SICS

How to Build your own OR-parallel Prolog System

Roland Karlsson

SICS research report
R92:03
ISSN 0283-3638

How to Build your own OR-parallel Prolog System

March 1992

Roland Karlsson

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

THIS paper 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.

Contents

1 Introduction

The goal of this paper 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 paper can also be seen as an introduction to the Muse interface paper [17].

Here we assume the reader is familiar with the WAM [26] and with the Muse model [3, 2].

We start in Section 2 giving some terminology used in this paper. Section 3 describes our layered approach for gradually extending any sequential Prolog system to an OR-parallel (full) Prolog system.

Sections 4, 5, and 6 show how to extend a sequential Prolog system to a OR-parallel (pure) Prolog system. Section 4 describes very briefly the Muse model and the required extensions to the sequential Prolog execution model. Section 5 describes in some detail the modifications to the WAM and its data structures. Section 6 gives a simple and rather efficient scheduler for pure Prolog programs.

Section 7 shows how to get an OR-parallel (full) Prolog system.

Section 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 9 points out a reason for keeping the sequential semantics of findall construct in parallel Prolog systems.

Sections 10 and 11 discuss adding non-Prolog features to OR-parallel Prolog systems.

Sections 12 and 13 discuss machine dependent aspects of Muse on shared and distributed memory machines.

Section 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 15 discusses some advanced topics like efficient mechanisms for cuts, commits, and scheduling of speculative work. Section 16 discusses features that should be considered in a commercial OR-parallel Prolog system.

2 Inherited Terminology

In this paper 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 [20]. 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.

3 The Layered Approach

An OR-parallel Prolog system is best described (and implemented) using the layered approach shown in Figure 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.

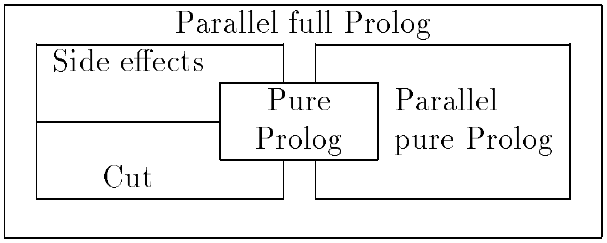


Figure 1: The layered approach.

4 The Muse Model

The Muse execution model [3] is based on having a number of workers, each with its own local memory space, and some global memory space 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 4.1 an efficient way of getting alternatives from frames associated with shared choicepoints is described.

Section 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 4.3 discusses what we call *incremental copying* — the method used in Muse to copy the difference between the states of two workers.

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 5.1.

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.

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 [27].

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 [3].

5 Extending the WAM

The sequential Prolog system we have extended to explore OR-parallelism is SICStus Prolog [9], a slightly modified WAM [26] system. How to extend the SICStus WAM and the ordinary WAM is now shown.

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 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 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.

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 choi-

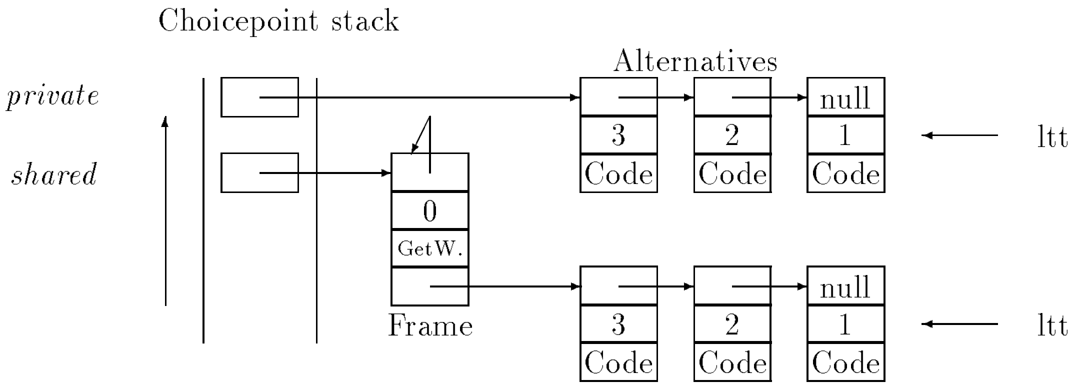


Figure 2: The SICStus predicate data structure.

cepoints. 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.

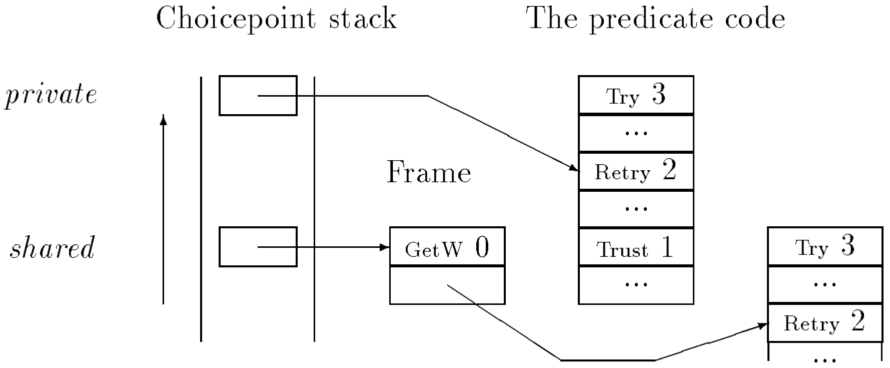


Figure 3: The WAM predicate data structure.

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 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.

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 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 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 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.

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

```
  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 5: Iteration 2 of the WAM emulator.

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 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 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 [17].

The engine part of the initialization macro `Sch_Init()`, in Figure 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 8. The first is used to keep the load register updated and the 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.

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 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.

The `Sch_Get_Work()` macro is the the most central macro in the scheduler code.

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

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

Figure 8: The `Sch_Set_Load()` and the `Sch_Get_Top()` macros.

```
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 9: The `Sch_Check()` macro.

Its function is described in Sections 6 and 8.4. If the macro succeeds the register `alt` shown in Figure 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.

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 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 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 6.3 a correct but inefficient scheduler is described. An improved scheduler is described in Section 6.4 and further optimizations are described in Section 8.4.

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 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.

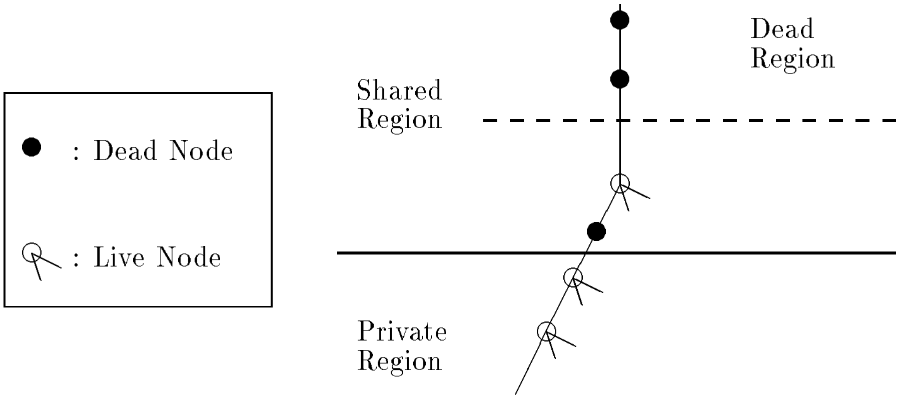


Figure 10: The dead region.

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.

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 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.

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 6.4 and the other two in Section 8.4.

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.

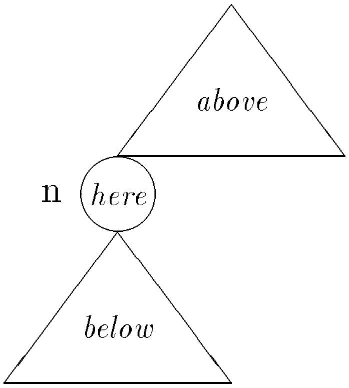


Figure 11: Here, above, and below.

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 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 12 can be described. The macro is called by the engine when backtracking to a shared choicepoint as shown in Figure 6 (page 11). 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 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 [12]. 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 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

```
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 12: The `Sch_Get_Work()` macro called from the emulator switch shown in Figure 6 (page 11).

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 [21, 24, 23, 25] 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 7.3), cut (described in Section 7.4), and all-solutions predicates (described in Section 7.6), all requiring the leftmost check described in Section 7.1.

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 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 2 (page 7).)

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

```
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 13: The leftmost check performed by a worker.

the workers considered when examining `n->next`.

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 11.4.

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 7.1) one in the search tree before continuing. If the engine, while waiting, receives a pruning request (Section 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 8.5, 15.1, and 16.1. The macro `Sch_Synch()`, shown in Figure 14 is a very simplified version of the macro found in [17].

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

Figure 14: The Sch_Synch() macro.

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 15, 16, and 17 is a very simplified description of the cut algorithm. The code lacks some optimizations. Figure 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 [17].

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.

3. Otherwise, some cut information is stored in the youngest choicepoint where

```
  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 15: The Sch_Prune() macro.

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 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

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

fully performed. Otherwise, if there exists in any visited choicepoint some branch 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 16: The `Sch_Continue_Prune()` macro is added to the `GetWORK` code.

The last piece of the code in Figure 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 17: Adding receiving of a cut request to the `Sch_Check()` macro shown in Figure 9 (page 12).

7.5 Sequential Predicates

Although the described synchronization (Section 7.3) and cut (Section 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 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 18 and 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 15.1, or the method described in Section 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 18: Avoiding speculative work in iterative deepening programs.

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

Figure 19: Avoiding dummy branches.

Whenever a predicate has been declared sequential both Muse and Aurora [20] also treat the choicepoints created by its disjunctions as sequential choicepoints.

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 20. It is essentially the same parallelization as that found in [8].

```
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 20: Parallel implementation of findall/3.

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 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 7.4) 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 8.5.

8 Optimizations

It is very important to make the sharing session efficient. Two techniques for achieving this objective are described in Sections 8.1 and 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 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 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 8.5.

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 12.2 shows how this problem is solved and Section 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.

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 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.

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 [20] 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.

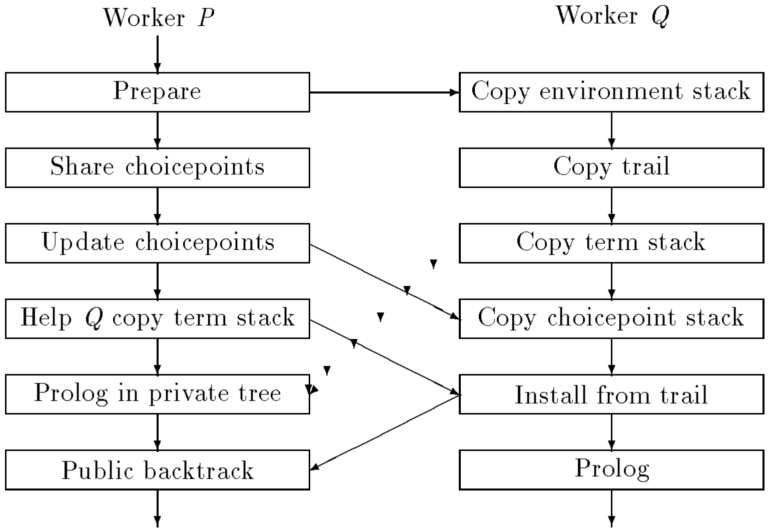


Figure 21: Phase 2 of the example parallel sharing session.

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 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 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).

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 de-

```
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 22: Extensions of the Sch_Check() macro (Figure 8 (page 11)) and the Sch_Set_Load() macro (Figure 9 (page 12)) to implement delayed release.

tail in the scheduler paper [2]. 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:

- **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 [2]. The last worker backtracking from a sequential choicepoint *must* stop and take care of the work found in that choicepoint.

Figure 23 is included to give a hint about how the scheduler loop in the unoptimized `Sch_Get_Work()` macro shown in Figure 12 (page 17) 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.

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 [8], which makes the semantics different in parallel and sequential Prolog systems. Some examples relying on the order of the solutions are presented in Section 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 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 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 5.1.) Each element of the list contains an alternative

```
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 23: The scheduler loop in the optimized `Sch_Get_Work()` macro. The vanilla macro was shown in Figure 12 (page 17).

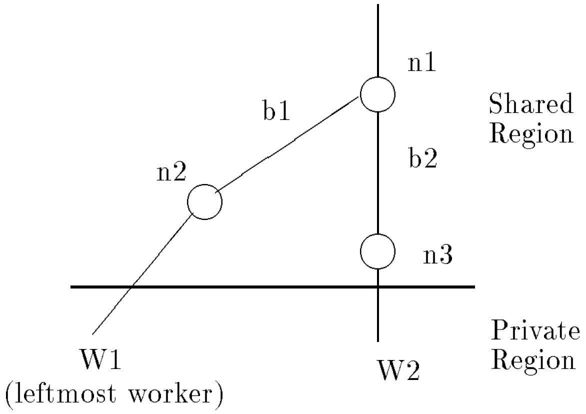


Figure 24: The speculative write approach.

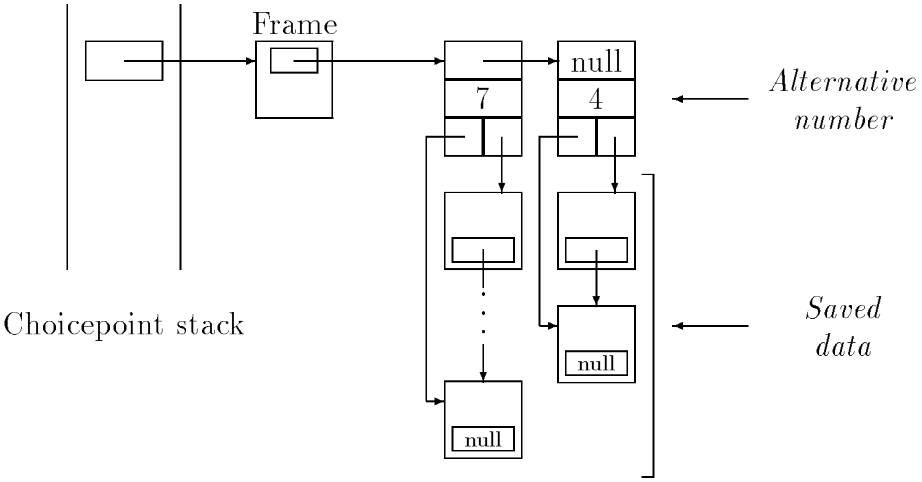


Figure 25: Data structures for speculative write.

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).

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 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 1: Comparison between strict and free findall implementations.

9 Independent AND- in OR-parallelism

This paper concerns mainly executing the different alternatives (clauses) in a predicate in parallel (OR-parallelism). Another important source for parallelism is called *independent* AND-parallelism [11, 15], 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 [8]. Our improved implementation of findall in Section 8.5, in conjunction with some optimizations described in [16] 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 [14]. 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 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 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.

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 [18]. Removing I/O buffering and switching to the *immediate* database view makes the implementation less efficient, possible removing the advantage of the parallelization.

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 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 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 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.

10.2 Asynchronous I/O

Implementing general I/O in a parallel UNIX environment is rather tricky, as described in Section 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 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).

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 [12] 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 [12] and is further discussed in Section 15.2.

It is very easy to implement a totally unrestricted commit (called *cavalier commit* [7]). 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.

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 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 metacall are also needed. The following example is taken from "ProLog by BIM" [6], a commercial Prolog now being parallelized using the Muse model.

Figure 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.

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.

```
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 28: Program that computes a maximum value using a global variable.

11 SICStus Specifics

This section presents some SICStus Prolog [9] specific topics. First three non standard Prolog constructs (`undo/1` in Section 11.1, `setarg/3` in Section 11.2, and `if/3` in Section 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 11.4.

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 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.

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 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.

```
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 29: The Muse implementation of setarg/3.

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 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.

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 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 30: A naive implementation of if/3.

The SICStus implementation shown in Figure 31 uses the built-in predicate SETARG/3 described in Section 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 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 32. This implementation should work without any extra modifications in the "ProLog by BIM" [6] 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 32: The global variable (parallel) implementation of if/3.

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.

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.

12.1 Lock

Muse uses the ordinary spin locks shown in Figure 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 33: The lock macros.

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 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 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 16.3. It also supports a sparse address space allowing the Prolog stack to grow without relocation.

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 [4]. 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.

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.

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.)

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 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 35: Serializing the scheduler loop for NUMA machines. The figure shows an extension of the macro in Figure 12 (page 17).

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 8.3) is increased from 5 to 10.

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 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 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()`.

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.

```
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.       */
}
```

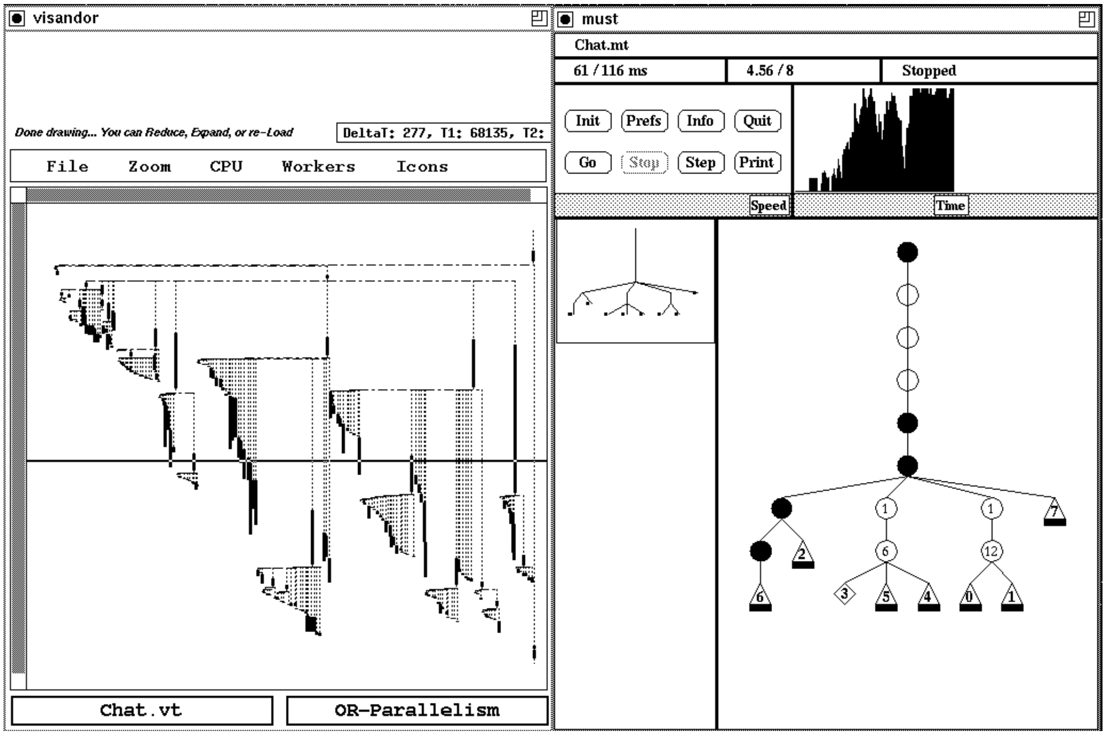
Figure 36: The cache coherency protocol for Butterfly II. The codes that are expanded can be found in Figures 9 (page 12) and 12 (page 17).

The first tool is briefly described in [3]. A more complete description can be found in [22]. Trace events are recorded in real time when executing a query 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 [10]. 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 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.



Done drawing... You can Reduce, Expand, or re-Load

DeltaT: 277, T1: 68135, T2:

Chat.vt

OR-Parallelism

Chat.mt

61 / 116 ms

4.56 / 8

Stopped

Figure 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 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.

```
                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 38: Sample overhead statistics.

Figure 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. 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 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 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.

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.

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 [28] 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 16.1 and 16.2.

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 7.4.

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 [12] 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 [5] 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.

15.3 Commit (the Real Thing)

Sometimes you are satisfied with finding just any solution to a goal. In Section 10.3 a brute force implementation (called cavalier commit) with that objective was described. If the extra information about speculative work described in Section 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 [12].

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 7.4. 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.

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 [18], 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.

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.

16.1 More Workers than Processors

Instead of suspending branches (as described in Section 15.1), a method using more workers than the number of processors can be implemented, as in [15]. 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 [13, 14]. 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.

16.2 Recompute Suspended Branches

To suspend branches as described in Section 15.1 is memory consuming and the number of workers, as described in Section 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.

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.

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.

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 [19].

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.

17 Conclusions

The Muse model is very simple. Using the already available code for the Muse-SICStus system in conjunction with this paper and the interface found in [17] 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.

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).

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 and Roland Karlsson. Full Prolog and Scheduling OR-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445 – 475, December 1990.

[3] 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.

[4] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. SICS Research Report R92:07, Swedish Institute of Computer Science, March 1992. Submitted to NGC Journal.

[5] 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.

[6] BIM. ProLog by BIM release 3.0. 3078 Everberg, Belgium, November 1990.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Bogumił Hausman. *Pruning and Speculative Work in OR-parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[13] 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.

[14] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Ewing Lusk. Remote I/O Handling Package specification. Internal Report, Gigalips Project, October 1989.

[20] 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.

[21] 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.

[22] Jan Sundberg and Claes Svensson. MUSE TRACE: A Graphic Tracer for OR-parallel Prolog. SICS Technical Report T90003, Swedish Institute of Computer Science, 1990.

[23] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, 1991.

[24] 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.

[25] Peter Lodewijk Van Roy and Alvin M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, pages 54–68, January 1992.

[26] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[27] 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.

[28] Patrick Weemeeuw. Garbage Collection for the Aurora System: A feasibility study, September 1991. Draft version published at the PEPMA (ESPRIT project 2471) review meeting.