Divided We Stand: Parallel Distributed Stack Memory Management

Kish Shen
Computer Science Department
Bristol University, UK
email: kish@acrc.bristol.ac.uk

Manuel Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid, SPAIN
email: herme@fi.upm.es

August 4, 1999

Abstract

We present an overview of the stack-based memory management techniques that we used in our non-deterministic and-parallel Prolog systems: &-Prolog and DASWAM. We believe that the problems associated with non-deterministic and-parallel systems are more general than those encountered in or-parallel and deterministic and-parallel systems, which can be seen as subsets of this more general case. We develop on the previously proposed "marker scheme", lifting some of the restrictions associated with the selection of goals while keeping (virtual) memory consumption down. We also review some of the other problems associated with the stack-based management scheme, such as handling of forward and backward execution, cut, and roll-backs.

Keywords: Parallelism, Logic Programming, Memory Management, And-parallelism, Or-parallelism, Implementation

1 Introduction

The efficiency of a sequential Prolog implementation is largely determined by two factors: the basic speed — i.e., the raw speed at which it is able to execute Prolog code; and memory usage — i.e., the amount of memory the system uses while executing a program.

Memory efficiency comes into play mainly when large, realistic application-type programs are executed, rather than in small benchmark-type programs. Memory usage is important for large programs because of the finite resources in any real computer system. In the worst case, a program will not run on a system that is not memory efficient, but will run on a more efficient one. In less extreme cases, the extra swapping a memory inefficient system introduces will have an important negative impact on performance.

The issue of memory performance is of even greater importance in a parallel system than in a sequential system, because a parallel system is likely to consume more total memory than a sequential system, both because more code is being executed at the same time, and because of the overhead needed to support parallelism. Moreover, the issue is of even greater importance for a declarative language such as Prolog, which is generally already considered to be less memory efficient than conventional imperative languages.

There are other issues that will affect the perceived performance of a parallel Prolog system, including the efficiency of the system in exploiting parallelism, and higher level issues such as the nature and availability of the parallelism. For a complete look at the performance of a parallel Prolog system, all these issues need to be examined. These other issues have been examined in greater or lesser detail elsewhere, but we feel that issues related to memory management and usage are usually not considered in detail in the parallel Prolog literature, and indeed performance evaluation of systems often consists of benchmarking the system with small programs, which do not stress the memory resources of the hardware. We therefore feel that there is a need to look at memory management in parallel Prolog systems in more detail. This is the purpose of this paper.

We present an overview of one way of tackling the memory usage problem of parallel Prolog system — that of using multiple stacks, concentrating on the methods used in systems we have implemented — PWAM [6] and DASWAM [18].[1] We examine these systems for concreteness, and also, as we shall show, because we feel that our systems represent a class of systems (those supporting non-deterministic and-parallelism) which have to tackle more general problems in memory management than other parallel Prolog systems, such as deterministic and-parallel and or-parallel systems.

The rest of the paper is organised as follows: first, we introduce the multiple stack model, and discuss its merits. We then briefly overview how the model is implemented, and finally, we examine some of the extra support mechanisms that are needed to deal with parallel execution of full Prolog: dealing with cuts, backward execution, and propagation of failure. Throughout this paper we assume that the reader is familiar with Prolog, and parallel Prolog. We also assume some familiarity with the implementation of parallel Prolog systems.

2 General Approach

The most efficient sequential logic programming systems obtain much of their performance from doing their own stack-based memory management and through compilation. Storage space is recovered automatically on backtracking, reducing the need for an explicit garbage collector. In addition, a compiled system is more memory efficient than an interpreted system because the compilation process reduces the amount of information that needs to be replicated from one procedure call to another. Moreover, in many systems (such as the DEC-10 Prolog machine [20] and the WAM [21]), further storage optimisation is obtained by the use of a *two stack model*, where the storage of variables is divided between two areas — the local and global stacks. This allows storage in the local stack to be recovered as soon as a clause has been completed without an alternative. Furthermore, through last call optimisation, local stack frames (the WAM "environments") can be often reused, effectively turning recursion into iteration.[2]

Ideally, we would like memory management on parallel systems to achieve similar results

[1]These are the abstract machines for &-Prolog [6] and DDAS [16], respectively.

[2]For a detailed description of the WAM, see [1].

to those achieved in sequential systems: recovery of storage space during backtracking, minimisation of the replication of state information, and early recovery of some additional storage space. A compiled parallel system is the first step to more efficient memory management, and we shall describe our approach in that context, although the techniques should be applicable to interpreted systems as well. Before we introduce our specific approach, we first discuss some general properties of the parallel systems we are considering.

We adopt the subtree-based approach to executing Prolog programs in parallel, which is common to many models. In this approach parallelism is achieved by allowing several entities –which are often called **workers**– to simultaneously explore the search tree of a program. Each such worker explores the search tree in much the same way as sequential Prolog: depth-first, left-to-right. Generally, each worker will be assigned to a different part of the tree. Thus, the search tree can be thought of as being divided into subtrees, each of which is executed sequentially and referred to as a **task**. In the case of or-parallelism these subtrees are generally branches of the tree, while in the case of and-parallelism they are contiguous parts of one or more branches. It is often the case that the subtrees are not determined a priori but rather as the tree is being dynamically constructed: as a worker works on a task, opportunities for parallelism are identified and thus marked. When a worker finishes exploring a subtree, it may start exploring another sub-tree which has been identified for parallel execution – this process is referred to as **stealing a task**. It should be noted that if there are no free workers the tasks or subtrees identified by a worker will (eventually) be explored by this worker.

The subtree-based approach has the advantage that the execution of each task is very similar to that of sequential Prolog, and thus many of the techniques (and advantages) of sequential Prolog implementations should carry over, including those related to memory management. However, the actual way in which the parallel workers are managed, and the actual mechanisms for memory management remain to be specified. For this, we will use a "distributed stack" scheme as our starting point. Variants of such a scheme (and its restricted version, the "cactus stack" scheme) have been used repeatedly in implementations because they offer the potential to achieve the above mentioned goal of approaching sequential memory efficiency [2, 8, 22, 23].

We assume the program is to be compiled into instructions which are quite similar to those of a Prolog engine, with perhaps some additional instructions related to parallelism. Following [9, 6], we view each of the workers as composed of two elements: an **agent**, which is a processing element, capable of executing such instructions in much the same way as a sequential Prolog engine, and a **stack set**, which represents the associated storage, i.e., a set of stacks, consisting of the normal sequential Prolog stacks plus perhaps some other areas needed for parallel execution, and a number of registers, as is shown in Figure 1, which represents a simplified layout with elements from both PWAM and DASWAM. A complete parallel system then consists of a number of agents, and the same number or greater of stack sets. Agents are free to attach to any stack set that does not already have an attached agent, and are also free to move to other stack sets. A stack set with an attached agent can then be viewed as a worker, and can actively perform computation. Efficient use of the agents, which are really representing the physical processors of the underlying parallel machine, is necessary to achieve good speedups. Efficient use of the stack sets is necessary to keep memory usage reasonable.

Following the subtree-based approach, an agent can execute a task and use the stack set it is attached to in much the same way as in standard Prolog execution, except for such differences as preparing work for and-parallel execution, and when a task is completed or suspended. When a task completes or suspends, then if more tasks are available, a new task can be started. In order to use agents and stack sets efficiently, the simplest thing is to use the same agent and stack set to perform the new task by using the space beyond that already used by the older task. Thus, the contents of a stack set can be seen as divided into areas, each corresponding to a task.

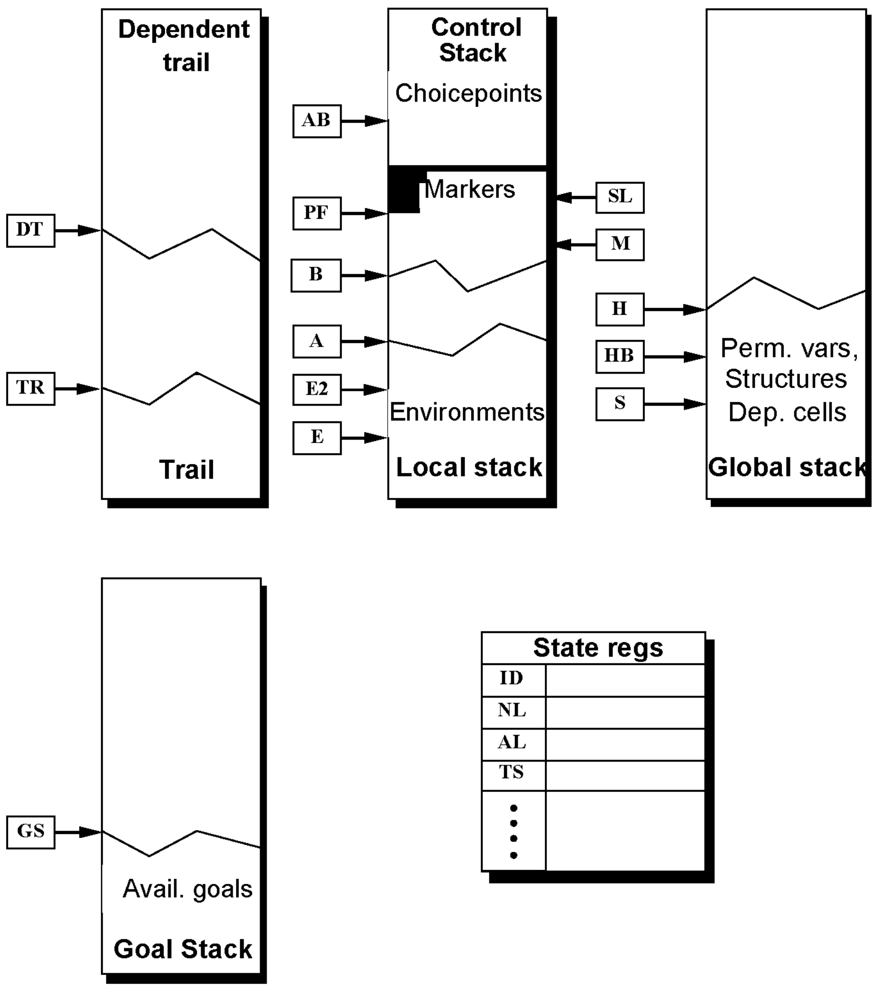

Figure 1: Possible Layout of One Stack Set

Each such area is referred to as a **stack section**.[3] The ordering of the stack sections on the worker's stack set is the chronological order in which the worker executed the tasks associated with the stack sections. In order to distinguish and manage such sections, they are separated from each other by **markers** [9]. Abstractly, we can consider that markers are placed in all the stacks in a stack set, separating the stack sections in each stack. However, in practice this is done by allocating markers on to only one stack (and the choice point (control) stack seems the most natural one to use), which in turn contain pointers to the corresponding boundaries of the stack sections in the other stacks. In addition, depending on the nature of the section above or below them, some markers may serve some additional special functions.

Thus, markers are used in a particular stack set to separate different stack sections, and in addition, to link the various stack sections that are distributed to the various stack sets logically. This logical link is important because it enables the linked stack sections to be viewed abstractly as one continuous stack.

The "marker scheme" [7, 9, 6] summarised above can be used for both or- and and-parallel systems. In an or-parallel system, specially marked ("public") choice points can serve as markers (as is done in e.g., Aurora [12]), each one corresponding to a "fork" in the parallel task tree. If and-parallelism is restricted to "deterministic" goals, such as in PNU-Prolog [13] and Andorra-

[3]Note that a task may be spread over several stack sections, because of backtracking and suspensions.

I [14],[4] then choice points can also be used as markers, as they mark points where there is no and-parallelism. However, in (don't know) non-deterministic and-parallel systems, where and-parallelism is allowed among goals which potentially can have more than one solution, the marker function cannot be fulfilled by choice points only. In addition, and unlike in an or-parallel system, not only forks but also "joins" have to be performed on the tasks representing sibling and-goals, and more coordination than for or-parallel systems is needed. Thus, additional data structures have to be provided to serve as several types of markers. Therefore, the marker scheme used for "non-deterministic" and-parallelism can be regarded as a generalisation of an or-parallel and a deterministic and-parallel marker scheme. In this paper we will concentrate on this scheme, with the understanding that the solutions proposed and results obtained can be applied to the other forms of parallelism, perhaps with simplifications.

2.1 Overview of the Marker Scheme in PWAM and DASWAM

For concreteness, we now overview the marker scheme as implemented in PWAM and DASWAM. Both of these schemes developed from the original scheme presented above, and contain similar extensions.[5] Both systems implement non-deterministic and-parallelism.[6]

In these implementations, five basic types of markers can be recognised:

Parcall Frame. This marker manages the and-parallel execution of body goals inside a clause, and is allocated just before scheduling a set of body goals which are determined to be executed in parallel (for concreteness, we will assume that &-Prolog's Conditional Graph Expressions — CGEs — [7], extending DeGroot's Execution Graph Expressions [5], are used for this purpose).[7]

Join Marker. This marks the end of an and-parallel execution: after spawning and-parallel execution for a particular CGE, solutions to the various and-goals being executed in parallel will be returned at different times, if the and-parallel execution does not fail. Eventually, solutions would be returned for all the and-goals, and the and-parallel execution has to be "joined" to allow the execution of goals following the CGE. For this purpose, the last agent to return a solution allocates a join marker on the stack set it is attached to, and then executes the continuation following the CGE.[8]

Suspend Marker. This marks the suspension of a task on the previous stack section, and is allocated by an agent on a stack set when it wants to use that stack set for another task. The suspended task can then (eventually) be continued at another location in the distributed stack.

Continuation Marker. This marks the continuation of a task, and is allocated when a task is resumed after a suspension. This enables the task to continue execution in a different location in the distributed stack.

Basic Marker. This marks the start of a stack section that is not of the above types, e.g., when a new task is started. These correspond to both the input goal markers and the local goal markers described in [7].

[4]The term "determinate" is used instead of "deterministic" in Andorra-I.

[5]There are some minor differences, but these are unimportant for the purposes of this paper.

[6]Independent and-parallelism in the case of &-Prolog, dependent and-parallelism (with independent and-parallelism as a subset), in the case of DASWAM.

[7]These frames are allocated on the environment stack in PWAM and on the control stack in DASWAM.

[8]Note that this is an extension of the original scheme proposed in [7], where the stack set that started the and-parallel execution had to be the one used for the task following the completion of the CGE.

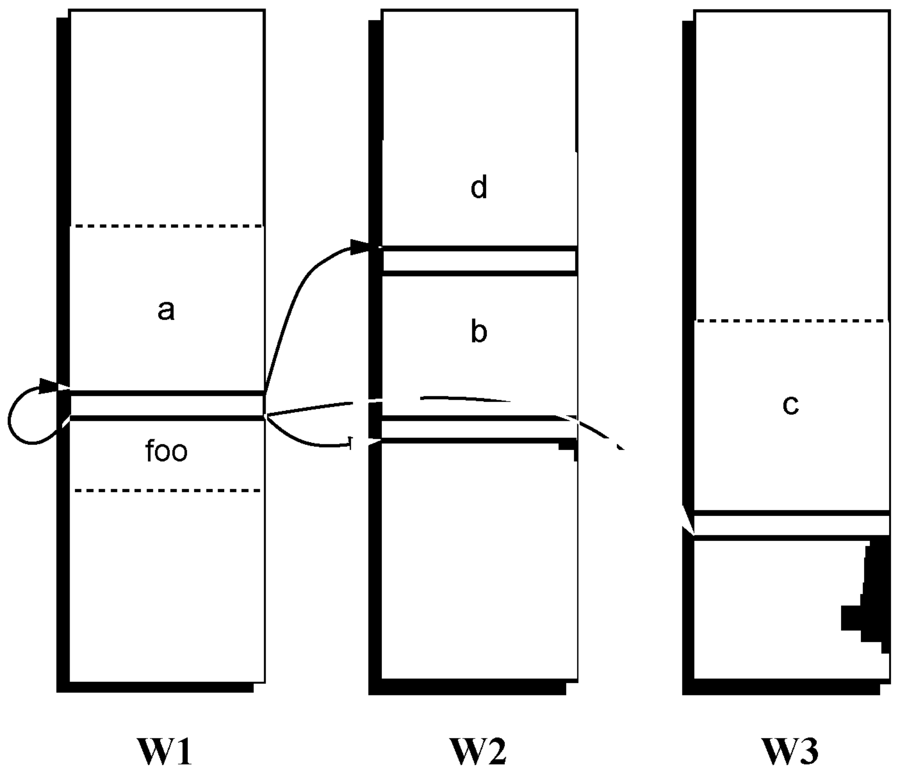

Figure 2: Stack States for a Distributed Stack Scheme

Figure 2 illustrates the use of the marker scheme to represent the following &-Prolog program fragment:

```
foo :- (a & b & c), d.
```

A possible parallel execution of this clause is shown in Figure 2, where each stack set is represented as a single stack for simplicity. Markers are represented as thin horizontal boxes, while the contents of a stack section are the larger shaded regions in each stack set. The and-task of concern is shaded in light grey in the different stacks. W1 is the worker that executes `foo`, which is pushed onto the top of W1's stack. At the CGE, `a` is executed locally, while `b` and `c` are executed remotely (in parallel with the execution of `a`) on W2 and W3 respectively. When `a`, `b` and `c` have all finished execution, W2 picks up the goal after the CGE, `d`, and continues the execution, leaving W1 and W2 idling, and thus ready to pick up more work. Before executing `b` or `c`, both W2 and W3 were idling and were therefore able to pick up `b` and `c`. Both have performed some work, and used their stack. This "old" work is separated from the current work by a *marker*. Parcall markers are used to mark the start of a CGE (e.g., the one separating `foo` from `a`), and contain pointers to link the stack sections of the sibling and-goals of the CGE, and a pointer to the stack section following the CGE.

All the markers also contain extra pointers for linking the various stack sections: markers on the same stack set are doubly linked, to facilitate the movement within a stack section, and markers also contain pointers to link the various stack sections (which may be on different stack sets) together logically. This is shown in Figure 3, where the lighter shaded stack sections represent a task which is split into two stack sections, with the appropriate linkages in the markers. These pointers allow the system to backtrack across different stack sections in the logically correct order by following the links.

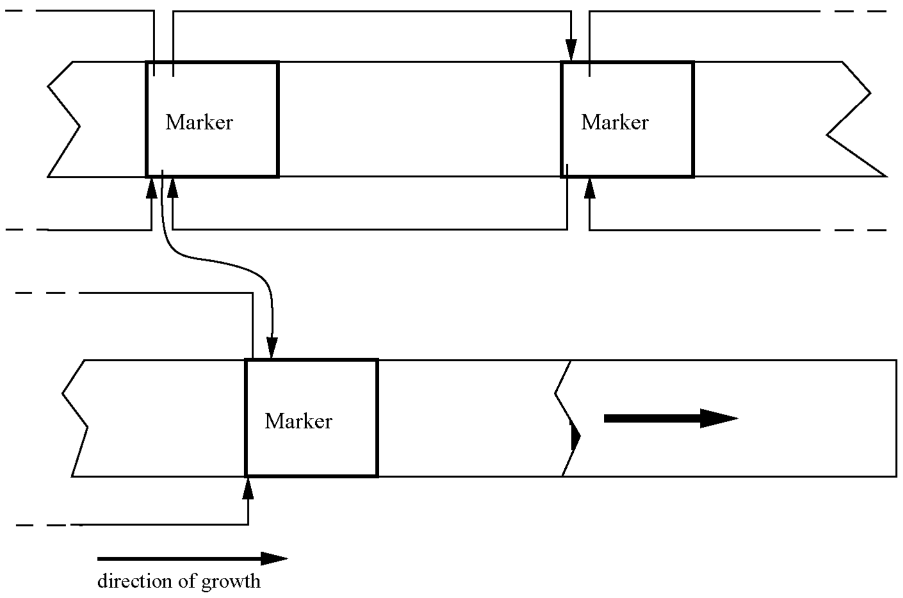

Figure 3: Linking of Markers

One major difference between a distributed stack scheme and a sequential stack scheme is that backtracking can occur in any of the stack sections in a stack set, so each stack set can have multiple points of backtracking (and potentially multiple points of growth) at the same time. The pattern of contraction and growth is thus affected by what each stack section represents, and this results in a close relationship between memory management and goal scheduling. This leads to the problems of "trapped goals" and "garbage slots" [9] (also later referred to as "holes" in or-parallel systems). In a previous paper [9], several solutions were proposed to solve these problems by placing some constraints on which goals could be executed on a particular stack set. This imposed extra overhead when selecting a goal, and either limited parallelism or imposed a high cost on virtual memory consumption, because of the creation of potentially large numbers of stack sets. In our actual implementations, we extended this previous work by the use of suspend/continuation markers, which allow the *suspension* and *resumption* of a task on different stack sets. Thus, the constraints on which goals can be selected are lifted, allowing the use of more flexible schedulers. Such schedulers should be able to give better performance (speedups), although they may use more memory.

2.2 Comparison between flexible and restricted schedulers

In order to gauge the effectiveness of the more flexible schedulers, two schedulers were implemented and compared: the 'flexible' and 'restricted' schedulers. The flexible scheduler uses an implementation that incorporates the suspend/continuation markers, and places no limitations on which goals may be selected. The restricted scheduler is an implementation of one of the schemes proposed in [9], where only 'appropriate' goals are allowed to be selected, resulting in the preservation of the sequential chronological ordering the stack sections in each stack set.

Some experiments were performed comparing these two schedulers, using the DASWAM simulator. The results, reported in [19], will be summarised here:

The flexible scheduler gave better speedups in programs in which more than one of the goals in a CGE contained CGEs, that is, in which recursive and-parallelism appeared in more than one of the goals in a CGE. Depending on the distribution of parallelism, the speedups ranged

# workers	orsim		boyer	
	flexible	restricted	flexible	restricted
10	9.8	8.1	6.6	2.8
20	19.1	16.3	8.7	3.2
30	27.9	23.6	9.3	3.4

Table 1: Comparison of speedups between restricted and flexible schedulers

	flexible		restricted	
	total	s.d.	total	s.d.
orsim	964580	±1499	967271	±13546
boyer	1764616	±764	175112	±19079

Table 2: Memory usages of the two schedulers

from being slightly to significantly better than those of the restricted scheduler. Table 1 shows the speedups achieved by the two schedulers for two programs at 10, 20, and 30 workers. One of the programs, boyer, a simple theorem prover, showed quite a large difference between the two schedulers. On the other hand, orsim, a simulator used to study the high-level characteristics of or- and independent and-parallelism [17], showed smaller differences.

We expect the differences between the two schedulers to be even more in favour of the flexible scheduler in a real system, because in our comparisons, we did not simulate the cost of maintaining the information necessary for the selection of appropriate goals in the restricted scheduler. This is expected to be either very expensive, or to place even more restriction on parallelism.

One possible problem with the flexible scheduler is that it may consume more memory. In order to quantify this effect, the amount of memory used by the two schedulers was also measured in our experiments. We found that the flexible scheduler did not use significantly more memory than the restricted scheduler, at least in the programs we examined.

Table 2 compares the memory usage for orsim and boyer running with 10 workers under the two schedulers. The measurements were taken at one instance of simulation 'time' just before the programs finished executing, when the memory usages for these programs were at their greatest. Two figures are given for each scheduler and program: the total amount of memory used by all the 10 stack sets of the 10 workers, and the standard deviation on the amount of memory each stack set was using. The greater the standard deviation, the greater the variation of memory usage for each worker.

The results suggest that the flexible scheduler does not use more memory than the restricted scheduler: in fact, the usages in both programs seem to be slightly smaller for the flexible scheduler, though this is probably due to the way the data was gathered. In addition, the memory usage is divided more evenly between the workers for the flexible scheduler.

Taken together, these results suggest that the flexible scheduler achieves better speedups with reasonable extra cost in memory usage.[9]

[9]Results from the or-parallel PEPSys system [4] also support the same conclusion for or-parallel systems. However, as mentioned before, or-parallel systems do not suffer from the trapped goal problem, so the problem is less severe than in the more general case of non-deterministic and-parallel systems.

3 Dealing with Executing Full Prolog in Parallel

In order to be able to execute full Prolog in parallel, mechanisms must be provided to handle such features as cuts and side-effects, and also mechanisms for dealing with failures: the links in the markers allow backtracking to take place, but the failure of a task in an and-parallel execution may affect other tasks as well, as described in [7, 17]. Mechanisms must thus be provided to coordinate the actions across different stack sets. In this section, we shall examine the impact of these issues on our distributed stack.

3.1 Dealing with Side-effects

In general, side-effects can be dealt with by the suspension of tasks, plus some extra synchronisations. These synchronisations are independent of the distributed stacks, and thus have no extra impact on it, so they will not be discussed further. One exception to this is the action of the cut, which does have an impact on the distributed stack.

3.2 Dealing with Cuts

In a sequential WAM such as SICStus' WAM [3], which has separate local and control stacks, the execution of a cut will be able to remove arbitrarily many choice points (up to the choice point representing the parent goal) from the top of the control stack. This is done by simply setting the top of the control stack register to point to the last choice point that is outside the scope of the cut. However, such a simple scheme is not sufficient for a distributed stack, as the choice point to cut to may be in a different stack section. In fact, there can be arbitrarily many stack sections between the current stack section and the stack section that choice point is located on.

Three general situations can be recognised when a cut is encountered:

- The cut cuts to a choice point within the current section. The normal sequential cut mechanism is used to deal with this.
- The cut cuts to a choice point outside the current section, but still within the same task. First, the top of control stack is reset to that of the current marker, removing any choice points allocated since this stack section was started. Next, choice points have to be removed from the previous stack sections, until the choice point to cut to is reached. This is done by following the markers in reverse chronological order, starting from the current stack section, and performing the cut operations on these previous stack sections.

  Each of these previous stack sections is bounded by markers both before and after the stack section. To facilitate the cut operation, each marker contains a pointer field which points to the last valid choice point (if any) on the stack section before it. Initially, when the marker is allocated, this field is set to point to the top of control stack. When a cut operation is performed, this last valid choice point pointer is set to point either to the choice point to cut to, if it is in this stack section; or to the marker before the stack section, if the choice point is outside this stack section. In the latter case, the marker before the stack section is used to locate the logically previous stack section, and the cut operation is performed recursively on that section.
- The cut cuts across sibling and-goals to its left. An example of this is:

```
foo :- (true => a & b & (c, !) & d).
```

> This cut cuts away the choices of `a`, `b`, `c`, as well as `foo`. The main problem is that `a` and `b` are executing in parallel, and may still be executing when the cut is encountered. The effect of the cut is performed in two stages: the choices of `c` are pruned when the cut is encountered, using the methods just described. The slots in the parcall marker associated with `a` and `b` are then marked with a 'cut' flag. The pruning of choices on `a` and `b` then takes place when all sibling and-goals between them and the cut have returned a solution, i.e., `b` is pruned when `b` returns a solution, `a` is pruned when both `a` and `b` have returned a solution (the finishing of the task that finishes later initiates the pruning). However, if an and-goal to the left of the cut fails, then the 'cut' flag is reset.

In our current systems, the space represented by the discarded choice points on the non-current stack sections cannot be immediately recovered, leaving 'garbage slots' in the control stack. The markers cannot simply be removed, because they have to be retained to allow detrailing of variables during the actual backtracking. Note that this is independent of what goal selection scheduling strategy is being used.

The space can be recovered by a garbage collector, or, alternatively, if the control stack is separated into a choice point stack for choice points only, and a marker stack for markers only. In this case, the markers would not be 'blocking' the space recovered by the cut. Some redesign of the existing scheme would be needed, but in principle this would make the recovery of the space occupied by the choice points easier.

Note that no parallelism is lost (except for whatever overhead is needed to perform the cut) in dealing with cuts. This is in contrast to dealing with other side-effects, where the task performing the side-effect must in general suspend until it is leftmost.[10]

3.3 Dealing with Signals

In and-parallel execution, events that take place on one task can affect the behaviour of other tasks. For example, under the "restricted" intelligent backtracking scheme used for pure goals in RAP-WAM [10], when a goal in a CGE fails, all sibling and-goals are "killed". Even if no intelligent backtracking is used, standard backtracking among parallel and-goals involves coordination among agents and tasks. In DDAS, there is even more interaction between and-goals because of the dependent and-parallelism [17, 16].

Such communication among tasks can be implemented by allowing tasks to send signals to each other.[11] For example, when a task is told to undo its computation (referred to as "unwinding" or **roll-back**), a 'kill' or 'redo' signal is sent to the task. A 'kill' signal informs the task that receives the signal that it is to be killed. A 'redo' signal means that after undoing the computation to the previous alternative, the task starts forward execution again. A 'kill' signal does not restart execution of the task. The decision of which signal to send is determined by the exact backward execution scheme used, and will not be discussed further here. Here our interest is in how memory can be recovered and the signals handled.

[10]Note that this does not apply to systems where deterministic goals are executed early, and are allowed to bind variables, such as PNU-Prolog and Andorra-I. In such systems, the search space explored can be different from Prolog, and for correct full Prolog behaviour, goals should not be executed in and-parallel across cuts [15].

[11]Such "signals" are of course conceptual and do not in general imply using actual operating system signals – more often the action involved is setting a bit in a signal word of an agent or stacking a value on to a "signal buffer."

Many approaches can be taken to deal with the complexities that arise from the interactions of signals. A simple approach is to delay the killing of a task until it has finished: the task finds out that it has been killed when reporting back success or failure to the parcall marker. This solution is attractive in that it completely avoids the complexities and synchronization overheads, at the cost of using more resources and performing more (wasted) work. This extra work is unfortunately potentially infinite, unless care is taken at compilation time to only allow the parallel execution of goals which can be proved to terminate. Since this property can only be approximated, the number of goals which can be executed in parallel is restricted in this approach. Moreover, even then the system may still perform a large amount of wasted work. Another approach, used in APEX [11], is to suspend all forward execution whenever such interaction occurs. This simplifies the problem, but can potentially greatly affect performance as many of the workers may be doing work that would be completely unaffected. It also requires global synchronisation, which is probably undesirable with any significant number of workers.

A third solution is to tackle the complexity head-on. This is clearly the most complex approach, but it can potentially give the best performance in resources and time. Many variations on this approach are possible, but, for concreteness, we shall describe one of them, namely the approach taken in DASWAM (the approach taken in PWAM, although another variation, is quite similar).

As already discussed, a task is represented in the distributed stack by one or more sections that are logically linked by the continuation markers. The task receiving the signal may not be active, i.e., it may not be actively being worked on as some worker's top-most stack-section. Indeed, a task may have started its own and-parallel execution, and thus it would composed of a number of descendant and-tasks. Thus, there is no simple representation for a task. However, the start of a task is well defined: a task begins when it picks up an and-goal and starts execution on it. The start of a task is thus represented by the first stack section of the task. The parcall marker contains pointers to the first stack section of every and-goal that has been executed in the CGE that it represents. Signals are sent to all the sibling and-tasks in a CGE, or to the sibling and-tasks to the right of the task that generated the signal, depending on the nature of the signal. Thus, when a signal is generated, the local parcall marker is used to determine which and-task the signal needs to be sent to, and the signal is then sent to the stack set containing the first stack section of the and-task. This signal is treated by the receiving stack section as an interrupt, such that if an agent is attached to the stack set, the agent stops its current work and processes the signal before returning to the original work. If no agent is attached, then if there are idle agents, one of these is immediately attached to the stack set to process the signal. Otherwise, the signal would be processed by the agent that was attached to the stack set that sent the signal. Henceforth, for simplicity, we will assume that an agent is attached to the stack set that receives the signal, and simply refer to it as a worker.

Once a worker receives a signal for a task that it started, the signal must be propagated to the following stack sections of the task, if the task is split into more than one stack section. This is done by following the pointers in the various markers to the other stack sections. Note that this propagation is distributed: once the signal is propagated to a new stack section, the processing is handled by the worker to which that stack section belongs.

Once a signal is propagated as far as it can go (i.e. to the last stack section representing a task), then the action associated with the signal can take place. For both 'kill' and 'redo' signals, the work done by the task receiving the signal is rolled-back in much the same way as the undoing of work during backtracking, except that alternatives represented by choice points are not tried. The process of undoing a piece of work may lead to more 'kill' signals, e.g., if there are nested CGEs inside one of the tasks being killed. However, in practice, many of these signals apply to the same tasks, and the system can filter out signals that are sent to a task that has already received the same signal. The task is rolled-back in semi-chronological order in that stack sections representing later work of a task are undone before those representing the earlier work. The exception is that work done by sibling and-goals can be rolled-back in parallel.

One apparent problem comes from the fact that the propagation of a signal takes a finite amount of time, while the affected task can spawn new tasks before receiving the signal. Thus, in principle, it might be that such new tasks are produced quicker than the speed of propagation of the signal, and the overall killing process does not terminate. Note, however, that the scheme presented effectively avoids such "race conditions" since the propagation is distributed, with the worker receiving the signal performing the processing of the signal. Thus, if a signal affects many stack sections, more and more workers (or more precisely, agents) become involved in the processing of signals, so that in the extreme case, all the workers are involved in processing the signal, and no forward execution is performed. Because there is a finite number of agents or workers, propagation of signals will always be completed and acted upon. Note that the system gracefully degrades to the approach taken by APEX of stopping all agents at the moment of processing any signal, but only when such drastic action is needed. At the same time, as the signal propagation is a simple and distributed operation, it should occur quickly, so the amount of wasteful work performed is kept low, and is never infinite.

The actual mechanism used can best be illustrated by an example. Figure 4 shows an example stack state for a still executing CGE. The lightly shaded stack sections are all executing the same and-task: At W1, during the execution of stack section "a", a CGE is encountered, and two descendant and-tasks, executing section "b" (on W1) and "c" (on W2), are started. At some point, section "b" is completed, and a new section started on top of it. Section "c" encounters another CGE, spawning sections "d" (executed locally on W2) and "e" (executed on W3). Section "d" is completed, but no new work is available, so W2 goes into the idle state. Task "e" is for some reason (e.g., a suspension that has been resumed) split into two sections: "e1" on W3, and "e2" on W4. e1 has been partially backtracked, and section "e2" is in the process of forward execution. At this point, the task associated with section "a" receives a 'kill' signal.

The roll-back has to undo the states of sections "a" to "e2". A child section is undone before its parent — i.e., starting from "b", "d" and "e2", and working up the hierarchy to "a". The reason for this is that the propagation of the kill signal to descendant and-tasks is asynchronous and takes a finite amount of time, so it is quite involved to undo an ancestral stack state when its descendant may still be running (because they have not yet received the kill signal). For example, if section "e2" is still running, it might access its ancestral stack sections "e1", "c" and "a". Thus the kill signal is propagated to the youngest child sections before the killing starts. In this example, "b", "e2" and "d" are rolled-back, when "e2" has been rolled-back, "e1" is rolled-back. Section "c" is rolled-back when both its descendant sections ("c" and "e1") are undone. Again, "a" is not rolled-back until both its children — "b" and "c" — are rolled-back.

Each worker is responsible for performing the roll-back in its stack set. One reason for this is to keep the roll-back algorithm relatively simple. Another reason is to exploit the opportunities for parallelism: e.g., sections "b", "d" and "e2" can be rolled-back in parallel with each other.

The case is simple for sections "d" and "e2", as they are the topmost sections. The same applies to section "c", as by the time it is allowed to be roll-back, section "d" would be undone already, and "c" would have become the topmost section. In the cases of "b" and "e1", they are not the topmost sections of their worker's stack set during the roll-back. In these cases, the worker has to freeze the current work it is doing, perform the roll-back, and then go back to its current work.

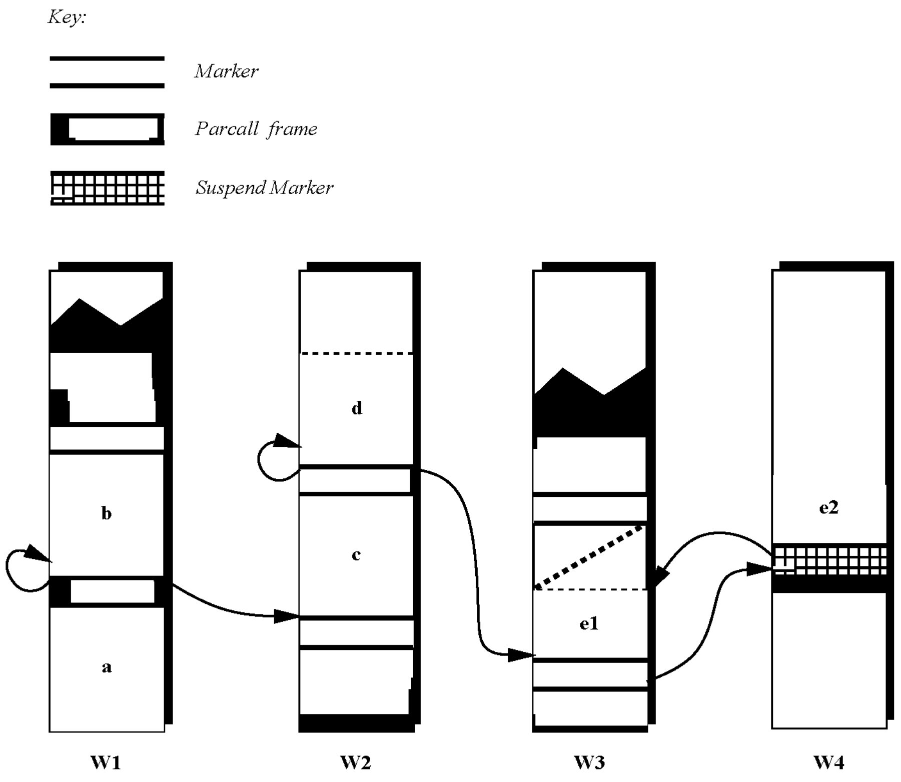

Figure 4: Example Stack State Before Roll-Back

3.4 Multiple Kill/Redo Signals

During a roll-back, a worker may receive other 'kill' or 'redo' signals. Some of these will be to other parts of the stack set, and are independent of the current roll-back. These are accumulated and dealt with one after the other. However, some kill/redo signals would interact with the current roll-back, because they affect the and-task being rolled-back. For example, in figure 4, consider the case of section "a" receiving a kill signal and section "e1" receiving a redo signal when the roll-back of "a" is being performed. Another possible interaction is section "a" first receiving a redo, and later a kill signal.

When a signal is sent to a task, the marker representing the start of that task is marked with a flag (saying that the task is 'to be killed' or 'to be redone'). If a subsequent signal is sent to the task (either propagated from another signal to an ancestral task, or a direct signal to this task), then a 'kill' signal would override any 'redo' signal. This simply means setting the flag to 'to be killed'. Otherwise the new signal is filtered out, as the correct action is already taking place.

4 Conclusions

We have overviewed aspects of memory management in the context of non-deterministic and-parallel systems, which we showed can be considered as a generalisation of memory management in or- and deterministic and-parallel systems. We also discussed how cuts and roll-backs can be handled in our scheme. Although we have concentrated on WAM-derived models which preserve environment stacking, we believe most of our findings should also apply to other stack-based approaches such as pure goal stacking models and, in general, to any system supporting both and-parallelism and don't know non-determinism.[12]

The mechanisms used for suspension are useful for many other purposes, such as implementing constraints, and allowing dynamic expansions of stacks. We are actively researching many of these possibilities that the scheme has opened up for us.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

[2] P. Borgwardt. Parallel prolog using stack segments on shared memory multiprocessors. In *International Symposium on Logic Programming*, pages 2–12, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[3] M. Carlsson. *SICStus Prolog Internals Manual.* Swedish Institute of Computer Science, Box 1263, S-163 12 Spånga, Sweden, Jan. 1989.

[4] J. Chassin de Kergommeaux. Measures of the PEPSys Implementation on the MX500. Technical Report CA-44, European Computer-Industry Research Centre, Arabellaastr. 17, D-8000 München 81, Germany, 1989.

[5] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, ICOT, November 1984.

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

[7] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, The University of Texas At Austin, 1986.

[8] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[12]Note that RAP-WAM uses goal stacking for parallel goals and environment stacking for sequential goals. PWAM (and thus DASWAM) adopt the same general scheme but use the environment information to minimise the memory consumption in the goal stacking part, thus implementing an environment-based goal stacking model for parallel goals [6].

[9] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[10] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[11] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. University of Washington, MIT Press, August 1988.

[12] E. L. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[13] L. Naish. Parallelizing NU-Prolog. In *International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[14] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Proceedings of the Eighth International Conference on Logic Programming*, 1991.

[15] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *Proceedings of the Eighth International Conference of Logic Programming*, 1991.

[16] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Joint International Conference and Symposium on Logic Programming*, pages 717–731, 1992.

[17] K. Shen. *Studies of And/Or Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[18] K. Shen. Implementing Dynamic Dependent And-parallelism. In *International Conference of Logic Programming*, pages 167–183. The MIT Press, 1993.

[19] K. Shen and M. V. Hermenegildo. A Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Internal Report, 1993.

[20] D. H. D. Warren. Implementing prolog - compiling predicate logic programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[21] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025, USA, 1983.

[22] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues. In *Proceedings 1987 Symposium on Logic Programming*, pages 92–102. Computer Society Press of the IEEE, Sept. 1987.

[23] D. S. Warren. Efficient prolog memory management for flexible control strategies. In *International Symposium on Logic Programming*, pages 198–203, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.