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).
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.

<table>
<thead>
<tr>
<th>Queens</th>
<th>No. of elements in the list</th>
<th>Aurora (Bristol) Free findall</th>
<th>Muse Strict findall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1W</td>
<td>10W</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>1.42</td>
<td>0.19 (7.47)</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>5.32</td>
<td>0.61 (8.72)</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>22.35</td>
<td>2.41 (9.27)</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>91.66</td>
<td>9.50 (9.65)</td>
</tr>
<tr>
<td>11</td>
<td>2680</td>
<td>412.05</td>
<td>42.87 (9.61)</td>
</tr>
</tbody>
</table>

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([],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 effi-
cient (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 imple-
ment 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 de-
scribed 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

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

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

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

performs the sequence (write, read, write) 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 op-
eration 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, O) 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.
\texttt{findsum(X,Goal,Sum) :-}
\hspace{1em} \texttt{get\_unique\_identifier(S),}
\hspace{1em} \texttt{record(S,0),}
\hspace{1em} \texttt{findsum\_internal(X,Goal,Temp,S),}
\hspace{1em} \texttt{Sum=Temp.}
\hspace{2em} :- \texttt{sequential findsum\_internal/4.}
\texttt{findsum\_internal(X,Goal,\_,S) :-}
\hspace{3em} \texttt{call(Goal),}
\hspace{3em} \texttt{mutex([ recorded(S,X1), X2 is X1 + X, rerecord(S,X2) ]),}
\hspace{3em} \texttt{fail.}
\texttt{findsum\_internal(\_,\_,Sum,S) :-}
\hspace{4em} \texttt{recorded(S,Sum),}
\hspace{4em} \texttt{erase(S).}

\% Example of usage
\texttt{prog(Sum) :- findsum(X,generate(X),Sum).}

\textbf{Figure 28:} Program that computes a maximum value using a global variable.

\section{SICStus Specifics}

This section presents some SICStus Prolog \cite{sicstus} specific topics. First three non standard Prolog constructs (\texttt{undo/1} in Section 11.1, \texttt{setarg/3} in Section 11.2, and \texttt{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.

\subsection{Undo}

The SICStus built-in predicate \texttt{undo(Goal)} is used to execute the goal "\texttt{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 un binding on backtracking it also checks for \texttt{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 \texttt{setarg/3} in Section 11.2) variant of \texttt{undo} that may execute the \texttt{undo} goal several times (called \texttt{multi\_undo/1}) and a correct version that preserves the sequential semantics.

The implementation of the \texttt{undo} with sequential semantics is based on allocating (in global memory) an \texttt{undo frame} for each \texttt{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 implement 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.

\[
\begin{align*}
\text{if}(G1, G2, _) & :\ - \ exists(G1), !, \ call(G1), \ call(G2). \\
\text{if}(_, _, G3) & :\ - \ call(G3). \\
\text{exists}(G) & :\ - \ not(not(G)). \\
\text{not}(G) & :\ - \ call(G), !, \ fail. \\
\text{not}(_) & .
\end{align*}
\]

**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.

\[
\begin{align*}
\text{if}(G1, G2, G3) & :\ - \ if(G1, G2, G3, flag(no)). \\
\text{if}(G1, G2, _, Flag) & :\ - \ call(G1), \ SETARG(1, Flag, yes), \ call(G2). \\
\text{if}(_, _, G3, flag(no)) & :\ - \ call(G3).
\end{align*}
\]

**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.

\[
\begin{align*}
\text{if}(G1, G2, G3) & :\ - \ get_unique_identifier(S), \\
& \quad \ record(S, anything), \\
& \quad \ if(G1,G1,G3,S). \\
\end{align*}
\]

:- sequential if/4.

\[
\begin{align*}
\text{if}(G1, G2, _, S) & :\ - \ call(G1), \ erase(S), \ call(G2). \\
\text{if}(_, _, G3, S) & :\ - \ is_a_key(S), \ erase(S), \ call(G3).
\end{align*}
\]

**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 (nor-
mally) 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.

```c
#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)=i) 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 \( n \):th 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

\[ pa_i = i \times size. \]
\[ va^n = base + ((N + i - n) \mod N) \times size. \]

One example

<table>
<thead>
<tr>
<th>The file</th>
<th>pa(0)</th>
<th>pa(1)</th>
<th>pa(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker 0</td>
<td>va(0,0)</td>
<td>va(1,0)</td>
<td>va(2,0)</td>
</tr>
<tr>
<td>Worker 1</td>
<td>va(1,1)</td>
<td>va(2,1)</td>
<td>va(0,1)</td>
</tr>
<tr>
<td>Worker 2</td>
<td>va(2,2)</td>
<td>va(0,2)</td>
<td>va(1,2)</td>
</tr>
</tbody>
</table>

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 structures (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.

```c
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_Shar() {
    ...
    flush all areas that Q will copy; /* Replaces 'Copy to Q' */
    ...
}

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

Eng_Q_Shar(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.

![Figure 37: The combined tool ViMust.](image)

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

Total time = 6120 = 612*10

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>6120</td>
</tr>
<tr>
<td>Predicate calls</td>
<td>43825</td>
</tr>
<tr>
<td>Tasks</td>
<td>90(par)/2(seq)</td>
</tr>
<tr>
<td>Accepted sharing req.</td>
<td>62</td>
</tr>
<tr>
<td>Workers</td>
<td>10</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Tasks (Pieces of work without asking the scheduler for more)</th>
<th>from</th>
<th>to</th>
<th>num ( % )</th>
<th>calls ( % )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-&gt; 7: 3 (3.3 %) 14 (0.0 %)</td>
<td>8-&gt; 15: 2 (2.2 %) 22 (0.1 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-&gt; 31: 5 (5.4 %) 119 (0.3 %)</td>
<td>32-&gt; 63: 12 (13.0 %) 561 (1.3 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64-&gt; 127: 16 (17.4 %) 1613 (3.7 %)</td>
<td>128-&gt; 255: 23 (26.0 %) 3899 (8.9 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256-&gt; 511: 11 (12.0 %) 3930 (9.0 %)</td>
<td>512-&gt; 1023: 10 (10.9 %) 8290 (18.9 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024-&gt; 2047: 4 (4.3 %) 4593 (10.5 %)</td>
<td>2048-&gt; 4095: 5 (5.4 %) 16498 (37.6 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096-&gt; 8191: 1 (1.1 %) 4286 (9.8 %)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum: 92 (100.0 %) 43825 (100.0 %) [476.36]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunks (Pieces of undisturbed work)</th>
<th>from</th>
<th>to</th>
<th>num ( % )</th>
<th>calls ( % )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-&gt; 0: 2 (1.3 %) 0 (0.0 %)</td>
<td>1-&gt; 1: 2 (1.3 %) 2 (0.0 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-&gt; 3: 5 (3.2 %) 13 (0.0 %)</td>
<td>4-&gt; 7: 11 (7.1 %) 62 (0.1 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-&gt; 15: 8 (5.2 %) 86 (0.2 %)</td>
<td>16-&gt; 31: 12 (7.8 %) 279 (0.6 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-&gt; 63: 28 (18.2 %) 1340 (3.1 %)</td>
<td>64-&gt; 127: 43 (27.9 %) 3945 (9.0 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-&gt; 255: 13 (8.4 %) 2223 (5.1 %)</td>
<td>256-&gt; 511: 11 (7.1 %) 3971 (9.1 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512-&gt; 1023: 10 (6.5 %) 8373 (19.1 %)</td>
<td>1024-&gt; 2047: 3 (1.9 %) 3358 (7.7 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048-&gt; 4095: 5 (3.2 %) 16016 (36.5 %)</td>
<td>4096-&gt; 8191: 1 (0.6 %) 4157 (9.5 %)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum: 154 (100.0 %) 43825 (100.0 %) [284.58]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: \textit{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 \textit{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 asynchronous 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), Försvarsvets Materielverk FMV (Defence Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

References


