scheduling:* time spent doing scheduling activities. This includes time spent sharing nodes, grabbing work from shared nodes, selecting busy workers for sharing nodes, copying data, synchronization, moving up within the shared region, acquiring a spin lock, sending requests, etc.

Table 4.5 shows the total time spent in each activity, and the corresponding percentage of the total time, for the first set of benchmarks. Times shown in Table 4.5 have been obtained from an instrumented system of Muse on Sequent Symmetry. Those times include the time spent on measurements. The times obtained from an instrumented system are longer than those obtained from an uninstrumented system by around 10%. 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	80885(90.0)	83158(75.7)	84829(63.8)	86424(56.2)
Idle	2893(3.2)	13010(11.8)	28488(21.4)	41765(27.2)
Scheduling	6069(6.8)	13684(12.5)	19731(14.8)	25576(16.6)
Total	89847(100.0)	109852(100.0)	133048(100.0)	153765(100.0)

Table 4.5: Total times (in milliseconds) spent in the basic activities of a Muse worker for the first set of benchmarks.

As mentioned in Section 4.7.1, 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 contains benchmarks that lack parallelism. Lack of parallelism explains the decrease in the *Prolog* percentage and increase in the *Idle* percentage with increasing number of workers in Table 4.5. The sum of these two percentages is almost constant (only 4.1% difference) from 8 workers to 15 workers. The *Scheduling* overhead increases from the 4 workers case to the 8 workers case by 5.7%, from the 8 workers to the 12 workers by 2.3%, and from the 12 workers to the 15 workers by 1.8%.

A possible explanation for the increase of the overhead with the number of workers is shown in Table 4.6, which shows the effect of increasing the number of workers on the number of tasks and task sizes (expressed as the number of Prolog calls per task) for the first set of benchmarks. In Table 4.6 the granularity of parallelism is decreased from the 4 workers case to the 8 workers case by a factor of around 2 whereas from 8 workers to 12 workers by a factor 1.2, and from 12 workers to 15 workers also by a factor 1.2.

	Muse Workers			
	4	8	12	15
Total Number of Tasks	13807	28145	34530	39537
Prolog Calls per Task	64	31	26	22

Table 4.6: Average number of tasks and task sizes for the first set of benchmarks.

The number of Prolog calls equivalent to the scheduling overhead per task are shown in Table 4.7. The numbers in Table 4.7 are calculated from Tables 4.5 and 4.6. Table 4.7 indicates that the scheduling overhead is equivalent to around 5 – 7 Prolog calls per task, where the time of a Prolog procedure call is between 83 to 100 microseconds. Similar figures for the scheduling overhead in terms of Prolog calls per task have been reported for Aurora with the Manchester scheduler [83]. The time of a Prolog procedure call for Aurora is longer than the corresponding time for Muse by a factor of around 1.20 – 1.25, the ratio of the speed of the Muse engine to that of the the Aurora engine. That is, the scheduling overhead of Muse is less than the scheduling overhead of Aurora with the Manchester scheduler by approximately the same factor.

	Muse Workers			
	4	8	12	15
Scheduling Overhead per Task in Prolog calls	4.80	5.10	6.05	6.51

Table 4.7: Scheduling overhead per task for the first set of benchmarks.

Both Muse and Aurora with the Manchester scheduler attempt to minimize the increase of the scheduling overhead as the number of workers is increased. They do this by supporting mechanisms for avoiding a steady decrease in task sizes as the number of workers grows. The idea used in the Muse system is that when a busy worker reaches a situation in which it has only one private parallel node, it will make its private load visible to the other workers only when that node is still alive after a certain number n of Prolog procedure calls. The value of n is a constant chosen to be about equal to the number of Prolog procedure calls equivalent to the scheduling overhead per task. It is 5 on Sequent Symmetry.

The corresponding idea used in the Manchester scheduler is for each busy worker to check for the arrival of requests from other workers on every N Prolog procedure calls. The value of N is also 5 on Sequent Symmetry.

The Muse solution entails execution time overhead only when a worker has only one private parallel node. It also limits the parallelism in this situation only, which in turn prevents the steady decrease in task sizes. The Manchester scheduler solution entails overhead on every Prolog procedure call in the form of updating a counter. The private work is released on every N Prolog procedure calls.

4.8 Future Work

Our future work on the current Muse system is to use more advanced implementation schemes for cut, commit, findall, and sequential side effects. (In the current Muse implementation, commit is translated into cut.)

In the current implementation of cut, we do not maintain information for locating alternatives with cut. Cut schemes based on maintaining such information reduce speculative work. An advanced cut scheme based on one of the schemes presented in [45] will be implemented.

In the current implementation of *assert*, when a worker is going to assert a rule, it waits until its branch is leftmost in the whole tree. One idea is to allow the worker to perform most of the work needed for assert and delay insertion of the rule until the branch is leftmost in the whole tree, having the worker proceed with operations following assert. (This idea is presented in Section 7.8.5.) Information describing the uncompleted assert is saved in the nearest upper node M in the branch with a branch to its left. The last worker that backtracks to M from left branches that have delayed the assert takes care of that uncompleted assert, if it has not already been removed by a cut. That worker either moves the information describing the uncompleted assert to another upper node with a branch to its left, or completes it if the current branch is leftmost in the tree.

Similarly, adding a solution generated by *findall* requires the current branch to be leftmost in a proper subtree. The above idea could be used here also to perform most of the work needed for generating a solution and delay insertion of the solution until its branch is leftmost in the subtree, with the worker proceeding with the next operation.

Regarding calling dynamic predicates and the other side effects, like *retract* and *input/output*, a mechanism for suspending branches is needed. One method of suspension is to save in the shared-memory space the difference between the current state of a worker that is going to suspend its branch and the state corresponding to the nearest upper node M with a branch to its left. Information describing the suspended branch and the location of the saved part of state will be stored in M. As described above, the last worker that backtracks to M from left branches that have caused the suspension of a branch will take care of the suspended branch, if it has not already been removed by a cut. That worker either moves information describing the suspended branch(es) to another upper node N with a branch to its left along with the difference between the computation state at M and N, or restarts the suspended branch if it is leftmost in the whole tree. When a worker is going to restart a suspended branch, it first gets the state corresponding to that branch from the shared-memory space by using information stored in the node M.

The difference in states between two nodes can be computed in a similar way as when doing incremental copying (see Section 3.2.3 for details).

4.9 Conclusions

The principles and implementation of scheduling work and supporting full Prolog in Muse have been presented. Many of the algorithms presented are also applicable to other OR-parallel Prolog approaches. The performance results are very encouraging and the overhead for maintaining information that supports cut, findall, and sequential side effect constructs in Muse on Sequent Symmetry is very low (around 0% for one worker, and 2% – 3% for 15 workers). The total scheduling overhead per task on Sequent Symmetry for the set of benchmarks used in [83], is equivalent to around 5 – 7 Prolog calls per task. For programs with cuts, dispatching on the bottommost gave much better performance results than dispatching on the topmost. The suggestions for improvements and new constructs mentioned in the chapter will be implemented. The implementation of an efficient and simple form of suspension of branches and speculative work will be examined. Using bitmaps in the presented algorithms limits the use of these algorithms for systems with a large number of workers. Mechanisms for large systems (like larger configuration of the Butterfly machine) will also be examined.

4.10 Acknowledgments

We would like to thank Mats Carlsson and Andrzej Ciepielewski for helpful comments on an earlier draft of this chapter.

Chapter 5

Performance of Muse on Switch-Based Multiprocessor Machines

Khayri A. M. Ali *Roland Karlsson* *Shyam Mudambi*

A long version [11] of a paper published in PARLE'92 [12]

THE Muse (multiple sequential Prolog engines) approach has been used to make a simple and efficient OR-parallel implementation of the full Prolog language. The performance results of the Muse system on bus-based multiprocessor machines have been presented in previous chapters. This chapter discusses the implementation and performance results of the Muse system on switch-based multiprocessors (the BBN Butterfly GP1000 and TC2000). The results of Muse execution show that high real speedups can be achieved for Prolog programs that exhibit coarse-grained parallelism. The scheduling overhead is equivalent to around 8 – 26 Prolog procedure calls per task on the TC2000. The chapter also compares the Muse results with corresponding results for the Aurora OR-parallel Prolog system. For a large set of benchmarks, the results are in favor of the Muse system.

5.1 Introduction

With the availability of multiprocessors many approaches to parallel implementations of the Prolog language have been developed. Some of them exploit either OR-parallelism [6, 14, 23, 28, 54, 60, 65, 95] or independent AND-parallelism [32, 50, 61] or a combination of both [19, 30, 56, 94]. The Muse approach described in Chapter 3 exploits OR-parallelism only. In OR-parallel execution branches of a Prolog search tree are executed in parallel.

Muse based on SICStus Prolog [26] (a fast and portable Prolog system) is currently implemented on both bus-based and switch-based shared memory machines. The bus-based machines are TP881V (a 4 (88100) processor machine from Tadpole Technology), a machine constructed at SICS (a 7 (68020) processor VME-bus machine with local/shared memory), Sequent Symmetry S81 (a 26 (i386) processor machine from Sequent). The switch-based machines are BBN Butterfly I (GP1000) and II

(TC2000), with 96 (68020) and 45 (88100) processors respectively. The overhead associated with this adaptation is very low in comparison with the other approaches. It is around 3% to 5% for the bus-based machines. The performance results of Muse on the BBN Butterfly machines will be presented in this chapter. The Muse implementation and its performance results on the constructed prototype and the Sequent Symmetry machine have been presented in Chapters 3 and 4.

The chapter is organized as follows. Section 5.2 briefly describes the Muse approach. Section 5.3 discusses Muse implementation on the Butterfly machines. Section 5.4 presents performance results of the Muse implementation on the Butterfly machines. Section 5.5 compares the Muse results with the corresponding results for another OR-parallel Prolog system, named Aurora. Section 5.6 concludes the chapter.

5.2 The Muse Approach — An Overview

This section briefly describes the Muse approach presented in in Chapters 3 and 4. In the Muse approach (as in other OR-parallel Prolog approaches, e.g. Aurora [65] and PEPSys [14]), OR-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. The Muse execution model is based on having a number of 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 stacks[1]. The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks.

This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the difference between the two states. This reduces copying overhead. Nodes are shared between workers to allow dynamic load balancing, which reduces the frequency of copying.

A node in a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared (private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each shared node is accessible only to workers within the subtree rooted at the node. Private nodes are only accessible to the worker that created them.

[1]The assumed worker's stacks are: *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 [91].

Each worker can be in either engine mode or scheduler mode. In engine mode, the worker works exactly like a sequential Prolog engine on private nodes, but is also able to respond to requests from other workers. In scheduler mode, the worker establishes the necessary coordination with 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 part of a worker state, (2) making local nodes shareable, and (3) grabbing a piece of work from a shared node. The Muse scheduling work strategies to minimize the overhead are as follows.

- The scheduler attempts to share a chunk of nodes between workers on every sharing. This maximizes the amount of shared work between the workers and allows each worker to release work from the bottommost node in its branch (dispatching on the bottommost) by using backtracking with almost no overhead. Dispatching on the bottommost also entails less speculative work than dispatching on the topmost, where work is taken from the topmost live node in a branch (see [45]). (Speculative work is defined as work which is within the scope of a cut and therefore may never be done in sequential execution.)
- 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 by the positions of workers in 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 have low overhead.
- An idle worker is responsible for selecting the best busy worker for sharing and positions itself at the right position in the tree before requesting sharing from the busy worker. This allows a busy worker to concentrate on its task[2] and to respond only to requests that have to be handled by it.

The current implementation of Muse supports the full Prolog language with its standard semantics. It also supports asynchronous (parallel) side effects and internal database predicates. In this implementation we have simple mechanisms for supporting cut and all standard Prolog side effect predicates (e.g. read, write, assert, retract, etc).

A simple way to guarantee the correct semantics of sequential side effects is to allow execution of such side effects only in the leftmost branch of the whole search tree. The current Muse implementation does not support suspension of branches. Hence a worker that executes a sequential side effect predicate in a branch that is not

[2]A task is a continuous piece of work executed by a worker.

leftmost, will wait until that branch becomes leftmost in the entire tree. Similarly, for supporting the *findall* predicate, a worker that generates a findall solution will wait until its branch is leftmost in a proper subtree. (A novel method has been implemented in the current (1992) version of Muse to remove the need to wait until becoming leftmost for assert and findall. This method is described in Section 7.8.5.)

The effect of *cut N* in a branch is to prune all branches to the right of the branch in the subtree rooted at the node N. In the current implementation of cut, when a worker executing a cut is not leftmost in the relevant subtree, it will prune as much as it can and leave the pruning of the remaining branches to a worker to its left and then proceed with executing operations following the cut. When a worker detects that all left branches have failed, it will try to prune as much as it can until all branches within the scope of the cut have been pruned.

5.3 Muse on the Butterfly

The Butterfly GP1000 is a multiprocessor machine capable of supporting up to 128 processors. The GP1000 is made up of two subsystems, the processor nodes and the Butterfly switch, which connects all nodes. A processor node consists of an MC68020 microprocessor, 4 MBytes of memory and a Processor Node Controller (PNC) that manages all references. A non-local memory access across the switch takes about 5 times longer than local memory access (when there is no contention). The Butterfly switch is a multi-stage omega interconnection network. The switch on the GP1000 has a hardware supported block copy operation, which is useful when implementing the Muse incremental copying strategy. The peak bandwidth of the switch is 4 MBytes per second per switch path.

The Butterfly TC2000 is a similar but newer machine. The main difference is that the processors used in the TC2000 are the Motorola 88100s. They are an order of magnitude faster than the MC68020 and have two 16 KBytes data and instruction caches. Thus in the TC2000 there is actually a three level memory hierarchy: cache memory, local memory and remote memory. Unfortunately no support is provided for cache coherence of shared data. Hence by default shared data are not cached on the TC2000. The peak bandwidth of the Butterfly switch on the TC2000 is 9.5 times faster than the Butterfly GP1000 (around 38 MBytes per second per path).

The main optimization of Muse for the Butterfly machines was the creation of separate copies of the WAM code for each processor. The copies of the code were placed in non-shared memory, thus making them cachable on the TC2000. We have also tried to optimize the scheduler by reducing non-local busy waits and by optimal placement of memory. All the shared memory used is spread across all the nodes to avoid switch contention. We also identified and removed some hot spots. For example, data associated with a shared choicepoint can be simultaneously accessed by many workers (when looking for work), hence on the Butterfly machines access to this data was serialized. Similarly data which are most frequently accessed by a single worker (such as the global registers associated with each worker) are stored in its local memory.

To reduce copying overheads, the local address space of each worker is mapped into a separate part of the global address space of the system. This enables the two workers involved in copying to copy parts of the WAM stacks in parallel. A cache coherence protocol for the WAM stack areas has also been implemented on the TC2000. The basic idea of this protocol is as follows. Every worker keeps a list of all stack areas that it reads from any other worker during its last copying operation. When a worker Q is going to copy data from another worker P in the next copying operation, Q invalidates the listed areas in its cache and P flushes the areas to be copied by Q from its cache.

In the current Muse implementation, there is one cell associated with every choice-point frame for supporting leftmost operations. These cells are accessible by all the workers. To simplify caching of the WAM stacks, these cells are saved in a separate stack associated with every worker. This stack is not cachable.

Another optimization of the Butterfly Muse implementation is in the installation of bindings. In the Muse model, bindings made by P in its stack parts common with Q need to be installed in Q's stacks. Since block copying is more efficient over the Butterfly switch, instead of installing cell by cell from P to Q, P first assembles those bindings, then Q copies them as a block into its local memory and installs the bindings.

5.4 Performance Results

In this section we discuss the performance of the Muse implementation on the Butterfly machines for a large group of benchmarks. The group of benchmarks used in this chapter is the same group of benchmarks that has already been used for the evaluation of OR-parallel Prolog systems on bus-based multiprocessor architectures (Section 3.9 and [14, 44, 24, 83]). We extend this group with some benchmarks that have been used for the Aurora OR-parallel Prolog system on the Butterfly machines [68]. The group of benchmarks used in this chapter can be divided into two sets: the first set (*11-queens1, 11-queens2, semigroup, 8-queens1, 8-queens2, tina, salt-mustard, parse2, parse4, parse5, db4, db5, house, parse1, parse3, farmer*) does not contain major cuts. This set contains benchmarks with different characteristics. It is divided into four groups which reflect the amount of parallelism exhibited by the benchmarks: (*11-queens1, 11-queens2, semigroup*), (*8-queens1, 8-queens2, tina, salt-mustard*), (*parse2, parse4, parse5, db4, db5, house*), and (*parse1, parse3, farmer*). The four groups are referred to in the following sections as *GI, GII, GIII*, and *GIV* respectively. The group *GI* represents programs with the highest amount of parallelism, and the group *GIV* represents programs with the lowest amount of parallelism.

N-queens1 and *N-queens2* are two different *N* queens programs from ECRC. *semigroup* is a theorem proving program for studying the R-classes of a large semigroup from Argonne National Laboratory [64]. *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 database searching parts of the fourth and fifth Chat-80 queries. *house* is the "who owns the zebra" puzzle from ECRC. *farmer* is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC. All benchmarks of the first set look for all solutions of the problem.

The benchmarks in the second set (*mm1, mm2, mm3, mm4, num1, num2, num3, num4*) contain major cuts and have been used for studying different cut schemes [44]. *mm* is a master mind program with four different secret codes. The numbers program (*num*) generates the two largest numbers consisting of given digits and fulfilling specified requirements. The numbers program was run with four different queries. This set is divided into two groups known in the following sections as *mm* and *num*. All benchmarks of the second set look for the first solution of the problem.

In this section we present the Muse results for the first set of benchmarks. The Muse results for the second set of benchmarks are presented in Section 5.5.

5.4.1 Timings and Speedups

5.4.1.1 Butterfly GP1000

Table 5.1 shows the execution times (in seconds) from the execution of the first set of benchmarks for Muse on the GP1000 machine. The execution times given in this chapter are the mean values obtained from eight runs. On the Butterfly machines, mean values are more reliable than best values due to variations of timing results from one run to another. These variations are due mainly to switch contention and are greatest in the smaller benchmarks.

In Table 5.1, times are shown for 1, 10, 30, 50 and 64 workers with speedups given in parentheses. These speedups are relative to the execution times of Muse on one worker. The last column shows execution times of SICStus0.6 on one GP1000 node with the ratio of execution times on Muse for the 1 worker case to the SICStus0.6 times. For benchmarks with small execution times the timings shown refer to repeated runs, the repetition factor being shown in the first column. $\sum$ in the last row corresponds to the goal: (*11-queens1, 11-queens2, semigroup, 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 benchmarks as one benchmark. In all tables, the last row for each group of a set of benchmarks represents the whole group as one benchmark.

	Muse Workers					
Benchmarks	1	10	30	50	64	SICStus0.6
`semigroup`	7948.49	797.91(9.96)	271.54(29.3)	168.08(47.3)	134.55(59.1)	7363.96(1.08)
`11-queens1`	1706.24	171.23(9.96)	57.99(29.4)	35.93(47.5)	28.54(59.8)	1595.00(1.07)
`11-queens2`	5195.67	521.06(9.97)	175.47(29.6))	108.36(47.9)	86.93(59.8)	4829.65(1.08)
● `GI` Σ	14850.40	1490.07(9.97)	504.69(29.4)	312.25(47.6)	249.81(59.4)	13788.61(1.08)
`8-queens1`	13.25	1.49(8.89)	0.74(17.9)	0.71(18.7)	0.82(16.2)	12.36(1.07)
`8-queens2`	33.00	3.67(8.99)	1.53(21.6)	1.27(26.0)	1.40(23.6)	31.66(1.04)
`tina`	27.28	3.38(8.07)	1.83(14.9)	1.70(16.0)	1.80(15.2)	23.90(1.14)
`salt-mustard`	4.29	0.54(7.94)	0.38(11.3)	0.43(9.98)	0.51(8.41)	3.44(1.25)
○ `GII` Σ	77.82	9.07(8.58)	4.46(17.4)	4.10(19.0)	4.47(17.4)	71.36(1.09)
`parse2*20`	10.27	3.27(3.14)	3.55(2.89)	3.63(2.83)	3.78(2.72)	9.38(1.09)
`parse4*5`	9.42	1.91(4.93)	1.83(5.15)	1.88(5.01)	1.91(4.93)	8.67(1.09)
`parse5`	6.65	1.05(6.33)	0.92(7.23)	1.00(6.65)	1.02(6.52)	6.10(1.09)
`db4*10`	4.24	0.98(4.33)	1.07(3.96)	1.13(3.75)	1.14(3.72)	3.80(1.12)
`db5*10`	5.16	1.20(4.30)	1.25(4.13)	1.32(3.91)	1.40(3.69)	4.64(1.11)
`house*20`	6.94	2.92(2.38)	3.19(2.18)	3.37(2.06)	3.43(2.02)	8.86(1.01)
◇ `GIII` Σ	42.68	11.30(3.78)	11.80(3.62)	12.31(3.47)	12.61(3.38)	39.53(1.08)
`parse1*20`	2.72	1.63(1.67)	1.73(1.57)	1.80(1.51)	2.03(1.34)	2.48(1.10)
`parse3*20`	2.33	1.60(1.46)	1.64(1.42)	1.90(1.23)	1.78(1.31)	2.13(1.09)
`farmer*100`	6.02	4.82(1.25)	4.86(1.24)	5.30(1.14)	5.26(1.14)	5.58(1.08)
∗ `GIV` Σ	11.07	8.04(1.38)	8.22(1.35)	8.82(1.26)	9.03(1.23)	10.19(1.09)
Σ	14981.97	1518.51(9.87)	529.21(28.3)	337.68(44.4)	276.01(54.3)	13909.69(1.08)

Table 5.1: Muse execution times (in seconds) for the first set of benchmarks on the GP1000.

As shown in the last column of Table 5.1, on one worker, Muse is about 8% slower than SICStus0.6, the sequential Prolog system from which Muse is derived. This overhead is mainly due to the maintaining of the private load and the checking of some global flags by each Muse worker. This overhead is higher for programs that access the Prolog tables[3] heavily. The Prolog tables are partitioned into sections and each section resides in the local memory of one processor. Accessing remote memory is much more expensive than accessing local memory. The ratio of local to remote memory access time on the GP1000 is 1 to 5. The *salt-mustard* benchmark has 25% overhead per worker, because it makes heavy use of meta calls, which require accessing the predicate table. The corresponding overhead in the other OR-parallel Prolog systems is much higher (see Section 5.5).

The performance results that Table 5.1 illustrates are encouraging: on 64 processors the average speedup factor is 59.4 for the *GI* programs, 17.4 for the group *GII*, 3.38 for the group *GIII*, and around 1.23 for the group *GIV*. The speedup factor for the entire first set of benchmarks on 64 processors is 54.3. The average real speedups on 64 processors, in comparison to SICStus on one GP1000 processor, are 55.2 for the *GI* programs, and 50.4 for the entire first set of benchmarks. For all programs in the group *GI*, increasing the number of workers results in shorter execution times. For programs in the groups *GII, GIII* and *GIV*, increasing the number of workers beyond a certain limit results in no further improvement of execution times. Actually, increasing the number of workers results in slightly longer execution times for the

[3]By Prolog tables we mean the atom table, the predicate table, and also the allocated records associated with occupied items in the tables.

latter three groups. This degradation is due to the increased scheduling overhead (see Section 5.4.2).

Figure 5.1 shows the speedup curves for the four groups: *GI, GII, GIII* and *GIV*. These curves correspond to speedups obtained from Table 5.1. Variations around the mean value are shown by a vertical line with two short horizontal lines at each end. Variations of less than 0.4 are not shown (in all figures).

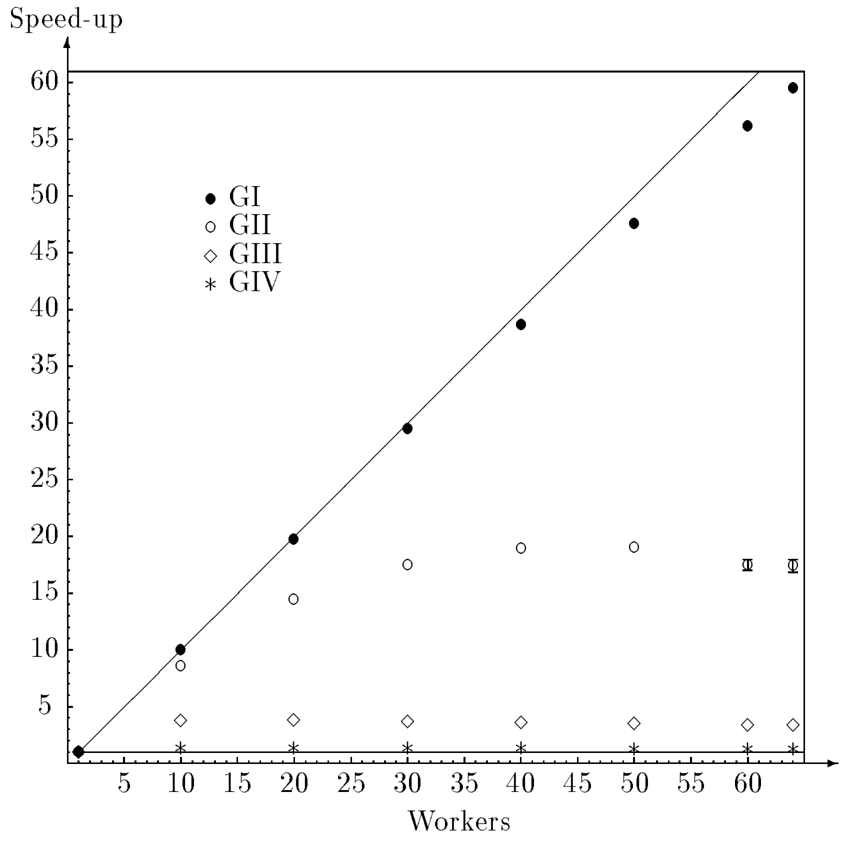


Figure 5.1: Speedups of Muse for the first set of benchmarks on GP1000.

We observe from Figure 5.1 that when the amount of parallelism is not enough for all workers in the system to find work, the speedup curves level off and reach an almost constant value. This characteristic is very important for any parallel system (such as Muse) that dynamically schedules work at runtime.

5.4.1.2 Butterfly TC2000

On the TC2000 we have two versions of the Muse system: one caches the WAM stacks and the other does not. The version that caches the WAM stacks has a faster Prolog engine. The timing results on the two versions will be presented in this section to illustrate the effect of engine speed on the obtained speedups.

Tables 5.2 and 5.3 present the execution times of Muse for the first set of benchmarks on the TC2000 machine when the WAM stacks were and were not cached respectively. Times are shown for 1, 10, 20, 30 and 32 workers with speedups given in parentheses. These speedups are relative to the execution times of Muse on one worker as in Table 5.1. Figures 5.2 and 5.3 show speedup curves corresponding to Tables 5.2 and 5.3.

	Muse Workers					
Benchmarks	1	10	20	30	32	SICStus0.6
`semigroup`	1502.60	151.18(9.94)	76.44(19.7)	51.62(29.1)	48.47(31.0)	990.89(1.52)
`11-queens1`	313.89	31.66(9.91)	16.00(19.6)	10.80(29.1)	10.11(31.0)	190.87(1.64)
`11-queens2`	965.24	96.68(9.98)	48.67(19.8)	32.71(29.5)	30.75(31.4)	574.27(1.68)
• `GI` Σ	2781.73	279.50(9.95)	141.10(19.7)	95.11(29.2)	89.32(31.1)	1756.03(1.58)
`8-queens1`	2.47	0.29(8.52)	0.19(13.0)	0.15(16.5)	0.15(16.5)	1.48(1.67)
`8-queens2`	6.36	0.69(9.22)	0.39(16.3)	0.30(21.2)	0.30(21.2)	3.77(1.69)
`tina`	5.57	0.71(7.85)	0.45(12.4)	0.39(14.3)	0.39(14.3)	3.07(1.81)
`salt-mustard`	0.84	0.11(7.64)	0.09(9.33)	0.09(9.33)	0.09(9.33)	0.44(1.91)
∘ `GII` Σ	15.24	1.79(8.51)	1.10(13.9)	0.94(16.2)	0.93(16.4)	8.76(1.74)
`parse2*20`	2.36	0.84(2.81)	0.93(2.54)	0.93(2.54)	0.94(2.51)	1.43(1.65)
`parse4*5`	2.16	0.49(4.41)	0.47(4.60)	0.52(4.15)	0.53(4.08)	1.31(1.65)
`parse5`	1.53	0.29(5.28)	0.26(5.88)	0.26(5.88)	0.27(5.67)	0.93(1.65)
`db4*10`	0.90	0.21(4.29)	0.22(4.09)	0.23(3.91)	0.23(3.91)	0.53(1.70)
`db5*10`	1.09	0.25(4.36)	0.26(4.19)	0.26(4.19)	0.26(4.19)	0.64(1.70)
`house*20`	1.33	0.62(2.15)	0.66(2.02)	0.66(2.02)	0.68(1.96)	0.94(1.41)
⋄ `GIII` Σ	9.37	2.70(3.47)	2.80(3.35)	2.85(3.29)	2.89(3.24)	5.78(1.62)
`parse1*20`	0.63	0.44(1.43)	0.45(1.40)	0.47(1.34)	0.46(1.37)	0.38(1.66)
`parse3*20`	0.54	0.43(1.26)	0.44(1.23)	0.45(1.20)	0.46(1.17)	0.33(1.64)
`farmer*100`	1.13	1.04(1.09)	1.05(1.08)	1.07(1.06)	1.11(1.02)	0.66(1.71)
∗ `GIV` Σ	2.30	1.90(1.21)	1.93(1.19)	1.98(1.16)	2.02(1.14)	1.37(1.68)
Σ	2808.64	285.89(9.82)	146.93(19.1)	100.89(27.8)	95.18(29.5)	1771.94(1.59)

Table 5.2: Muse execution times (in seconds) for the first set of benchmarks on TC2000 without caching of the WAM stacks.

From results shown in Tables 5.2 and 5.3, we observe that the absolute times on the Muse version that cached the WAM stacks are shorter than the corresponding ones on the version that did not. We also observe that speedups in Table 5.2 are somewhat better than the corresponding ones in Table 5.3. The reason speedups are poorer in Table 5.3 is that the Muse version that caches the WAM stacks has a faster Prolog engine, and in general when the ratio of the speed of the engine to that of the scheduler increases, the relative cost of scheduling overhead increases, which in turn decreases the speedups.

From the last column in Table 5.2, Muse on one worker is about 59% slower than SICStus0.6. This is because in the SICStus system all code, data, and tables are cachable while in this version of Muse only the code area is cachable. (The TC2000 processor is provided with data and instruction caches.) The overhead per Muse worker has been reduced from 59% to 22% by caching the data area (i.e. the WAM stacks) as shown in the last column in Table 5.3. The Prolog tables on Muse system are not cachable. The reason the overhead per Muse worker is lower on the GP1000 (8%) than on TC2000 (22%) is that the former does not have any cache memory. So, on GP1000 there is no difference in access time between cachable and noncachable data while on TC2000 there is a difference. In order to verify that it was the Prolog tables that caused the overhead, we compared the execution times of Muse with one worker (making the Prolog tables cachable) to those of SICStus0.6 on the TC2000

	Muse Workers					
Benchmarks	1	10	20	30	32	SICStus0.6
`semigroup`	1178.63	118.56(9.94)	60.06(19.6)	40.72(28.9)	38.35(30.7)	990.89(1.19)
`11-queens1`	225.23	22.78(9.89)	11.58(19.4)	7.88(28.6)	7.41(30.4)	190.87(1.18)
`11-queens2`	729.77	73.12(9.98)	36.92(19.8)	24.93(29.3)	23.43(31.1)	574.27(1.27)
• `GI` Σ	2133.63	214.46(9.95)	108.53(19.7)	73.52(29.0)	69.18(30.8)	1756.03(1.22)
`8-queens1`	1.79	0.21(8.52)	0.14(12.8)	0.13(13.8)	0.13(13.8)	1.48(1.21)
`8-queens2`	4.80	0.53(9.06)	0.31(15.5)	0.26(18.5)	0.25(19.2)	3.77(1.27)
`tina`	4.28	0.58(7.38)	0.39(11.0)	0.34(12.6)	0.34(12.6)	3.07(1.39)
`salt-mustard`	0.71	0.10(7.10)	0.08(8.88)	0.09(7.89)	0.10(7.10)	0.44(1.61)
∘ `GII` Σ	11.58	1.42(8.15)	0.91(12.7)	0.82(14.1)	0.81(14.3)	8.76(1.32)
`parse2*20`	1.82	0.80(2.27)	0.84(2.17)	0.85(2.14)	0.87(2.09)	1.43(1.27)
`parse4*5`	1.67	0.42(3.98)	0.43(3.88)	0.45(3.71)	0.47(3.55)	1.31(1.27)
`parse5`	1.18	0.24(4.92)	0.22(5.36)	0.24(4.92)	0.24(4.92)	0.93(1.27)
`db4*10`	0.68	0.20(3.40)	0.21(3.24)	0.21(3.24)	0.21(3.24)	0.53(1.28)
`db5*10`	0.83	0.23(3.61)	0.24(3.46)	0.25(3.32)	0.25(3.32)	0.64(1.30)
`house*20`	0.98	0.58(1.69)	0.61(1.61)	0.63(1.56)	0.63(1.56)	0.94(1.04)
⋄ `GIII` Σ	7.16	2.46(2.91)	2.54(2.82)	2.62(2.73)	2.65(2.70)	5.78(1.24)
`parse1*20`	0.48	0.40(1.20)	0.40(1.20)	0.41(1.17)	0.42(1.14)	0.38(1.26)
`parse3*20`	0.41	0.39(1.05)	0.41(1.00)	0.42(0.98)	0.42(0.98)	0.33(1.24)
`farmer*100`	0.83	1.01(0.82)	1.03(0.81)	1.07(0.78)	1.05(0.79)	0.66(1.26)
∗ `GIV` Σ	1.72	1.80(0.96)	1.83(0.94)	1.89(0.91)	1.88(0.91)	1.37(1.26)
Σ	2154.09	220.14(9.79)	113.83(18.9)	78.85(27.3)	74.53(28.9)	1771.94(1.22)

Table 5.3: Muse execution times (in seconds) for the first set of benchmarks on TC2000 with caching of the WAM stacks.

(Table 5.4). We see that the overhead for this version of Muse is between 4% to 7%, which is quite close to the overhead observed on the Sequent Symmetry.

Benchmarks	Muse	SICStus0.6
GI	1829.03	1756.03(1.04)
GII	9.36	8.76(1.07)
GIII	6.19	5.78(1.07)
GIV	1.47	1.37(1.07)
Σ	1846.05	1771.94(1.04)

Table 5.4: Comparison of the execution times (in seconds) between 1 Muse worker (with Prolog tables cached) and SICStus0.6.

The average real speedups on 32 processors, in comparison to SICStus on one TC2000 processor, are 25.4 for the *GI* programs, and 23.8 for the whole first set of benchmarks (calculated from Table 5.3). It should be noted that the Muse system has always been shown to have the best real speedups in comparison to the other existing OR-Parallel Prolog systems (see Section 5.5 for further discussions).

The role of the ratio of the speed of the engine to that of the scheduler in the Muse system on the Butterfly machines can also be seen in the speedup curves those of Figures 5.1, 5.2 and 5.3. The best speedup curves are those of Figure 5.1 (for the same number of workers), and the worst curves are in Figure 5.3. From the first column in Tables 5.1 and 5.3, the engine speed on the TC2000 is around 7 times the engine speed on the GP1000 due to the difference in the speeds of the processors.

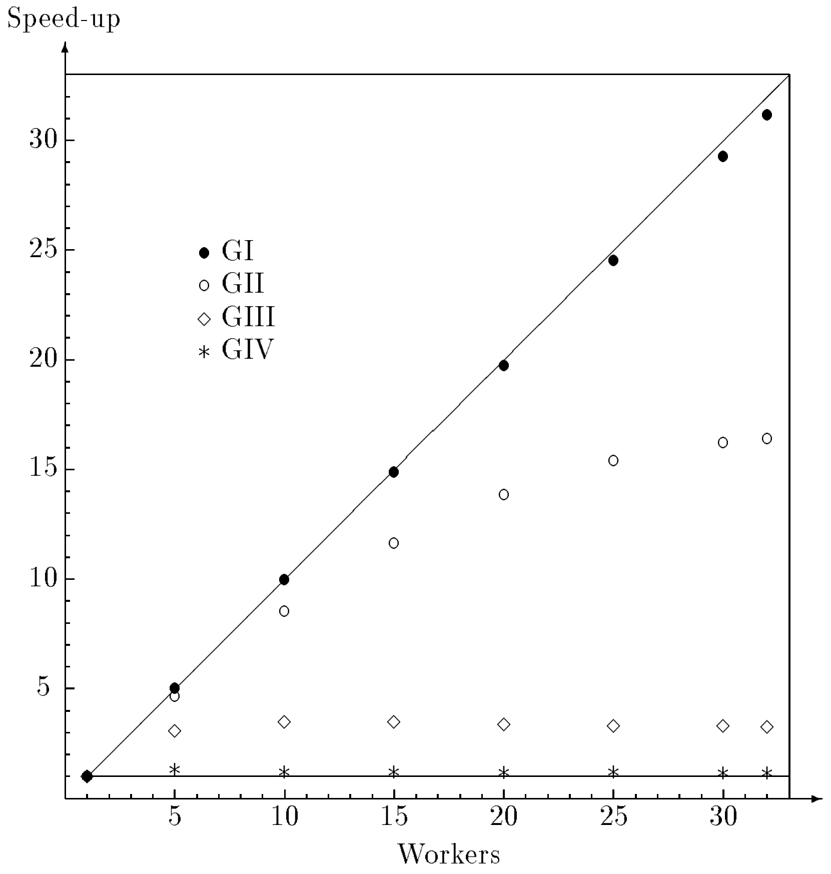


Figure 5.2: Speedups of Muse for the first set of benchmarks on TC2000 without caching of the WAM stacks.

The scheduler speed on the TC2000 is not increased by the same factor even though the switch speed is increased by a similar factor. There could be two reasons for this:

1. The scheduler data on the TC2000 are not cached (for obvious reasons), whereas most of the engine data are cached. This results in the engine to scheduler speed ratio being higher on the TC2000, as the GP1000 does not have a cache.

2. Unlike the GP1000 switch, the switch on the TC2000 does not have a hardware supported block copy operation. This has the effect of reducing the real bandwidth of the switch in incremental copying.

5.4.2 Worker Activities and Scheduling Overhead

In this section, we present and discuss the time spent in the basic activities of a Muse worker on the TC2000 machine for the version of Muse that caches the WAM stacks. A worker's time is distributed over the following activities:

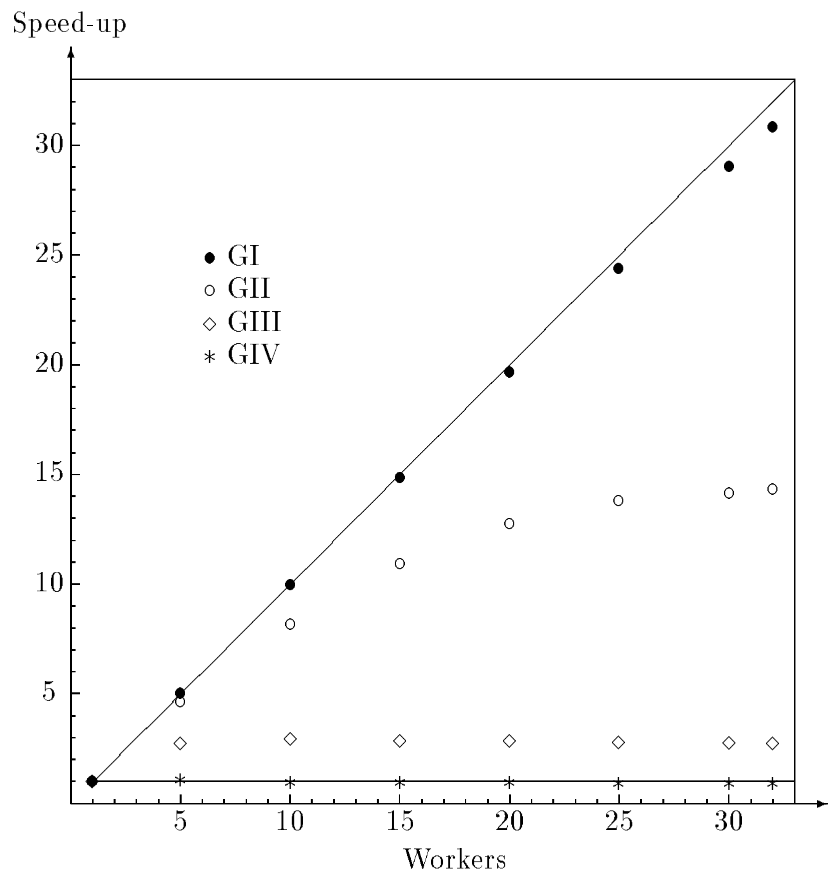


Figure 5.3: Speedups of Muse for the first set of benchmarks on TC2000 with caching of the WAM stacks.

1. *Prolog:* time spent executing Prolog, checking for the arrival of requests, and keeping the value of the private load up to date.
2. *Idle:* time spent waiting for work to be generated when there is temporarily no work available in the system.
3. *Grabbing Work:* time spent grabbing available work from shared nodes.
4. *Sharing:* time spent making private nodes shared with other workers, copying parts of the WAM stacks, binding installation, or synchronization with other workers while performing the sharing operation.
5. *Looking for work:* time spent looking for a worker with private load.
6. *Others:* time spent in other activities such as acquiring spin locks, sending requests to other workers – either requesting sharing or performing commit/cut to a shared node, etc.

Table 5.5 shows the time spent in each activity as a percentage of the total time, for one selected benchmark from each group of the first set of benchmarks. The last row of each table gives the scheduling overhead, which is the sum of all the activities excluding *Prolog* and *Idle*. The *11-queens2* is selected from *GI*, *8-queens2* from *GII*, *parse5* from *GIII*, and *farmer* from *GIV*. Results shown in Table 5.5 have been obtained from an instrumented system of Muse on the TC2000. The times obtained from an instrumented system are longer than those obtained from an uninstrumented system by around 13%. We believe that the percentage of time spent in each activity obtained from the instrumented system reflects what is happening in the uninstrumented system.

In all benchmarks, the *Prolog* percentage of time decreases and the *Idle* percentage of time increases as the number of workers is increased. This is because each benchmark allows a limited amount of parallelism and increasing the number of workers decreases the amount of work assigned to each worker. For *11-queens2*, a program with high OR-parallelism, the *Prolog* percentage decreases by 1.1% from 3 workers to 32 workers and the *Idle* percentage increases by 0.4% from 3 workers to 32 workers. The total percentage of scheduling overhead (activities 3 – 6) on 32 workers is only 0.7%.

For *8-queens2*, the scheduling overhead increases from 0.9% on 3 workers to 17% on 32 workers. The rate by which the *Prolog* percentage decreases and the *Idle* percentage increases (w.r.t. to the number of workers) is much higher in *8-queens2* than in *11-queens2*. This is due to the fact that *8-queens2* has much less parallelism.

For *parse5* on 32 workers, the *Prolog* percentage is only 20.8%, while the *Idle* percentage reaches 52.4%. This benchmark has a reasonable amount of parallelism up to 10 – 15 workers, but not beyond. The scheduling overhead is higher than in the previous two benchmarks (5.3% on 3 workers and around 26% on 10 or more workers). *parse5* contains finer grain parallelism than in *8-queens2* and *11-queens2*.

For *farmer*, the amount of parallelism is only enough for 2 – 3 workers. The scheduling overhead reaches its maximum value (33.9%) for 3 workers and then decreases with an increasing number of workers. The reason for the decrease in the percentage of scheduling overhead is that adding more than 3 workers just increases the total *Idle* time in the system, which then dominates the total execution time.

In all benchmarks, the sharing overhead is the dominating part of the scheduling overhead. Sharing overhead ranges from 0.0% on the *11-queens2* to 22.3% on the *parse5*.

A possible explanation for the increase in overhead with the number of workers is shown in Table 5.6, which shows the effect of increasing the number of workers on the average task sizes (expressed as the number of Prolog procedure calls per task). A task is a continuous piece of work executed by a worker.

In Table 5.6 the task size decreases as the the number of workers is increased until it reaches a constant value, 20 for *parse5* and 11 for *farmer*. The reason the task size is almost constant after a certain number of workers is that the Muse system supports a form of delayed release of work that tries to avoid a continuous decrease in task size

	Muse Workers								
Activity	1	3	5	10	15	20	25	30	32
11-queens2									
Prolog	100	100	99.9	99.8	99.7	99.5	99.4	98.9	98.9
Idle	0	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.4
Grabbing Work	0	0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.3
Sharing Work	0	0.0	0.0	0.0	0.1	0.2	0.2	0.3	0.3
Looking for Work	0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1
Others	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Sched. Overhead	0	0.0	0.0	0.1	0.2	0.4	0.4	0.7	0.7
8-queens2									
Prolog	100	98.9	97.6	93.0	87.5	81.3	74.0	67.0	64.3
Idle	0	0.2	0.5	2.0	4.2	8.1	11.5	16.6	18.7
Grabbing Work	0	0.3	0.7	1.8	3.0	3.6	5.0	5.5	5.2
Sharing Work	0	0.5	0.9	2.3	3.9	5.1	6.3	7.7	8.0
Looking for Work	0	0.1	0.2	0.7	1.2	1.6	2.7	2.6	3.1
Others	0	0.0	0.0	0.2	0.3	0.3	0.4	0.7	0.7
Sched. Overhead	0	0.9	1.8	5.0	8.4	10.6	14.4	16.5	17.0
parse5									
Prolog	100	93.9	84.6	60.6	46.5	34.0	27.5	22.4	20.8
Idle	0	0.8	2.1	13.3	21.3	30.5	41.0	48.4	52.4
Grabbing Work	0	1.8	2.9	3.4	3.9	3.9	3.5	3.3	3.0
Sharing Work	0	3.1	8.3	16.8	21.0	22.3	19.7	18.1	16.8
Looking for Work	0	0.4	2.0	5.7	6.9	8.7	7.8	7.2	6.4
Others	0	0.0	0.1	0.2	0.3	0.5	0.6	0.6	0.6
Sched. Overhead	0	5.3	13.3	26.1	32.1	35.4	31.6	29.2	26.8
farmer									
Prolog	100	44.8	25.7	12.0	7.8	5.8	4.6	3.8	3.5
Idle	0	21.3	42.6	70.3	80.6	85.8	88.7	90.6	91.2
Grabbing Work	0	5.0	4.1	2.1	1.4	1.0	0.8	0.7	0.7
Sharing Work	0	21.5	20.6	11.4	7.5	5.4	4.3	3.7	3.5
Looking for Work	0	6.0	6.1	3.6	2.4	1.7	1.4	1.1	1.1
Others	0	1.3	1.0	0.6	0.4	0.2	0.2	0.2	0.2
Sched. Overhead	0	33.8	31.8	17.7	11.7	8.3	6.7	5.7	5.5

Table 5.5: Percentage of time spent in basic activities of a Muse worker on TC2000.

	Muse Workers								
Benchmark	1	3	5	10	15	20	25	30	32
11-queens2	31410506	52004	20530	8412	3839	2712	2267	1463	1382
8-queens2	207778	980	618	174	124	91	66	61	58
parse5	39119	132	58	43	28	23	20	20	20
farmer	33486	16	11	11	11	11	11	11	11

Table 5.6: Task sizes for some programs of the first set of benchmarks on TC2000.

as the number of workers is increased. Supporting such a mechanism is crucial in order to avoid a continuous high increase in scheduling overhead with an increasing number of workers. The idea of the delayed release mechanism supported in the Muse system is as follows. When a worker reaches a situation in which it has only one private parallel node, it will make its private load visible to the other workers only when that node is still alive after a certain number, k, of Prolog procedure calls. The value of k is a constant selected to be larger than the number of Prolog procedure calls equivalent to the scheduling overhead per task (see below).

Table 5.7 shows the scheduling overhead per task in terms of Prolog procedure calls for the four selected benchmarks. The scheduling overhead (all activities except *Prolog* and *Idle*) is equivalent to around 8 – 26 Prolog procedure calls per task. It is almost constant (around 10) for *11-queens2*, a program with coarse-grained parallelism. It is higher and increases with the number of workers for the other three, programs with finer grain parallelism. For *parse5*, the scheduling overhead per task is somewhat higher in comparison with the other benchmarks. *parse5* generates a search tree with two long branches and each branch has many short branches.

	Muse Workers								
Benchmark	1	3	5	10	15	20	25	30	32
11-queens2	0	10.8	10.2	9.0	9.2	9.8	10.2	10.7	11.4
8-queens2	0	9.0	11.9	9.3	11.8	11.9	12.9	14.9	15.4
parse5	0	7.5	9.1	18.5	19.3	24.0	22.9	26.0	25.8
farmer	0	12.1	13.6	16.2	16.4	16.0	16.2	16.5	17.0

Table 5.7: Scheduling overheads per task in terms of Prolog procedure calls for some programs of the first set of benchmarks on TC2000.

To conclude, the scheduling overhead per task of Muse for the selected four benchmarks on TC2000 when caching the WAM stacks is around 8 – 26 Prolog procedure calls per task. This value is affected by the relative speeds of the Prolog engine and the scheduler. For instance, the corresponding value for the Muse system on Sequent Symmetry is around 5 – 7 Prolog procedure calls per task. The time of a Prolog procedure call is between 23 and 30 microseconds on TC2000, and between 83 and 100 microseconds on Sequent Symmetry. So, in order to avoid losing any gain obtained by exploiting parallelism, the Muse system on TC2000 should avoid letting the task sizes fall below 26 Prolog procedure calls. That is, the value of k should be larger than 26. Actually, all results presented in this chapter correspond to k equal to 10.

We have also experimented with increasing k to 15, 25 and 30, the result being that some improvements were obtained for programs with very fine granularity (e.g. *farmer*).

5.5 Comparison with Aurora

In this section we compare the timing results of Muse with the corresponding results for Aurora with the Manchester scheduler [65]. Both Aurora and Muse are based on the same sequential Prolog, SICStus version 0.6, and have been implemented on the same Butterfly machines. Neither Muse nor Aurora with the Manchester scheduler [24] do any special treatment of speculative work. (A new Aurora scheduler which handles speculative work better is under development at the University of Bristol.) The main difference between Muse and Aurora in the implementation of cut, findall, and sequential side effect constructs is that Aurora supports suspension of branches whereas Muse does not. For instance, an Aurora worker executing a cut in a non–leftmost branch of the cut node will suspend the branch and try to find work outside the cut subtree. In Muse, the pruning operation suspends while the worker proceeds with the next operation following the cut as described in Section 5.2.

Another difference between Aurora and Muse is that Aurora is based on a different model for OR-parallel execution of Prolog, namely the SRI model [92]. The idea of the SRI model is to extend the conventional WAM with a binding array for each 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. A binding is conditional if a variable can get several bindings. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around in 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.

Many optimizations have been made of the implementation of Aurora on the Butterfly machines. These optimizations are the same as the Muse optimizations, with the exception of caching the WAM stacks. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. Therefore, it is straightforward in Muse to make the WAM stack areas cachable whereas in Aurora it requires a complex cache coherence protocol to achieve this effect.

The Manchester scheduler for Aurora also tries to avoid a continuous decrease in task sizes with an increasing number of workers. The idea used by the Manchester scheduler is that each busy worker checks for the arrival of requests from other workers on every N Prolog procedure calls. The best value of N on TC2000 is 20. All Aurora timing results presented in this chapter correspond to $N = 20$.

Table 5.8 shows the execution times of Aurora with the Manchester scheduler for the first set of benchmarks on TC2000. The last column of Table 5.8 shows the execution times of SICStus0.6 on one TC2000 node with the ratio of execution times on Muse for the 1 worker case to the SICStus0.6 times.

	Aurora Workers					
Benchmarks	1	10	20	30	32	SICStus0.6
`semigroup`	1699.78	169.62(10.0)	87.03(19.5)	58.90(28.9)	54.68(31.1)	990.89(1.72)
`11-queens1`	369.14	36.92(10.00)	18.54(19.9)	12.47(29.6)	11.70(31.6)	190.87(1.93)
`11-queens2`	1044.49	105.38(9.91)	52.83(19.8)	35.37(29.5)	33.19(31.5)	574.27(1.82)
• `GI` Σ	3113.41	311.91(9.98)	158.38(19.7)	106.68(29.2)	99.55(31.3)	1756.03(1.77)
`8-queens1`	2.85	0.33(8.64)	0.21(13.6)	0.22(13.0)	0.23(12.4)	1.48(1.93)
`8-queens2`	6.97	0.76(9.17)	0.43(16.2)	0.35(19.9)	0.36(19.4)	3.77(1.85)
`tina`	6.33	0.83(7.63)	0.65(9.74)	0.88(7.19)	1.16(5.46)	3.07(2.06)
`salt-mustard`	1.47	0.19(7.74)	0.12(12.2)	0.12(12.2)	0.14(10.5)	0.44(3.34)
∘ `GII` Σ	17.62	2.10(8.39)	1.40(12.6)	1.55(11.4)	1.81(9.73)	8.76(2.01)
`parse2*20`	2.46	1.30(1.89)	1.50(1.64)	1.74(1.41)	1.87(1.32)	1.43(1.72)
`parse4*5`	2.25	0.74(3.04)	0.81(2.78)	1.01(2.23)	1.08(2.08)	1.31(1.72)
`parse5`	1.59	0.46(3.46)	0.47(3.38)	0.54(2.94)	0.59(2.69)	0.93(1.71)
`db4*10`	0.91	0.28(3.25)	0.35(2.60)	0.55(1.65)	0.58(1.57)	0.53(1.72)
`db5*10`	1.11	0.30(3.70)	0.36(3.08)	0.54(2.06)	0.60(1.85)	0.64(1.73)
`house*20`	1.55	0.79(1.96)	1.12(1.38)	1.99(0.78)	2.63(0.59)	0.94(1.65)
⋄ `GIII` Σ	9.87	3.86(2.56)	4.60(2.15)	6.24(1.58)	7.17(1.38)	5.78(1.71)
`parse1*20`	0.66	0.62(1.06)	0.73(0.90)	0.85(0.78)	0.84(0.79)	0.38(1.74)
`parse3*20`	0.57	0.62(0.92)	0.71(0.80)	0.70(0.81)	0.74(0.77)	0.33(1.73)
`farmer*100`	1.14	1.80(0.63)	2.14(0.53)	2.33(0.49)	2.37(0.48)	0.66(1.73)
∗ `GIV` Σ	2.37	3.03(0.78)	3.56(0.67)	3.80(0.62)	3.91(0.61)	1.37(1.73)
Σ	3143.27	320.92(9.79)	167.95(18.7)	118.47(26.5)	112.60(27.9)	1771.94(1.77)

Table 5.8: Aurora execution times (in seconds) for the first set of benchmarks on TC2000.

The average real speedups on 32 processors, in comparison to SICStus on one TC2000 processor, are 17.6 for the *GI* programs, and 15.7 for the whole first set of benchmarks (calculated from Table 5.8). The corresponding figures for Muse, calculated from Table 5.3 (page 80), are 25.4 and 23.8 respectively. It can also be seen from Tables 5.2 (page 79), 5.3 (page 80) and 5.8 that Muse is faster than Aurora in all benchmarks. The Muse system has somewhat better speedups for hard benchmarks, i.e. benchmarks of groups *GII, GIII* and *GIV*. These hard benchmarks are a good test of the schedulers. The average real speedups on 32 processors of Aurora are 4.84 for *GII*, 0.81 for *GIII*, and 0.35 for *GIV*. The corresponding real speedups for Muse are 10.8, 2.2, and 0.73 respectively. These results illustrate that the Muse scheduler performs better by at least a factor 2.

Table 5.9 shows the ratio of the execution times on Aurora to the execution times on Muse for each group of the first set of benchmarks. Aurora timings are longer than Muse timings by 39% to 171% for 1 to 32 workers.

Tables 5.10 and 5.11 present the execution times (in seconds) from the execution of the second set of benchmarks for Aurora and Muse on the TC2000 machine. Figures 5.5 and 5.6 show the corresponding speedup curves for the two groups of the second set of benchmarks: *mm* and *num*. The reason there is considerable

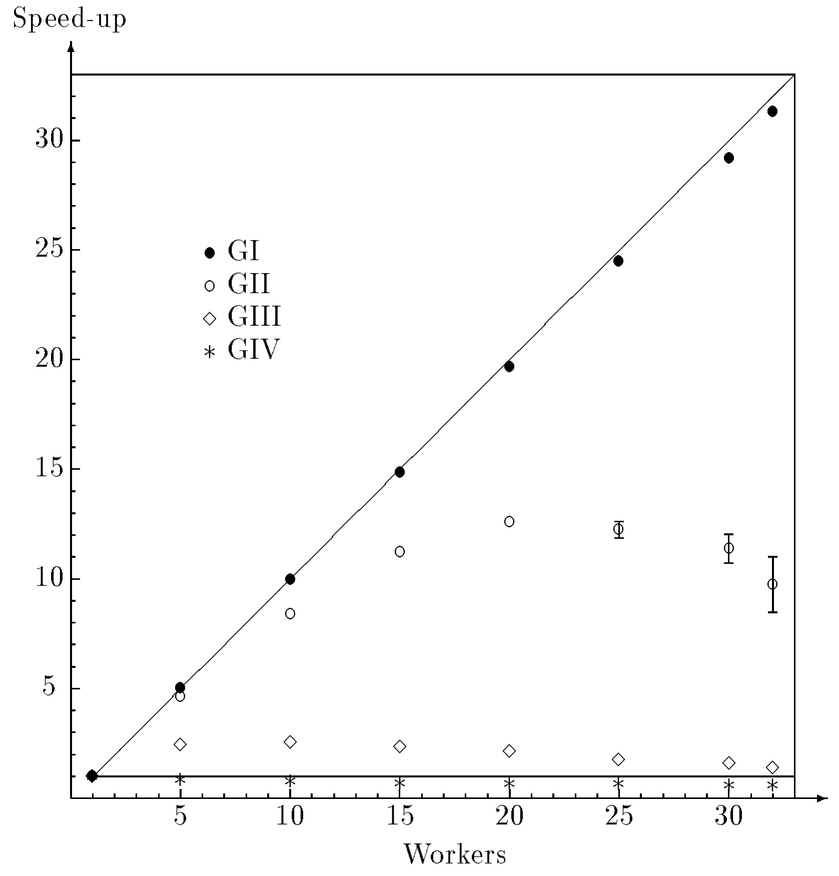


Figure 5.4: Speedups of Aurora for the first set of benchmarks on TC2000.

Benchmarks	Workers				
	1	10	20	30	32
GI	1.46	1.45	1.46	1.45	1.44
GII	1.52	1.48	1.54	1.89	2.23
GIII	1.38	1.57	1.81	2.38	2.71
GIV	1.38	1.68	1.95	2.01	2.08
Σ	1.46	1.46	1.48	1.50	1.51

Table 5.9: The Aurora to Muse ratio of execution times for the first set of benchmarks on TC2000.

variation around the mean values on the cut curves is that the two systems (Muse and Aurora) do not make any special treatment of speculative work.

	Aurora Workers					
Benchmarks	1	10	20	30	32	SICStus0.6
`mm1`	1.69	0.80(2.11)	0.68(2.49)	0.75(2.25)	1.39(1.22)	0.88(1.92)
`mm2`	1.23	0.63(1.95)	0.58(2.12)	0.43(2.86)	0.45(2.73)	0.69(1.78)
`mm3`	3.63	1.28(2.84)	1.03(3.52)	1.08(3.36)	1.16(3.13)	1.99(1.82)
`mm4`	6.39	1.59(4.02)	1.11(5.76)	1.22(5.24)	1.12(5.71)	3.36(1.90)
$\circ$ `mm` Σ	12.94	4.28(3.02)	3.36(3.85)	3.38(3.83)	3.92(3.30)	6.92(1.87)
`num1`	0.52	0.19(2.74)	0.14(3.71)	0.15(3.47)	0.16(3.25)	0.36(1.44)
`num2`	0.84	0.19(4.42)	0.13(6.46)	0.16(5.25)	0.18(4.67)	0.59(1.42)
`num3`	0.90	0.15(6.00)	0.11(8.18)	0.13(6.92)	0.13(6.92)	0.63(1.43)
`num4`	1.16	0.16(7.25)	0.12(9.67)	0.14(8.29)	0.17(6.82)	0.82(1.41)
$\diamond$ `num` Σ	3.42	0.68(5.03)	0.50(6.84)	0.57(6.00)	0.63(5.43)	2.40(1.43)
Σ	16.36	4.97(3.29)	3.86(4.24)	3.97(4.12)	4.56(3.59)	9.32(1.76)

Table 5.10: Aurora execution times (in seconds) for the second set of benchmarks on TC2000.

	Muse Workers					
Benchmarks	1	10	20	30	32	SICStus0.6
`mm1`	1.18	0.29(4.07)	0.25(4.72)	0.21(5.62)	0.22(5.36)	0.88(1.34)
`mm2`	0.92	0.21(4.38)	0.15(6.13)	0.14(6.57)	0.14(6.57)	0.69(1.33)
`mm3`	2.67	0.68(3.93)	0.50(5.34)	0.44(6.07)	0.45(5.93)	1.99(1.34)
`mm4`	4.57	0.72(6.35)	0.45(10.2)	0.40(11.4)	0.39(11.7)	3.36(1.36)
$\bullet$ `mm` Σ	9.34	1.89(4.94)	1.34(6.97)	1.18(7.92)	1.20(7.78)	6.92(1.35)
`num1`	0.45	0.11(4.09)	0.08(5.62)	0.08(5.62)	0.09(5.00)	0.36(1.25)
`num2`	0.74	0.14(5.29)	0.11(6.73)	0.11(6.73)	0.11(6.73)	0.59(1.25)
`num3`	0.79	0.11(7.18)	0.08(9.88)	0.09(8.78)	0.08(9.88)	0.63(1.25)
`num4`	1.02	0.12(8.50)	0.08(12.8)	0.09(11.3)	0.09(11.3)	0.82(1.24)
$\star$ `num` Σ	3.00	0.48(6.25)	0.35(8.57)	0.36(8.33)	0.37(8.11)	2.40(1.25)
Σ	12.34	2.36(5.23)	1.69(7.30)	1.54(8.01)	1.57(7.86)	9.32(1.32)

Table 5.11: Muse execution times (in seconds) for the second set of benchmarks on TC2000.

It can be seen from Tables 5.10 and 5.11 that Muse is faster than Aurora for all benchmarks in the second set. The Muse system also has better speedups for all benchmarks. The last row of Table 5.12 shows the ratio of the execution times on Aurora to the execution times on Muse for the entire second set of benchmarks. Aurora timings are longer than Muse timings by 33% to 190% for 1 to 32 workers.

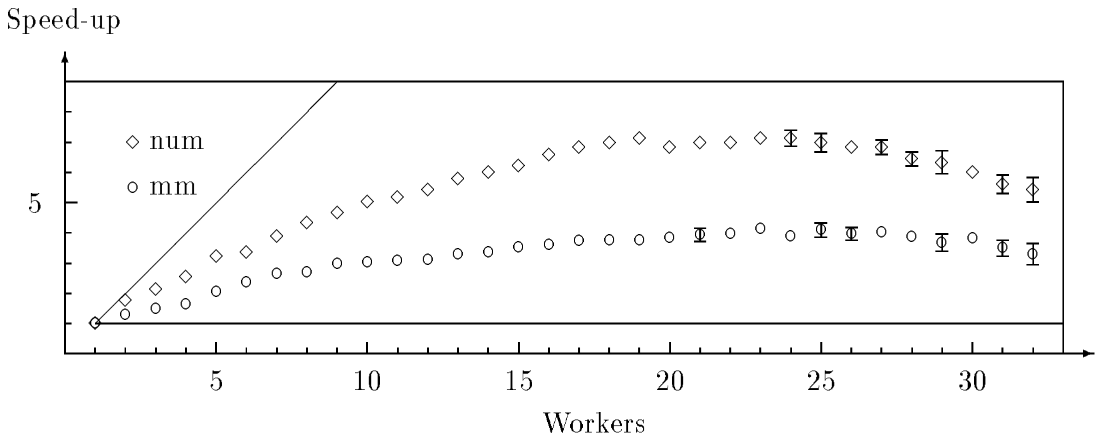


Figure 5.5: Speedups of Aurora for the second set of benchmarks on TC2000.

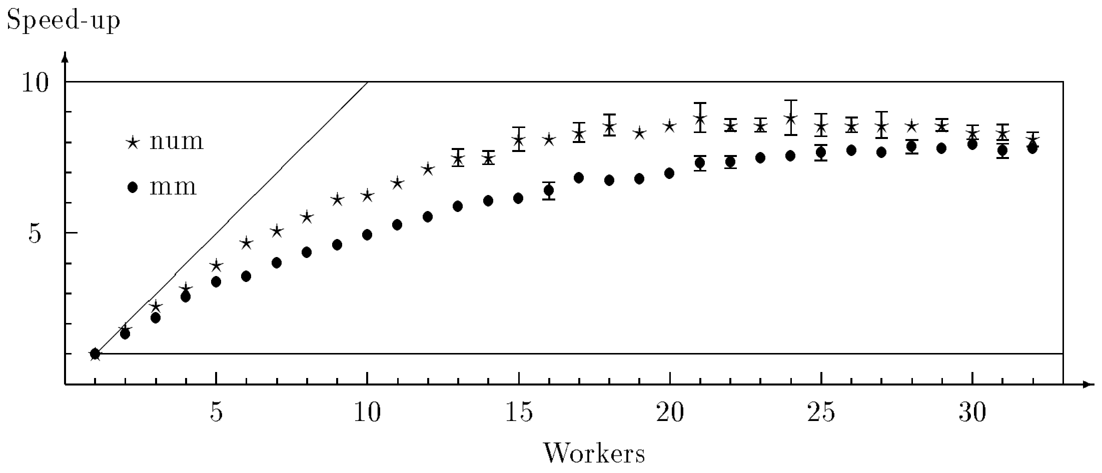


Figure 5.6: Speedups of Muse for the second set of benchmarks on TC2000.

	Workers				
System	1	10	20	30	32
`Aurora`	16.36	4.97	3.86	3.97	4.56
`Muse`	12.34	2.36	1.69	1.54	1.57
`Aurora/Muse`	1.33	2.11	2.28	2.58	2.90

Table 5.12: Aurora and Muse execution times (in seconds) and the ratio between them for the second set of benchmarks on TC2000.

5.6 Conclusions

The Muse implementation and its performance results on the Butterfly machines have been presented and discussed. A large set of benchmarks with different amounts of parallelism have been used in the evaluation of Muse performance on the Butterfly machines. The performance results of Muse have also been compared with the corresponding results for the Aurora OR-parallel Prolog system.

The results obtained for the Muse system are very encouraging: almost linear speedups (around 60 on 64 GP1000 processors, and 31 on 32 TC2000 processors) for Prolog programs with coarse grain parallelism, and a speedup factor of almost 1 for programs with very low parallelism. The average real speedup on 64 GP1000 processors, in a comparison with SICStus on one GP1000 processor, is 55.2 for the programs with coarse grain parallelism. The corresponding average real speedup on 32 TC2000 processors is 25.4. The obtained speedups, for the same number of processors, on the GP1000 are better than those obtained on the TC2000. This is due to the difference in the relative speed of the engine and the scheduler on the two machines. This relative speed is better on the TC2000, due to the fact that most engine data are cachable. This is not a factor on the GP1000 as it does not have cache memory.

The overhead of Muse associated with adapting the sequential SICStus Prolog system to an OR-parallel implementation is low, around 8% on the GP1000 and around 22% on the TC2000. The higher overhead on the TC2000 is due to the fact that each of its processors is provided with cache memory while the GP1000 processor is not. In Muse the Prolog tables are not cachable while in SICStus system all code, data, and tables are cachable. In Muse the Prolog tables are partitioned into sections and each section resides in the local memory of a processor. Access to cache memory is much faster than to local (and remote) memory.

The corresponding overhead per worker for the Aurora system on TC2000 is around 77%. One reason why the overhead per worker is higher in Aurora (than in Muse) on the TC2000 is that in Aurora the WAM stacks are shared by the all workers, and thus the stacks are not cachable. In Muse each worker has its own copy of the WAM stacks, and thus it is straightforward to make the WAM stacks cachable. There is also the overhead of maintaining the binding arrays used in Aurora. The average real speedup on 32 TC2000 processors, for the programs with coarse grain parallelism, is 17.6 for Aurora and 25.4 for Muse. For programs with finer parallelism, the real speedups are clearly in favor of the Muse system by at least a factor 2. For all benchmarks used in this chapter, the Muse system is faster than the Aurora system.

The delayed release mechanism supported by the Muse system avoids a continuous decrease in task sizes as the number of workers grows. Without such a mechanism the gains due to parallel execution can be lost as the number of workers is increased, due to the increasing overheads for scheduling work because of decreasing task sizes. The scheduling overhead of the Muse system on TC2000 for a set of benchmarks is around 8 – 26 Prolog procedure calls per task. The corresponding cost for Muse on Sequent Symmetry is around 5 – 7 Prolog procedure calls per task. The relative cost of scheduling overhead is higher on TC2000 than on Sequent Symmetry due to the relatively high ratio of processor speed to communication speed in the former. Thus, the task size should be controlled at runtime to be not less than 26 Prolog procedure calls on TC2000 in order to avoid performance degradation.

In the near future we plan to support handling of speculative work better on the Muse system by using one of the schemes presented in [45]. The Muse group at SICS is collaborating with a group at BIM to extend the BIM sequential Prolog system to an OR-parallel implementation using the Muse approach. We believe that extending the BIM Prolog system will be as simple and efficient as extending the SICStus Prolog system.

5.7 Acknowledgments

We would like to thank the Argonne National Laboratory group for allowing us to use their Butterfly machines. Shyam Mudambi was supported by NSF grant CCR-8718989.

Chapter 6

OR-parallel Speedups in a Knowledge Based System: on Muse and Aurora

Khayri A. M. Ali   *Roland Karlsson*

In proc. of FGCS'92 [10]
Revised March 1992

THIS chapter presents the results of running a knowledge based system that applies a set of rules to a circuit board design and reports any design errors, on two OR-parallel Prolog systems, Muse and Aurora, implemented on a number of shared memory multiprocessor machines. The knowledge based system is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. When the system was tested on Muse and Aurora, without any modifications, the OR-parallel speedups were very encouraging for this large practical application. The number of processors used in our experiment was 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The results obtained show that the Aurora system is much more sensitive to the machine architecture than the Muse system, and the latter is faster than the former on all the three machines used. The real speedup factors of Muse, relative to SICStus, are 24.3 on S81, 31.8 on TC2000, and 46.35 on GP1000.

6.1 Introduction

Two main types of parallelism can be extracted from a Prolog program. The first, AND-parallelism, utilizes possibilities for simultaneous execution of several subproblems offered by Prolog semantics. The second, OR-parallelism, utilizes possibilities for simultaneous search for multiple solutions to a single problem. This chapter is concerned with two systems exploiting only the latter type of parallelism: Muse and Aurora, described in Chapters 3 and 4 and in [65]. Both systems support the full Prolog language with its standard semantics, and they have been implemented on a number of shared multiprocessor machines, ranging from a few processors up to around 100 processors. Both systems show good speedups, in comparison with good sequential Prolog systems, for programs with a high degree of OR-parallelism. The two systems are based on two different memory models. Aurora is based on the SRI model [92] and Muse on incremental copying of the WAM stacks, as described in Section 3.2. The two systems have been implemented by adapting the same sequential Prolog system, SICStus version 0.6. The overhead associated with this adaptation is low and depends on the Prolog program and the machine architecture. For a large set of benchmarks, the average overhead for the Muse system on one processor is around 5% on Sequent Symmetry, 8% on BBN Butterfly GP1000, and 22% on BBN Butterfly TC2000. For the Aurora system with the same set of the benchmarks, it is around 25% on Sequent Symmetry, 30% on BBN Butterfly GP1000, and 77% on BBN Butterfly TC2000. Earlier results presented in Sections 3.9, 4.7 and 5.5 show that the Muse system is faster than the Aurora system for a large set of benchmarks on the above mentioned machines.

In this chapter we investigate the performance results of Muse and Aurora systems on those multiprocessor machines for a large practical knowledge based system [52, 41]. The knowledge based system is used to check a circuit board (or a gate array) design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or testability requirements. The knowledge based system has been written in SICStus Prolog [26], by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. The gate array used in our experiment consists of 755 components. The system was tested on Muse and Aurora without any modifications. One important goal that has been achieved by Muse and Aurora systems is getting OR-parallel speedups with almost no user annotations.

The speedup results obtained are very good on all the machines used for the Muse system, but not for Aurora on the Butterfly machines. We found that this application has high OR-parallelism. In this chapter we are going to present and discuss the results obtained from the Aurora and Muse systems on the three machines used.

The chapter is organized as follows. Section 6.2 briefly describes the three machines used in our experiment. Section 6.3 briefly describes the two OR-parallel Prolog systems, Muse and Aurora. Section 6.4 presents the knowledge based system. Sections 6.5 and 6.6 present and discuss the experimental results. Section 6.7 concludes the chapter.

6.2 Multiprocessor Machines

The three machines used in our study are Sequent Symmetry S81, BBN Butterfly TC2000, and BBN Butterfly GP1000. Sequent Symmetry is a shared memory machine with a common bus capable of supporting up to 30 (i386) processors. Each processor has a 64-KByte cache memory. The bus supports cache coherence of shared data and its capacity is 80 MByte/sec. It presents the user with a uniform memory architecture and equal access time to all memory.

The Butterfly GP1000 is a multiprocessor machine capable of supporting up to 128 processors. The GP1000 is made up of two subsystems, the processor nodes and the Butterfly switch, which connects all nodes. A processor node consists of an MC68020 microprocessor, 4 MByte of memory and a Processor Node Controller (PNC) that manages all references. A non-local memory access across the switch takes about 5 times longer than local memory access (when there is no contention). The Butterfly switch is a multi-stage omega interconnection network. The switch on the GP1000 has a hardware supported block copy operation, which is used to implement the Muse incremental copying strategy. The peak bandwidth of the switch is 4 MBytes per second per switch path.

The Butterfly TC2000 is similar to the GP1000 but is a newer machine capable of supporting up to 512 processors. The main differences are that the processors used in the TC2000 are Motorola 88100s. They are an order of magnitude faster than the MC68020 and have two 16 KBytes data and instruction caches. Thus in the TC2000 there is actually a three level memory hierarchy: cache memory, local memory and remote memory. Unfortunately no support is provided for cache coherence of shared data. Hence by default shared data are not cached on the TC2000. The peak bandwidth of the Butterfly switch on the TC2000 is 9.5 times that of the Butterfly GP1000 (at 38 MBytes per second per path). The TC2000 switch does not have hardware support for block copy.

6.3 OR-Parallel Systems

In Muse and Aurora, OR-parallelism in a Prolog search tree is explored by a number of *workers* (processes or processors). A major problem introduced by OR-parallelism is that some variables may be simultaneously bound by workers exploring different branches of a Prolog search tree. Two different approaches have been used in Muse and Aurora for solving this problem. Muse uses incremental copying of the WAM stacks as described in Section 3.2 while Aurora uses the SRI memory model [92].

The idea of the SRI model is to extend the conventional WAM with a binding array for each worker and modify the trail to contain address-value pairs instead of just addresses. The binding array contains the conditional bindings associated with the workers' branch. A binding is conditional if a variable can get several bindings. The WAM stacks are shared by all workers. The nodes of the search tree contain extra fields to enable workers to move around in 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.

The incremental copying of the WAM stacks used in Muse is based on having a number of 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 WAM stacks. The stacks are not shared between workers. Thus, each worker has bindings associated with its current branch in its own copy of the stacks. This simple solution allows the existing sequential Prolog technology to be used without loss of efficiency. But it requires copying data (stacks) from one worker to another when a worker runs out of work. In Muse, workers incrementally copy parts of the (WAM) stacks and also share nodes with each other when a worker runs out of work. The two workers involved in copying will only copy the differing parts between the two states. The shared memory space stores information associated with the shared nodes in the search tree. Workers get work from shared nodes through the normal backtracking mechanism of Prolog. Each worker having its own copy of the WAM stacks simplifies garbage collection and caching the WAM stacks on machines, like the BBN Butterfly TC2000, that do not support cache coherence of shared data.

A node in a Prolog search tree corresponds to a Prolog choicepoint. Nodes are either *shared* or *nonshared (private)*. These nodes divide the search tree into two regions: *shared* and *private*. Each worker can be in either engine mode or scheduler mode. In engine mode, the worker works like a sequential Prolog system on private nodes, but is also able to respond to requests from other workers. Anytime a worker has to access the shared region of the search tree, it switches to scheduler mode and establishes the necessary coordination with other workers. The two main functions of a worker in scheduler mode are to maintain the sequential semantics of Prolog and to match idle workers with the available work with minimal overhead.

The two systems, Muse and Aurora, have different working schedulers on the three machines used in our experiment. Aurora has two schedulers: the Argonne scheduler [22] and the Manchester scheduler [24]. According to the reported results, the Manchester scheduler always gives better performance than the Argonne scheduler [68, 83]. So the Manchester scheduler was used for Aurora in our experiment. Muse has only one scheduler. It is described in Chapter 4.

The main difference between the Manchester scheduler for Aurora and the Muse scheduler is in the strategy used for dispatching work. The strategy used by the Manchester scheduler is to take work from the topmost node in a branch, and only one node at a time is shared. In Muse, several nodes at a time are shared and work is taken from the bottommost node in a branch. The bottommost strategy approximates the execution of sequential implementations of Prolog within a branch. Another difference between the two schedulers is in the algorithms used in the implementation of cut and side effects to maintain the standard Prolog semantics.

Many optimizations have been made of the implementation of the Aurora and Muse systems on all three machines. The only optimization that has been implemented for Muse and not for Aurora is caching the WAM stacks on the BBN Butterfly TC2000. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. Therefore, it is straightforward for Muse to make the WAM stack areas cachable whereas in Aurora it requires a complex cache coherence protocol to achieve this effect.

6.4 The Knowledge Based System

An important process in the design of circuit boards and gate arrays is the checking of the design with respect to a set of rules. These rules may for example be imposed by the development tool, by company standards or by testability requirements. Until now, many of these rules have only been documented on paper. The check is performed manually by people who know the rules well. Increasing the number of gates in circuit boards (or gate arrays) makes the manual check a very difficult process. Computerizing this process is very useful and may be the most reliable solution. The knowledge based systems group at SICS, in collaboration with groups from some Swedish companies, has been developing a knowledge based system that applies a set of rules to a circuit board (or gate array) design and reports any design errors [41, 52]. The groups have developed two versions of the knowledge based system. The first version has been developed using a general purpose expert system shell while the second has been developed using SICStus Prolog. The latter, which will be used in our experiment, is more flexible and more efficient than the former. It is around 10 times faster than the first version on single processor machines. When it has been tested, without any modifications, on Muse and Aurora systems on Sequent Symmetry, the speedups obtained are linear up to 25 processors. Thus an OR-parallel Prolog implementation is around 250 times faster than the expert system shell implementation.

One reason for the high degree of OR-parallelism in this kind of application is that all of the rules applied to the circuit board (or gate array) design are independent of each other. The second source of OR-parallelism is the application of each rule to all instances of a given circuit sub-assembly on the board. A circuit sub-assembly can be either a component (like *buffer, inverter, nand, and, nor, or, xor,* etc.) or a group of interconnected components. The knowledge based system consists mainly of an inference engine, design rules, and a database describing the circuit board (or gate array). The inference engine is implemented as an interpreter with only 8 Prolog clauses. The gate array used in our experiment consists of 755 components (Texas gate array family TGC–100), and is described by around 10000 Prolog clauses. The design rules part with its interface to the gate array description is around 200 Prolog clauses. Eleven independent rules are used in this experiment. The interpreter applies the set of rules to the gate array description. For a larger gate array more OR-parallelism is expected. It should be mentioned that the people who developed the knowledge based system did not at all consider parallelism, but they tried to make their system easy to maintain by writing clean code. They avoided using side effects, but they have used cuts (embedded in If_Then_Else) and findall constructs. The user interface part of this application is not included in our experiment.

Since Muse and the Aurora system are also running on larger machines, the BBN Butterfly machines, it was natural to test the knowledge based system on those machines. The speedup results obtained differ for the Muse and the Aurora system. On 37 TC2000 processors, Muse is 31.8 times faster than SICStus, while Aurora is only 7.3 times faster than SICStus. Similarly, on 70 GP1000 processors Muse is 46.35 times faster than SICStus, while Aurora is only 6.68 times faster than SICStus. The low speedup for the Aurora system is surprising since this application is rich in OR-parallelism. Is this a scheduler problem for Aurora or an engine problem? The following two sections will present and analyze the results of Muse and Aurora, in order to try to answer this question.

6.5 Timings and Speedups

In this section we present timing and speedup results obtained from running the knowledge based system on Muse and Aurora. The execution times given in this chapter are the mean values obtained from eight runs. On Sequent Symmetry, there is no significant difference between mean and best values, whereas on the Butterfly machines, mean values are more reliable than best values due to variations of timing results from one run to another. (These variations are due mainly to switch contention.) Variations around the mean value will be shown in the graphs by a vertical line with two short horizontal lines at each end. The speedups given in this section are relative to execution times of Muse on one processor on the corresponding machine. The SICStus one-processor execution time on each machine will also be presented to determine the overhead associated with adapting the SICStus Prolog system to the Aurora and Muse systems. Sections 6.5.1, 6.5.2, and 6.5.3 present those results on Sequent Symmetry, GP1000, and TC2000 machines respectively.

6.5.1 Sequent Symmetry

Table 6.1 shows the execution times of Aurora and Muse on Sequent Symmetry, and the ratio between them. Times are shown for 1, 5, 10, 15, 20, and 25 workers with speedups (relative to one Muse worker) given in parentheses. The SICStus execution time on one Sequent Symmetry processor is 422.39 seconds. This means that for this application and on the Sequent Symmetry machine the overhead associated with adapting the SICStus Prolog system to Aurora is 26.3%, and for Muse is only 1.0% (calculated from Table 6.1). The performance results that Table 6.1 illustrates are good for both systems, and Aurora timings exceed Muse timings by 25% to 26% for 1 to 25 workers. Figure 6.1 shows speedup curves for Muse and Aurora on Sequent Symmetry. Both systems show linear speedups with no significant variations around the mean values.

6.5.2 BBN Butterfly GP1000

Table 6.2 shows the execution times of Aurora and Muse on GP1000 for 1, 10, 20, 30, 40, 50, 60, and 70 workers. The SICStus execution time on one GP1000 node is 534.4 seconds. So, for this application and on the GP1000 machine the overhead

Workers	Aurora	Muse	Aurora/Muse
1	533.69(0.80)	426.74(1.00)	1.25
5	106.87(3.99)	85.67(4.98)	1.25
10	53.58(7.96)	42.94(9.94)	1.25
15	36.06(11.8)	28.73(14.9)	1.26
20	27.22(15.7)	21.65(19.7)	1.26
25	21.83(19.5)	17.39(24.5)	1.26

Table 6.1: Aurora and Muse execution times (in seconds) on Sequent Symmetry, and the ratio between them.

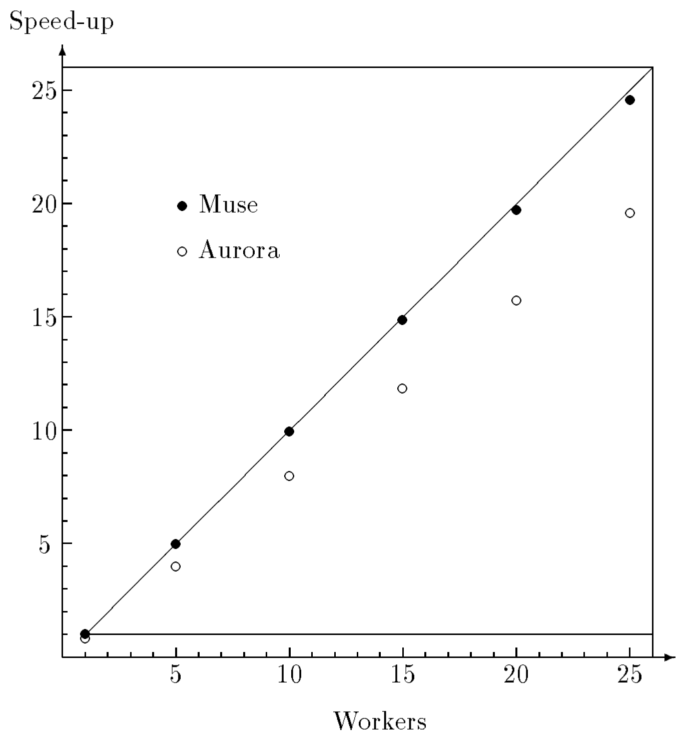


Figure 6.1: Speedups of Muse and Aurora on Sequent Symmetry, relative to 1 Muse worker.

associated with adapting the SICStus Prolog system to Aurora is 66%, and for Muse only 7%. Here the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 55% to 594% for 1 to 70 workers.

Workers	Aurora	Muse	Aurora/Muse
1	886.4(0.65)	572.3(1.00)	1.55
10	105.3(5.44)	58.3(9.82)	1.81
20	74.1(7.72)	29.8(19.2)	2.49
30	72.7(7.88)	20.7(27.7)	3.52
40	64.3(8.91)	16.1(35.5)	3.99
50	72.4(7.90)	13.8(41.6)	5.26
60	65.7(8.71)	12.4(46.1)	5.29
70	80.0(7.15)	11.5(49.6)	6.94

Table 6.2: Aurora and Muse execution times (in seconds) on GP1000 and the ratio between them.

Figure 6.2 shows speedup curves corresponding to Table 6.2 with variations around the mean values. The speedup curve for Aurora levels off beyond around 20 workers. On the other hand, the Muse speedup curve levels off as more workers are added.

6.5.3 BBN Butterfly TC2000

Table 6.3 shows the performance results of Aurora and Muse on TC2000 for 1, 10, 20, 30, and 37 workers. The SICStus execution time on one TC2000 node is 100.48 seconds. Thus, for this application and on the TC2000 machine the overhead associated with adapting the SICStus Prolog system to Aurora is 80%, and for Muse is only 5%. Here also the performance results are good for the Muse system but not for the Aurora system. Aurora timings are longer than Muse timings by 70% to 319% for 1 to 37 workers.

Workers	Aurora	Muse	Aurora/Muse
1	180.55(0.59)	105.97(1.00)	1.70
10	22.12(4.79)	10.81(9.80)	2.05
20	16.02(6.61)	5.56(19.1)	2.88
30	13.66(7.76)	3.93(27.0)	3.48
37	13.79(7.68)	3.29(32.2)	4.19

Table 6.3: Aurora and Muse execution times (in seconds) on TC2000 and the ratio between them.

Figure 6.3 shows speedup curves corresponding to Table 6.3. The speedup curves are similar to the corresponding ones shown in Figure 6.2.

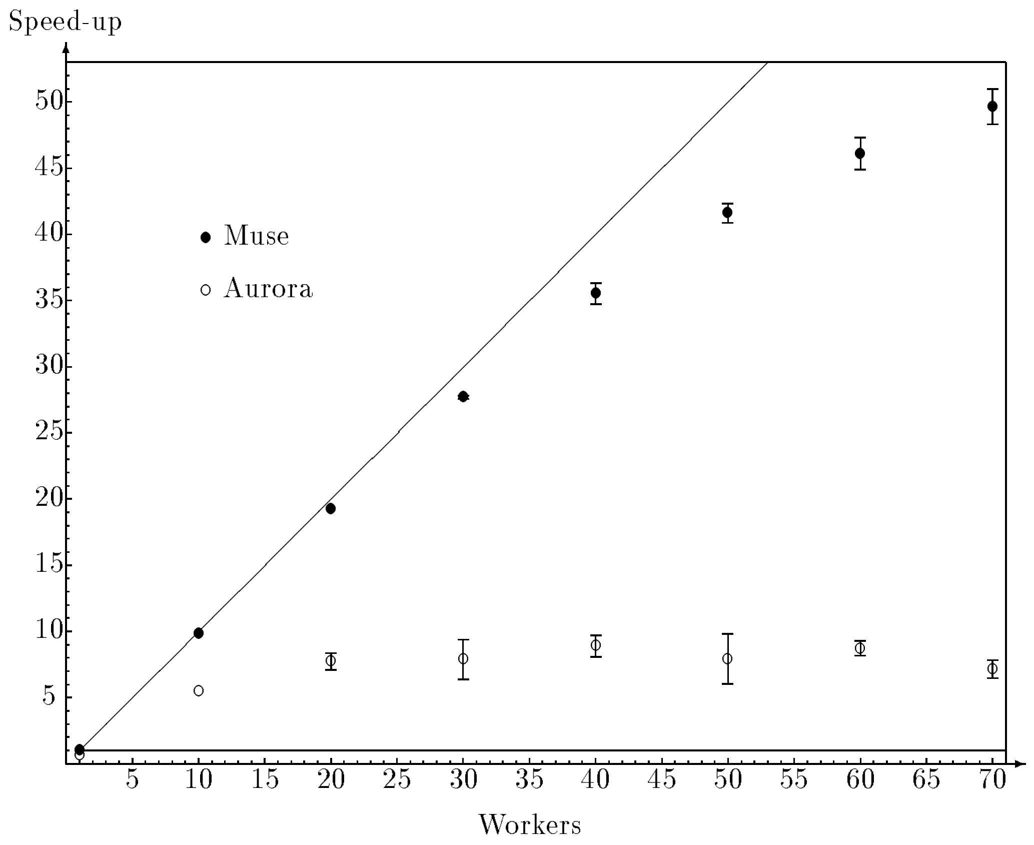


Figure 6.2: Speedups of Muse and Aurora on GP1000, relative to 1 Muse worker.

6.6 Analysis of the Results

From the results presented in Section 6.5 we found that the Muse system shows good performance results on the three machines, whereas the Aurora system shows good results on the Sequent Symmetry only. In this section, we try to explain these results by studying the Muse and Aurora implementations on one of the Butterfly machines (TC2000). The TC2000 has better support for reading the realtime clock than the GP1000. A worker's time can be divided into the following three main activities:

1. *Prolog:* time spent executing Prolog (i.e. engine time).
2. *Idle:* time spent waiting for work to be generated when there is temporarily no work available in the system.
3. *Others:* time spent in all the other activities (i.e. all scheduling activities) like acquiring a spin lock, sending requests to other workers, performing cut, grabbing work, sharing work, looking for work, binding installation (and copying in Muse), synchronization between workers, etc.

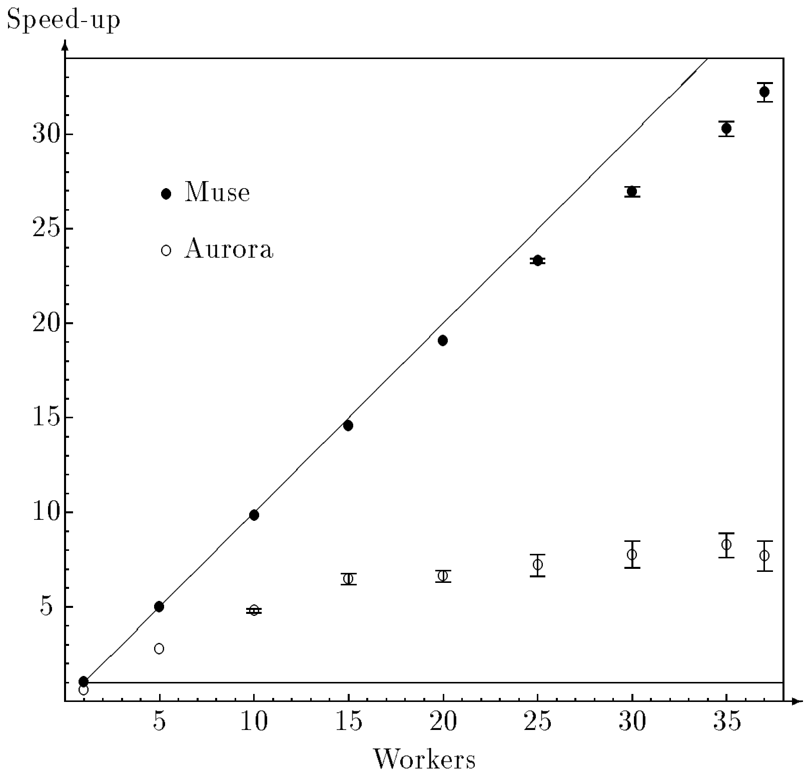


Figure 6.3: Speedups of Muse and Aurora on TC2000, relative to 1 Muse worker.

Tables 6.4 and 6.5 show time spent in each activity and the corresponding percentage of the total time. Results shown in Tables 6.4 and 6.5 have been obtained from instrumented versions of Muse and Aurora on the TC2000. The times obtained from the instrumented versions are longer than those obtained from uninstrumented systems by around 19–27%. We believe that the percentage of time spent in each activity obtained from the instrumented system reflects what is happening in the uninstrumented system.

	Activity		
Muse Workers	Prolog	Idle	Others
1	128.36(100)	0	0
5	128.80(99.7)	0.09(0.1)	0.26(0.2)
10	129.28(99.1)	0.40(0.3)	0.71(0.5)
20	129.90(96.5)	3.56(2.6)	1.17(0.9)
30	130.32(95.4)	4.17(3.0)	2.11(1.5)

Table 6.4: Time (in seconds) spent in the main activities of Muse workers on TC2000.

Before analyzing the data in Tables 6.4 and 6.5 we would like to make two remarks on these data. The first remark is that in the Aurora system the overhead of checking for the arrival of requests is separated from the Prolog engine time, while in the Muse system there is no such separation. This explains why there is scheduling

	Activity		
Aurora Workers	Prolog	Idle	Others
1	210.42(98.2)	0	2.36(1.1)
5	221.24(98.3)	0.19(0.1)	2.03(0.9)
10	235.34(98.1)	0.43(0.2)	2.43(1.0)
20	329.60(98.1)	1.11(0.3)	3.61(1.1)
30	412.97(94.7)	13.70(3.2)	7.64(1.8)

Table 6.5: Time (in seconds) spent in the main activities of Aurora workers on TC2000.

overhead (*Others*) in the 1 worker case in Table 6.5 and not in Table 6.4. The other remark is that the figures obtained from the Aurora system do not total 100% of time, since a small fraction of the time is not allocated to any of the three activities. However, these two factors have no significant impact on the following discussion.

By careful investigation of Tables 6.4 and 6.5 we find that the total *Prolog* time of Muse workers is almost constant with respect to the number of workers whereas the corresponding time for Aurora grows rapidly as new workers are added. We also find that the scheduling time (*Others*) in Table 6.5 is not very large in comparison with the corresponding time in Table 6.4. Similarly, the difference in *Idle* time between Muse and Aurora is not so high. So the main reason for performance degradation in Aurora is the Prolog engine speed.

We think that the only factor that slows down the Aurora engine as more workers are added is the high access cost of non-local memory. Non-local memory access takes longer time than local memory access, and causes switch contention. Non-local memory accesses can refer to either the global Prolog tables[1] or the WAM stacks. In Muse and Aurora systems, the global tables are partitioned into parts and each part resides in the local memory of one processor. In Aurora the WAM stacks are shared by all workers while in Muse each worker has its own copy of the WAM stacks. The global Prolog tables have been implemented similarly in the Muse and Aurora systems. Since the Muse engine does not have any problem with the Prolog tables, the problem should lie in the sharing of the WAM stacks in Aurora, coupled with the fact that this application generates around 9.8 million conditional bindings, and executes around 1.1 million Prolog procedure calls. On the average, each procedure call generates around 9 conditional bindings. This may mean that the reason why Aurora slows down lies in the cactus stack approach, which causes a great many non-local accesses to the Prolog stacks. This results in a high amount of switch contention when executing on more than five workers. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and the copy is even cachable. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the TC2000 does not provide any support for measuring the stack variables access time.

[1]By Prolog tables we mean the atom table, the predicate table, and also the allocated records associated with occupied items in the tables.

6.7 Conclusions

The results of running a large practical knowledge based system on two OR-parallel Prolog systems, Muse and Aurora, have been presented and discussed. The number of processors used in our experiment was 25 on Sequent Symmetry (S81), 37 on BBN Butterfly II (TC2000), and 70 on BBN Butterfly I (GP1000). The knowledge based system used in our study checks a circuit board (or gate array) design with respect to a set of rules and reports any design errors. It is written in SICStus Prolog, by the Knowledge Based Systems group at SICS in collaboration with groups from some Swedish companies, without considering parallelism. It was used in our experiment without any modifications.

The results of our experiment show that this class of applications is rich in OR-parallelism. Very good real speedups, in comparison with the SICStus Prolog system, have been obtained for the Muse system on all three machines. The real speedup factors for Muse are 24.3 on 25 S81 processors, 31.8 on 37 TC2000 processors, and 46.35 on 70 GP1000 processors. The obtained real speedup factors for Aurora are lower (than for Muse) on Sequent Symmetry, and much lower on the Butterfly machines. The Aurora timings are longer than Muse timings by 25% to 26% for 1 to 25 S81 processors, 70% to 319% for 1 to 37 TC2000 processors, and 55% to 594% between 1 to 70 GP1000 processors.

The analysis of the obtained results indicates that the main reason for this great difference between Muse timing and Aurora timing (on the Butterfly machines) lies in the Prolog engine and not in the scheduler. The Aurora engine is based on the SRI memory model in which the WAM stacks are shared by the all workers. We think that the only reason why the Aurora engine slows down as more workers are added is the large number of non-local accesses to stack variables. This results in a high amount of switch contention as more workers are added. This is avoided in the Muse model, since each worker has its own copy of the WAM stacks in the processor local memory and the stacks are even cachable in the TC2000. Unfortunately, we could not verify this hypothesis because the current Aurora implementation on the Butterfly machines does not provide any support for measuring access time to stack variables.

6.8 Acknowledgments

We would like to thank the Argonne National Laboratory group for allowing us to use their Sequent Symmetry and Butterfly machines. We thank Shyam Mudambi for his work on porting Muse and Aurora to the Butterfly machines. We also would like to thank Fredrik Holmgren, Klas Orsvärn and Ingvar Olsson for discussions and for allowing us to use their knowledge based system.

Chapter 7

How to Build your own OR-parallel Prolog System

Roland Karlsson
SICS research report [57]

THIS chapter shows how to extend an existing Prolog system to automatically exploit OR-parallelism. The description starts with parallelizing pure Prolog, a Prolog version without either cut or side effects. The model is incrementally refined until finally reaching an efficient OR-parallel system for full Prolog with extra non-Prolog features. I have tried to keep the text as general as possible. When the text becomes too SICStus (my target Prolog system) specific some hints for the ordinary WAM is given. The chosen OR-parallel model is the Muse model. It is relatively straightforward using this model to extend most existing Prolog systems.

7.1 Introduction

The goal of this chapter is to show how to extend any sequential Prolog system based on the WAM to OR-parallel Prolog using the Muse execution model. It discusses and describes the implementation techniques used in the current Muse system. This chapter can also be seen as an introduction to the Muse interface found in Chapter 8.

Here we assume the reader is familiar with the WAM [91] and with the Muse model found in Chapters 3 and 4.

We start in Section 7.2 giving some terminology used in this chapter. Section 7.3 describes our layered approach for gradually extending any sequential Prolog system to an OR-parallel (full) Prolog system.

Sections 7.4, 7.5, and 7.6 show how to extend a sequential Prolog system to a OR-parallel (pure) Prolog system. Section 7.4 describes very briefly the Muse model and the required extensions to the sequential Prolog execution model. Section 7.5 describes in some detail the modifications to the WAM and its data structures. Section 7.6 gives a simple and rather efficient scheduler for pure Prolog programs.

Section 7.7 shows how to get an OR-parallel (full) Prolog system.

Section 7.8 discusses important optimizations including a novel approach for efficient implementation of assert and all-solutions predicates. The approach allows findall solutions to be constructed in the same order as in a sequential Prolog system, and with very good performance. Section 7.9 points out a reason for keeping the sequential semantics of findall construct in parallel Prolog systems.

Sections 7.10 and 7.11 discuss adding non-Prolog features to OR-parallel Prolog systems.

Sections 7.12 and 7.13 discuss machine dependent aspects of Muse on shared and distributed memory machines.

Section 7.14 discusses debugging and evaluation tools developed for the Muse system.

The last two sections discuss new mechanisms and features that could be implemented in the next generation of the Muse system. Section 7.15 discusses some advanced topics like efficient mechanisms for cuts, commits, and scheduling of speculative work. Section 7.16 discusses features that should be considered in a commercial OR-parallel Prolog system.

7.2 Inherited Terminology

In this chapter I try to use a consistent terminology. Most of the terminology is inherited from Mathematics, Computer Science, Prolog, OR-parallel Prolog, and the Aurora OR-parallel Prolog [65]. Sometimes there exist conflicting uses of some concept. Then I try (if it does not cause too much confusion) to use the definition from the more general field.

A Prolog program can be defined as follows. A *clause* consists of a head and a body. The *body* is a sequence (possibly empty) of literals. The *head* and the *literals* are Prolog structures of the form: $name(Arg_1 \ldots Arg_n)$. The *arity* of a structure is the number of arguments. The *name* and *arity* of a clause is the name and arity of the head of the clause. A *predicate* consists of a sequence of clauses, with the same name and arity. A *program* consists of a set of predicates.

As an optimization (e.g. indexing, decision tree) not all clauses of a predicate have to be considered. The actual clauses considered are called *alternatives*.

A Prolog execution constructs a virtual *execution tree*. The nodes in the tree consist of AND-nodes and OR-nodes and the edges correspond to the flow of control. Each Prolog predicate call creates an *AND-node* and each time a predicate with more than one alternative is called, an *OR-node* is created.

The OR-nodes are in Prolog implemented as *choicepoints*. In a choicepoint a *computation state* and a *next alternative pointer* are stored. The SICStus version of WAM (Warren Abstract Machine) uses four stacks. The *choicepoint stack* contains choicepoints, the *environment stack* contains control information and variables local to a procedure, the *term stack* contains long lived variables and compound terms, and the *trail stack* contains reset information for backtracking. In ordinary WAM the choicepoint stack and the environment stack are combined into a *control stack*.

A *worker* is (in our model) an ordinary sequential Prolog system with its own WAM stacks, extended with scheduler code and some global information for OR-parallel execution. A worker is normally associated with an operating system process. In Muse and Aurora the (somewhat extended) original sequential Prolog code is called the *engine* code and the code doing (mostly) scheduler activities is called the *scheduler* code.

Choicepoints may (in our model) be made shareable (among workers). Those choicepoints are then extended with a global *shared frame* and referred to as *shared choicepoints*. Choicepoints not shareable are called *private choicepoints*. Existing shared choicepoints constructs a *shared search tree*. Shared choicepoints is also referred to as *nodes* (of the current shared search tree).

In OR-parallel Prolog a *task* is a piece of Prolog work that can be executed without asking the scheduler for more work. The tasks can be found in shared choicepoints.

The execution of a task can be *suspended*. The suspension can be the result of a worker being unable to continue executing the task or the result of some scheduling decision. A suspended task may be *resumed* later on. A worker, currently responsible for a suspended task, may be *rescheduled* to some other task. Storing information in a shared choicepoint about a suspended task in order to make it possible to resume the task later on is called *suspending* the branch rooted at the choicepoint and associated to the task.

7.3 The Layered Approach

An OR-parallel Prolog system is best described (and implemented) using the layered approach shown in Figure 7.1. The kernel is a sequential pure Prolog system. This pure Prolog kernel can be extended to either sequential (full) Prolog (with a side effect/cut layer) or parallel pure Prolog (with extensions to explore OR-parallelism). The parallel pure Prolog layer can easily be implemented using an existing Prolog system, if only pure Prolog user programs are allowed to run in parallel. Adding a parallel full Prolog layer as a glue for the the other layers completes the system to an OR-parallel system for full Prolog. When the system is up and running, all layers can be optimized.

7.4 The Muse Model

The Muse execution model (described in Chapter 3) is based on having a number of workers, each with its own local memory space, and some global memory space

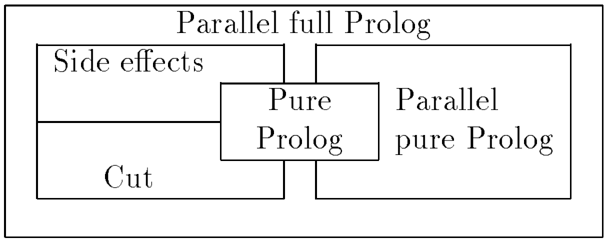


Figure 7.1: The layered approach.

shared by all workers. Each worker is a sequential Prolog system with its own WAM stacks. When a worker Q runs out of work, it tries to get work from another worker P having excess work. P allows Q to get some of its work by first sharing with Q all choicepoints that Q cannot reach, then copying the difference between the two workers' states. Now, the two workers have identical states and both share the same choicepoints. They can explore alternatives in the shared choicepoints by using the normal backtracking of Prolog.

When a worker takes an alternative from a shared choicepoint, it proceeds exactly like a sequential Prolog system, creating new choicepoints in its private choicepoint stack. It also maintains a measure of its load and responds to requests from the other workers.

The sharing operation associates one global frame with each choicepoint. The frame consists of a lock, available alternatives and other scheduling information. In Section 7.4.1 an efficient way of getting alternatives from frames associated with shared choicepoints is described.

Section 7.4.2 explains how each worker maintains its load measure. This measure is used to allow better selection of a worker with excess load.

Section 7.4.3 discusses what we call *incremental copying* — the method used in Muse to copy the difference between the states of two workers.

7.4.1 Failing to a Shared Choicepoint

From the description of the Muse model one can deduce that the Prolog system has to distinguish between backtracking to a private or to a shared choicepoint. When backtracking to a private choicepoint the next alternative is to be taken from the choicepoint and for a shared choicepoint from the associated global frame. There is a simpler and more efficient solution though. When a choicepoint is made shared, the next alternative pointer in the choicepoint can be replaced with a pointer to special code for taking the next alternative from the global frame. This solution introduces no overhead at all in sequential execution and no extra space is needed in the choicepoint. The details for implementing this method are presented in Section 7.5.1.

7.4.2 Keeping the Load Register Updated

Each busy worker needs to maintain a measure of the amount of work it can share with other workers. This measure is called the *load*, and it is stored in a global register associated with each worker.

The load is defined for the current Muse implementation as the the number of untried alternatives along the current branch and it can refer to one of two categories: alternatives belonging to shared choicepoints (*shared load*) and alternatives belonging to private choicepoints (*private load*).

It is very expensive to keep the measure exact. The amount of work in shared choicepoints is changed by other workers. The amount of work in private choicepoints changes very often. But, the measure is used for scheduling purposes only, mainly for optimizing choices. A compromise is made in the current implementation of Muse. An approximate value reflecting the private load is maintained while running Prolog. An exact value for the private load is calculated each time the busy worker gets a sharing request and when a choicepoint is created. The existence of any shared load (not the exact value) is calculated on demand.

To make it possible to compute the private load a value is stored in each private choicepoint. The value corresponds to the sum of the unexplored alternatives in the older private choicepoints. (The advantage in not including the current choicepoint when computing this value is that no change has to be made on backtracking.) The value in the load register is computed as the sum of the value saved in the youngest choicepoint and the untried alternatives in the youngest choicepoint. The private load changes at creation and destruction of choicepoints. For efficiency reasons it is computed and set at the creation of choicepoints only.

7.4.3 Incremental Copying

We can take advantage of one important aspect of the WAM memory management: it is easy to compute the difference between two states in the same branch. This makes it possible to copy only the differing parts between two workers' states.

As usual the worker P is copying to the worker Q. We refer to Q's stack segments as the *old* stack segments and to P's additional stack segments as the *new* stack segments. The information in the old stack segments is *almost* identical for P and Q. Pointers to all the differences can easily be found in P's trail.

First all the new stack segments are copied from P to Q. Then the old segment of Q's binding environment is updated, using the trail. For each trailed pointer in the new trail segment that refers to the old binding environment the corresponding binding environment cell is copied from P to Q. The latter function we call *installation*.

Both copying and installation cost are now proportional to the distance (measured as the sum of the stack sizes) when moving down along a branch, which is a very important property. Moreover, of practical importance is that copying of memory blocks is efficient. The Muse results show that the incremental copying approach is competitive with the installing from the binding array of the SRI model [92].

The only problem to solve is garbage collection. The two workers may not have the same layout of the stacks if either of the two workers has performed garbage collection. The solution in the current version of Muse is to do a total copy in this case. Another approach would be to let both P and Q perform garbage collection before doing the incremental copying. The SICStus garbage collection guarantees that the layout of the old stack is the same for both P and Q. A third alternative is described in Section 3.8.2.

7.5 Extending the WAM

The sequential Prolog system we have extended to explore OR-parallelism is SICStus Prolog [26], a slightly modified WAM [91] system. How to extend the SICStus WAM and the ordinary WAM is now shown.

7.5.1 Data Structures

The only two data structures of interest for OR-parallel pure Prolog are the representation of the list of alternative clauses in a predicate and the choicepoint stack.

In SICStus each clause has its own WAM code. Clauses of a predicate are represented by a set of linked lists. On calling a predicate, the SICStus system, via interpretation of indexing data structures chooses one of the linked lists. (Indexing is an optimization that, given the call arguments, limits the number of clauses that have to be considered.) Each element (called alternative) of the linked list contains a pointer to code for a clause. When calling a predicate having more than one alternative (in the chosen linked list) a choicepoint is created. The computation state and a reference to the next alternative are stored in the choicepoint. On backtracking the computation state is restored from the youngest choicepoint and the stored alternative is taken. If the last alternative was taken then the choicepoint is deallocated. Otherwise the next alternative is stored in the choicepoint.

The choicepoint stack could be left unchanged, but in the current Muse implementation there is an extra field called *lub* (Local Untried Branches) added in each choicepoint. This field contains an integer used to compute the current load, the measure of the amount of untried work to backtrack for in the private region, as described in Section 7.4.2. The list of alternative could also be unchanged. But here an extra field called *ltt* (alternatives Left To Try) is added to each alternative for the same purpose. (This extension could be avoided if the load were measured as the number of choicepoints instead of alternatives.) The ltt field contains an integer showing how many alternatives are untried when the alternative is chosen. (The extra fields ltt and lub are necessary when implementing full Prolog in the current Muse implementation, so it is not wasted space overhead.)

The changes to the predicate data structure are trivial. Figure 7.2 shows the data structures associated with one private and one shared choicepoint. Except for the ltt field, the data structure can be used unchanged when extending pure Prolog to explore OR-parallelism. When making a choicepoint shared the next-alternative pointer is copied to a field in the global allocated frame. A part of the frame looks just like an alternative (in the linked list). This "alternative" refers to a dummy WAM instruction (called `GetWORK`) that takes the next untried alternative. This code is executed each time someone backtracks to a shared choicepoint. The ltt field in the global frame is set to 0 to indicate that there is no private work in the shared region. There is a small trick involved though. The next element field of this "element" is a self reference to implement an infinite repeat. It is also prudent to include a reference counter in the global frame in order to determine when the frame can be deallocated.

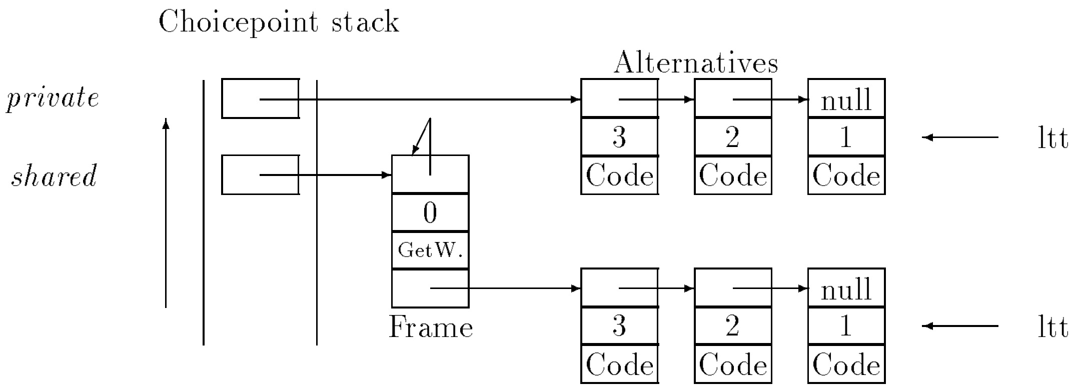


Figure 7.2: The SICStus predicate data structure.

In the WAM all the clauses of a predicate are compiled into one piece of WAM code, including indexing. Special WAM instructions (`try`, `retry`, `trust`) manipulate choicepoints. The `try` instruction creates a choicepoint and takes the first alternative. On backtracking the `retry` instruction takes a non last alternative and the `trust` instruction takes the last alternative (and removes the choicepoint). The choicepoint is similar to a choicepoint in SICStus with one exception: the next alternative pointer refers to code.

We have to add numbering of alternatives in a predicate. One way is to store the alternative number (ltt) in the WAM code as shown in Figure 7.3. An extra parameter is added to the WAM instructions `try`, `retry`, and `trust`. Its purpose is to contain the ltt. Making a choicepoint shared is very easy. Just copy the clause code pointer to the global allocated frame, and then replace the predicate code pointer, in the choicepoint, with a pointer to the pseudo WAM instruction `GetWORK` that takes the next alternative. The extra parameter associated with `GetWORK` is 0.

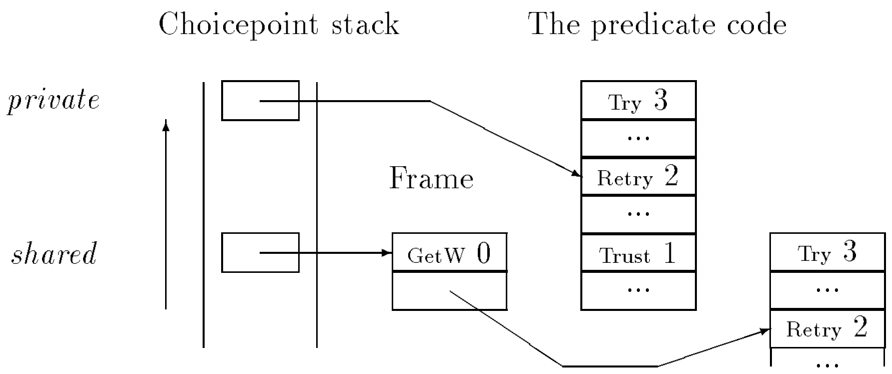


Figure 7.3: The WAM predicate data structure.

7.5.2 The Code

This section starts with a description of a generic WAM emulator and then stepwise refines it, adding the Muse extension. Topics that are irrelevant for adding OR-parallelism using the Muse model, such as indexing, unification, etc, are ignored. A WAM type emulator is only used for convenience. The text is also valid for e.g. native code Prolog systems, using minor transformations.

```
  pc = first address;                /* The program counter. */
  ch = initial (dummy) choicepoint;  /* The youngest choicepoint. */
loop:
  switch (instruction according to pc) {
    case TRUE: ...       goto loop;
    case CALL: ...       goto loop;
    case FAIL: fail: ... goto loop;
    ...
  }
```

Figure 7.4: Iteration 1 of the WAM emulator.

The emulator can be implemented (as in SICStus) as a big switch statement inside an infinite loop, as shown in Figure 7.4. The switch statement chooses one of the cases according to the WAM instruction (referred to by the program counter) and the emulator code for the chosen case is executed. The execution of the code performs those tasks that are necessary, updates the program counter, and (usually) jumps to the loop label. Some instructions have parameters generated at compile time. Those parameters are stored at the next addresses in the code. Some instructions might fail (e.g. unification). In that case the execution jumps to the code for the `FAIL` instruction, using the `fail` label. It is only necessary to consider the choicepoint manipulation made by the `CALL` and the `FAIL` instructions. The WAM instruction `TRUE` is added to show how always successful forward execution is emulated.

Pseudo C code for the WAM instructions `TRUE`, `CALL`, and `FAIL` is expanded in Figure 7.5. The `TRUE` instruction has no parameters and always succeeds. It only increments the program counter and then jumps to the loop label. The emulator can thereafter emulate the next WAM instruction.

```
  pc = first address;
  ch = initial (dummy) choicepoint;
loop:
  switch (instruction according to pc) {
    case TRUE:
      increment pc;
      goto loop;
    case CALL:
      pred = predicate according to (pc+1);
      if (pred is non-existing) goto fail;
      cl = first alternative according to pred;
      next_cl = next alternative according to cl;
      if (next_cl is not null) {
        ch = allocate choicepoint;
        save state, and next_cl in ch;
      }
      set pc to code according to cl;
      goto loop;
    case FAIL: fail:
      use the trail to undo bindings;
      restore state according to ch;
      cl = alternative according to ch;
      next_cl = next alternative according to cl;
      if (next_cl is null)  ch = deallocate choicepoint;
      else                  save next_cl in ch;
      set pc to code according to cl;
      goto loop;
    ...
  }
```

Figure 7.5: Iteration 2 of the WAM emulator.

Before executing the `CALL` instruction the procedure argument values are stored in the argument registers. The `CALL` instruction first fetches a pointer to the called predicate from the position directly following the `CALL` instruction. From that predicate pointer a pointer (called `cl`) to the first element in the list of alternatives is computed. If the next alternative field in `cl` is not null then there is more than one alternative. Then a choicepoint is allocated on the choicepoint stack. In this choicepoint a reference to the next alternative and a representation of the current computation state are stored. The computation state consists of a copy of the procedure arguments and pointers to the tops of the WAM stacks. There are some other, for us uninteresting, tasks that the `CALL` instruction performs. The program counter is set to point to the code of the alternative according to `cl`. Then a jump is made to the loop label.

The code for the `FAIL` instruction can be reached in one of two ways. Either a `FAIL` instruction is emulated or some other WAM instruction (e.g. a unification instruction) has failed. In the latter case the failing instruction uses the `fail` label. The task for the `FAIL` instruction is to reset the computation state and registers according to `ch` and then take the next alternative of `ch`. The choicepoint is removed from the choicepoint stack when the last alternative is taken.

In extending the emulator code to explore OR-parallelism there are several alternatives. The most obvious choice is adding code to the `CALL` instruction to make choicepoints shared and adding code to the `FAIL` instruction to get alternatives from shared choicepoints. Making and manipulating shared choicepoints are not very cheap operations. Therefore choicepoints are made shared on demand only, when receiving a request from another worker needing work. It is also a relatively expensive operation to test whether the current choicepoint is shared, at executions of the `FAIL` code. To avoid this test a trick is used. This trick makes it possible not to change the `FAIL` instruction at all.

The secret of the trick is to use the next alternative pointer in the choicepoint in an unorthodox way. When sharing a choicepoint the next alternative field (in that choicepoint) is changed to point to a special pseudo WAM instruction called `GetWORK`. This instruction corresponds to the `GetWORK` instruction found in Figure 7.2. (It could also be implemented as a call to some built-in predicate implemented in C, but that is more expensive.) This WAM instruction is never produced by the Prolog compiler.

After a sharing operation between two (or more) workers all but one simulate fail. This can be implemented as a jump to the `fail` label.

Figure 7.6 is the final iteration of the emulator for OR-parallel pure Prolog. This figure shows *all* Prolog engine modifications necessary to parallelize the pure Prolog system. Extra code, hidden in some macros, is also needed. There are five interface macros calling the parallel extension code: `Sch_Init()`, `Sch_Check()`, `Sch_Set_Load()`, `Sch_Get_Work()`, and `Sch_Get_Top()`. Not all `Sch_`-macros are pure scheduler code. Some of them need help from the engine to perform their tasks. Notice that the names of all scheduler macros are prefixed with **Sch_** and all engine macros are prefixed with **Eng_**, as a convention.

The macros used are slightly simplified versions of a set of macros defining an interface between the scheduler code and the engine code, described in Chapter 8.

The engine part of the initialization macro `Sch_Init()`, in Figure 7.7, is very system dependent. The main tasks are creating processes and shared memory, making the root choicepoint shared, and copying the state to all other workers. The main task for the scheduler code is to set the register top. This register is maintained (by the scheduler code) to refer to the youngest shared choicepoint.

Two trivial macros are the `Sch_Set_Load()` and the `Sch_Get_Top()` macros. They are shown in Figure 7.8. The first is used to keep the load register updated and the

```
  ...
  Sch_Init();
loop:
  switch (instruction according to pc) {
    case CALL:                                /* Modified */
      Sch_Check();
      ...
      if (next_cl is not null) {
        ch->lub = (parent(ch))->lub + (parent(ch))->alternative->ltt;
        Sch_Set_Load(ch->lub + ch->alternative->ltt);
        ...
      }
      ...
    shared_fail:                              /* Added */
      ch = Sch_Get_Top();
    fail:
    case FAIL: ...                            /* Unmodified */
    case GetWORK:                             /* Added */
      Sch_Get_Work(alt,goto shared_fail);
      pc = program counter according to alt;
      goto loop;
    ...
  }
```

Figure 7.6: The final iteration (3) of the WAM emulator.

```
top = undefined;

Sch_Init() {
  top = Eng_Init();
  Some other initializations.
}

Eng_Init() {
  Create shared memory;
  Create processes;
  Possibly lots of more initializations;
  Create ch and make it shared;
  Copy to all other workers;
  return ch;
}
```

Figure 7.7: The scheduler register top and the `Sch_Init()` macro.

second is used to provide the engine with the value of the current youngest shared choicepoint. Maintaining both the top and the load registers are scheduler tasks.

```
Sch_Set_Load(load) { load register = load; }
Sch_Get_Top()      { return top; }
```

Figure 7.8: The Sch_Set_Load() and the Sch_Get_Top() macros.

In the parallelization of pure Prolog the only request sent between workers is the sharing request. At every predicate call the request flag is checked. How a received request is treated is shown in Figure 7.9. The worker P can choose to either refuse or accept the worker Q's request. If the request is accepted then P makes its private choicepoints shared, updates shared choicepoints where Q is not a member, copies its state to Q, and sets the top.

```
Sch_Check() { if(any request for sharing) top = Eng_P_Share(); }

Eng_P_Share() {
  Q = worker who requested work;
  if (refuse sharing request) {
    refuse(Q);
    return top;
  }
  else {
    for (each choicepoint n of the private choicepoints) {
       Make n shared;
       Sch_Share_Node(n);
    }
    for (each choicepoint n of the shared choicepoints to update) {
       Sch_Update_Node(n,Q);
    }
    Copy to Q;
    Send (new) top to Q;
    return ch;
} }

Sch_Share_Node(n) { Fill in scheduler fields for n; }
Sch_Update_Node(n,Q) { Add Q to n; }
```

Figure 7.9: The Sch_Check() macro.

The Sch_Get_Work() macro is the the most central macro in the scheduler code. Its function is described in Sections 7.6 and 7.8.4. If the macro succeeds the register alt shown in Figure 7.6 contains a reference to the next alternative and if the macro fails the "failcode" goto shared_fail is executed. As usual the youngest choicepoint register ch is restored at shared_fail. Notice that a successful sharing for Q corresponds to failure.

7.6 The Scheduler

The main function of the pure Prolog scheduler is to match idle workers with available work. To avoid unnecessary backtracking the notion of a dead region is introduced in Section 7.6.1. The scheduler tasks can be divided into two parts: (1) finding shared work in the current branch on backtracking to a shared choicepoint and (2) a scheduling loop handling all other tasks. The first part is straightforward to implement and is described in Section 7.6.2. It is not so obvious how to implement the second part. There are several design choices and the different possible versions are more difficult to implement. The description is therefore divided into several parts. In Section 7.6.3 a correct but inefficient scheduler is described. An improved scheduler is described in Section 7.6.4 and further optimizations are described in Section 7.8.4.

7.6.1 The Dead Region

The notion of a dead region is introduced to avoid unnecessary backtracking. The dead region is the part of the branch (or the search tree) where no work can be found via backtracking, as shown in Figure 7.10. Without any knowledge of a dead region the worker has to backtrack to the root to find out that there is no more reachable shared work. To facilitate the implementation of the dead region the global frame associated with a shared choicepoint is extended to contain a pointer to the nearest older choicepoint that might have a job. When backtracking to a dead shared choicepoint the pointer is used to find jobs in older choicepoints. When the pointer refers to the root, no job can be found and the choicepoint is in the dead region. The pointer can be kept updated in a lazy fashion. Whenever a worker looks for a job it updates the pointer to the nearest choicepoint found containing work.

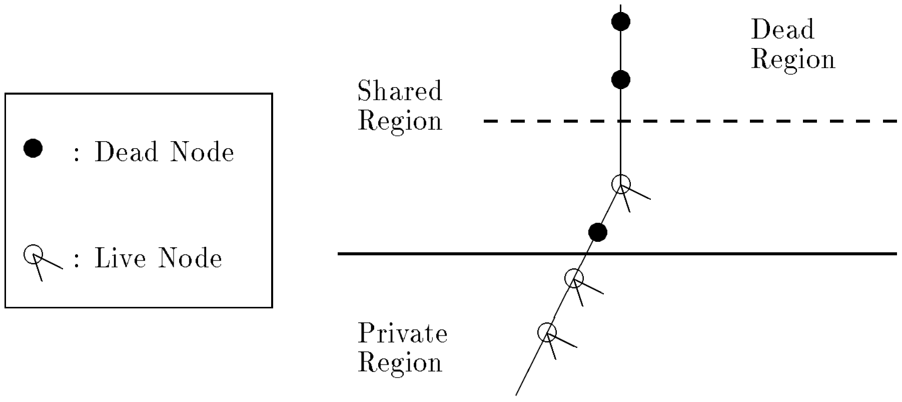


Figure 7.10: The dead region.

To implement the dead region efficiently the reference counter is replaced with three new fields in the global frame. The first field is a *member* set. This set contains the workers which can access the frame (i.e. workers that are at or below the node in the shared tree). To be able to calculate which workers are below the choicepoint a second set (called the *searching* set) is introduced. This set contains the workers that are (looking for jobs) at the choicepoint. The third field is a pointer to the nearest older choicepoint that might have work.

7.6.2 Finding Shared Work in the Current Branch

The first part, finding already shared work in the current branch, is common to all versions of the scheduler: basic, improved, and optimized. Whenever a worker dies back (backtracks) to a shared choicepoint the scheduler first tries to find work using these two steps:

- Try to find shared work in the current choicepoint.
- Try to find the nearest shared work in the older choicepoints (in the branch).

The two steps can be viewed as a shared region counterpart to ordinary Prolog backtracking. The first task, finding and taking work in the current choicepoint, is more or less a Prolog engine business. If there exists work (the choicepoint is not dead) then the engine takes the next available work (left to right). Some synchronization (enforced by the scheduler) is added though. The second task, finding work in older shared choicepoints, is performed thus: the scheduler, using the notion of a dead region (Section 7.6.1), finds the nearest choicepoint that might contain shared work and asks the engine if the choicepoint contains work. If so the scheduler moves to the choicepoint where the job was found and the engine backtracks until it reaches that choicepoint. Otherwise the scheduler tries again until it either finds work in any shared choicepoint or detects that no shared work can be found in the current branch.

7.6.3 A Very Basic Scheduler Loop

If the scheduler fails to find any shared work then the scheduler executes the scheduler loop. This section defines a very basic scheduler loop, repeating the following two steps:

- **Search:** Try to find a worker that is a member of the current shared choicepoint, having private load. If any worker is found, try to make it share some of its work.
- **Terminate:** Backtrack one choicepoint after N scheduler iterations.

This (very basic) scheduler loop terminates when either the worker has succeeded in getting some shared work from any worker below the current shared choicepoint or after a fixed number of scheduler iterations. If any worker has non zero private load the worker (called P) with maximum private load is asked to share some of its work. The request is either granted or refused. If the request is accepted then the worker moves down along P's branch. Termination of the execution is guaranteed as the worker, if it finds no work after N iterations, moves up one choicepoint.

The main problems with the basic scheduler are: (1) it is slow in finding workers above with private load, (2) the termination is slow, (3) it may miss the existence of shared work, and (4) it does not minimize movements in the shared search tree, The first two issues are treated in Section 7.6.4 and the other two in Section 7.8.4.

7.6.4 A Slightly Improved Scheduler Loop

The very basic scheduler loop described needs some improvements to become useful. The improved scheduler loop (1) looks both below and above for workers with private load and (2) improves the termination. Here is the outline of the scheduler loop repeating the following three steps:

- **Terminate:** If all work is exhausted at and below the shared choicepoint then move up.
- **Search Below:** For all workers below, try to find one with private load. If any worker is found, try to make it share some work.
- **Search Above:** For all workers above, try to find one with private load. If any worker is found, try to move to its branch.

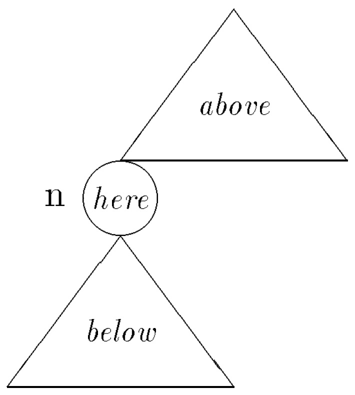


Figure 7.11: Here, above, and below.

The improved scheduler loop needs a notion of where the workers are. A worker can be in one of three positions relative to a shared choicepoint: *here*, *above*, and *below*. These concepts are defined in Figure 7.11. A worker P is defined to be *here* at a choicepoint n iff P is executing the scheduler loop at n. A worker P is defined to be *below* a choicepoint n iff n is a member of P's branch and P is not *here* at n. A worker P is defined to be *above* a choicepoint n iff it is not *below* n and not *here* at n. Now we are ready to explain the three different parts of the scheduler loop:

- **Terminate:** When there are no workers below, the worker positions its scheduler and backtracks its engine until it finds a busy worker in the member set.
- **Search Below:** The worker (called Q) tries to move down in the search tree, convincing some worker below to share its private work. The load is computed for all busy workers below, to find a suitable worker to ask to share work with. If any worker has non zero private load the worker (called P) with maximum private load is asked. The request is either granted or refused. If the request is accepted then a sharing operation is performed and Q moves down along P's branch.
- **Search Above:** The worker (called Q) tries to move up in the search tree, finding some worker above as a future candidate for a request to share its private work. The load is computed for all busy workers above, to find a suitable worker. If any worker has a non zero private load the worker (called P) with maximum private load is selected. The worker Q positions its scheduler and backtracks its engine to the first choicepoint where P is found as a member.

Now the most central scheduler macro `Sch_Get_Work()` shown in Figure 7.12 can be described. The macro is called by the engine when backtracking to a shared choicepoint as shown in Figure 7.6 (page 115). The macro can terminate with either success or failure. If the macro finds work in the shared choicepoint then the macro succeeds and a taken alternative is returned in the argument `alternative`. Otherwise the macro must terminate with failure, executing the code in the argument `FAILCODE` in one of the following ways: (1) move up for shared work on an older choicepoint in the current branch, (2) terminate when all work is exhausted in the current choicepoint and move up until finding a busy worker, (3) get shared work from a worker below, install new top, and backtrack for job, or (4) move up to the branch of some worker above with a load.

7.7 Extending to Full Prolog

So far we have covered pure Prolog only. Real Prolog implementations include cut and side effects. There are several design choices when implementing an OR-parallel full Prolog system. The main choice is between limiting parallelism and allowing speculative work. Speculative work can be defined as work that might not be carried out in a sequential execution [45]. The concept of speculative work can also be extended to include work that might need to be suspended. The current Muse system does not (by default) limit any parallelism. Thus the execution of side

```
Sch_Get_Work(alternative,FAILCODE) {
  if (no shared work found in the current choicepoint) { /* failure     */
    n = nearest choicepoint with work;
    if (n is not the root choicepoint)    /* search for shared work (1) */
      { top = move_up(to choicepoint n); FAILCODE; }

    while(true) {                             /* The scheduler loop     */
      if(any request for sharing) refuse request;
      if (no workers below) {                       /* Terminate    (2) */
        top = move_up(to a choicepoint with a busy worker);
        FAILCODE;
      }
      if (any busy worker below with excess load) { /* Search below (3) */
        P = worker with maximum load below;
        tmp = Eng_Q_Share(P);
        if (tmp) { top = tmp; FAILCODE; }
      }
      if (any busy worker above with excess load) { /* Search above (4) */
        P = worker with maximum load above;
        top = move_up(to a choicepoint with worker P);
        FAILCODE;
  } } }
  /* success */
  alternative = Eng_Take_One_Alternative(top);
}

Eng_Q_Share(P) {
  Request sharing from P and Wait for P to reply;
  if (the request is refused) return null;
  else { wait until P is ready; return new top; }
}

Eng_Take_One_Alternative(cp) {
  Update the next-alternative field in cp's shared frame;
  Return the old value of the next-alternative field;
}
```

Figure 7.12: The `Sch_Get_Work()` macro called from the emulator switch shown in Figure 7.6 (page 115).

effects might be suspended and the execution of cut operations might prune away branches containing work not carried out in a sequential execution.

The current Muse implementation of pure Prolog is very efficient, adding a negligible overhead to sequential programs and giving a very good speed up for parallel programs. In our opinion it is essential to maintain both properties (sequential speed and parallel speed up) even for the full Prolog parallelization. The only real measure of success is the absolute speed. It is also our opinion that the first goal (keeping sequential speed) is of overriding importance. This opinion is based on two facts: not all parts of all Prolog programs contain large grain parallelism, and it is only feasible to use a limited number of very high speed processors in a system. The most important goal must therefore be to minimize the overhead in sequential execution. The second (also very important) goal is to optimize parallel performance. The principle is even more important for the new generation [72, 87, 85, 88] of very efficient sequential Prolog systems. Then a very small absolute overhead may add a very high relative overhead.

In this section we add side effects (described in Section 7.7.3), cut (described in Section 7.7.4), and all-solutions predicates (described in Section 7.7.6), all requiring the leftmost check described in Section 7.7.1.

7.7.1 The Leftmost Check

The depth-first left to right search strategy in Prolog enforces a specific execution order of the side effects. This order has to be preserved in parallel (full) Prolog. The depth-first order is already maintained in each branch but the left to right order requires an extra mechanism in a parallel system: the *leftmost check*. A worker checks whether its branch is the leftmost one or not.

In the current Muse implementation a very simple representation of the search tree is used. There is a global *branch stack* associated with each worker. Each stack is a (simplified) representation of a worker's branch, showing the taken alternatives (in shared choicepoints) along this branch. Thus, for all members of a choicepoint, all taken alternatives can be calculated. For each shared choicepoint it is now easy to determine whether a worker's taken alternative corresponds to the leftmost branch or not. The leftmost check can now be performed as follows. Start at the youngest shared choicepoint (in the current branch) and traverse the choicepoints in the branch. If any choicepoint is found where the worker's taken alternative does not correspond to the leftmost branch then the leftmost check returns false. Otherwise it returns true.

To avoid traversing every choicepoint (in the leftmost check) an extra pointer field is added to each shared frame. This pointer (called *next*) is used to bypass choicepoints where there exist no branches to the left (of the current branch). The pointer is (lazily) updated to refer to the oldest choicepoint that cannot be bypassed. The Figure 7.13 shows a simplified implementation of the algorithm for the leftmost check using the bypass optimization. (Notice that branch numbers decrease from left to right as shown in Figure 7.2 (page 111).)

```
shared int branchstack[WORKERS][DEPTH];

/* The choicepoint scope is the youngest choicepoint where I do not
   care if I have taken the leftmost branch or not. */

leftmost(scope) {
  n = the youngest shared choicepoint in my branch;

  while (n is not older than the scope) {
    b = branchstack[my id][n];
    for (each worker w which is member of n)
      if (branchstack[w][n] > b) return NOT_LEFTMOST;
    n = n->next;
  }
  return LEFTMOST;
}
```

Figure 7.13: The leftmost check performed by a worker.

The code can be further optimized. In the code, all members are considered at each choicepoint. This means that the members of n also are considered at the parent choicepoint (`n->next`). As an optimization the members of n can be removed from the workers considered when examining `n->next`.

7.7.2 Internal (not order sensitive) Side effects

The Prolog system maintains global data. Some examples are loaded code, asserted code, the atom table, and the table of open file descriptors. Not all global data are visible to the user of the Prolog system. For invisible data the sequential order does not have to be preserved. One example is the atom table (if we disregard the peculiar built-in `current_atom/1` predicate). The synchronization of such side effects may be implemented using a simple scheme, accessing the table inside a locked region. It is even better if a read/write-lock scheme is used, allowing multiple readers but forcing any writer to be the sole accessor. An optimization for SICStus is described in Section 7.11.4.

7.7.3 Side effects

The current implementation of the side effect synchronization is very simple. As soon as an engine wants to execute a side effect it waits until its branch becomes the leftmost (Section 7.7.1) one in the search tree before continuing. If the engine, while waiting, receives a pruning request (Section 7.7.4) the side effect is never executed. This ensures that the same side effects as in a sequential execution are executed, and in the same order. Workers waiting to become leftmost are not efficiently utilized. Three remedies for this inefficiency are presented in Sections 7.8.5, 7.15.1, and 7.16.1. The macro `Sch_Synch()`, shown in Figure 7.14 is a very simplified version of the macro found in Chapter 8.

```
Sch_Synch(scope,FAILCODE)
  { while (!leftmost(scope)) { if (pruned()) FAILCODE; } }
```

Figure 7.14: The `Sch_Synch()` macro.

7.7.4 Cut

So far the main design principle has been: *keep it simple.* The most complicated parts have been incremental copying, the scheduler loop, and synchronization of side effects. Each new function introduced has been treated as a new layer not (seriously) affecting earlier layers introduced. There exists one more function to implement before we are done with full Prolog: cut. The purpose of this operation is to change the Prolog search order by pruning alternatives from the choicepoint stack. In SICStus the cut operation simply removes some choicepoints from the choicepoint stack, by changing the youngest-choicepoint pointer.

In an OR-parallel system complications arise when the choicepoints reside in the shared part: (1) the choicepoints cannot always be removed; (2) other workers may be affected by the cut operation; (3) the cut operation cannot always be completed; (4) other activities, even other cut operations, may be aborted.

The code showed in Figures 7.15, 7.16, and 7.17 is a very simplified description of the cut algorithm. The code lacks some optimizations. Figure 7.15 shows the part of the code that is executed when a cut operation is encountered. The macro `Sch_Prune()` is a very simplified version of the macro found in Chapter 8.

At execution of the `Sch_Prune()` macro three situations can occur: (1) the cut operation is local, (2) a non-local cut operation is completely performed, and (3) a non-local cut operation cannot be completely performed. Those three case are now described.

1. If only private choicepoints are affected by the cut operation, nothing except execution of the normal cut operation is done.

2. Otherwise, if the current branch is the leftmost branch within the subtree rooted at the "scope" choicepoint, then the pruning operation is fully performed by the function `prune_in_node()` and the macro returns the original scope. The function `prune_in_node()` prunes branches in the tree and sends pruning requests to workers on pruned branches.

```
  switch(instruction according to pc) { /* The emulator switch. */
    ...
    case CUT:
      scope = scope saved in a WAM register;
      scope = Sch_Prune(scope,goto shared_fail);         /* New */
      NormalCut(scope);
      increment pc;
      goto loop;
    ...
  }

Sch_Prune(scope,FAILCODE) {
  if (scope is older than top) {
    n = nearest where not leftmost;
    if (prune request) if (pruned()) FAILCODE;
    if (n is younger than scope) {
      n->cut_alternative = my alternative in n;
      n->cut_scope = scope;
      n->cut_token = any worker left of me in n;
      scope = n;
    }
    else n = child(scope);
    prune_in_node(n,my alternative in n);
  }
  return scope;
}
```

Figure 7.15: The `Sch_Prune()` macro.

3. Otherwise, some cut information is stored in the youngest choicepoint where the current branch is *not* the leftmost branch[1]. The cut information contains a reference to the current branch (the alternative taken in the choicepoint), the scope, and a token containing the name of a worker that has taken an alternative left of my taken alternative. (This worker may later on continue the cut operation.) After the cut information is stored the worker performs the cut operation only partially, using the function `prune_in_node()`, and then returns the choicepoint where the cut information was saved. This scope is then used by the normal cut operation.

Whenever a worker dies back (backtracks) to a shared choicepoint where cut information is stored, the continue pending cut operation, described in Figure 7.16, is executed. If there still exist alternatives to the left of the original cut branch then the token is transferred to some other worker that can continue the operation. Otherwise the worker (choicepoint by choicepoint) tries to kill all work to the right of the current branch. If the scope choicepoint is reached, then the cut operation is fully performed. Otherwise, if there exists in any visited choicepoint some branch

[1]As an optimization, saving of the cut information may be omitted when there exists no more work to prune above the choicepoint.

to the left, then the cut information must be placed in that choicepoint and the operation becomes pending again.

```
  switch(instruction according to pc) { /* The emulator switch. */
    ...
    case GetWORK:
      Sch_Continue_Prune();
      ...
    ...
  }

Sch_Continue_Prune() {
  if (top->cut_token == me) {
    top->cut_token = any worker left of top->cut_alternative;
    if (!top->cut_token) {
      s = top->cut_scope;
      top = move_up(one choicepoint);
      while (top is younger than s) {
        if (prune request) if (pruned()) return;
        prune_in_node(top,top->my_alternative-1);
        p = any worker left of me in top;
        if (p) {
          top->cut_alternative = my alternative in top;
          top->cut_scope = s;
          top->cut_token = p;
          return;
        }
        top = move_up(one choicepoint);
} } } }
```

Figure 7.16: The `Sch_Continue_Prune()` macro is added to the `GetWORK` code.

The last piece of the code in Figure 7.17 shows where the pruning request (produced by `prune_in_node()`) is received. A legal pruning request will override any sharing request received in the `Sch_Check()` macro, perform some local cleaning up of dead shared choicepoints, and then simulate failure.

```
Sch_Check() {
  if (any request) {
    if(any request for pruning) if(pruned()) goto shared_fail;
    ...
} }
```

Figure 7.17: Adding receiving of a cut request to the `Sch_Check()` macro shown in Figure 7.9 (page 116).

7.7.5 Sequential Predicates

Although the described synchronization (Section 7.7.3) and cut (Section 7.7.4) ensure that the correct Prolog semantics is maintained, it may sometimes be advantageous to declare a predicate as sequential to force the scheduler to traverse the shared search tree in a more left to right fashion. This can be useful when the programmer knows that it is best to finish one OR-alternative before starting the next one.

To implement sequential predicates, the ltt field (described in Section 7.5.1) stating how many alternatives are untried is set to 0. When making a choicepoint associated with a sequential predicate shared, the code `GetWORKsequential` is inserted in the global frame. This variant of the `GetWORK` code ensures that the alternatives are taken left to right one at a time.

Two examples where a sequential predicate is preferred are shown in Figures 7.18 and 7.19. In the first example the system is forced to execute the iterative deepening one level at a time. The sequential declaration in the second example avoids losing one worker that otherwise might become busy waiting at the `write/1` call. This problem can also be solved by rescheduling of the worker responsible for the suspended task, as described in Section 7.15.1, or the method described in Section 7.16.1.

```
search(Solution) :- iterate(1,Solution), !.

:- sequential iterate/2.
iterate(Depth,Solution) :- compute(Depth,Solution).
iterate(Depth,Solution) :- D is Depth+1, iterate(D,Solution).
```

Figure 7.18: Avoiding speculative work in iterative deepening programs.

```
:- sequential program/2.
program(Solution) :- compute(Solution), !.
program(_) :- write(error), nl.
```

Figure 7.19: Avoiding dummy branches.

Whenever a predicate has been declared sequential both Muse and Aurora [65] also treat the choicepoints created by its disjunctions as sequential choicepoints.

7.7.6 All-solutions Predicates

All-solution predicates (e.g. `findall/3`, `bagof/3`, `setof/3`) can be implemented (in the Prolog system) using various techniques. I will cover the parallelization of the SICStus implementation only. The parallel version of findall can be found in Figure 7.20. It is essentially the same parallelization as that found in [25].

A findall root, a reference to the save area, is allocated in the global memory space. All solutions are generated by the `save_instances/3` predicate, and saved in the global memory space, when the worker generating a solution is leftmost within the

```
findall(Template, Generator, List) :-
    '$allocate'(Root),
    save_instances(Template, Generator, Root),
    list_instances(Root, L),
    '$deallocate'(Root),
    L=List.

:- sequential save_instances/3.
save_instances(Template, Generator, Root) :-
    '$current_scope'(Chpt),
    call(Generator),
    '$save_solution'(Chpt, Template, Root),
    fail.
save_instances(_,_,_).

:- sequential list_instances/2.
list_instances(Root, List) :-
    '$fetch_and_remove_oldest_solution'(Root, Term), !,
    List=[Term|L],
    list_instances(Root, L).
list_instances(_, []).
```

Figure 7.20: Parallel implementation of findall/3.

findall scope. When all solutions have been generated and saved they are collected in a list, in the binding environment of the executing worker, by the predicate `list_instances/3`. Finally the findall root is deallocated.

The execution of findall can be terminated from outside, either by an abort request or by a pruning (Section 7.7.4) request. To facilitate the deallocation of the root and saved solutions, all allocated roots form a linked list.

Workers waiting to become leftmost reduce the parallelism. One very attractive remedy for this inefficiency is presented in Section 7.8.5.

7.8 Optimizations

It is very important to make the sharing session efficient. Two techniques for achieving this objective are described in Sections 7.8.1 and 7.8.2.

Some programs contain phases where the search tree consists of a multitude of very small tasks. The granularity of forward execution work is then very small. How to avoid the sharing of work in such situations is described in Section 7.8.3.

Scheduling tasks is the topic that invites the largest amount of experimentation. You can spend nearly unlimited time on refining the scheduler. But the slightly improved basic scheduler already described is good. In Section 7.8.4 the layout of the optimized scheduler loop used in the current Muse system is presented.

An important and very attractive optimization for assert, write, and all-solutions predicates is presented in Section 7.8.5.

7.8.1 Copying without Using a Buffer

It is desirable to copy directly from one worker's stack area to another one's, avoiding copying via a buffer, and thereby minimizing the copying time. One way of implementing this feature is to make each worker map the stack areas of all other workers into its virtual memory space. This approach limits the usable virtual memory area for the workers' own stacks. This becomes a problem for systems supporting a huge number of processors (e.g. one hundred), such as the Butterfly machines. A (for our purposes) minor problem is that there are limitations on the allocation of shared memory in the DYNIX operating system. Section 7.12.2 shows how this problem is solved and Section 7.16.3 indicates an even better solution.

The operating system Mach can provide virtual memory copying methods. The actual contents of physical pages is then copied on demand only. We have not found any programs so far that require copying of large blocks of memory, comparable to the page size. Thus we believe this technique may not be useful for most programs.

7.8.2 Parallelizing the Sharing Session

There are two workers involved in a sharing session. Let us, as usual, call them P and Q. The worker Q is the one that wants to get shared work from P. It is obvious that the two workers can divide the work among themselves. But to make the choices concerning the division in an optimal way is somewhat tricky. There are several aspects to consider.

The worker P was disturbed while busy performing (hopefully) productive work. It is very important that P can continue as quickly as possible. Otherwise a situation may arise where a set of non busy workers seriously hamper the busy workers by making them spend most of their time not executing Prolog code.

To make the trade-off which worker should copy what areas is very difficult but fortunately we have found that an optimal choice is not necessary. Some tasks in a sharing session are interdependent. The choicepoint stack cannot (easily) be copied before the choicepoints have been made shared, the installation from the trail cannot be made before the term stack and environment stack are copied, etc. It is faster to copy a memory area as one contiguous block, but if the block is big it may be a good choice to divide the task.

Parallelizing the sharing session also requires a more complex synchronization. The synchronization needed is best explained by an example. I therefore describe the sharing session for SICStus Muse on shared memory machines. The worker P is initially executing Prolog code and the worker Q detects that P may have some untaken job to share. At the end of the session both are busy executing Prolog.

In the first phase Q and P tries to find out if it is a good choice for P to share some work with Q. First Q makes some tests to determine whether to send a request to P. If the tests give a positive answer then Q sends the request and waits for the reply. The worker P eventually detects the request and makes some tests to decide whether to accept Q's request or not. If P decides to refuses Q's request then it tells Q so and the sharing session is over.

If the worker P accepts the sharing request the sharing session enters the second phase. The work is divided as follows: (1) the worker P decides what to do, makes the private choicepoints shared, updates some shared choicepoints, and maybe helps Q copy the term stack; (2) the worker Q copies the environment stack, copies the trail, copies the term stack, copies the choicepoint stack, and makes the installations of newly bound variables in the environment and the term stack according to the trail.

There are some synchronization points, shown as a dependency graph in Figure 7.21. The worker P must not start executing Prolog before the term stack, the environment stack, and the (private part of the) trail are copied. It also must not backtrack into the shared region before Q is completely done with the sharing session. The worker Q must not start any copying before P has decided what to do. It also must not copy the choicepoint stack before P is done with the sharing session. If P is helping Q to copy the term stack then Q must not do any installation before P is done with the copying.

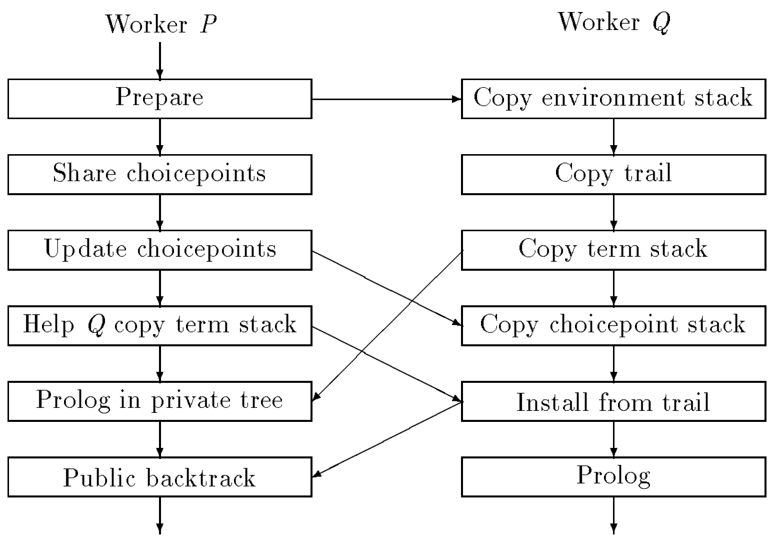


Figure 7.21: Phase 2 of the example parallel sharing session.

7.8.3 Delayed Release of Load

Some programs contain phases where the search tree consists of a multitude of very short tasks. The granularity of forward execution work is then very small. Avoiding sharing small pieces of work is necessary in order not to degrade performance.

A simple method for granularity control (used in the Aurora [65] system) consists in increasing the interval between polling for incoming requests. This approach has two drawbacks: (1) the overhead for incrementing a counter at each procedure call and (2) the performance lost by making the detection of the request slower. Moreover, the method does not remove the problem, it just reduces it.

The current implementation of Muse uses another method. This method eliminates the problem of short branches for many programs. It also does not add the constant overhead at every procedure call. The main drawback is that programs that normally show a large parallel slow down due to short branches now show a small sequential slow down. The parallel performance is substantially enhanced though.

We call the method *delayed release*. The idea is to delay the release of the load (to the global load register) for a number of predicate calls, when a worker changes its private load (Section 7.4.2) from zero to nonzero. A busy worker with small tasks only is then never going to show any private load to the other workers.

The implementation shown in Figure 7.22 uses the request sending facility of Muse. Whenever a choicepoint is created the load register is *not* updated if the previous private load was zero. Instead a counter is initialized to a suitable value and the worker sends a *release* request to itself. When the worker receives the request (at the next predicate call) it decrements the counter. If the counter becomes zero then the private load is released. Otherwise the worker sends a new request to itself. The chosen counter value for the current Muse system on a Sequent Symmetry is 5 (procedure calls).

7.8.4 Refining the Scheduler Loop

The scheduler loop optimizations mainly concern three topics: (1) making the "nearest" idle worker share work with a busy worker, (2) finding shared load in other branches, and (3) distributing workers over the tree. The topics are discussed in detail in Chapter 4. I have chosen to just give the outline of the scheduler loop used in the current Muse scheduler.

We have found that it is hard to make significant improvements over the previously described scheduler loop but two optimizations are important. The first one is finding shared work in other branches. The other one is the concept of nearness, which is useful for large systems.

Here is the outline of the scheduler loop repeating the following steps:

```
Sch_Set_Load(load) {
  if (previous load is zero) {
     count register = 5;
     send request for delayed release to myself;
  }
  else load register = load;
}

Sch_Check() {
  if (any request) {
    if(any request for delayed release)
      if (count register-- == 0) {
        load register = compute_load();
        clear request for delayed release;
      }
    ...
} }
```

Figure 7.22: Extensions of the `Sch_Check()` macro (Figure 7.8 (page 116)) and the `Sch_Set_Load()` macro (Figure 7.9 (page 116)) to implement delayed release.

- **Terminate:** If all work is exhausted at and below the current choicepoint then move up to the nearest choicepoint where work may be generated. New work may be generated at either a sequential choicepoint or at a choicepoint with any busy worker.
- **Search Below:** For all workers below me to whom I am the nearest worker, try to find the one with the maximum private load. If any worker is found, try to make it share some work with me.
- **Search Above:** For all workers above me to whom I am the nearest worker, try to find "the best" one having private load. (We have used several criteria for choosing "the best" worker: the nearest one, the one with maximum load, etc.) If any worker is found, try to move to its branch.
- **Find a Better Position:** When there exists a busy worker with no idle worker in its branch and I am the nearest idle worker (to that busy worker) then try to move to the worker's branch (to be in the correct position when work becomes available). Notice that moving up may entail having to reconstruct parts of the former state, so it is best to do it slowly one choicepoint at a time.
- **Search for Shared Work:** At each N:th loop do: For all workers below me to whom I am the nearest worker, try to find one having shared load. If any worker is found, try to make it share some work with me.

Notice that workers moving upwards have to take sequential choicepoints into consideration. The last worker backtracking from a sequential choicepoint *must* stop and take care of the work found in that choicepoint.

Figure 7.23 is included to give a hint about how the scheduler loop in the unoptimized `Sch_Get_Work()` macro shown in Figure 7.12 (page 121) looks when the optimizations are added. The part of the code used for finding a better position needs some explanation. The code computes the set `b` containing the busy workers that need an idle worker in their branch and to whom I am the nearest worker.

7.8.5 Speculative Write Type Side effects

A novel method for optimizing write type side effects, such as write and assert predicates, is introduced in this section. It makes it possible to implement efficiently the findall construct, which is very important in Prolog, while preserving the same order as in a sequential Prolog system, in contrast to the Aurora implementation, where the findall construct does not preserve the order, for efficiency reasons [25], which makes the semantics different in parallel and sequential Prolog systems. Some examples relying on the order of the solutions are presented in Section 7.9.

The method is based on the following observation: write type side effects can be (speculatively) saved for (possible) later execution. We can use the fact that write type side effects do not alter the binding environment. The engine can save the side effect and continue its execution, as if the side effect was executed.

One solution is to save the side effect in the nearest shared choicepoint where the branch is not leftmost. Worker W2 in Figure 7.24 is working in a branch which is not the leftmost one, so it saves the side effect in the choicepoint n1. The side effect can then later on be executed when the branch b2 becomes leftmost.

The chosen data structure, allocated in shared memory space, is shown in Figure 7.25. Each global frame (the shared choicepoint extension) contains an extra list pointer. This pointer is the start of a null terminated list. Each element in the list corresponds to an alternative from which delayed side effects have been saved. The list is sorted leftmost alternative first. (Notice that the alternative number is decrease from left to right as discussed in Section 7.5.1.) Each element of the list contains an alternative number, a sublist pointer, and an end-of-sublist pointer. The sublists are non empty null terminated lists, containing the saved side effects in chronological order. Here is an algorithm:

1. A leftmost worker performs the side effect immediately.
2. A non-leftmost worker saves the side effect in the youngest shared choicepoint where the worker's branch is not leftmost. The saved side effect is to be associated with the current branch.
3. Whenever a worker dies back to a choicepoint, it tries to execute, applying rules 1 and 2, all non-dead saved side effects that are associated with branches now leftmost in the choicepoint. All side effects thus dealt with are removed (moved to an older choicepoint or executed).

```
Sch_Get_Work(alternative,FAILCODE) {
    ...
    cnt = 1;
    while(true) {                               /* The scheduler loop */
      if(any request for sharing) refuse request;
      if (no workers below) {                         /* Terminate    */
        top = move_up(to a choicepoint with a busy worker
                      or a sequential choicepoint);
        FAILCODE;
      }
      b = the set of near busy workers below;         /* Search below */
      if (any worker in b with excess load) {
        P = worker with maximum load in b;
        tmp = Eng_Q_Share(P);
        if (tmp) { top = tmp; FAILCODE; }
      }
      b = the set of near busy workers above;         /* Search above */
      if (any worker in b with excess load) {
        P = worker with maximum load in b;
        top = move_up(to a choicepoint with worker P);
        FAILCODE;
      }
      b = the set of busy workers above;   /* Find a better position */
      i = the set of idle workers above;
      for (all workers w in i) {
        if (b is the empty set) break;
        if (I am below w) b = the empty set;
        else remove all workers below w from b;
      }
      if (any worker in b) {
        top = move_up(one choicepoint);
        FAILCODE;
      }
      if (cnt++ == N) {                    /* Search for shared work */
        b = the set of workers below that may have shared work;
        if (b is not empty) {
          P = any worker in b;
          tmp = Eng_Q_Share(P);
          if (tmp) { top = tmp; FAILCODE; }
        }
        cnt = 1;
    } }
  ...
}
```

Figure 7.23: The scheduler loop in the optimized Sch_Get_Work() macro. The vanilla macro was shown in Figure 7.12 (page 121).

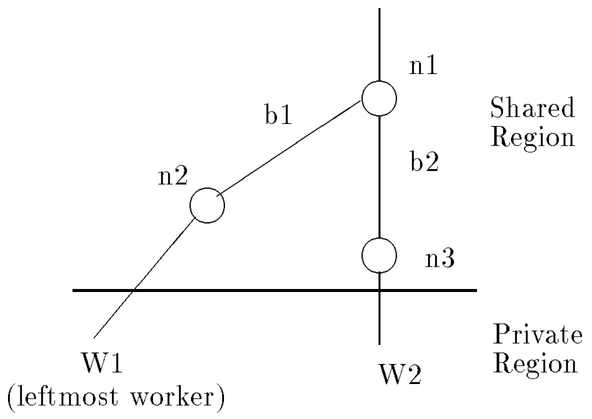


Figure 7.24: The speculative write approach.

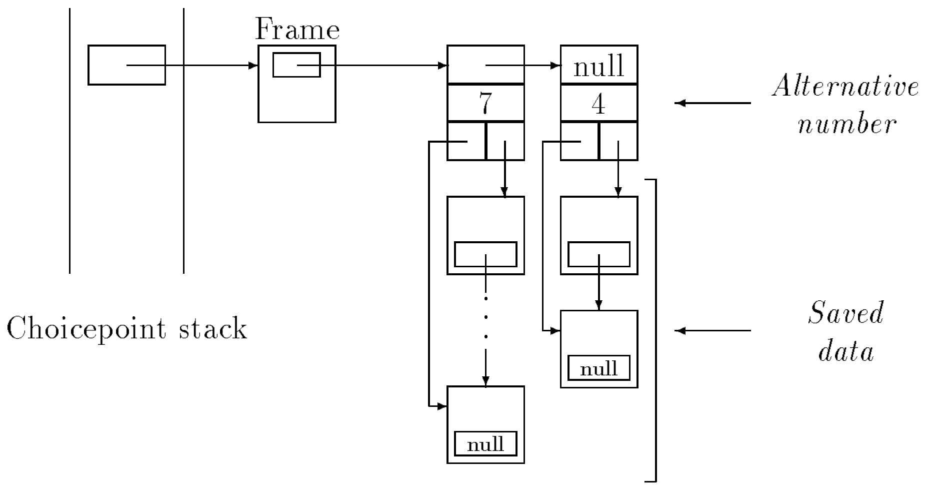


Figure 7.25: Data structures for speculative write.

4. Whenever a choicepoint is marked as dead as a result of a pruning operation, all saved side effects to the right of the current branch are removed (marked as dead or deallocated).

5. Whenever a worker receives a (legal) pruning request when trying to execute a side effect, the attempt is aborted.

6. All saved side effects are deallocated when the choicepoint is deallocated.

Information about type and scope is also associated with the saved side effects. The side effects are of different types (e.g. write, assert, and findall). The scope within which the branch must be the leftmost one can also vary (e.g. in the case of findall).

Notice that saving a side effect in a shared choicepoint usually does not require any substantial overhead in comparison with the sequential implementation, neither in time consumption nor in memory space. It is just a matter of creating a header and updating pointers. Moving a saved side effect, or even a block of saved side effects, is just moving pointers from one choicepoint to another. (The write side effect may need some intermediate format when saved in the tree though.) Very promising results have been obtained. Table 7.1 makes a comparison for findall programs between Muse using speculative write and preserving the sequential order and Aurora without preserving the order. Times shown in the table are measured in seconds. When collecting all solutions, using the unoptimized strict findall (not shown in the table), the speed up is 2.3 for the 8 queens case. The results shown in the table indicate that we can achieve a good performance without losing the sequential semantics for findall, using the speculative approach presented in this section.

Queens	No. of elements in the list	Aurora (Bristol) *Free findall*		Muse *Strict findall*	
		1W	10W	1W	10W
7	40	1.42	0.19 (7.47)	1.05	0.14 (7.50)
8	92	5.32	0.61 (8.72)	3.95	0.47 (8.40)
9	352	22.35	2.41 (9.27)	16.41	1.81 (9.07)
10	724	91.66	9.50 (9.65)	67.07	7.07 (9.49)
11	2680	412.05	42.87 (9.61)	305.83	31.77 (9.63)

Table 7.1: Comparison between strict and free findall implementations.

7.9 Independent AND- in OR-parallelism

This thesis concerns mainly executing the different alternatives (clauses) in a predicate in parallel (OR-parallelism). Another important source for parallelism is called *independent* AND-parallelism [32, 50], where goals in a clause body are independent and can be executed in parallel.

This section describes how an OR-parallel system can exploit independent AND-parallelism by using the built-in predicate `findall/3`. The topic has already been treated in [25]. Our improved implementation of findall in Section 7.8.5, in conjunction with some optimizations described in [53] has made this approach suitable for some programs. We have used the same benchmarks as in a system specially built to exploit independent AND-parallelism, called &-Prolog [49]. The resulting AND-parallel system is not as good as the dedicated system but the results are encouraging.

A very simple example showing the principle of the parallelization is shown in Figure 7.26. The original (sequential) program `p1(In,Out)` generates a new list `Out` applying "some kind of task" to all elements of the list `In`. The second (parallel) program `p2(In,Out)` uses findall to apply (in parallel) "some kind of task" to all elements of the list `In` and collect the solutions in the list `Out`.

```
%%% The normal program.
p1([],[]).
p1([X|In],[Y|Out]) :- some_kind_of_task(X,Y), p1(In,Out).

%%% A (very) naive parallelization.
p2(In,Out) :- findall(S,p2a(In,S),Out).

p2a([H|_],S) :- some_kind_of_task(H,S).
p2a([_|T],S) :- p2a(T,S).
```

Figure 7.26: Small example showing AND in OR.

This simple implementation of independent AND shows one example that requires the sequential semantics of the findall construct.

7.10 Adding Non-Prolog Features

The main purpose in parallelizing a Prolog system is to enhance its performance. The user is to get the impression that he is using a faster Prolog system. But the semantics of Prolog may, for some problems, introduce unnecessary constraints. Finding just any solution to a given query may be sufficient, the ordering of the side effects may be unimportant etc. When partially removing the constraints we are faced with some design problems. What syntax and semantics shall we use? Should special non-Prolog predicates (e.g. asynch_write/1) or a global annotation of goals (e.g. asynch(program)) or a lower level of parallelization based on an atomic exchange and destructive assignment to global data be used? Should the system implement everything that anyone can ever imagine? Should it solve the problems of *cold fusion*? Sorry, I got somewhat carried away.

I think that the main advice is: *keep it simple*. Prolog is a simple and elegant programming language and the OR-parallel version of Prolog using the Muse model is also simple. Another advice is: *keep it visible*. It may be regarded as elegant to use global annotation changing the semantics of e.g. side effects or cut, but it is nearly impossible to understand.

A reasonable set of non-Prolog extensions includes asynchronous I/O predicates (with constraints on usage), a cavalier oneof predicate (based on cavalier commit), global variables, and a simple mutex call. More complicated to implement, but also useful, are commit (symmetric cut) and a more complex mutex call.

General asynchronous I/O and assert/retract are hard to implement and of questionable value for the programmer. It is also hard to define a usable syntax and semantics. It is hard to figure out the meaning of asynchronous buffered read and asynchronous assert using the *logical* database view [62]. Removing I/O buffering and switching to the *immediate* database view makes the implementation less efficient, possible removing the advantage of the parallelization.

7.10.1 Mutex

Some operations in a parallel system need to be atomic. One example is incrementing a global counter. For that purpose a `mutex/1` (mutual exclusion) metacall is useful. This call can be implemented in several ways. Figure 7.27 show three possible implementations. They are ordered by increasing functionality and inefficiency. All three calls demand that the worker shall ignore all requests for sharing and pruning when executing the goal. No synchronization is allowed in the goal. So the system *must* ignore synchronization of side effects.

```
mutex1(G) :- lock, call(G), unlock.

:- sequential mutex2/1.
mutex2(G) :- lock, call(G), !, unlock.
mutex2(_) :- unlock, fail.

:- sequential mutex3/1.
mutex3(G) :- ( lock ; undo(unlock),fail ),
              call(G),
             ( unlock ; undo(lock),fail ).
```

Figure 7.27: Three versions of mutex/1.

The first call is very efficient but the goal must have exactly one solution. Otherwise the program may crash the system. The second implementation is slightly less efficient (it creates a choicepoint) but it cannot crash the system. The solutions to the goal are limited to the first one, via cut. The last implementation is very expensive (and requires the `undo/1` metacall described in Section 7.11.1 to be implemented) but it can be used as a general metacall.

One global lock is added (used by the `lock/0` and `unlock/0` calls above) to implement mutual exclusion. The worker ignores all requests when it has acquired the lock.

7.10.2 Asynchronous I/O

Implementing general I/O in a parallel UNIX environment is rather tricky, as described in Section 7.16.5. The following is only applicable to terminal I/O if general I/O is not implemented.

I propose that the first implementation include asynchronous I/O for a limited set of I/O functionality only: input buffering shall be inhibited, output shall be flushed, and sequences of related I/O operations shall be made atomically. As a matter of fact, turning off input buffering and using the metacall `mutex/1` introduced in Section 7.10.1 is enough. The call

```
mutex((write(X),ttyflush))
```

writes and flushes the term X atomically without synchronization and the call

```
mutex((write(:),read(X),write(X),ttyflush))
```

performs the sequence $\langle write, read, write \rangle$ atomically and without synchronization (In SICStus a read from the terminal flushes the terminal output).

7.10.3 Cavalier Oneof

Sometimes you want to relax the sequential semantics of the cut operation to seek any solution to a goal instead of the leftmost one. This operation is called *commit* (or symmetrical cut) and can (using an informal description) be implemented as follows in an OR-parallel system: kill all other work rooted at the commit scope choicepoint.

As discussed in [45] it is necessary to impose some restrictions on the commit operation to make the semantics of the operation defensible. A worker (executing speculative work) is not supposed to kill branches that should have survived if no speculative work were permitted. The choice to execute speculative work is an implementation issue that should be invisible to the user.

It is not easy to know what work is speculative. This issue has been thoroughly penetrated in [45] and is further discussed in Section 7.15.2.

It is very easy to implement a totally unrestricted commit (called *cavalier commit* [22]). In the current Muse implementation we have chosen not to implement a special cavalier commit operator. Instead we have implemented a special built-in metacall predicate called `cavalier_oneof(Goal)`. The predicate finds any solution to the goal, even if the solution is speculative. The programmer using this predicate must be aware of the risk and only use it when solutions to the goal are known not to be in speculative branches. This predicate can be useful to a programmer that knows the limitations but it is our belief that the effects of using a cavalier commit operator are almost impossible to comprehend.

7.10.4 Global Variables

Sometimes a program may need non-backtrackable data for communication between OR-branches, e.g. a global register containing the maximum value found so far. In Prolog the database predicates assert/retract provide a very flexible (and inefficient) way to implement global variables. This method can be used in a parallel environment also if we preserve (via synchronization) the sequential semantics. But sometimes the updating of the global variable is not order sensitive, as in the case of the maximum value previously mentioned.

Unfortunately the implementation of assert/retract is not trivial to parallelize, as described in Section 7.15.4. There are also some problems in defining the semantics for an asynchronous parallel assert/retract.

One easy solution suitable for parallel programming (and also sequential programming) is to introduce global variables. The variables are allocated and changed using built-in predicates. The variable can be, for ease of implementation, typed (e.g. int, float, etc) at both allocation and usage. Some Prolog systems support untyped global variables. Some complex operations (e.g. atomic exchange) or a mutex meta-call are also needed. The following example is taken from "ProLog by BIM" [18], a commercial Prolog now being parallelized using the Muse model.

Figure 7.28 is an example of a higher level built-in predicate using the basic built-in predicates. The predicate `findsum/3` computes the sum of all solutions to a goal. The calls `allocate_unique_identifier(S)` and `record(S,0)` allocates a global variable with the name S and the initial value 0. Each new value computed is added to the global variable inside the `mutex/1` call. The call `recorded/2` reads the old value and the call `rerecord/2` writes the new value.

```
findsum(X,Goal,Sum) :-
    get_unique_identifier(S),
    record(S,0),
    findsum_internal(X,Goal,Tmp,S),
    Sum=Tmp.

:- sequential findsum_internal/4.
findsum_internal(X,Goal,_,S) :-
    call(Goal),
    mutex(( recorded(S,X1), X2 is X1 + X, rerecord(S,X2) )),
    fail.
findsum_internal(_,_,Sum,S) :-
    recorded(S,Sum),
    erase(S).

% Example of usage
prog(Sum) :- findsum(X,generate(X),Sum).
```

Figure 7.28: Program that computes a maximum value using a global variable.

It may be difficult to foresee all necessary higher level predicates (e.g. `findsum/3`) needed, so it is prudent of the system programmer to provide the lower level predicates for the user. A set of higher level predicates is to be provided as library routines, to use directly or to use as a model for similar constructs.

7.11 SICStus Specifics

This section presents some SICStus Prolog [26] specific topics. First three non standard Prolog constructs (`undo/1` in Section 7.11.1, `setarg/3` in Section 7.11.2, and `if/3` in Section 7.11.3) are discussed. All three demand special treatment in an OR-parallel environment. Then an algorithm for minimizing lock collisions when using the SICStus hash tables is presented in Section 7.11.4.

7.11.1 Undo

The SICStus built-in predicate `undo(Goal)` is used to execute the goal "`Goal,fail`" on backtracking. The predicate saves the goal on the term stack and saves a reference to the saved goal on the trail. When the Prolog system traverses the trail to perform unbinding on backtracking it also checks for *undo references*. When such a reference is found, the goal is executed.

In a parallel system the same trail segment may be traversed several times for segments belonging to shared choicepoints. This is done each time a worker moves up along a shared branch. The current Muse implementation supports both the seemingly useless (but needed for implementing `setarg/3` in Section 7.11.2) variant of undo that may execute the undo goal several times (called `multi_undo/1`) and a correct version that preserves the sequential semantics.

The implementation of the undo with sequential semantics is based on allocating (in global memory) an *undo frame* for each undo goal. This frame has a reference counter, indicating how many workers refer to the undo frame. When a backtracking worker decrements the counter to zero the undo goal is executed and the undo frame is deallocated.

7.11.2 Setarg

The SICStus built-in predicate `setarg(N,Struct,Term)` is used to replace the contents of the `N`:th argument of the structure `Struct` with the term `Term`. The old value is restored on backtracking.

The backtrackable predicate `setarg` is implemented using a destructive assignment version, `SETARG(N,Struct,Term)`, and using `multi_undo/1` as shown in Figure 7.29. (The real SICStus and Muse implementations implements the predicate `setarg` as one built-in predicate, implemented in C for efficiency.) In calling `setarg` an undo goal resetting the argument `N` of `Struct` to the old value using `SETARG` is stored on the term stack before changing argument `N` of `Struct` to the new value using `SETARG`.

Notice that the predicate `SETARG`, is not always possible to use in an OR-parallel system since it destructively changes the binding environment. Those changes are not stored on the trail and not restored on backtracking thus violating the basic principles of both the Muse model and the incremental copying method. But the destructive assignment call `SETARG` can be used to implement the backtrackable assignment call `setarg`. At the call to `setarg` an extra dummy reference (pointing

```
setarg(N,Struct,New) :-
   arg(N,Struct,Old),               % Get the old value.
   multi_undo(SETARG(N,Struct,Old)), % Prepare for restoring the
                                    %   old value on backtracking.
   'trail the reference'(Struct,N),  % Save a dummy reference for
                                    %   incremental copying and GC.
   SETARG(N,Struct,New).            % Set the new value.
```

Figure 7.29: The Muse implementation of setarg/3.

to the changed term stack cell) is stored on the trail stack to force the term stack cell to be copied at incremental coping. (The trailed reference is also needed in SICStus by the garbage collector.) The undo goal resets the changes made on backtracking.

7.11.3 If

The SICStus predicate `if(G1,G2,G3)` is a mighty queer creature. If there exists any solution to the goal `G1` then the goal "G1,G2" is executed. Otherwise the goal "G3" is executed. The `if` call cannot be implemented in Prolog without using side effects or repeating the search for the first solution to `G1`.

The naive implementation that repeats the search for the first solution to the goal `G1` is shown in Figure 7.30. The repeated execution introduces an inefficiency and also an error if the goal G1 contains side effects.

```
if(G1,G2, _) :- exists(G1), !, call(G1), call(G2).
if(_ ,_ ,G3) :- call(G3).
                                     not(G) :- call(G), !, fail.
exists(G) :- not(not(G)).            not(_).
```

Figure 7.30: A naive implementation of if/3.

The SICStus implementation shown in Figure 7.31 uses the built-in predicate `SETARG/3` described in Section 7.11.2. This predicate cannot be used in the current version of Muse, so the predicate `if` is not currently implemented.

```
if(G1,G2,G3) :- if(G1,G2,G3,flag(no)).

if(G1,G2, _,Flag    ) :- call(G1), SETARG(1,Flag,yes), call(G2).
if(_ ,_ ,G3,flag(no)) :- call(G3).
```

Figure 7.31: The SICStus implementation of if/3.

It is possible to implement the `if` predicate in an OR-parallel system using global variables, as shown in Figure 7.32. This implementation should work without any extra modifications in the "ProLog by BIM" [18] version of Muse. Notice that `erase(S)` always succeeds, even if `S` does not exist.

```
if(G1,G2,G3) :- get_unique_identifier(S),
                record(S,anything),
                if(G1,G1,G3,S).

:- sequential if/4.
if(G1,G2, _,S) :- call(G1),    erase(S), call(G2).
if( _, _,G3,S) :- is_a_key(S), erase(S), call(G3).
```

Figure 7.32: The global variable (parallel) implementation of if/3.

7.11.4 Minimize Locking of Hash Tables

In SICStus Prolog the atom table and other tables are implemented as expandable hash tables. No garbage collection is supported though. The hash tables are (normally) frequently searched and infrequently updated. The updating adds a new item and may expand the table.

An improved algorithm for access to hash tables has been implemented in the current Muse system. Before entering the hash table search, a copy of the global pointer to the current hash table is made. This copy is used in the algorithm. All searching in the atom table is done without acquiring the table lock. Whenever a hash table miss is encountered, the table is updated. Before performing this update a lock is acquired. Now one of three situations can occur:

- If the new item can be added to the table then add it and release the lock.
- If the item cannot be added (the hash table position is occupied or the hash table has been expanded) then release the lock and retry the hash table search.
- If the table must be expanded before adding the item, then make an expanded copy of the table, add the item to the new table, update the global hash table pointer to point to the new table, and release the lock.

The only drawback introduced by the algorithm is that old copies of the hash table cannot be removed at expansion of the hash table. They can be deallocated later at some situation known to be safe. The problem is less serious than it may appear. If the new size after expansion is twice the old size then the sum of all older hash tables sizes is approximately the same as the size of the new one. Without doing any deallocation at all the memory consumption is then twice as large only.

7.12 Machine Dependent Issues

Writing portable programs may be achievable when writing sequential algorithms using the language C. Writing parallel programs is an altogether different issue. Allocating shared memory and using locks is highly machine dependent. There are both operating system and hardware differences.

7.12.1 Lock

Muse uses the ordinary spin locks shown in Figure 7.33. The lock is implemented as a repeated attempt to make an atomic exchange from 0 (unlocked) to 1 (locked). If the exchange succeeds the lock is acquired. If the lock is already marked as acquired the exchange fails and the lock remains marked as acquired. One machine dependent macro is needed: `try_lock()`. This macro returns 0 if the lock is already acquired and 1 otherwise. As an example `try_lock` is shown for a Sun4 machine. The SPARC processor has an atomic exchange instruction called `swap` taking two arguments. The first argument is a value and the second is an address. The value is atomically exchanged with the value at the address in the memory. The macro returns the previous value at the address. The unlock operation simply writes 0 to the lock. This is a correct operation iff the worker doing the unlocking operation holds the lock. The initialization of the lock is equivalent to the unlocking operation.

```
#define try_lock(p)     (swap(1,(p))==0)     /* Sun 4 */
#define init_lock(p)    (*(p)=0)
#define un_lock(p)      (*(p)=0)

#define _lock(p) do { if (try_lock(p)) break;    \
                      while(*(p)==1) continue;   \
                    } while(1)
```

Figure 7.33: The lock macros.

7.12.2 Shared Memory Allocation

Any shared memory multiprocessor machine supports allocation of memory shared between processes. The syntax for the operation may differ: UNIX BSD/mmap(), UNIX System V/shmat(), Mach/vm_map(), etc. The usage is normally trivial. An important optimization in Muse introduces a complication though: it is advantageous to be able to copy from one worker's stack area to another worker's without using an intermediate buffer. On the SunOS (UNIX BSD version), on UNIX System V, or on Mach this is not a problem. But on DYNIX (UNIX BSD version) on the Sequent Symmetry you have to use some tricks.

The shared memory mapping implementation in DYNIX is best described as adequate. Its main purpose is either to allocate some shared memory or to map files for efficient access. The Muse optimization is based on a more complicated memory mapping. Different workers (processes) have different views of the shared memory. To maintain the multiple sequential Prolog processes (Muse) model all workers must access their own stack area at similar addresses. But, all workers must also be able to access all (other) stack areas. There exists a problem with the DYNIX mmap(): new file descriptors are opened whenever either the physical or the virtual memory is not contiguous. The number of allowed file descriptors in DYNIX is limited. The following gives a quick sketch of the solution currently used in Muse.

First some definitions. The number of workers, size of stack area, and base address to a worker's own stack area are N, $size$, and $base$ respectively. The first address in the mapping file is 0. The address in the mapping file is also called the physical address or simply pa. The worker's view is called the virtual address or va. The formulas giving the n:ths worker's view of the i:th worker's stack area are shown in Figure 7.34. An example map for three workers is also included. Each discontinuity is marked with a double vertical line (‖). Worker 0 has one discontinuity and the others two. The maximum number of file descriptors used is constant (i.e. independent of the number of workers).

Calculating addresses

$$pa_i = i * size.$$
$$va_i^n = base + ((N + i - n) \bmod N) * size.$$

One example

The file	pa(0)	pa(1)	pa(2) ‖

Worker 0	va(0,0)	va(1,0)	va(2,0) ‖
Worker 1	va(1,1)	va(2,1) ‖	va(0,1) ‖
Worker 2	va(2,2) ‖	va(0,2)	va(1,2) ‖

Figure 7.34: The current memory mapping on DYNIX.

The main drawback of the allocation scheme currently used on the DYNIX machines is its static nature. It is not suited for an expandable shared memory. A more flexible memory mapping that supports expansion of shared memory easily is described in Section 7.16.3. It also supports a sparse address space allowing the Prolog stack to grow without relocation.

7.13 NUMA Specific Optimizations

In ordinary shared memory machines the bus may become a bottleneck. Some machines solve this problem using distributed memory, still keeping the shared memory concept. One example of such an architecture is the two Butterfly machines described in Section 5.3. In such machines a *processor node* consists of a processor and local memory. The local memory of remote nodes is accessible through a connection network. Accessing nearby memory is therefore faster than accessing far away memory. Those machines are called "Non Uniform Access Memory" (NUMA) machines. This class of machines introduces some new problems though. One has to minimize the remote accesses, and caching coherence of shared data is normally not provided.

7.13.1 Better Locality

In NUMA machines memory allocation must be designed to minimize access to non local memory. Each worker (which is mapped to one processor node) has its own copy of the Prolog program. Memory areas of global usage are best distributed over all workers. Otherwise switch contention may occur in the Butterfly switch. The allocation routine therefore allocates global memory in a round robin fashion. It is also possible to allocate (normally huge) data structurs (like the atom table) distributed over the processor nodes. Allocation of (possibly) temporary shared data structures (such as global frames, save areas for findall, etc) is mainly in the processor's own memory. This is done to increase the probability that future accesses to the structures is made by the processor having the data.

7.13.2 Decreasing the Polling Frequency

Many handshaking protocols in Muse are implemented using busy polling at a global address. One example is busy waiting for a spin lock. This is no problem on a shared memory machine supporting cache coherency of shared data, such as the Sequent Symmetry. During busy polling, the value of this memory location is in the cache. Whenever some other processor writes to that memory location the cache line is invalidated and the correct value cached in.

On the two Butterfly machines this is not possible. A global read/writable memory location cannot be cached. Butterfly I in fact does not support any caching at all. So either the polling processor repeatedly reads its local memory or else reads remotely from some other processor's memory. It is obvious that polling via the network from a remote memory location might cause network capacity degradation. But there is one other reason for not doing too eager polling. The processor node's local memory is double ported. It can be reached both from the local processor and from the network. But accessing the memory from one side blocks the access from the other side. Polling in my own memory might slow down someone that wants to access my memory and polling in some other processor's memory might make that processor slower.

There are two methods used to avoid too extensive polling: inserting delay code in the polling loop and avoiding polling. One way in which polling can be avoided is to test for the possibility of acquiring a lock before really trying to acquire it. If the lock is already acquired then something else might be done. The latter optimization has in fact also made the Sequent Symmetry implementation more efficient. (The scheduler never acquires the lock to request sharing from a processor when the processor's request lock is already acquired. It is then likely that a request will never be accepted.)

7.13.3 Switch Contention

The Butterfly switch puts some constraints on the connections to remote memory, i.e. there can only be a limited number of processors accessing the same memory block at the same time. When the number of processors accessing the same memory block reaches about 15 the performance degrades rapidly. The only situation in the Muse scheduler where that problem might occur is when many workers are idle. The problem is exacerbated when many workers are idle and staying in the same shared choicepoint.

A simple and efficient solution to the problem is to partly serialize the scheduling activities. Only one worker per choicepoint in the search tree is allowed to search for work in the scheduler loop, limiting the number of workers repeatedly accessing the same shared frame to one. Allowing for more than one worker may be better, but we have chosen this simple solution, which results in good performance. The code for the algorithm is shown in Figure 7.35. The idea is implemented by associating an extra field containing the name of the worker (called the *scheduling-worker*) currently allowed to execute the scheduling loop, with each shared frame. Workers that are not the scheduling-worker enter a sleep loop. The field can be initialized to any value. If the scheduling-worker is not scheduling at the shared choicepoint any worker in the sleep loop can choose to be the new scheduling-worker.

```
Sch_Get_Work(...) {
  ...
  while (true) {                              /* The scheduler loop. */
    ...
    while (I am not the scheduling-worker) {  /* The sleep loop */
      if (the scheduling-worker is here) sleep(some milliseconds);
      else the scheduling-worker = me;
  } }
  ...
}
```

Figure 7.35: Serializing the scheduler loop for NUMA machines. The figure shows an extension of the macro in Figure 7.12 (page 121).

7.13.4 Higher Value for Delayed Release

Scheduling activities are relatively more expensive in NUMA machines than in ordinary shared memory machines. This is because the scheduler relies on shared data structures which are frequently accessed by all processors. The smallest usable task size is increased and it is therefore harder to get speed up for programs with small granularity. The number of predicate calls before releasing new load (Section 7.8.3) is increased from 5 to 10.

7.13.5 Caching on Butterfly II

The Butterfly II machine can declare any page in memory (both local and remote) as cached or not. But coherence for physical pages shared between processors is not guaranteed. The easy solution is to turn caching for shared data off. But in the current version of Butterfly II Muse there are two conflicting demands. The WAM stacks should be cached for Prolog execution efficiency, and the WAM stacks should also be remotely accessible (i.e. shared) for copying efficiency as shown in Section 7.8.1.

A good (but tricky) solution is to declare the own stacks as copy back cached and to implement a cache coherence protocol for the sharing session. Say that parts of worker P's stacks are copied to worker Q. Worker P first flushes to memory the parts to be copied to ensure that the physical memory contains the correct information. Then worker Q performs the copying from the remote stacks to its private stacks.

The copying for worker Q can be made even more efficient. If worker Q has declared the remote stacks as cached then the copying loop performs the transfers one cache line at a time, resulting in fewer remote accesses and a higher cache hit ratio. This solution complicates the cache coherence protocol. In the current version of Butterfly II Muse every worker keeps a list of areas that are copied from remote stacks. Those areas cannot be copied again without first invalidating the area. We have chosen to invalidate the whole list of areas while the worker Q is waiting for the worker P to respond to a sharing request. That time is usually just wasted time anyhow.

Some parts of the code implementing the total cache coherency protocol are shown in Figure 7.36. At sharing Q does all the copying. In the macro `Eng_P_Share()`, P therefore flushes all data that Q shall copy. Information about all areas that Q copies is recorded by the the function `add_to_copied_areas()`. Before Q can do any copying it must invalidate all previously copied areas with the function `invalidate_copied_areas()`.

7.14 Debugging and Evaluation Tools

It is very important to have tools for debugging and evaluating a big system like Muse. Otherwise it is very hard to make an efficient implementation. Unfortunately our main programming environment (DYNIX) does not (in our opinion) support any useful tools for debugging or evaluating parallel programs. We have used four tools during the development of Muse: (1) the Muse graphic tracing facility *Must*, (2) the visualization tool VisAndOr, (3) the Muse built-in statistics package, and (4) a benchmarking package.

The first tool is briefly described in Section 3.10. A more complete description can be found in [82]. Trace events are recorded in real time when executing a query

```
Eng_P_Share() {
  ...
  flush all areas that Q shall copy;  /* Replaces 'Copy to Q' */
  ...
}

Copy_from(P) {
  for (All area:s to copy) {
    ptr = remote_area(P,area);
    copy(ptr,size);
    add_to_copied_areas(ptr,size);  /* New code. */
} }

Eng_Q_Share(P) {
  ...
  invalidate_copied_areas();  /* New code while waiting for reply. */
  ...
  Copy_from(P);  /* New code added when sharing is accepted due to the  */
  ...            /* fact that in NUMA machine Q does the copying.       */
}
```

Figure 7.36: The cache coherency protocol for Butterfly II. The codes that are expanded can be found in Figures 7.9 (page 116) and 7.12 (page 121).

on a special version of Muse. Some trace events include a time stamp. When the execution of the query terminates (or at an exception) the recorded trace events can be written to a file. A graphical tool called Must is used to display the dynamic behavior of the shared part of the search tree and also the processor utilization. Repeatable bugs related to the scheduler are very easy to track down using the tracer. Scheduler inefficiencies are also easy to find.

The VisAndOr is a visualization tool developed at the Computer Science department of the University of Madrid [27]. The tool uses a subset of the Must tool traces, and it shows a static view of the *whole* execution tree of a query. The displayed tree is ordered from left to right on the x-axis and by increasing time stamp on the y-axis. We have developed a program that converts Must files to VisAndOr files. The Must files contain information about the shared search tree only, so the VisAndOr tool shows a static view of the shared part of the whole search tree.

We have also integrated the two tools into one tool called *ViMust*. The two tools send and receive the current time stamp. The ViMust tool is mainly used in two modes: (1) the Must tracer, while showing an animated view of the execution, sends the current time stamp to VisAndOr, and VisAndOr displays the current time stamp as a horizontal line, and (2) the user chooses, with the mouse, an interesting part of the search tree shown in VisAndOr and Must moves to the corresponding time stamp. The ViMust tool is very useful when examining the behavior of and the amount of parallelism in the parallel execution of a Prolog program.

Figure 7.37 shows a snapshot of ViMust for an execution using 8 workers. In the right window the Must tool shows the current search tree at the time 61 milliseconds and in the left window the VisAndOr tool shows the total shared search tree with a horizontal line corresponding to the same time.

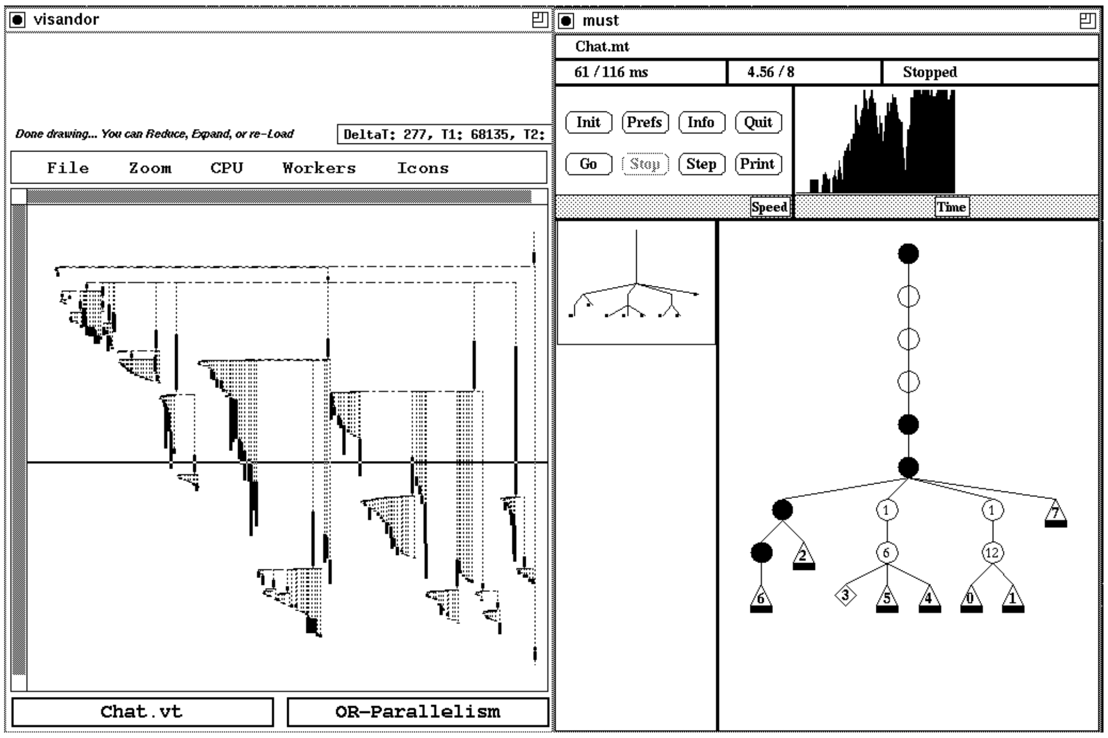


Figure 7.37: The combined tool ViMust.

The third tool, the statistics package, is mainly used to find inefficiencies in the system. During a parallel execution of a query statistics are collected regarding the amount of time spent in several "modes", amount of information copied, task size in predicate calls, etc.

Figure 7.38 is a typical output showing timing information and the amount of copied data for a Prolog program executed by 10 workers on Sequent Symmetry. The main information to be found in this figure is: (1) the workers are executing Prolog 86.6% of the time, (2) no work could be found 10.6% of the time, (3) the rest (2.8%) is scheduler overhead, (4) the number of tasks are 90+2, (5) the number of accepted sharing requests is 62, (6) and the total amount of copied information is around 80 KBytes (less than 2 KBytes per sharing session). The average task size is around 476 (43825/92) predicate calls per task.

Figure 7.39 is a typical output for granularity information showing two histograms for task sizes and two histograms for chunk sizes, both measured in number of predicate calls. A task is a piece of work executed without asking the scheduler for more work. A *chunk* is a part of a task undisturbed by performing sharings.

```
                Type        #         ms
WORK
                busy      498       5285 (  86,4 % )
            bcktrack      241         12 (   0,2 % )       5297 (  86,6 % )
IDLE
                idle      162        650 (  10,6 % )        650 (  10,6 % )
GRAB
                 gsb       20          1 (   0,0 % )
                 gpb      329         19 (   0,3 % )
                 grb        9          0 (   0,0 % )         20 (   0,3 % )
Q SHARE
              qfound       88          3 (   0,1 % )
              qwait1       88         23 (   0,4 % )
               qcopy      248         34 (   0,6 % )
              qwait3        3          0 (   0,0 % )
              qwait5        7          1 (   0,0 % )
            qinstall       53          5 (   0,1 % )         68 (   1,1 % )
P SHARE
            pshare_d       86          6 (   0,1 % )
         pfind_invis       37          3 (   0,1 % )
            pprepare       62          4 (   0,1 % )
              pshare       49          7 (   0,1 % )
             pupdate       62          7 (   0,1 % )
               pcopy       20          7 (   0,1 % )
              pwait1       43         11 (   0,2 % )         46 (   0,7 % )
FIND
               f_jmp       64          5 (   0,1 % )
              search      215         18 (   0,3 % )         24 (   0,4 % )
OTHER
            r_signal       98          3 (   0,1 % )
              commit        2          0 (   0,0 % )
             r_prune       10          1 (   0,0 % )
                 cut        2          0 (   0,0 % )
           spec_save       79          3 (   0,0 % )
          spec_claim       27          3 (   0,0 % )
            lck_wait      158          5 (   0,1 % )         15 (   0,3 % )

Total time = 6120 = 612*10      term stack (by P)    19496 bytes
Predicate calls = 43825         term stack (by Q)    48668 bytes
Tasks = 90(par)/2(seq)          environment stack     4496 bytes
Accepted sharing req. = 62            trail stack     1984 bytes
Workers = 10                    choicepoint stack     9300 bytes
```

Figure 7.38: Sample overhead statistics.

The average task size is 476.36 predicate calls and the average chunk size is 284.58 predicate calls. For systems like Muse where the sharing operation is expensive, the chunk size is the most interesting information. With some computation it can be deduced that around half of the time is spent in chunks greater than 1000 predicate calls. That is a fairly huge chunk size. The delayed release function described in Section 7.8.3 tries to avoid chunk sizes that are smaller than 5 predicate calls.

```
Tasks (Pieces of work without asking the scheduler for more)
  from      to       num             calls          [average calls]
     4->     7:        3 (  3,3 %)      14 (  0,0 %)
     8->    15:        2 (  2,2 %)      22 (  0,1 %)
    16->    31:        5 (  5,4 %)     119 (  0,3 %)
    32->    63:       12 ( 13,0 %)     561 (  1,3 %)
    64->   127:       16 ( 17,4 %)    1613 (  3,7 %)
   128->   255:       23 ( 25,0 %)    3899 (  8,9 %)
   256->   511:       11 ( 12,0 %)    3930 (  9,0 %)
   512->  1023:       10 ( 10,9 %)    8290 ( 18,9 %)
  1024->  2047:        4 (  4,3 %)    4593 ( 10,5 %)
  2048->  4095:        5 (  5,4 %)   16498 ( 37,6 %)
  4096->  8191:        1 (  1,1 %)    4286 (  9,8 %)

           Sum:       92 (100,0 %)   43825 (100,0 %)   [476.36]

Chunks (Pieces of undisturbed work)
  from      to       num             calls          [average calls]
     0->     0:        2 (  1,3 %)       0 (  0,0 %)
     1->     1:        2 (  1,3 %)       2 (  0,0 %)
     2->     3:        5 (  3,2 %)      13 (  0,0 %)
     4->     7:       11 (  7,1 %)      62 (  0,1 %)
     8->    15:        8 (  5,2 %)      86 (  0,2 %)
    16->    31:       12 (  7,8 %)     279 (  0,6 %)
    32->    63:       28 ( 18,2 %)    1340 (  3,1 %)
    64->   127:       43 ( 27,9 %)    3945 (  9,0 %)
   128->   255:       13 (  8,4 %)    2223 (  5,1 %)
   256->   511:       11 (  7,1 %)    3971 (  9,1 %)
   512->  1023:       10 (  6,5 %)    8373 ( 19,1 %)
  1024->  2047:        3 (  1,9 %)    3358 (  7,7 %)
  2048->  4095:        5 (  3,2 %)   16016 ( 36,5 %)
  4096->  8191:        1 (  0,6 %)    4157 (  9,5 %)

           Sum:      154 (100,0 %)   43825 (100,0 %)   [284.58]
```

Figure 7.39: Sample granularity statistics.

An invaluable benchmarking tool has been developed. All speed-up graphs and tables presented in several sections of this paper are (more or less) automatically generated by the tool. The tool can generate combined information for e.g. Muse, Aurora, and SICStus, relative speed-ups, absolute speed-ups, best values, mean values (with standard deviation), etc. The user has to make a file defining the benchmarks to execute, the number of workers used, what Prolog system to use, etc. The benchmarking suite is then executed, statistics computed, LaTeX tables and graphs produced and printed. All as one batch job.

All those statistics (speed-ups, timings, granularity, etc.) might for non parallel-programmers look boring and meaningless, but for us it is a way of life. The main motivation for parallelizing the execution is to get better performance. Without collecting (usually lots of) statistics it is very hard to evaluate and improve the parallel system.

7.15 Advanced Topics

So far the implementation of the system has followed the main rule: *keep it simple*, the only slight exceptions being the implementation of cut and the optimizations. In this section some more difficult problems are described. The more interesting topics are suspending branches and optimizing for finding the first solution only.

7.15.1 Suspending Branches

A situation may occur where the system would benefit from rescheduling a busy worker from one part of the search tree to another. Information about suspended tasks must then be saved in a way that allows the tasks to be resumed later. Saving information in a shared choicepoint on how to resume a suspended task is called *suspending* the branch associated with the task and rooted at the choicepoint.

The suspension-resumption cycle of a branch is (potentially) expensive in any OR-parallel system. It is time consuming or memory consuming or both, depending on the implementation. Even though there exist situations where rescheduling tasks is crucial to the performance, it is always a risk. It may be a waste of resources and it may even be a disaster. The execution may be substantially slowed down due to excess resumptions or page faults. The available virtual memory space may also be totally consumed. Some kind of moderator for rescheduling activities has to be introduced.

To implement suspension of branches and to implement garbage collection are two conflicting goals in an OR-parallel system. The garbage collection changes the shared parts of the Prolog stacks. The problem is evident both in Aurora [93] and in Muse. A garbage collector is a crucial part of any serious Prolog system.

In Muse, each worker can perform local garbage collection, changing the representation of the worker's state. A similar conflict as for incremental copying exists for suspension. You can either save information about a suspended branch as a difference between two computation states or save the whole state. In the latter case a branch can be resumed (without problems) by any worker, even if it has performed garbage collection. But it is too expensive to always save the whole state.

One method is to use garbage collection to get a canonical representation of the state. The worker suspending a branch performs garbage collection before it computes and saves the difference. All workers also perform garbage collection before resuming the branch. The SICStus garbage collector guarantees that this procedure will work. Other solutions to the problem of suspended work are discussed in Sections 7.16.1 and 7.16.2.

7.15.2 Scheduling of First Solution Goals

Sometimes you are only interested in finding the first solution to a goal. For this purpose the cut operation is used to remove alternatives that are no longer needed. How cut is implemented is described in Section 7.7.4.

In an OR-parallel system ongoing work sometimes is in danger of being aborted. This work is called *speculative work*. It is obvious that the possibility of speculative work causes scheduling problems: doing speculative work may be a waste of time. This topic has been thoroughly investigated. The paper [45] introduces a method for computing which branches correspond to speculative work. The paper also investigates and evaluates several scheduling principles. One main conclusion is that workers shall use a more directed scheduling strategy, searching for new work from left to right in the search tree.

The paper [15] describes three main methods to increase the performance of programs containing speculative work: (1) the cut operation can be more completely performed, (2) idle workers can choose a left-to-right scheduling strategy, and (3) busy workers may get rescheduled to work found in a branch to the left in the tree.

Scheduling of speculative work is an issue where more research is needed. The overhead added by generating more information and by making non optimal scheduling decisions slows down the system considerably when no speculative work exists. For programs that generate lots of speculative work special scheduling is beneficial. Solutions for programs containing a limited amount of speculative work and for programs containing phases with and without speculative work have not yet been found.

7.15.3 Commit (the Real Thing)

Sometimes you are satisfied with finding just any solution to a goal. In Section 7.10.3 a brute force implementation (called cavalier commit) with that objective was described. If the extra information about speculative work described in Section 7.15.2 is introduced, then it is very easy to implement a more useful version of commit. This version is referred to as *commit* only [45].

The implementation of commit can (in an informal description) be implemented as follows: kill all other work rooted at the scope choicepoint so long as the killing of the work cannot be prevented by a cut operation. This implementation (in contrast to cavalier commit) keeps the execution of speculative work invisible to the user. Notice that lack of information about speculative work reduces commit to cut.

How to implement cut is described in Section 7.7.4. A similar method can be used when implementing commit. The worker performing a commit traverses the choicepoints in the shared part of its branch. If the worker does not detect any choicepoint where the branch is endangered by a cut operation (from a branch to the left of the branch) then it can perform the whole commit operation. Otherwise the worker performs as much as possible of the commit operation and marks the choicepoint where the operation could not be continued with a pending commit. The pending commit operation can then be continued later on.

7.15.4 Asynchronous Assert

The implementation of assert/retract is not trivial to parallelize. The main difficulty derives from the fact that the Prolog system is able to backtrack to several alternative asserted clauses. In a parallel environment, without synchronization, several workers may backtrack for more clauses for a dynamic predicate, a predicate for which at the same time several other workers may both assert and retract clauses.

Another problem is the semantics of assert/retract. Different Prolog systems have different sequential semantics. Not all types of semantics are suitable for using assert/retract as a communication means between OR-branches. The most common semantics, the *logical* database view [62], hides any modifications of the database during a database search. In Prolog, using this semantics, backtracking for more solutions to a call to a dynamic predicate is affected by neither assert nor retract. The asynchronous version of this semantics is very hard to implement in an OR-parallel system and it is also almost useless. The *immediate* database view, where changes made by assert/retract take effect immediately, is better suited as an OR-parallel communication means. But it is also not easy to implement and it is expensive: a choicepoint is needed for every call to a dynamic immediate predicate. Naively implemented, the retracted clauses cannot be (without very expensive tests) deallocated (but only marked as dead) when the system has more than one active worker. I have to admit that the current implementation in Muse is such a naive one. It also contains a (hard to fix) bug. I now present an improved algorithm capable of deallocating retracted clauses almost immediately. The question now is: should this new algorithm be implemented or shall immediate dynamic predicates be excluded from Muse?

The new algorithm (for immediate dynamic predicates) is based on having one reference counter for each asserted clause, marking referenced retracted clauses as dead, and removing unreferenced dead clauses. For simplicity the cut operation is first ignored and later on introduced. The predicate `asserta/1` adding a clause before all asserted clauses is also ignored for the same reason. All operations are assumed to be atomic.

The allocated clauses (both alive and dead) are put in one linked list per predicate. Each clause element contains the clause code, a pointer to the next clause element, an alive flag, and a reference counter. Whenever the reference counter is decremented to 0 and the clause is dead the clause is deallocated.

The following is a description of the algorithm (disregarding cut). (1) When a clause is asserted a clause element is put last in the linked list of clauses. The reference counter is initiated to 0 and the alive flag to true. (2) When a retract is made a search for the first alive clause is made. If no clause is found then the call fails. Otherwise a choicepoint is allocated and a reference to the found clause is added to the choicepoint, the reference counter in the found clause is incremented, and the clause is marked as dead. (3) At backtracking (on retract) a search for the next alive clause is made. After the search the reference counter for the previous clause is decremented (and the clause if dead deallocated). If no new clause is found then the call fails. Otherwise a reference to the found clause is added to the choicepoint, the reference counter in the found clause is incremented, and the clause is marked as dead. (4) The call and backtracking (on call) is performed in a similar way except that the clause is not marked as dead. (5) On sharing the reference counters are updated to reflect the number of choicepoints now referring to the clauses.

The algorithm becomes more complex if cut is introduced. Let us call choicepoints referring to dynamic clauses *dynamic choicepoints*. One solution is to keep all dynamic choicepoints on the same choicepoint stack in a linked list. Let us call this list the *dynamic clause reference* list. Normally a cut operation (in SICStus) is a constant time operation just changing a WAM register. Now the items in the dynamic clause reference list corresponding to removed choicepoints must be examined to update reference counters in asserted clauses.

An algorithm for parallel assert/retract using the logical view can be implemented in a similar but much more complex way, involving allocating frames in shared memory for each dynamic choicepoint. The complexity and inefficiency of the algorithm in conjunction with the dubious value of parallelizing makes it unnecessary to describe.

7.16 Further Thoughts

In this section I discuss some topics that I find interesting. Nothing here discussed has been implemented and sometimes the discussed topic generates open questions. Some topics related to implementing a real production system are also discussed. The current research version of Muse does not try to solve those problems.

7.16.1 More Workers than Processors

Instead of suspending branches (as described in Section 7.15.1), a method using more workers than the number of processors can be implemented, as in [50]. Some of the workers situated at the root choicepoint are initially sleeping. (To avoid confusion I do not call the sleeping workers suspended.) Whenever a busy worker suspends its task it wakes up one of the sleeping workers and then goes to sleep. The new worker then uses the now free processor resource to execute some non suspended task. Whenever the suspended task becomes leftmost its associated worker can be waken up to resume the task.

It is hard to foresee whether this very simple implementation will be useful. The number of excess workers must be limited, so any/some/most programs may run out of workers nevertheless. Going to sleep and waking up is also expensive. One solution to the latter problem is to *not* allocate a UNIX process to each worker. The number of processes is instead the same as the maximum number of awake workers, similar to methods used in &-Prolog [48, 49]. A process shall then be able to change identity from one worker to another. Changing identity involves the process remapping its shared memory and also saving and changing the contents of some registers. But remapping etc. may also be too expensive.

7.16.2 Recompute Suspended Branches

To suspend branches as described in Section 7.15.1 is memory consuming and the number of workers, as described in Section 7.16.1, is limited. But there exist (at least) two more alternatives for treating suspended tasks. Both are based on recomputation.

The first, and simplest, alternative is to do a total recomputation when resuming suspended branches. When a branch is suspended the total state of the branch is thrown away. The alternative number corresponding to the suspended branch is stored in the shared choicepoint where the branch is rooted. At resumption the execution of the alternative is restarted.

The second alternative relies on the workers keeping a *history path* when executing Prolog as described in [1]. The history path is a “road map” showing the route from the root choicepoint to the current position (in the total search tree). At suspension of a branch a copy of the history path is stored in the choicepoint where the branch is rooted. At resumption the history path is used in conjunction with a special version of the WAM emulator to recompute the state.

The first alternative is very simple but it may be very expensive to do the recomputation since thrown away branches are thrown away work. The second alternative solves the problem of potentially very expensive recomputations since the resumption time is proportional to the length of the suspended branch. But the overhead associated with maintaining the branch stack is high and modifying the WAM complicates the implementation.

All models relying on recomputation have problems with asynchronous side effects. The same side effect might be executed several times.

7.16.3 Dynamic Memory Size and Worker Number

Allocating the maximum number of workers and amount of memory at start up of a production OR-parallel Prolog system is an intolerable waste of resources. Neither the version running on the DYNIX operating system on the Sequent Symmetry nor our earliest version running on a dedicated VME-bus based hardware can dynamically change the resource allocation. The DYNIX `mmap()` system call is not easy to use and the original VME hardware was not software configurable at all.

Other operating systems (e.g. Mach, SunOS, and System V) do not have this kind of limitations. Using shared memory mapping and expandable or more paging files makes it very easy to add and remove both workers and memory. I now present a new and very simple scheme for allocating memory, which even works with DYNIX (the proposed scheme requires four more file descriptors per worker for DYNIX than the earlier scheme).

The allocation scheme is based on all shared memory blocks being allocated from a common shared memory. All workers map this shared memory identically. One special memory block is allocated. This block contains a table indicating the address and size for some blocks (e.g. the WAM stack blocks and scheduler areas for all workers). Remember that all workers shall be able to reach their own WAM stack blocks at the same addresses. This is accomplished by each worker mapping its own WAM stack areas at an alias address. Thus each worker in SICStus Muse needs to make 4 memory mappings. One for the whole shared memory space, one for the environment stack, one for the term stack and one for the combined choicepoint and trail stacks.

7.16.4 Executing on a Loaded System

The current Muse implementation is optimized for executing on a non loaded machine. It assumes that the worker is always running. Several of the communication protocols, especially those at sharing, are based on one worker (say P) busy waiting for another one (say Q). If the process associated with worker Q is moved from the run state to the ready queue (by the operating system) while P is waiting for Q to finish some task, P is prevented from continuing for an intolerably long time. At the sharing session it may be a good choice in a production system to let P produce a block of data without waiting for Q. The worker Q can use this block to install itself to the same state as P.

7.16.5 Parallel UNIX I/O

In the operating system UNIX file descriptors (the reference to an I/O channel) can not be exported from one process to another. The only way to export file descriptors is to create them before forking (creating) a child process. The child then gets a copy of the file descriptor. But the file descriptor copied is not shared. Shared I/O between processes is also not supported. Several processes using the same I/O channel are not synchronized, use their own I/O buffers etc. In a real parallel Prolog system this problem must be solved.

One solution is to perform all I/O via an I/O server. The workers are the clients to this server. All I/O manipulations are made via this server: opening file descriptors, performing the I/O etc. There is naturally some overhead associated with using this approach. An extra process is needed. If this process is busy waiting for any I/O then a processor is lost in the system and if it is sleeping waiting for I/O then it is likely that the process is swapped out when needed. To let all I/O go via a server also introduces some delay. The solution is simple though. The current version of Aurora includes such a server [63].

Another solution is to let the workers themselves implement parallel I/O, removing most of the overhead associated with the server approach. The overhead can be completely removed if no asynchronous I/O is allowed. Then the same methods as for doing sequential I/O can be used if the I/O buffers are shared. For asynchronous I/O some locking to assure atomic updating of the shared I/O buffers has to be introduced. One problem with using this method is that information about open file descriptors must be kept identical in all workers. This can be managed using the interrupt mechanism of UNIX and duplicating all file descriptor manipulation. Although this method is more complex I think it is the preferable one.

7.17 Conclusions

The Muse model is very simple. Using the already available code for the Muse-SICStus system in conjunction with this chapter and the interface found in Chapter 8 makes it possible to adapt virtually any existing Prolog system to explore OR-parallelism. With just minor efforts it should be possible to create a system that executes pure Prolog programs, containing medium to large granularity, with high efficiency. Extending this system for full Prolog and optimizing it for programs of smaller granularity should also be feasible. I guess that the effort needed to extend a real Prolog system (like SICStus) is around one man year. When adding extra features (like asyncronous I/O), do remember the two design philosophies: keep it simple and keep it visible. Do not add anything that complicates the design and keep the use of the features clearly visible in the Prolog code.

7.18 Acknowledgments

The work was made possible by the funding by the PEPMA project (ESPRIT project 2471) and by the SICS sponsors. The cooperation with André Mariën and Luc Maes (both from the BIM Prolog company) has been very essential. I am indebted to

Khayri A. M. Ali, Torkel Franzén, and Mats Carlsson (all at SICS) for extensive proof reading.

⋆ ⋆ ★ ⋆ ⋆

The sponsors of SICS are Asea Brown Boveri AB, NobelTech Systems AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defence Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

Chapter 8

The Scheduler-Engine Interface used in Muse

Roland Karlsson Khayri A. M. Ali
SICS research report [58]

ALMOST any sequential Prolog system is in principle easy to extend for OR-parallelism, using the Muse execution model. To reduce your programming effort we have implemented the Muse scheduler, with a clean interface to the Prolog sequential engine. This interface is implemented as a set of C macros. The sequential Prolog system to be parallelized uses some of those macros provided by the Muse scheduler and must also provide some macros for the Muse scheduler. This chapter contains a definition and description of the required macros, emphasizing information needed by the Prolog engine programmer.

8.1 Introduction

This chapter documents the interface between the Muse scheduler and Prolog engines based on the WAM. The interface has been implemented for the SICStus Prolog engine (version 0.6) [26] and tested on a number of multiprocessor machines (and Sun workstations).

In this document, we assume that the reader is familiar with the WAM [91], and the Muse approach found in Chapters 3 and 4. We also recommend reading Chapter 7 describing the actual Muse implementation. In Muse, a number of workers (processes) explore OR-parallelism in a Prolog search tree. Each worker has two components – an engine, which executes Prolog code, and a scheduler, responsible for distributing the available work among the workers. There is a well defined interface between the engine and the scheduler code that enables different Prolog engines based on the WAM to be used with the Muse scheduler. At present there is only one Prolog engine used with the Muse scheduler. The engine is based on SICStus version 0.6. There is ongoing work for using the sequential BIM Prolog engine with the Muse scheduler.

In Muse, the search tree consists of a number of nodes of two kinds – shared and private. Shared nodes make up the shared portion of the search tree. Private nodes are accessible only to the worker that created them. The tree is divided into the upper shared section and the lower private section. Each worker's engine performs a sequential traversal of its private region. Anytime a worker has to access the shared part of the search tree, it calls its scheduler. Thus the scheduler provides synchronization between the engines.

In Muse, each Prolog engine has its own copy of the WAM (or Prolog) stacks. There is also some shared memory for representing shared nodes, global tables (e.g. atom and predicate tables), and some global registers. When a worker runs out of work, it shares some nodes with another worker and copies the difference between the two workers' states. Then through the normal backtracking of Prolog, a worker gets work from the shared nodes.

The work documented here is much influenced by the Aurora interface [84]. One motivation for defining an interface between the Muse scheduler and the Prolog engines based on WAM is that the BIM company has assigned a subcontract to SICS in the PEPMA framework for extending the sequential BIM Prolog to OR-parallel execution using the Muse approach. SICS defines the algorithmic interface between the Muse scheduler and the BIM engine, and the BIM group implements the engine part of the interface. The existing Muse scheduler is used, providing a set of macros for the engine.

Ideally, the interface definition should be general enough to enable different schedulers and Prolog engines to be used with each other. The Aurora interface allows different schedulers to be plugged into the same engine, whereas the Muse interface is directed towards allowing different Prolog engines to be plugged into the same scheduler. Defining and implementing a more general interface that combines both could be a next step.

We have tried to reuse as much as possible from the Aurora interface work in order to take advantages of the Aurora results. Since the Muse model differ from the SRI model [92] (used in Aurora) and the search tree is represented differently in Muse than in Aurora, many of the interface macros are different.

The definition documented here does not fully cover some aspects of the engine/scheduler interface within Muse (e.g. those related to debugging and performance analysis). We are also expecting further feedback from the BIM group which may lead to some more modifications in the interface.

As a bonus for us, the interface work has improved the structure and readability of the Muse code, and also enabled us to discover and remove redundant code. The interface work has not introduced any negative effect of the performance results. Some experimental results of the new Muse system are found in Sections 5.4 and 6.5.

In the next two sections, we describe and define the engine and scheduler duties. Section 8.4 presents an overview of the Muse interface and discusses the differences compared to the Aurora interface. In Section 8.5 we describe the Muse interface. Section 8.6 concludes the chapter.

8.2 The Prolog Engine

The core of the Prolog engine is a conventional sequential Prolog engine. This core is extended for OR-parallel execution.

Since different sequential Prolog engines can be implemented differently and the representation of data and code is known only to the engine, it is natural for the engine to perform memory management. The memory management has to make a distinction between local and global memory. Some of the information stored in the local memory of the sequential Prolog system has to be made globally known to the other workers. This information includes the Prolog program, hash tables for atoms, the dynamic predicates etc. The engine also provides shared memory areas for storing information associated with the shared nodes and for communication between workers (defined in Section 8.5.1).

The engine also does process management. This includes forking workers, killing workers, and providing functions for suspending and resuming processes.

Some global information (e.g. atom tables) can be accessed by any worker asynchronously as an atomic operation. This can be achieved by using locks (or semaphores, etc) to serialize the access. The engines perform this access without any help from the scheduler. For synchronous access to other global information the engine has to call the scheduler to make sure that the worker is in the leftmost branch of the Prolog search tree.

Whenever there might be some interaction with other workers the engine calls an appropriate scheduler function. This includes: 1) checking arrival of (prune or share) requests from other workers, 2) entering the shared section of the search tree, 3) performing a pruning operation.

In the Muse model, each worker has its own copy of the Prolog stacks. After sharing, the two workers involved in the sharing operation have roughly identical memory images. Since the representation of the Prolog stacks is known to the engine, the engine performs copying and installation of the Prolog stacks.

The scheduler needs some help macros to be able to fulfill its purpose. There are macros providing information about the state of the engine and also macros that perform certain engine specific tasks.

8.3 The Muse Scheduler

> *"The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with available work with minimal overhead."*

The central idea is the notion of a shared search tree. Choicepoints in the sequential Prolog machine will become nodes in this tree. The Muse scheduler maintains a shared search tree, where each worker only has access to nodes within its own branch. The shared part of the search tree grows when a worker makes its private nodes shareable and it shrinks when the last worker backtracks from a shared node.

Whenever an engine runs out of work within its local subtree, it calls the scheduler for work. The scheduler first tries to find work on shared nodes in the current branch. Then it tries to find another worker that has either private excess load or shared excess load. In either case, it requests sharing work from that worker. If no work can be found then the scheduler stays at a suitable place in the search tree. When no work remains within a subtree all workers leave that subtree.

Whenever an engine performs a synchronous side effect, it calls the scheduler to check whether or not the current worker is in the leftmost branch of the search tree. So, the scheduler has to support that test.

When an engine performs a cut (or commit) operation, it calls the scheduler to prune branches according to the semantics of the cut operation. Since the Muse scheduler allows execution of speculative work (i.e. work that might not be executed by the sequential Prolog system), the scheduler provides detection and correction for the case when speculative work has to be aborted.

8.4 An Overview of the Interface

In this section we give an overview of the Muse interface and point out the relations between the Muse interface and the Aurora interface [84].

In the current Muse interface, as in the current Aurora interface, the execution of a Prolog program is governed by the engine: whenever the engine runs out of work, it calls an appropriate scheduler macro to provide a new piece of work. As pointed out in [84], the advantage of this scheme is that the overhead for switching between the engine and the scheduler is much smaller than it is when the Prolog execution is governed by the scheduler. The reason for the reduced overhead is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when the engine runs out of work and need not be rebuilt when the engine gets a new piece of work.

We give in Sections 8.4.1 and 8.4.2 an overview of the macros defined by the scheduler and the engine respectively. In Section 8.4.3 we give the scheduler and the engine macros used for sharing nodes between workers. Notice that the names of all scheduler macros are prefixed with **Sch_** and all engine macros are prefixed with **Eng_**, as a convention.

8.4.1 Scheduler Macros

Figure 8.1 shows the main five macros provided by the Muse scheduler for the engine during work. The scheduler macros are arranged in three groups, following the Aurora classification [84]:

- finding work (**Sch_Get_Work**);
- communication with other workers (**Sch_Check, Sch_Prune, Sch_Synch**);
- events of interest to the scheduler (**Sch_Set_Load**).

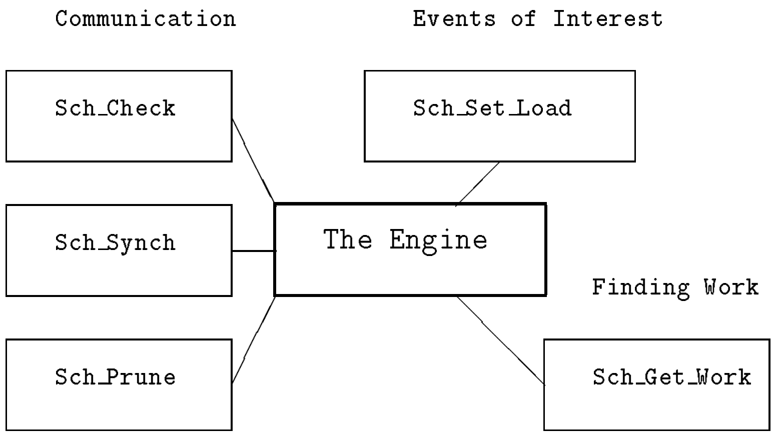


Figure 8.1: Macros called by the Prolog engine.

Sch_Get_Work is called when the engine dies back to a shared node. Its purpose is to find a new piece of work.

Sch_Check is called at every Prolog procedure call to check arrival of requests from other workers.

Sch_Prune is called when a cut or commit is executed.

Sch_Synch is called when a side effect is encountered.

Sch_Set_Load is called when the engine creates a choicepoint. The scheduler maintains information about the current local load of the engine.

This set of macros can roughly be seen as a simple subset of the scheduler macros of the Aurora interface [84].

8.4.2 Engine Macros

Let us now look at the other side of the interface: macros provided by the engine for the scheduler. The main data structure shared between the engine and the scheduler is the representation of the Prolog search tree. Nodes in the search tree are either private or shared. Each private node is represented by the normal Prolog choicepoint. Each shared node is represented by two physically separated data structures: the normal Prolog choicepoint and a shared frame.

The engine provides the scheduler with information about **choicepoints** (e.g. the parent of a given choicepoint), the type of a predicate (sequential or parallel) associated with a choicepoint, etc. The engine also maintains the representation of the Prolog clauses. So, the engine provides the scheduler with macros that manipulate **alternative clauses**. It also provides the scheduler with the current **load**.

Figure 8.2 shows macro groups provided by the engine. They are classified into **clause** macros, **choicepoint** macros and local **load** macros.

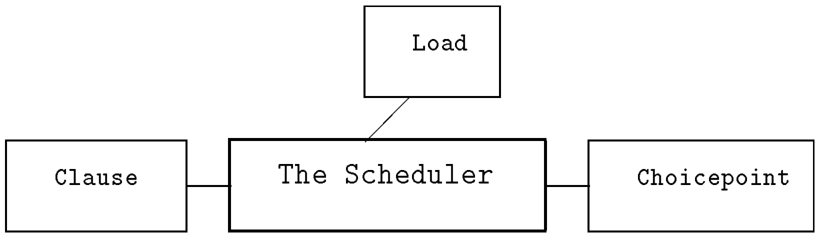


Figure 8.2: Macro categories provided by the Prolog engine.

Each shared frame contains scheduler specific information associated with the node. The scheduler also maintains the boundary between the shared and the private sections and tells the engine when it is updated.

If we compare this side of the Muse interface with the engine side of the Aurora interface, we find many differences between the two interfaces. In Aurora, all nodes are identical and maintained by the engine. The notion of *embryonic* nodes in Aurora does not exist in Muse. The representation of the search tree in Aurora is more general than the one used in Muse. Moreover the SRI model (used in Aurora) and the incremental stack-copying model (used in Muse) require different installation methods when moving down in the shared search tree. These differences explain why the engine side of the Muse interface is different from the engine side of the Aurora interface.

8.4.3 Sharing Macros

The main scheduler and engine macros used for starting and performing the sharing session are outlined in Figure 8.3. Q and P stand for a worker requesting sharing and a worker requested for sharing, respectively.

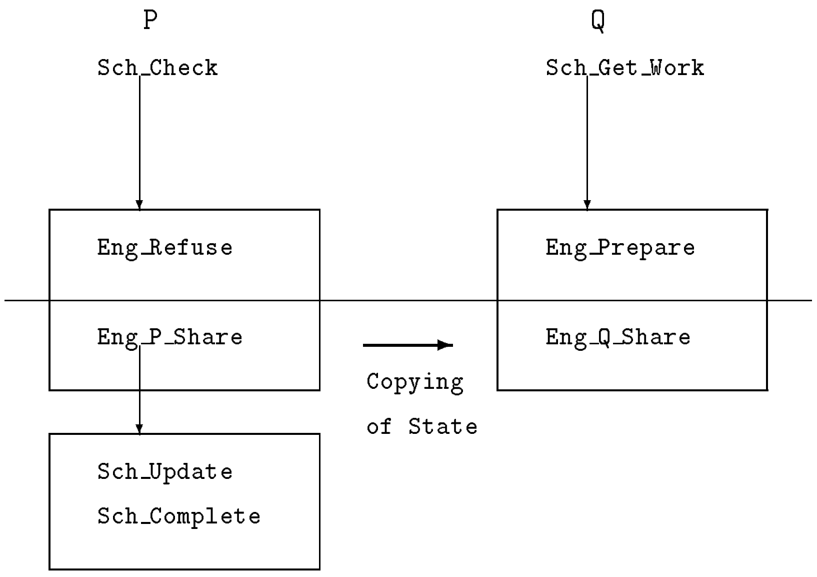


Figure 8.3: The sharing macros.

When Q's engine finishes processing its current piece of work (task), it calls the scheduler macro **Sch_Get_Work()** for finding a new job. If the scheduler cannot find work in the current branch and there exists another worker P with excess load, Q starts the sharing session with P as follows. The **Sch_Get_Work()** macro asks the engine to prepare for the sharing session through invoking the **Eng_Prepare()** macro and then it requests sharing from P, through invoking the **Eng_Q_Share()** macro. Upon a successful sharing session the **Eng_Q_Share()** macro returns the new top. Otherwise it returns zero.

The worker P detects the sharing request when its engine invokes the **Sch_Check()** macro. P may accept or refuse the sharing request. In the latter case, the **Sch_Check()** macro tells the engine to refuse the sharing request through invoking the **Eng_Refuse()** macro. Otherwise, the **Sch_Check()** macro starts the sharing session by invoking the **Eng_P_Share()** macro.

During the sharing session, the **Eng_P_Share()** macro will request the scheduler to add Q in all nodes that are already shared by P but not by Q through invoking the **Sch_Update()** macro. It will also request the scheduler to fill in the new shared choicepoints with the scheduler information through invoking the **Sch_Complete()** macro.

Copying of the state from P to Q will be performed by the two engines as a combined effort through the two macros **Eng_P_Share()** and **Eng_Q_Share()**.

8.5 The Interface

In this section we define the data structures used by the interface and the macros defined by the scheduler and the engine. Notice that a node is an abstraction that is implemented by a choicepoint and for a shared node, also a shared frame.

8.5.1 Data Structures

The Muse scheduler requires the engine to define the `struct` data structures shown in Figure 8.4. Those data structures can be used by the engine programmer for engine specific data. The data structures can be empty.

```
struct Eng_Private {
  Any data strictly private to the worker.

  Accessed by pw.Eng.'fieldname'.
};

struct Eng_Local {
  Any data stored locally that is remotely accessible.   One example
  is data used for communication at the sharing session.

  Accessed by lw->Eng.'fieldname'
        or by rw[worker_id]->Eng.'fieldname'.
};

struct Eng_Global {
  Any globally stored data.  One example is data that has to
  be globalized such as an atom table pointer (and its
  associated lock).

  Accessed by gw->Eng.'fieldname'.
};
```

Figure 8.4: Various convenient data structures.

The Muse scheduler also requires the engine to define the `struct` data structure shown in Figure 8.5. This data structure can be used by the engine programmer for storing engine specific data associated with each shared choicepoint. The data structure can (in principle) be empty, but we assume that the engine programmer stores the `next-clause-to-try` pointer in the `Eng_Stry_Node` structure.

Some data types are supposed to be known both to the Muse scheduler and to the Prolog engine. These are listed in Figure 8.6. The first column contains the C data type used. The second column gives a short form used in this chapter. The third column is used for a short description of the data types. It is important to understand that the Muse scheduler expect the Prolog engine to define the `struct`

```
struct Eng_Stry_Node {
  Any globally stored data associated with a shared frame.

  TRY *next-clause-to-try.

  accessed by 'framepointer'->Eng.'fieldname'.
};
```

Figure 8.5: The shared frame data structure.

`node` and the `struct try_node` data types. The Muse scheduler defines the `struct stry_node` data type.

Data type	Short form	Description
struct try_node	TRY	A Prolog clause.
struct node	CP	A Prolog choicepoint.
struct stry_node	STRY	A shared frame.
-	CODE	A piece of C code.
int	BOOL	An integer used as a boolean.
unsigned int	u_int	An unsigned integer.

Figure 8.6: Used data types.

8.5.2 Prolog Engine → Muse Scheduler

The Prolog engine can call some macros defined by the Muse scheduler, outlined in Figure 8.1. Some of the macros are mandatory to use and some are auxiliary. The auxiliary macros are marked with a †.

8.5.2.1 Macros for Communication with Other Workers

The Muse scheduler provides some macros for communication with other workers. The macros might move the border between private and shared choicepoints. This border is called the *top*. The engine may have to change its pointer to the youngest choicepoint (called *current-CP*) according to the top when the top changes.

void Sch_Check(CODE failcode)

This macro is called regularly (e.g. at every procedure call). The scheduler uses this macro to detect requests for sharing and requests to abort the current local execution. It is also used to implement the delayed release function.

When this macro detects a sharing request, the scheduler either refuses or accepts the request (Section 8.5.3.4). A successful sharing may change the top. In the case of a delayed release request the scheduler releases the current load after a fixed number of calls (Section 8.5.2.3). In the case of a legal request for abortion of the current local execution, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 8.5.2.4) and then it backtracks.

void Sch_Prune
(CP *scope, BOOL is_commit, CODE failcode, CODE prunecode)

This macro is called before executing a cut instruction. The scheduler uses the macro to perform pruning of subtrees that would not have been visited in a sequential execution. It also requests abortion of the work of workers that are staying in those subtrees. This macro generally sets the value of the local load register.

The **scope**[1] of the cut operation is provided. This parameter is both input and output. A boolean flag (called **is_commit**) that is true if the cut operation shall be treated as cavalier commit, and false otherwise, is also provided.

A pruning operation is performed only when the **scope** reaches into the shared region. At a successful pruning operation the top may be changed, the **scope** may be changed, and then the code **prunecode** is executed. The **scope** value, when executing **prunecode**, is the same as the output value for **scope**. The pruning operation may be aborted by another pruning operation. In this case, the top may be changed, the **scope** is set to 0, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 8.5.2.4) and then it backtracks.

When the cut operation is local the **scope** value is unchanged. In the case of a local cut or a successful pruning operation, the output **scope** value is used when performing the local cut operation.

void Sch_Synch(CP *scope, CODE failcode)

This macro is called whenever the engine cannot continue forward execution until it becomes leftmost in a search tree (e.g. before a side effect). It is used to maintain the (sequential) Prolog semantics. No side effect is executed until the current branch is leftmost in a proper subtree. Most side effects (e.g. write, assert) use the search tree defined by the whole computation, but closed sub-computations (e.g. bagof) can use a subtree defined by the computation.

The **scope**[2] of a proper subtree is provided. In case of a legal request for abortion of the current local execution, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 8.5.2.4) and then it backtracks.

[1]A pointer to the youngest choicepoint that is alive after performing the cut operation

[2]A pointer to the choicepoint which is the common root that defines the subtree

8.5.2.2 Macros for Finding Work

The Muse scheduler provides some macros for finding work when the engine backtracks to a shared choicepoint. The macros might move the top. Whenever work is found the engine is supposed to set the program counter (or equivalent) according to a returned clause pointer.

void Sch_Get_Work(CODE failcode, TRY *try)

The macro is called whenever the engine backtracks to a shared choicepoint. It is a very central macro to the Muse scheduler. (As an optimization this macro consists of a set of three macros **Sch_Get_Work_Parallel()**, **Sch_Get_Work_Sequential()**, and **Sch_Get_Work_Root()**. They all have the same syntax and semantics. But the Muse scheduler expects the Prolog engine to call the correct one depending on the current shared choicepoint type.) It is used to find shared work. The scheduler may take shared work from the current (shared) choicepoint or it may move to an older choicepoint containing shared work or it may, with the help of another worker, install a new environment extending its branch. When no shared work can be found the worker may stay at the choicepoint or move to an older choicepoint for a better position.

There are several reasons for the engine to backtrack after the return from this macro; the scheduler wants to move up in the tree; a pruning request has been detected; backtracking for job after a successful sharing session. In all cases, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 8.5.2.4) and then it backtracks.

The scheduler may find job in the current choicepoint. The engine, for that purpose, provides an output field of the type **TRY**, a clause pointer. The scheduler fills in that field with a pointer to the found clause. The engine is supposed to execute that clause. Then, if the last alternative in a sequential choicepoint is taken, top is changed. The engine changes its current-CP accordingly.

8.5.2.3 Macros for Events of Interest to the Scheduler

The Prolog engine is supposed to inform the Muse scheduler about some events. Currently the only event is change of the current load. For more efficient handling of cut this group of macros might grow substantially.

void Sch_Set_Load(u_int load, BOOL stable_load)

This macro is called whenever creating a (complete) choicepoint. Then the macro informs the scheduler about both the current load and whether the current load is stable or not. Stability is defined, in the current Muse implementation, as having an older parallel private choicepoint (with extra alternatives). The purpose of providing information about the stability is to avoid showing excess load that may disappear after a very short period of time.

The **load** argument contains the current load. This value is (normally) copied to the load register. The **stable_load** argument contains a boolean stating whether the provided load shall be regarded as stable or not. The exact definition of **stable_load** can be different from one engine implementation to another, but it shall reflect whether there already did exist any local load before calling the macro. (The **stable_load** argument can contain the previous load value. This is consistent with using a C boolean.) If the load is not stable the release of the load is delayed for a fixed number of calls to the macro **Sch_Check()**.

8.5.2.4 Macros for Asking the Scheduler

Some of the scheduler information is relevant to the engine. In this group of macros the engine asks the scheduler to get such information. So far, there is only one macro. It is used for getting a pointer to the current top. In contrast to Aurora, Muse stores information about the current top (the boundary between the shared and the private regions) in the scheduler part.

CP *Sch_Get_Top()

The Prolog engine calls this macro each time it needs to update the engine state according to a (possibly) new top.

This macro returns a pointer to the current top, the youngest shared choicepoint. Calling this macro has no effect on the Muse scheduler.

8.5.2.5 Sharing Macros

As described in Section 8.4.3, the scheduler can find another worker to get shared work from through the macro **Sch_Get_Work()**. The scheduler then request sharing from the other worker. We call the requesting worker Q and the other worker P. The worker P detects the request in the macro **Sch_Check()**.

void Sch_P_Update_Nodes(CP *old_ptop, CP *qtop, u_int q_id)

This macro is called from **Eng_P_Share()** during the sharing session. It adds Q as a member to all already shared choicepoints accessed by P but not by Q.

Three arguments are provided. The **old_ptop** is the old top for P, the **qtop** is Q's top, and **q_id** is the name of Q.

void Sch_P_Complete_Stry(STRY *stry, CP *next_job)

This macro is called from **Eng_P_Share()** for every choicepoint that is made shared by P. The purpose of the macro is to fill in the extra scheduler fields in the newly allocated `STRY` structure.

A pointer to the newly allocated `STRY` (**stry**) is provided. A pointer to a good choicepoint (**next_job**), to start looking for work (when running out of work in the current choicepoint) is also provided. This choicepoint must be older than the current choicepoint and not older than the nearest parallel choicepoint that contains available work. In the current Muse implementation the **next_job** is the parent of the current choicepoint.

8.5.2.6 Initialization Macros

The start up of the Muse system consists of a number of steps. The start up (master) process allocates shared memory (including the stack areas) for all processes. It performs initializations of the Prolog engine (and the scheduler) before executing any Prolog code. The Prolog system is loaded using the master process, the root choicepoint is made shareable, the stacks are copied to all other workers, and then the processes are forked. The master then waits until all other processes are created, initialized correctly, and sharing the root choicepoint. Thereafter the system is ready.

void Sch_Init(u_int num_workers)

This macro is called by the master process (worker id 0) before running any engine code and before forking any other worker, but after allocating and setting up of shared memory and creating the initial choicepoint (older than the root). The number of workers (**num_workers**) is provided.

void Sch_Add_Remote(u_int wid)

This macro is called by the master process just before fork, for every other worker. The worker id (**wid**) is provided.

void Sch_Add_Private(u_int wid, u_int num_workers)

This macro is called by each worker, except the start up worker, just after fork. The worker id (**wid**) and the number of workers (**num_workers**) are provided.

void Sch_Wait_For_Forked_Children()

This macro is called by the start up worker just after fork and before continuing running Prolog. The purpose is to wait until all other workers have been started.

void Sch_P_Initial_Share()

This macro is called just before the initialization macro **Sch_Add_Remote()**. The purpose is to make the root choicepoint shared.

void Sch_Q_Initial_Share()

This macro is called just after calling the initialization macro **Sch_Add_Private()**. The purpose is to propagate information about the shared root choicepoint.

8.5.2.7 † Auxiliary Macros

The following macros are not necessary to use. The Scheduler provides them for auxiliary use.

† void Sch_Set_Load_Now(u_int load)

When the engine wants to (temporarily) make itself uninteresting (e.g. at GC) it calls this macro with a load of 0. When the engine wants to make itself interesting again it calls this macro with the current load. The macro can replace the **Sch_Set_Load()** macro, if delayed release is not wanted.

† u_int Sch_Num_Workers()

This macro returns an unsigned integer indicating the number of workers currently in the system.

† u_int Sch_Worker_Id()

This macro returns an unsigned integer indicating the worker identification. The identification is in the range $\langle 0 \ldots Workers - 1\rangle$.

† void Sch_Write
(NODE *scope, STRY *stry, CODE execcode, CODE savecode,
CODE failcode,BOOL retry)

This macro can be used as an optimization for write type side effects (e.g. assert, write, and saving solutions in findall). Instead of using **Sch_Synch()** to suspend the execution of the current task non-leftmost workers can save the side effect for later execution.

The **scope** and the **failcode** have the same meanings as for **Sch_Synch()**. The `STRY` **stry** is an output field pointing to an appropriate `STRY` to save non-executed side effects in. It is only valid iff the `CODE` **savecode** is executed. The `CODE` **execcode** is the code to execute iff the side effect can be executed. The `CODE` **savecode** is the code to execute iff the side effect has to be saved until later execution. The boolean flag **retry** is true iff the call to the macro is made after claiming saved side effects.

The scheduler needs some help from the engine, so if this macro is used the engine must implement the macros **Eng_Any_Saved_Writes()** and **Eng_Propagate_Writes()**.

8.5.3 Muse Scheduler → Prolog Engine

The Prolog engine provides some macros for the Muse scheduler, outlined in Figure 8.2 (page 166). Most of the macros are mandatory to define. But some, if not defined, are replaced with dummy macros that always return a default value. Macros that do not have to be implemented are marked with a †.

8.5.3.1 Load Macros

It is up to the Prolog engine to define and maintain the current excess load. The engine provides a macro that can return such a value. The load shall be a measure of the amount of untried local work that can be found via backtracking (e.g. the number of untried local alternatives).

u_int Eng_Get_Load()

This macro returns the current excess local load. When no load exists the macro returns the value 0. The macro is used for some important scheduler optimizations.

8.5.3.2 Clause Macros

Some macros are used for getting and manipulating available alternatives in shared choicepoints. For this purpose, the engine maintains the `next-clause-to-try` field for shared choicepoints, in the associated `STRY` structure shown in Figure 8.5 (page 169). The Muse scheduler must be able to get the current `next-clause-to-try` for shared choicepoints. The scheduler must also be able to advance it to the next alternative and to clear it. Clearing corresponds to executing a cut operation in the sequential Prolog engine.

Other macros are used for implementing the leftmost (in the search tree) check. For this purpose, the engine for each worker in a parallel shared choicepoint maintains information about which alternative is taken. This information shall be accessible to the scheduler. The scheduler must also be able to clear the value, indicating that no alternative is taken. The value returned by **Eng_My_Alternative()** and **Eng_Remote_Alternative()** is 0 for sequential choicepoints and when no alternative is currently taken.

TRY *Eng_Get_Next_Alternative(STRY *stry)

This macro returns a pointer to the `next-clause-to-try` for a shared choicepoint, referenced by **stry**. A returned null pointer means that there are no more alternatives. The `next-clause-to-try` value shall *not* be advanced.

void Eng_Adv_Next_Alternative(STRY *stry)

This macro tells the engine to advance the `next-clause-to-try` for the shared choicepoint referenced by **stry**.

void Eng_Clear_Next_Alternative(STRY *stry)

This macro tells the engine to mark the shared choicepoint referenced by **stry** as empty. All subsequent calls to **Eng_Get_Next_Alternative()** for this choicepoint, shall return the value null.

u_int Eng_My_Alternative(STRY *stry)

The purpose of this macro is to give an ordering of taken alternatives. For sequential choicepoints this ordering is not interesting as only one alternative can be taken. This macro is very central for implementing the sequential Prolog semantics.

This macro returns an unsigned integer representing my-current-alternative for a shared choicepoint referenced by **stry**. A returned non-zero value means that I have currently taken a alternative. For parallel choicepoints the value then returned is one monotonically decreasing with the corresponding clause number. For sequential choicepoints the value is not important as long as the same value is always returned for a alternative. A returned 0 means that I have currently not taken any alternative in this choicepoint.

u_int Eng_Remote_Alternative(STRY *stry, u_int workerid)

This macro is equivalent to the previous one, except for returning the value for a worker given by **workerid**.

void Eng_Take_Alternative(STRY *stry, TRY *try)

This macro informs the engine that the scheduler has reserved the clause **try** in the parallel shared choicepoint referenced by **stry**. The engine then updates the my-current-alternative value accordingly.

void Eng_Take_No_Alternative(STRY *stry)

This macro informs the engine that the scheduler has not taken any alternative in the parallel shared choicepoint referenced by **stry**. The engine updates the my-current-alternative value to 0.

8.5.3.3 Choicepoint Macros

Only the Prolog engine knows the choicepoint representation. The engine therefore has to provide some macros for the Muse scheduler. The **CP** type is a choicepoint reference (usually an address). The choicepoint stack may be moved, so using a **CP** reference in the scheduler might not always be a good idea. The engine provides a location independent choicepoint reference called **offset**. Conversion macros back and forth between the **CP** and the **offset** representations are provided. The engine also provides age comparison macros both for the **CP** and the **offset** representations. Some information about a choicepoint, such as its parent, is also provided by the engine.

The set of macros concerning the age of choicepoints, direct or via offset, may look somewhat arbitrarily chosen. Only those actually needed by the scheduler are described. The set of macros could be made more complete if desired.

The engine maintains means to get pointers to the child and the shared frame of a shared choicepoint.

CP *Eng_Parent(CP *cp)

This macro returns the parent of a choicepoint.

BOOL Eng_Is_Younger_CP(CP *cp1, CP *cp2)

This boolean macro returns true, iff the choicepoint **cp1** is younger than the choicepoint **cp2**.

CP *Eng_Get_Oldest_CP()

This macro returns a pointer to the oldest existing choicepoint. This choicepoint must be older than the oldest shared choicepoint (called root).

CP *Eng_Get_Youngest_CP()

This macro returns a pointer to the youngest choicepoint now existing in the engine.

u_int Eng_CP_To_Offset(CP *cp)

This macro converts from a **CP** pointer to a Prolog engine independent offset. The offset for a given choicepoint must never change during the execution and it must also be an unique value for all choicepoints in the same branch.

CP *Eng_Offset_To_CP(u_int offset)

This macro converts from a Prolog engine independent offset to a **CP** pointer.

BOOL Eng_Is_Older_Offset(u_int o1, u_int o2)

This macro returns the value true, iff the offset **o1** represents an older choicepoint than the offset **o2**.

u_int Eng_Infinitely_Young_Offset()

This macro returns an offset that represents a choicepoint that is younger than any real choicepoint.

CP *Eng_Child(CP *cp)

This macro returns the (shared) child of the shared choicepoint **cp**. The scheduler calls this macro only for shared choicepoints that have a shared choicepoint as child. This restriction makes it possible for the engine in the SICStus-Muse system to use the same choicepoint field for calculating the local load and as child reference.

STRY *Eng_Stry(CP *cp)

This macro returns a pointer to the shared frame associated with the shared choice-point **cp**. The scheduler calls this macro only for shared choicepoints.

BOOL Eng_Is_Parallel(CP *cp)

This macro returns true iff the predicate that has created the choicepoint **cp** is a parallel predicate.

8.5.3.4 Sharing Macros

The workers P and Q are defined in Section 8.5.2.5.

void Eng_Q_Prepare_For_Sharing(u_int wid)

This macro prepares for the protocol used by the engine at the sharing session. It is called from **Sch_Get_Work()** by Q before the scheduler sends the request for sharing request to P. The worker id (**wid**) for P is provided.

CP *Eng_Q_Share(u_int wid)

This macro is called from **Sch_Get_Work()** by Q after the sending of a request to P (**wid**) for sharing request. After a successful sharing session it returns the new top. If sharing was refused it returns 0.

CP *Eng_P_Share(u_int load, CP *cp, u_int wid)

This macro is called from **Sch_Check()** by P when detecting a request for sharing. The correct **load** and an id to Q (**wid**) are provided. If **load** is zero then the scheduler may have found work in some shared choicepoint. A pointer to this choicepoint is provided in **cp**. At a successful sharing session the macro returns a pointer to the new top. If sharing was refused it returns 0. In that case, the scheduler calls the macro **Eng_P_Refuse_Sharing_Request()**.

void Eng_P_Refuse_Sharing_Request(u_int wid)

This macro is called from **Sch_Check()** by P when a detected request for sharing is to be refused. The id of Q is provided.

void Eng_Stry_Dealloc(STRY *stry)

This macro is called when the scheduler wants to deallocate the STRY **stry**.

8.5.3.5 Initialization Macros

The role of these macros is described in Section 8.5.2.6.

CP *Eng_P_Initial_Share_Root(u_int wid)

This macro is called from **Sch_P_Initial_Share()**. The id of the start up worker (**wid**) is provided. The macro makes the root choicepoint shared and then returns a pointer to it.

void Eng_P_Initial_Copy_Stacks()

This macro is called from **Sch_P_Initial_Share()** for each worker other than the start up worker. The id of the other worker (**wid**) is provided. The macro performs the operations by P that are necessary to copy the state from P to Q.

CP *Eng_Q_Initial_Copy_Reg()

This macro is called from **Sch_Q_Initial_Share()** by each worker other than the start up worker. The macro performs the operations by Q that are necessary to copy the state from P to Q.

8.5.3.6 † Auxiliary Macros

These macros are used to implement non-Prolog features. The engine can choose not to implement them. They are then automatically replaced with dummy macros that return default values.

† BOOL Eng_Is_Atomic()

This macro returns the value true iff the Prolog engine is in atomic mode. The default value is false. This engine mode makes the scheduler ignore requests in the macro **Sch_Check()**. The engine shall only be in this mode for short periods and only when executing Prolog code with exact one solution.

There may be some use of this macro besides implementing non-Prolog features. The Prolog engine may have to force a sequence of Prolog predicate calls to be made as an atomic operation. In the SICStus implementation there exists a special metacall for this purpose. It is implemented as ...

```
atomic_call(Goal) :- atomic_on, call(Goal), atomic_off.
```

... where the built-in predicates `atomic_on` and `atomic_off` sets and resets the atomic flag. Remember the limitations for the goal `Goal`. It is not a good idea to make the predicate `atomic_call/1` public.

† BOOL Eng_Is_Mutex()

This macro returns the value true iff the Prolog engine is in mutex mode. The default value is false. If the engine implements the mutex function, only one worker at a time shall be able to return the value true to this macro. (If this macro returns true then the engine must not call the macro **Sch_Synch()** with a scope that may suspend current job. This to avoid deadlock.)

† BOOL Eng_Is_Mutex_Remote(u_int workerid)

This macro returns the value true iff the Prolog engine of the worker with id **workerid** is in mutex mode. The default value is false.

† BOOL Eng_No_Sharing()

This macro returns the value true iff the Prolog engine does not allow sharing. The default value is false.

† BOOL Eng_No_Sharing_Remote(u_int workerid)

This macro returns the value true iff the Prolog engine of the worker with id **workerid** does not allow sharing. The default value is false.

† BOOL Eng_Any_Saved_Writes(STRY *stry)

This macro returns the value true iff there exists any saved side effects in the `STRY` **stry**. When the **Sch_Write()** optimization is not used then the macro returns the default value false

† BOOL Eng_Propagate_Writes(STRY *stry, u_int alternative)

This macro is called when the scheduler detects (with **Eng_Any_Saved_Writes()** that the `STRY` **stry** may contain saved side effects. The **alternative** argument contains the alternative number for the leftmost alternative in the shared choicepoint referred to by **stry**. (The `STRY` **stry** is locked when the calling the macro.) For all saved side effects the macro calls **Sch_Write()** to propagate the side effects further in the tree or eventually perform the side effects. When the **Sch_Write()** optimization is not used then the macro is a dummy macro, expanding to the empty macro.

8.6 Conclusions

The engine-scheduler interface in the Muse system has been described. This interface is working in the current Muse system, which is based on SICStus0.6 Prolog engine with the Muse scheduler. Performance results on six different machines indicate that the interface has not introduced any negative effect on the performance results. As a result of the interface work, the Muse code becomes more readable and redundant code has been discovered and removed.

It remains to establish that the engine-scheduler interface documented here fits other Prolog engines. There is ongoing work interfacing the sequential BIM Prolog engine with the Muse scheduler using this interface. There exist important differences between the BIM Prolog engine and the SICStus Prolog engine (e.g in the representation and manipulation of the Prolog stacks and the Prolog code). So, we are expecting further feedback from the BIM group which may lead to some further refinements in the interface definition.

8.7 Acknowledgments

This work is based on the Aurora interface as described by Péter Szeredi (from the University of Bristol) and Mats Carlsson (from the Swedish Institute of Computer Science) in [84].

The work is made possible by the funding by the PEPMA project (ESPRIT project 2471) and by the SICS sponsors. The discussions with André Mariën (from the BIM Prolog company) have also been of good help.

The sponsors of SICS are Asea Brown Boveri AB, NobelTech Systems AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defence Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

Chapter 9

Conclusions

We have studied techniques for efficient OR-parallel implementation of the full Prolog language on both UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access) computers. A simple and efficient execution model, Muse, based on extending the state-of-the-art sequential Prolog implementation, WAM, has been developed. The Muse model assumes a number of extended WAMs (called workers), each with its own local address space, and some global space shared by all workers. The model requires copying parts of the WAM stacks when sharing work. The copying operation is made efficient by utilizing the stack organization of the WAM. Workers make a number of choicepoints shareable on each copying operation, and they get work from those shared choicepoints (nodes) through the normal backtracking mechanism of Prolog. Each node is associated with a global frame containing a lock, available work, and scheduling information. A worker consists of two components: the *engine*, which performs the actual Prolog work, and the *scheduler*. The schedulers, working together, divide the available work between engines and support the sequential semantics of Prolog.

⋆ ⋆ ★ ⋆ ⋆

The first Muse engine was produced by extending the SICStus Prolog version 0.6. Extensions have been carefully added to preserve the high efficiency of SICStus leading to a negligible overhead which is significantly lower than in other OR-parallel models.

Efficient algorithms for scheduling work have been developed. Dynamic load balancing and efficient grabbing of work in the current branch is achieved by making several nodes shareable at a time and dispatching the work at the *bottommost* node in a branch. The cost of the task switching is reduced by making busy workers invisible to all but the nearest idle worker. This also prevents more than one idle worker from getting work from the same busy worker. The task size is maximized by idle workers getting work from busy workers with maximum excess load. Making very small tasks shareable is avoided through a run-time method for controlling the granularity of jobs.

An idea for supporting cut, first proposed in [24], has been further developed and efficiently implemented in the Muse scheduler. The handling of side effects that cannot be executed immediately has been implemented in two different ways. For *read* and *retract* the execution suspends and the worker is busy waiting until the worker's branch becomes leftmost. For *write* and *assert* a novel and very efficient method has been developed. A side effect that a worker is unable to execute is temporarily saved in a suitable node on the search tree, and then the worker proceeds as if the side effect was executed. The same algorithm has been used for saving the solutions in the *findall* predicate, showing very promising results and also preserving the strict sequential ordering of the produced list of solutions. The predicates *bagof* and *setof* are implemented similarly.

The Muse system has been evaluated using a large set of benchmarks including those used for evaluating similar systems. The performance results have been compared with the results of another state-of-the-art OR-parallel Prolog system (Aurora) for the same set of benchmarks and on the same platforms. The comparison shows that the Muse system outperforms the Aurora system on all platforms used. The difference in performance is greater on the NUMA than on the UMA machines. This is due to the high locality of memory references in the Muse model. The Muse system is the most efficient OR-parallel Prolog system existing today.

An interface between a Prolog engine based on WAM and the Muse scheduler has been defined. The design was influenced by the Aurora interface [84]. The interface is used in the current Muse system, greatly enhancing the structure of the code and still preserving the efficiency of the system. It is also being used for extending the "ProLog by BIM" Prolog engine according to the Muse model.

The Muse system has been implemented on six different multiprocessor machines: a VME system, Sequent Symmetry, TP881V from Tadpole Technology, Sun Galaxy, BBN Butterfly I, and BBN Butterfly II. Tuning the Muse system for each architecture has been discussed. In order to make the Muse system maintainable and portable machine dependent stuff is grouped and separated from the rest of the code. Important implementation details and how to extend any sequential Prolog engine based on WAM to OR-parallel execution using the Muse model have been presented.

Tools for debugging and evaluating the Muse system have been developed. Without such tools it is very difficult to produce an efficient parallel system.

In the near future we plan to integrate the Muse system with the latest vesion of SICStus, with native code generation (version 2.1). The enhancements discussed in Chapter 7 will be considered. We also plan to exploit both OR-parallelism and independent AND-parallelism using an idea found in [38].

Chapter 10

Epilog

THE Muse model has been described in this thesis and the performance of the implementation has been analyzed. The main conclusion is clear: *the stack based incremental copying model is well suited for implementing OR-parallel Prolog.* The results achieved by the Muse system are more than competitive with those of systems using other approaches. In the following two sections I will try to estimate (although this is difficult) the future potential of the model.

10.1 Sequential Execution

The hardware evolution of general purpose processor chips is very impressive. The speed (measured in MIPS or whatever unit you prefer) is rapidly increasing each year, as a result of improved technology and better designs. The research on Prolog program analysis and programming techniques also shows very promising results. Compilation to machine code results in substantial performance improvements in comparison with WAM based emulators. This makes the speed of Prolog programs competitive with the speed of C programs. *Prolog is here to stay.*

10.2 Parallel Execution

There are two requirements that have to be fulfilled to guarantee the success of OR-parallel Prolog systems: (1) there must exist price/performance cost effective shared memory multiprocessors and (2) the OR-parallel Prolog system must be able to show interesting speed-ups on those machines. How many foresaw the current situation, regarding the development of computers, 20 years ago? Not many. I shall nevertheless try to make some predictions.

To get maximum performance a small set of tightly coupled very efficient processors can be used (as in the Sun Galaxy machine using up to 8 75 MIPS Viking SPARC-processors) or a huge number of (maybe not so fast) loosely coupled processors (like the CM5 connection machine). I think that both type of machines will survive for a long time. The former type of machine is potentially suited for OR-parallel Prolog. The speed of processors is rapidly increasing but the speed of shared memory does not increase at the same rate. This leads to more memory being moved closer to the processor (i.e. the caches get bigger). The scheduling activities of the OR-parallel Prolog system involve frequent access to the shared memory, making the scheduler slower relative to the Prolog system.

⋆ ⋆ ★ ⋆ ⋆

I do think that the Muse OR-parallel model is very well suited to cope with the challenge of having faster processors and better Prolog compilers. Its simplicity makes it easy to add OR-parallelism to any new efficient Prolog system. The low overhead introduced (it can theoretically approach zero) results in speed ups being *real* speed ups. The results achieved on the Butterfly machines (which have a big overhead for scheduler activities) show that the increased gap between the efficient Prolog system and the scheduler for the Muse system is of little importance. The Prolog company BIM is currently using the Muse model to make a commercial product running on Sun Galaxy. *Parallel Prolog is here to stay.*

⋆ ⋆ ★ ⋆ ⋆

References

[1] Khayri A. M. Ali. OR-parallel Execution of Horn Clause Programs Based on WAM and Shared Control Information. SICS Research Report R88010, Swedish Institute of Computer Science, November 1986.

[2] Khayri A. M. Ali. OR-parallel Execution of Prolog on a Multi-Sequential Machine. *International Journal of Parallel Programming*, 15(3):189 – 214, June 1986.

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

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

[5] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445 – 475, December 1990.

[6] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[7] Khayri A. M. Ali and Roland Karlsson. The Muse OR-parallel Prolog Model and its Performance. In *the Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[8] Khayri A. M. Ali and Roland Karlsson. MUSE: A Parallel Prolog System. In *the Proceedings of the Al-azhar Engineering Second International Conference*, volume XI, pages 271 – 282, December 1991.

[9] Khayri A. M. Ali and Roland Karlsson. Scheduling OR-parallelism in Muse. In *the Proceedings of the 1991 International Conference on Logic Programming*, Paris, June 1991.

[10] Khayri A. M. Ali and Roland Karlsson. OR-parallel Speedups in a Knowledge Based System: on Muse and Aurora. In *the Proceedings of FGCS'92 (the International Conference on Fifth Generation Computer Systems*, June 1992.

[11] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. *New Generation Computing*, 11(1):81–103, 1992.

[12] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of Muse on the BBN Butterfly TC2000. In *the Proceedings of PARLE'92 (Parallel Architectures and Languages Europe)*, June 1992.

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

[14] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *the Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 841 – 850. ICOT, November 1988.

[15] Anthony Beaumont. Scheduling Strategies and Speculative Work. In *the Proceedings of ICLP91 (International Conferance on Logic Programming) Preconferance Workshop on Parallel Execution of Logic Programs*, June 1991.

[16] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I: An Implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, July 1990, University of Oregon.

[17] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Scheduling OR-Parallelism in Aurora with the Bristol Scheduler. Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.

[18] BIM. ProLog by BIM release 3.0. 3078 Everberg, Belgium, November 1990.

[19] 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 *the Proceedings of the Fifth International Conference' and Symposium on Logic Programming*, pages 1160–1179. MIT Press, August 1988.

[20] P. Borgwardt. Parallel Prolog using Stack Segments on Shared-Memory Multiprocessors. In *the Proceedings of the International Symposium on Logic Programming*. IEEE Computer Society, 1984.

[21] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[22] 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 on Logic Programming*, pages 1590–1605. MIT Press, August 1988.

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

[24] 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.

[25] Mats Carlsson, Ken Danhof, and Ross Overbeek. A Simplified Approach to the Implementation of AND-parallelism in an OR-parallel Environment. In *the Proceedings of the Fifth International Conference on Logic Programming*, pages 1565–1577. MIT Press, August 1988.

[26] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual (for version 0.6). SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.

[27] M. Carro, L. Gomez, and M. Hermenegildo. VISANDOR: A Tool for Visualizing And- and Or-parallelism in Logic Programs. Technical report, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, 1991.

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

[29] John Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.

[30] John Conery. Binding Environments for Parallel Logic Programs in Non-shared Memory Multiprocessors. *International Journal of Parallel Programming*, 17(2):125–152, April 1988.

[31] Quintus Corporation. Manual for Quintus Prolog release 3.1. February 1991.

[32] Doug DeGroot. Restricted AND-Parallelism. In Hideo Aiso, editor, *International Conference on Fifth Generation Computer Systems 1984*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[33] Terrence 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.

[34] Terrence Disz, Ewing Lusk, and Ross Overbeek. Experiments with OR-parallel Logic Programs. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 576–600. MIT Press, 1987.

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

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

[37] Gopal Gupta, Vitor Santos Costa, Rong Yang, and Manuel V. Hermenegildo. IDIOM: Integrating Dependent AND-, Independent AND-, and OR-parallelism. In *the Proceedings of the 1991 ISLP (International Symposium on Logic Programming)*, 1991.

[38] Gopal Gupta and Manuel Hermenegildo. ACE: AND/OR-parallel Copying-based Implementation of Logic Programs. In *the Proceedings of the ICLP '91 Pre-conference Workshop on Parallel Execution of Logic Programs*, pages 146–158. Springer-Verlag, June 1991.

[39] Gopal Gupta and Bharat Jayaraman. Compiled AND-OR Parallelism on Shared Memory Multiprocessors. In *the Proceedings of the 1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[40] Gopal Gupta and Bharat Jayaraman. On Criteria for OR-parallel Execution Models of Logic Programs. In *the Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756. MIT Press, October 1990.

[41] G. Hagert, F. Holmgren, M. Lidell, and K. Orsvärn. On Methods for Developing Knowledge Systems — an Example in Electronics, 1988. Mekanresultat 88003 (in Swedish).

[42] Seif Haridi and Per Brand. Andorra Prolog—an integration of Prolog and committed choice languages. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

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

[44] Bogumił Hausman. Handling of Speculative Work in OR-parallel Prolog: Evaluation Results. In *the Proceedings of the 1990 North American Conference on Logic Programming*, pages 721–736. MIT Press, October 1990.

[45] Bogumił Hausman. *Pruning and Speculative Work in OR-parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[46] Bogumił Hausman, Andrzej Ciepielewski, and Alan Calderwood. Cut and Side-Effects in OR-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*. ICOT, 1988.

[47] Bogumił 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.

[48] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[49] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[50] 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.

[51] 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 on Logic Programming*, pages 263–279. MIT Press, August 1988.

[52] Fredrik Holmgren and Klas Orsvärn. Towards a Domain Specific Shell for Design Rule Checking. In *the Proceedings of the IFIP TC 10/WG10.2 Working Conference on the CAD Systems Using AI Techniques*, pages 221–228, Tokyo, June 1989.

[53] Sverker Janson, Roland Karlsson, and Khayri A. M. Ali. AND-in-OR using Findall Revisited. SICS Research Report (in preparation), Swedish Institute of Computer Science, 1992.

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

[55] 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.

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

[57] Roland Karlsson. How to Build Your Own OR-parallel Prolog System. SICS Research Report R92:03, Swedish Institute of Computer Science, March 1992.

[58] Roland Karlsson and Khayri A. M. Ali. The Engine-Scheduler Interface used in the Muse OR-parallel Prolog System. SICS Research Report R92:04, Swedish Institute of Computer Science, March 1992.

[59] Robert Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, Amsterdam, 1979.

[60] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma. KABU-WAKE: A New Parallel Inference Method and Its Evaluation. *COMPCON Spring 86*, 1986.

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

[62] T. Lindholm and R. A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In *the Proceedings of the Fourth International Conference on Logic Programming*, pages 21–39. MIT Press, 1987.

[63] Ewing Lusk. Remote I/O Handling Package specification. Internal Report, Gigalips Project, October 1989.

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

[65] 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.

[66] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. Sepia - an extendible Prolog system. In *the Proceedings of the 11th World Computer Congress IFIP'89*, San Fransisco, August 1989.

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

[68] Shyam Mudambi. Performance of Aurora on NUMA machines. In *the Proceedings of the 1991 International Conference on Logic Programming*, pages 793–806. MIT Press, June 1991.

[69] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[70] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

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

[72] M. Pazzani, T. Cain, B. Silver, and K.C. Wong. BIM [Prolog by BIM] is an impressive Prolog with several good features. *AI Expert Magazine*, page 46, January 1991.

[73] Balkrishna Ramkumar. *Machine Independent "AND" and "OR" Parallel Execution of Logic Programs.* PhD thesis, Univerity of Illinois at Urbana-Champaign, 1991.

[74] Michael Ratcliffe. A Progress Report on PEPSys. Presentation at the Gigalips Workshop, Manchester, July 1988.

[75] Dan Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1991.

[76] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both AND- and OR-Parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, April 1991.

[77] Ehud Shapiro. An OR-parallel Execution Algorithm for Prolog and its FCP Implementation. In *the Proceedings of the Fourth International Conference on Logic Programming*, pages 311–337. MIT Press, 1987.

[78] Ehud Shapiro, editor. *Concurrent Prolog—Collected Papers.* MIT Press, 1987.

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

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

[81] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[82] Jan Sundberg and Claes Svensson. MUSE TRACE: A Graphic Tracer for OR-parallel Prolog. SICS Technical Report T90003, Swedish Institute of Computer Science, 1990.

[83] Péter Szeredi. Performance Analysis of the Aurora OR-parallel Prolog System. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[84] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing Engines and Schedulers in OR-parallel Prolog Systems. In *PARLE 91, Conference on Parallel Architectures and Languages Europe.* Springer-Verlag, June 1991.

[85] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, 1991.

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

[87] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, University of California, 1990. Available as Report No. UCB/CSD 90/600.

[88] Peter Lodewijk Van Roy and Alvin M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, pages 54–68, January 1992.

[89] D. S. Warren. Efficient Prolog Management for Flexible Control Strategies. In *the Proceedings of the International Symposium on Logic Programming*. IEEE Computer Society, 1984.

[90] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

[91] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[92] 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.

[93] Patrick Weemeeuw. Garbage Collection for the Aurora System: A feasibility study, September 1991. Draft version published at the PEPMA (ESPRIT project 2471) review meeting.

[94] Harald Westphal, Philippe Robert, Jacques Chassin, and Jean-Claude Syre. The PEPSys model: combining backtracking, AND- and OR-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*. IEEE, 1987.

[95] H. Yasuhara and K. Nitadori. ORBIT: A Parallel Computing Model of Prolog. *New Generation Computing*, 2(3):277–288, 1984.

Printed in Malmö, Sweden by Graphics Systems using a camera-ready copy typeset in Computer Modern by the author, using a LaserWriter II, the LaTeX document preparation system, the emacs editor, and a SPARCstation 1 workstation.