A Proposal for a Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs

Kish Shen[§]
Computer Science Department
Bristol University, United Kingdom
kish@acrc.bristol.ac.uk

Manuel Hermenegildo
Facultad de Informática
U. of Madrid (UPM), Spain
herme@fi.upm.es

Abstract

In this paper, we examine the issue of memory management in the parallel execution of logic programs. We concentrate on non-deterministic and-parallel schemes which we believe present a relatively general set of problems to be solved, including most of those encountered in the memory management of or-parallel systems. We present a distributed stack memory management model which allows flexible scheduling of goals. Previously proposed models (based on the "Marker model") are lacking in that they impose restrictions on the selection of goals to be executed or they may require consume a large amount of virtual memory. This paper first presents results which imply that the above mentioned shortcomings can have significant performance impacts. An extension of the Marker Model is then proposed which allows flexible scheduling of goals while keeping (virtual) memory consumption down. Measurements are presented which show the advantage of this solution. Methods for handling forward and backward execution, cut and roll back are discussed in the context of the proposed scheme. In addition, the paper shows how the same mechanism for flexible scheduling can be applied to allow the efficient handling of the very general form of suspension that can occur in systems which combine several types of and-parallelism and more sophisticated methods of executing logic programs. We believe that the results are applicable to many and- and or-parallel systems.

1 Introduction

Systems which exploit implicit parallelism in logic programs are attractive because they allow the achievement of faster execution speeds without an increase in programming complexity. Desirable objectives of such systems are to preserve the programming model, offer actual speedups with respect to state of the art sequential systems, and achieve these results with resource efficiency comparable to that of such sequential systems. Understandably most of the research so

[§]Some of the research reported in this paper was carried out while this author was at the Computer Laboratory, University of Cambridge, Cambridge, U.K.

far has concentrated on the first two objectives. The state of the art is such that several models and some actual implementations have shown a good measure of success at offering a standard programming model and good speedups (*e.g.* [3, 15, 12, 2, 30], *etc.*). With respect to the third issue, resource efficiency, attention has concentrated mostly on processor efficiency, since it is so related to speedup when the number of processors is limited. On the other hand comparatively little effort has been devoted to other issues which can dramatically affect the practicality of an implementation. Perhaps the most important of these issues is *memory efficiency.*

In fact, effective memory management is one of the main reasons for the efficient performance of sequential Prolog systems for realistic, large application programs. It is expected that this issue will become of increasing importance in parallel systems: the need to allocate storage for several execution threads can multiply storage requirements. The evaluation of many parallel implementations to date, focussed as mentioned above mainly on the execution speed and the speed up achieved, has been done generally for comparatively small programs that did not consume large amounts of memory. While memory management inefficiencies will not have a large effect on the execution of these programs if enough raw memory is available, they may be a serious limiting factor when real applications are tackled. Without efficient memory management the favourable speedup results obtained to date with parallel logic programming systems will simply not easily carry over to larger programs which consume large amounts of memory.

In this paper, we examine the basic issues and problems of memory management of certain classes of parallel logic programming systems. We consider mainly "distributed-stack" [5, 10] systems, *i.e.* those which mimic in many ways the sequential execution model (based generally on the WAM [1] or a derivative of it) with the objective of preserving speed and resource efficiency at least during sequential threads of execution. These models are generally based on having multiple sets of stacks ("multiple WAMs") on which the processors can work concurrently. In a previous paper [11] we showed how memory management and scheduling are intimately related. Thus we will also necessarily consider scheduling issues. Our aim is to achieve efficient memory usage while allowing flexibility in scheduling. We propose several more flexible solutions to some of the problems raised in [11], and we provide data which we hope can aid an implementor in choosing a scheme. For conciseness our discussion will concentrate on the particular issues that arise during the exploitation of non-deterministic and-parallelism, given their generality: because of the tree topology of or-parallel execution, and as shown in [11], from the memory management point of view, the problems which appear during exploitation of this type of parallelism are a subset of those which appear during arbitrary non-deterministic and-parallelism, which is topologically more general than a tree.

2 General Approach

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 [27] and the WAM [1]), 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 [1], local stack frames (the WAM "environments") can be often reused, effectively turning recursion into iteration.

Ideally, we would like memory management on parallel systems to achieve the same results as 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 system 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 and/or 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.

In our attempt to devise efficient memory management schemes for the subtree-based approach our starting point is the "distributed stack" scheme (and its restricted version, the "cactus stack" scheme) [5, 29, 10, 28] which has been used repeatedly in implementations because it offers the potential to achieve the above mentioned goal of approaching sequential memory efficiency. This approach is based on the following observation: as each worker executes a task much as a sequential Prolog system would, many of the implementation techniques used in sequential a Prolog system can be adopted for use in parallel execution, hopefully retaining many of the advantages of the sequential implementation. In particular, the stack-based storage model of sequential Prolog systems is extended for parallel execution in the form of a distributed stack model.

Following on the distributed stack idea we assume the program to be compiled into instructions which are quite similar to those of a Prolog engine, with perhaps some additional instructions related to parallelism. We view each of the workers introduced above as composed of a processing element, capable of executing such code in much the same way as sequential Prolog, and its 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 we shall show, it is often useful to distinguish between these two components of a worker. Following [11, 12] we refer to all the storage areas mentioned above as a **stack set** and to the processing element as an **agent**. A worker can be then seen as a given agent attached to a given stack set. 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.

Given a task to work on, an agent can execute the related instructions and use the stack set it is attached to in much the same way as in standard Prolog execution. On the other hand a

significant deviation can appear for example when the task is finished (also when it suspends): If there are more tasks available, and in order to use the 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. Each such area is referred to as a **stack section**. 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** [11]. In addition, depending on the nature of the section above or below them, some markers may serve some additional special functions. The marker scheme 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 [16]), each one corresponding to a "fork" in the parallel task tree. However, in an and-parallel system, not only forks but also "joins" have to be performed on the tasks representing sibling and-goals and more coordination is needed. Thus additional data structures have to be provided to serve as several types of markers.[1] Therefore, an and-parallel marker scheme can be regarded as a generalisation of an or-parallel marker scheme. In this paper we will concentrate on the more general and-parallel scheme, with the understanding that the solutions proposed and results obtained can be applied to an or-parallel scheme, perhaps with simplifications.

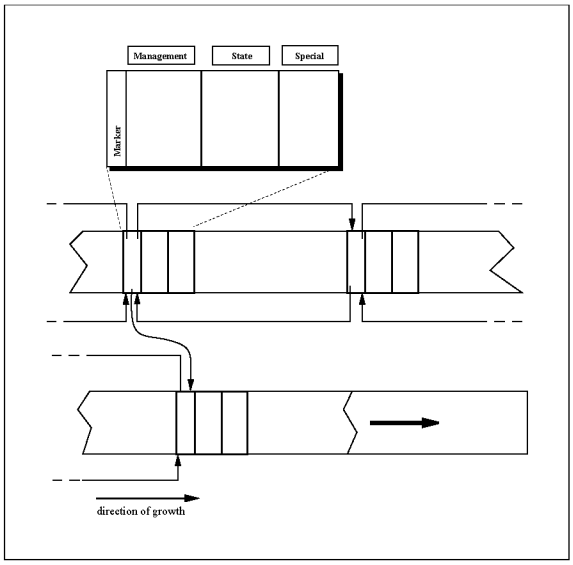

Figure 1: Structure of a marker

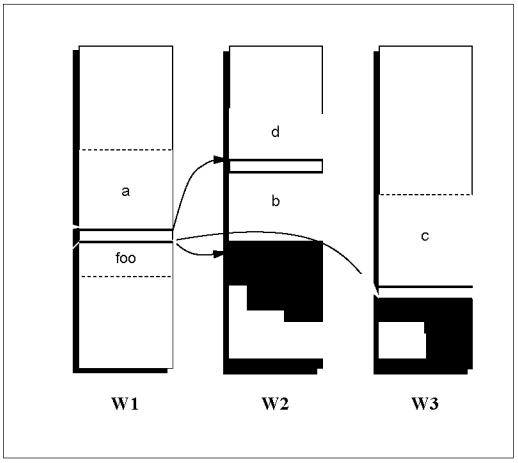

Figure 2: Stack states for a distributed stack scheme

The general structure of a marker is shown in Figure 1. A marker is divided into three general parts, a *management* part for managing the marker and sections, a *state* part which stores the pointers to the various stacks, and the values of some other state registers when the marker was allocated, and a *special* part, which serves the special requirements of the various types of markers (and is empty for the basic markers).

Markers are linked to the next and previous markers in the same stack set. The marker can contain a third pointer, the continuation pointer, the use of which will become apparent in the

[1] It is possible to use only choice-points as markers if and-parallelism is restricted to "deterministic" goals, as in systems such as PNU-Prolog [18] and Andorra-I [30].

following sections. Figure 1 shows a task (shaded gray) started in the top stack fragment, and continuing in the bottom stack section.

For concreteness, we assume the and-parallel execution of body goals inside a clause is managed by a P_Call Frame, which can be viewed as being part of the environment (and is thus implemented in &-Prolog [12]), or as a type of marker (and is implemented as such in DASWAM [21]). Each and-goal is represented by a "slot" in the frame, which holds enough information to allow the forward and backward execution of the goal. In addition, the P_Call Frame holds a pointer to the task following the completion of the and-parallel execution. This allows the last agent to finish an and-goal in the CGE to continue the execution after the and-parallel goals without changing stack set. Figure 2 illustrates the use of the marker scheme (with the P_Call Frame considered as a marker) to represent the following &-Prolog program fragment:[2]

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

A possible parallel execution of this clause is shown in Figure 2. 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. 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*. Markers are also used to mark the start of an CGE (*e.g.* the one separating `foo` from `a`). These specialised *parcall markers* (which correspond to P_Call Frames) 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.

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" [11] (also referred to as "holes" in or-parallel systems). In a previous paper [11], solutions were proposed to solve these problems by placing some constraints on how goals should be scheduled. In this paper, we extend this previous work by developing mechanisms that allow for more flexible schedulers.

In addition, we discuss the implementation of *suspension* and *resumption* of a task. The scheme proposed allows a worker to suspend a task and work on another task, with the suspended task resumed later on by any worker. Suspension is very important for at least the following reasons:

- it allows the implementation of Prolog side-effects by allowing a task to suspend when it tries to perform a side-effect. This approach can give rise to more parallelism than in a synchronisation block approach (*e.g.* [9, 17]).
- it allows the implementation of more flexible scheduling strategies, *e.g.* strategies that give lower priority to potentially wasted work (*e.g.* [23, 4]).

[2]The scheme described here is an extension of the original scheme, where the stack set that started the and-parallel execution had to be the one used for the task following the completion of the and-parallel execution.

- it allows the implementation of dependent and-parallelism, by allowing a task to suspend when it does not meet the condition of and-parallel execution (*e.g.* [21]).

- it allows the delaying of execution of some goals, which can be used to implement more sophisticated control than in Prolog, *e.g.* delayed execution ([18, 30]) and constraint satisfaction [26].

- By generalising the suspension mechanism further, then when the space used by a stack is exhausted, additional memory for the stack can be allocated somewhere else in the virtual memory. This allows stacks to be expanded dynamically during execution. Unlike conventional stack shifting in sequential Prolog, no copying of the old stack to a new area is needed, as disjointed stacks are already allowed. Stack shifting is important for a practical sequential Prolog system, but is of even more importance in a parallel system where different stack sets in a system may have very different stack usage requirements.

3 Scheduling Issues

As already mentioned, two important problems which may arise in the distributed stack scheme are the "garbage slot" problem and the "trapped goal" problem. Under the full generality of the distributed stack scheme, each stack set can have many stack sections, and backtracking can occur in these stack sections at any time. Essentially, the "garbage slot" problem comes from the fact that if backtracking is not performed at the top of the stack set, it will leave "holes" in the stack, essentially areas of wasted memory. The "trapped goal" problem occurs when forward execution is to be resumed on a non-topmost stack section that has been backtracked. As the section is not topmost, the task cannot simply proceed blindly using the space available in the stack section as its growth will be blocked by later sections.

As shown in [11] these problems simply will not occur if "age" constraint is imposed on the sections that can be allocated on a stack set. The simple rule is that a task can only be executed on a stack set if it is guaranteed that it will always be backtracked over before all the tasks allocated before it in that stack set. This amounts in practice to a notion of a task being "appropriate" for execution on a given stack set and to restricting scheduling in such a way that the age constraints are respected – this is referred to as the "goal restriction" method. In addition to avoiding the problems mentioned above this restriction has the additional advantages of minimising the number of markers, and the communication needed during backtracking.

The greatest concern with the "goal restriction" approach is that, assuming a simple arrangement of n workers (agent/stack set pairs), parallelism can become limited since these workers may not find appropriate tasks even though other tasks were available. In order to evaluate the impact of goal restriction on speedup we have implemented both a goal restriction scheduler and a flexible scheduler (using the method to be described later) in one of our systems (the pseudo-parallel DASWAM) and compared the speedups obtained from both schedulers for the benchmark programs "boyer", "orsim", "tak", and "bt_cluster". The first three were used in our previous high-level simulation study [22], but, since the limitations of the high-level simulator are not present larger problems are tackled. Boyer is *boyer_nsi* from that study, but here the theorem used in Tick's original query [25] is used. Orsim is the same as the *orsim* program from the simulation study (*i.e.* one of the simulators themselves), and it simulates the program *atlas* with its full database.[3] Tak is the same program as in the simulation study. The last program,

[3] orsim(sp2) in [22] was simulating the atlas program with a reduced database.

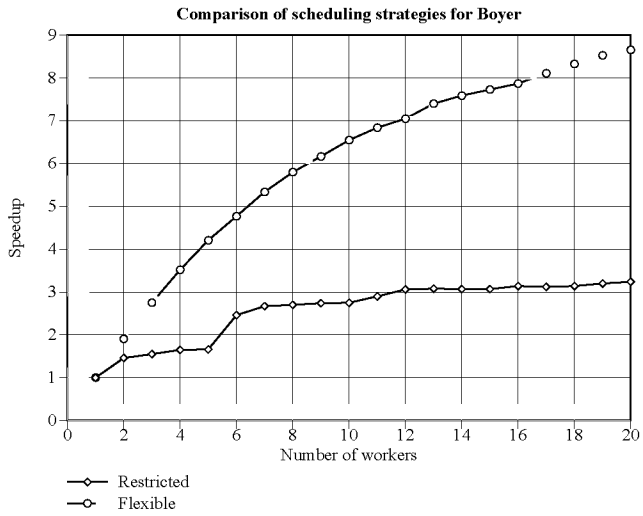

Figure 3: Speed-up for Boyer under the two scheduling strategies

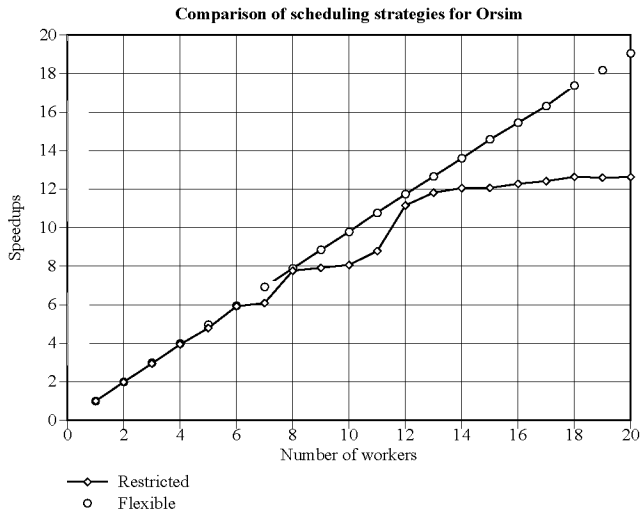

Figure 4: Speed-up for Orsim under the two scheduling strategies

bt_cluster, is Tony Beaumont's modified version of British Telecom's original clustering algorithm. As can be observed in Figures 3–4, the results for the first three programs show that the flexible scheduling is substantially better (bt_cluster showed the same results for both scheduling strategies).[4] In most cases, the graph for the restricted scheduling shows regions where the speedup curve becomes nearly flat, *i.e.* where adding workers does not seem to improve the

[4]Note that the speedups are simulated speedups. The cost for locking is not taken into account. In addition, the cost for determining an appropriate goal in the restricted scheduling, is also not taken into account.

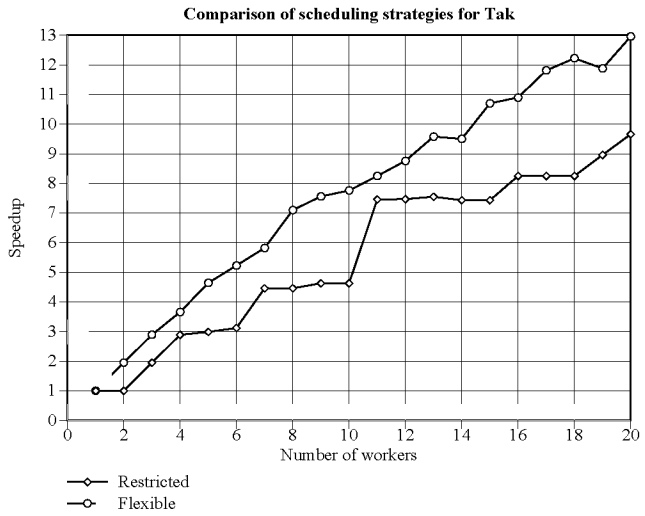

Figure 5: Comparison of speed-ups for Tak under the two scheduling strategies

execution time of the program, until suddenly the performance jumps.

The reason why the restricted strategy can sometimes be much worse than the flexible strategy, as in the flat speedup areas encountered above, is probably because it can limit the ability for a worker to seek new work very early on. Consider the a simple clause "`foo :- (a & b)`" where `a` generates more parallelism. A worker which executed `b` (or its subgoals) would not be able to exploit the parallelism created by `a` since all the computation in `a` and its descendants is younger than those in `b` and therefore it is not appropriate for the worker's stack set. From the results presented it appears that such situations can be quite common. The reason that the speedup curve for the restricted scheduling flattens out frequently is because `b` in the above case may initially contain some small amounts of parallelism early in the execution, whereas the more significant parallelism occurs deeper in `a`. Thus new workers will initially exploit the parallelism in `b`, giving only small improvements, and then being restricted from exploiting the more profitable parallelism in `a`. Finally, a newly added worker (with empty and thus unrestricted stack set) can exploit the parallelism in `a` giving a performance jump. It should be noted that, as workers are added, both scheduling strategies will eventually achieve the same maximum speedup, but a much larger number of workers is needed.

While speedup alone can be a determinant in choosing a flexible scheduling strategy, another problem is the overhead involved in checking tasks for appropriateness, which was not included in the previous measurements. Methods for doing this have been suggested in [11], by Lin [14], and (under a slightly different context) by Tebra [24]. However, in the first scheme the cost can become unbounded both in terms of space and time, and in the other two cases the amount of exploitable parallelism is artificially limited. This is therefore an additional argument in favour of flexible scheduling.

Although the exact impact of goal restriction was not known at the time, a scheme for allowing a certain degree of flexible scheduling was outlined in [11]. This scheme is based on the above mentioned observation that the speedup of both schemes will be identical if enough

workers are present. However, of the parts that comprise a worker (agent/stack set) it is really only the stack set that has an age: the agent has no state and could work on another stack set. Stack sets correspond to memory and agents to processors. Of the two, processors are in principle the scarcest resource in a typical parallel machine. The scheme improves the utilisation of agents at the expense of memory consumption: it allows the creation of more stack sets than agents, and the movement of agents from one stack set which is too old to run the currently available goals to another one which is free (no agent is working on it) and is young enough. However, there are still problems with this approach:

- As there are more stack sets than workers, not all stack sets will be used at any point in time. Thus, the storage available in the unused stack sets can actually be also seen as a "hole". As a significant amount of memory is needed for each stack set, and the requirements of the scheduling strategy can require the creation of a large number of stack sets, the multiple stack set approach can lead to a large consumption of (virtual) memory. This may be less of a problem perhaps in future systems, but is certainly a problem in current ones. In particular, consider the case for Boyer: the speedup here for the flexible scheduler increases much quicker than for the restricted scheduler, so many more stack sets would be needed in the restricted strategy before the same speedup as the flexible strategy is achieved. This could be ameliorated by allocating small stack sets and allowing them to be continued in other stack sets, but, as we will see, this amounts to a similar level of complexity than the scheme we propose, which would however leave less holes.
- The problem of the overhead involved in checking for appropriate goal remains.
- Goal selection can still be restricted in the end in practice. In a real system, the number of stack sets will necessarily be limited, possibly to at most a few times the number of agents available.
- The same inefficiencies will appear when dealing with suspended tasks. In order for backtracking to occur only at the top of each stack set, only "younger" tasks can be allocated on top of a suspended task. If suspension is frequent (as is the case in many dependent and-parallel programs), then the stack sets will rapidly all become suspended, preventing further parallel work.

Because of all the reasons above, and the already discussed usefulness of suspension, clearly a scheme that allows flexible scheduling is desirable. We describe such a scheme in the following section.

4 Dealing with Garbage Slots and Trapped Goals

As an alternative to the appropriate goal schemes, we propose a flexible scheduling scheme which places no restrictions on what task can be started on a stack set after the current task completes or suspends. As we have shown in the previous section this lack of restrictions on scheduling will allow better speedups than with previous approaches. However, the "trapped goal" and "garbage slot" problems, which were effectively avoided in the appropriate goal scheme, will now have to be dealt with in some way.

It is first important to note that these two problems are not of the same importance. Garbage slots in themselves are not a great problem, as their space can eventually be recovered through

backtracking and/or by garbage collection. In fact, in or-parallel systems (where only the "garbage slot" problem can occur) it is often tolerated: Aurora [15] allows garbage slots inside its worker's stack. A study of actual performance of the or-parallel PEPSys system suggests that selecting only "appropriate" goals also gives poorer performance in such systems than a more flexible goal selection scheme which can select any goal, even after accounting for the overhead involved in the more complex stack management and the memory loss due to the garbage slots in the stack [8].

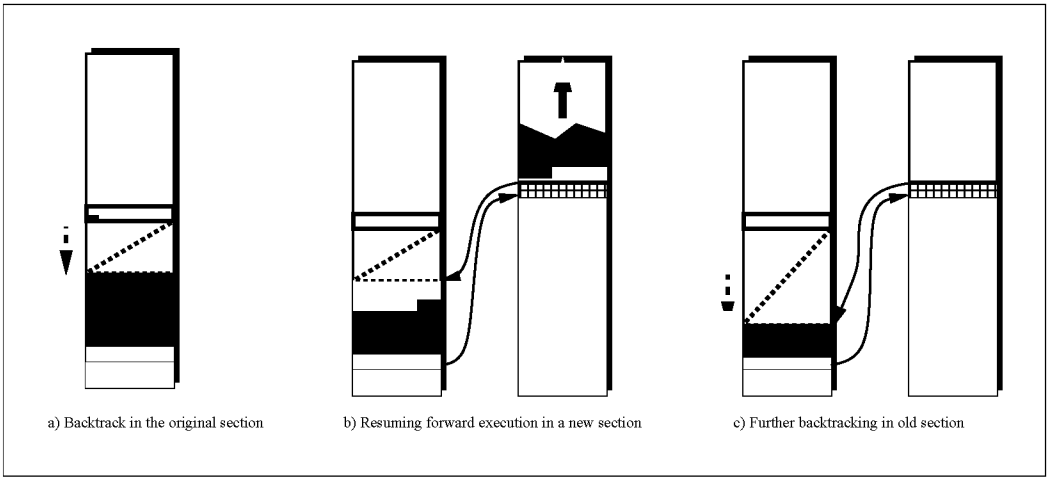

Figure 6: Solving the trapped goal problem

On the other hand the trapped goal problem is more serious: it must be dealt with in some way if a flexible goal scheduling scheme is to be used because trapped goals may simply result in overwriting of valid memory areas and thus incorrect execution. We propose to deal with the problem by introducing a special type of marker: the *continuation marker*.

Essentially, a continuation marker allows the execution of a task to continue in another part of the distributed stack. Thus, the state of a task can spread across one or more disjointed sections. These sections are linked into one logically contiguous section by the continuation markers. The use of the continuation marker for solving the trapped goal problem is shown in Figure 6. The darkly shaded areas correspond to the stack space currently used by a goal, and "garbage" space is indicated by a diagonal dotted line. At first, backtracking takes place in the original stack section, creating some "garbage" space as indicated in Figure 6a. When forward execution is resumed, a continuation marker is allocated on the top of the stack of a free worker, and execution and growth of the stack is resumed at the new worker, instead of at the original stack section. The continuation marker contains a pointer pointing to the top of the still valid work in the old stack section, as shown in Figure 6b.

The system has to be able to handle the backtracking of a goal across the disjointed sections. If backtracking occurs in the new stack section, and it continues to the continuation marker, further backtracking is facilitated by following the pointer in the continuation marker to the old section, the old section is shrunk further, and forward execution is resumed with the pointer in the continuation marker updated, as shown in Figure 6c. Space used by a goal can of course be spread across many stack sections, so the backtracking on a "trapped" section may recursively lead to a continuation marker that points to an even older section for the same task. This is easily handled by the continuation markers.

A question remains however: who should be responsible for performing backtracking on the

old section? One possibility is to allow only one agent to work on any stack set at a time — the "owner" of the stack set. This agent can be interrupted from whatever it is doing to do the backtracking. Another alternative is to allow the originally backtracking agent to continue the backtracking. This backtracking is called **remote backtracking**, as it is performed by an agent "remotely", on a stack set that it does not currently own. The advantage of this approach is that the owner of the stack set is free to continue its own work, and multiple backtrackings can occur at the same time in any stack set, giving more scope for parallelism during backward execution. This second approach was taken in both &-Prolog / PWAM [12] and DASWAM [21], in which the scheme proposed has been implemented. In this approach, the general organisation of a stack set is such that there is only one point at any time where the stack can grow — the top of the stack in the topmost section (the **stack tip**); but there can be multiple remote backtrackings (up to one per each non-topmost section) going on in the non-topmost sections at the same time.

One disadvantage of remote backtracking is that it is more complicated and requires more locking and synchronisation than ordinary backtracking. A marker stores part of the state of a worker — the value of various state registers (including all the top of stack pointers) at the time the marker is allocated. This state facilitates the start of remote backtracking. The marker following a section stores enough state information to allow backtracking to start on that section before (below) it. When remote backtracking is first initiated, as in Figure 6a, the new worker reads the values of the various state registers in the marker into its own corresponding registers (having saved the original values beforehand), and remote backtracking is started. When an alternative is found, the original values of the registers for the remotely backtracking worker are restored, and forward execution is started in that worker's top of stack.

With the flexible strategy, a limited form of selecting 'appropriate' goals can still be performed at very little cost: when a task is successfully completed, the worker may then try and select an appropriate sibling and-goal of the just completed task as its next task. That is, it can select a goal in the same CGE that is to the right of the just completed task. This can be done very cheaply because all sibling and-goals of a CGE are recorded in the P_Call Frame. If no such goal is available, any available goals can then be selected.

5 Dealing with suspension

As already discussed, suspension is very useful for many purposes. The same mechanism used for the flexible goal scheduling can be extended to allow new work to be started on top of a suspended stack section so that the suspended task can later be continued elsewhere in the distributed stack when the task is allowed to resume. The continuation marker can be used to allow this continuation. An additional marker is needed to record the state of the worker at suspension so that the state can be restored when the task is resumed. The suspension marker saves enough of the state of a worker to allow the goal to be resumed elsewhere. The amount of information that needs to be recorded depends on how general the suspension is (*i.e.* where it is allowed to take place). For example in &-Prolog and in many other systems, suspension takes place before a goal is called, and in this case, the argument registers do not need to be recorded. In the case of DDAS [21], suspension can occur at any point from the call of a goal to when head unification succeeds, and thus the argument registers need to be preserved. This new marker is called a *suspension marker*. An alternative is to push the state on to the global stack, using mechanisms similar to those used to implement delay primitives in standard Prolog

implementations [6].

6 Dealing with Variable Age

Using the flexible scheduling scheme outlined above, there is in principle no need to keep track of task age for scheduling purposes. However, there is still another reason why some scheme for detecting certain ages of objects may be needed: the treatment of unbound variables. In the WAM, when two unbound variables are unified, the older variable is set to point to the younger variable where, since the memory areas are contiguous (and the local and global stack arranged in the appropriate way) is done by a simple address comparison. This is needed to prevent dangling pointers when the environment is deallocated because of last call optimisation or during backtracking. In our parallel approach, we maintain all sequential optimisations inside the execution of each task, thus we also need to deal with the variable age problem. However, it is complicated by the possibility of binding two unbound variables in different stack sections. The mechanism for recognising precedence for selecting "appropriate" goals can also be used to decide how two unbound variables should be bound together [11]. However, once the need for this mechanism is obviated, in the flexible scheduling scheme it may be an unnecessary overhead to preserve this mechanism only for variable age comparison purposes.

We propose three alternative ways to solve these problems. One way is to allow two variables in different stack sections to bind in an arbitrary order (address comparisons can still be used for variables in the same stack section). This means that such variables would have to be globalised and/or trailed in order to avoid dangling pointer problems. As it cannot in general always be known at compile time which variables would be bound to other variables outside their own section, this probably means that an undetermined number of variables will have to be globalised. This scheme has the advantage that it does not require modification to the variable representation. The main disadvantage is that the benefit of splitting the variables into local and global variables is lost and that some additional trailing may be done. Furthermore, it is not clear if the separation into local and global variables is such a great advantage, and recent high-performance sequential Prolog systems such as Van Roy's BAM [19] does not distinguish local and global variables. Thus, this may be a reasonable approach.

Another way to deal with this problem is based on the observation that in order to ensure the correct binding of the local variables, a much less general scheme is needed than that required for determining when a task is "appropriate". In the WAM, a sequentially older local variable can only be directly unified with a younger local variable if the younger variable occurs in a subgoal of the goal the older variable occurs in. Thus, to maintain the correct binding order in local variables, only the ordering of stack sections within the same and-goal is needed. This can be done by associating an age level with each stack section. The age is incremented each time a parallel conjunction is encountered (with each sibling and-goal given the same age to start with), and also each time a new section is started. The age level must also be associated with each variable, indicating when the variable was created. This is implemented by introducing an "unbound" tag to represent the unbound variable, with the "value" part representing the age level of the variable. The mechanism for doing this is similar to the unbound tag needed for maintaining binding arrays in the or-parallel SRI model [28].

The third solution is a combination of the two solutions proposed above: rather than keeping the age of all unbound variables, tags are kept only for the stack sections (storing them in the corresponding marker). Addresses can then be directly compared if they are in the same stack

section. Otherwise the addresses of the markers are compared. The success of this approach (as that of the first one) depends on how easy it is to determine to which segment an address belongs. This is rather easy if stack section allocation is segmented, but that can result in inefficient use of storage.

7 Dealing with Cuts

In a sequential WAM such as SICStus' WAM, 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 control stack register to point at 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 SICStus WAM 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 point has to be removed from the previous stack sections, until the choice point to cut to is reached. This is done by following the pointers in the marker 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 section are bounded by a 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 was allocated, this field is set to point at the top of control stack. When a cut operation is performed, this last valid choice point pointer is set to point either at 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 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` is pruned when the cut is encountered, using the methods just described. The slot 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 has returned a solution, *i.e.* `b` is pruned when

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
Atlas, seq.	17200	-	15	9	8	7	-	39
Atlas, flex	17800	0	17	54	8	10	2	91
Atlas, flex2	17800	0	17	54	8	12	0	91
		1	2	12	0	0	0	12
		T	19	66	8	12	0	103
Atlas, res2	17800	0	17	54	8	12	0	91
		1	2	12	0	0	0	12
		T	19	66	8	12	0	103

Table 1: Absolute Memory Usage - Atlas

`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. As the markers 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. 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.

8 Memory Performance

The results corresponding to flexible scheduling in Figures 3–5 were obtained from direct measurements of our implementation of the memory management and flexible scheduling schemes presented in the previous sections, using remote backtracking and variable age tags. These results show that the approach proposed can effectively achieve better speedups than the existing restricted scheduling based approaches. However, and as mentioned before, we are interested not only in speedups, but also in memory efficiency. In this section we study the efficiency of the memory management and flexible scheduling proposed and compare it to that of sequential systems. We also present results using the proposed memory management scheme but with restricted scheduling, and compare them to the results for unrestricted scheduling. The results are presented in Tables 1 to 3 and Figures 7 to 9 for the boyer, orsim, and atlas programs. The atlas program is a standard Prolog benchmark program, and has been used as a benchmark also in our high-level simulation studies [22]. It is chosen because it is a program with a little and-parallelism and much backtracking, which is unlike the other two programs.

Tables 1 to 3 show memory usage figures, all of them taken just before the end of the execution of the program (at the end of the execution all storage is recovered through backtracking). The first column indicates the program. The second column is the time, expressed in terms of abstract machine instructions executed, at which the stack sizes were measured, the third column is the

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
orsim seq.	9050000	-	3683	1608	600832	148365	-	754488
orsim flex	9069226	0	45965	89366	601548	230624	0	967503
orsim flex10	926152	0	4666	9172	61135	22562	0	97335
		1	4465	8792	59563	22938	0	95758
		2	4841	8669	60726	22610	0	96846
		3	4465	8464	58930	23230	0	95089
		4	4880	8568	61487	22798	0	97733
		5	4277	8824	58133	23230	0	94464
		6	4503	9202	59113	23382	0	96200
		7	4541	8545	59655	22510	0	95251
		8	4390	8915	59379	23318	0	96002
		9	4955	9400	62551	22996	0	99902
		T	45956	88551	541017	229574	0	905098
orsim res10	1124508	0	4517	8462	57861	21628	0	92468
		1	4466	8468	57913	21814	0	92661
		2	5590	11768	72294	28284	0	117936
		3	4203	8084	55921	21428	0	89636
		4	5591	10862	74033	28202	0	118688
		5	3677	8001	49708	19358	0	80744
		6	3790	7039	49873	19142	0	79844
		7	4542	7629	57593	21516	0	91280
		8	5329	9991	69849	26636	0	111805
		9	4278	9037	56362	22532	0	92209
		T	45983	89341	601407	230548	0	967271

Table 2: Absolute Memory Usage - Orsim

id. of the stack set for which the measurement is taken (we are assuming one agent per stack set). This is followed by the actual memory utilisation for the for the four main stack areas used by WAM/PWAM/DASWAM, in words (32 bits in size) and for the PWAM/DASWAM goal stack. The last column is the total memory usage in that stack set. Individual stack utilisation is for each stack set, in the cases where there is more than one. In those cases the last row represents the global utilisation for the corresponding type of stack. The last column of this row indicates the total overall amount of storage used. In addition to the precise figures for one instance of time, we also present the evolution of memory usage over the execution of the whole program in Figures 7–9. In these cases, only the total memory usages of all workers (in the cases where there are more than one) for the particular stack is shown. The data for the goal stack is not shown as its usage was relatively quite small.

Results are given for a sequential WAM model based on the SICStus abstract machine [7], which contains several optimisations over the original WAM [1], running the unannotated program (*i.e.* with no parallel constructs); for PWAM/DASWAM running the annotated program on one worker, using flexible scheduling (the restricted scheduling results are the same for this case), and for the annotated program on ten workers both using flexible and restricted scheduling. In the case of atlas, the parallel cases are for two workers, since that is the maximum amount of parallelism. Note that the behaviour of PWAM/DASWAM when running an unannotated program by a single worker is virtually identical to that of the WAM. In any case the worst case scenario of the larger markers and the use of variable age tags, which requires the trailing of two words per variable instead of one, is measured. In addition, the P_Call Frames are allocated on the control stack.

The figures show that the memory usages for the flexible and restricted strategies are very

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
Boyer, seq.	138000	-	14	0	7351	2542	-	9907
Boyer, flex.	138410	0	22465	131712	7243	13776	0	175196
Boyer, flex10	21134	0	2350	13811	813	1430	0	18404
		1	2303	12999	733	1344	0	17379
		2	2085	13078	658	1346	0	17167
		3	2279	13264	734	1402	0	17679
		4	2021	12892	642	1328	0	16883
		5	2018	12729	607	1298	0	16652
		6	1975	12133	602	1334	0	16044
		7	2601	13698	893	1452	0	18644
		8	2517	13432	783	1322	0	18054
		9	2320	13208	758	1424	0	17710
		T	22469	131244	7223	13680	0	174616
Boyer, res10	50292	0	5872	34341	1921	3620	0	45754
		1	565	2946	169	286	0	3966
		2	2	48	0	0	0	50
		3	1500	8531	517	880	0	11428
		4	1298	7480	441	800	0	10019
		5	6023	35415	1919	3694	0	47051
		6	1069	6401	358	658	0	8486
		7	318	1855	110	210	0	2493
		8	5782	34359	1796	3608	0	45545
		9	54	336	12	18	0	420
		T	22483	131712	7243	13674	0	175112

Table 3: Absolute Memory Usage - Boyer

similar, showing that "garbage slots" are not a real problem, at least for these examples. One additional advantage of the flexible strategy is shown by the stack usages of individual workers. In orsim and boyer, which show good speedups, work is divided reasonably evenly between the workers, so the memory usage is divided also reasonably evenly: each worker uses about $\frac{1}{N}$ the amount of memory of the 1 worker case, where N is the number of workers. In the case of the restricted scheduler, the memory usage is divided less evenly as some workers are prevented from working because no appropriate work is available. This difference in utilisation can make the system run out of memory in one stack set when there is still plenty of space in others and force potentially expensive stack set shifting.

Turning now to a comparison between the memory usages of the parallel case for different numbers of workers (the tables show this for one and ten or two workers) we observe that the total amount of memory used is in fact very similar. That is, using more workers does not seem to increase the total memory usage. In fact, the detailed figures would suggest that utilisation decreases slightly, though this might be just a side effect of sampling at one particular point. This preservation of the amount of memory usage is unlike many other parallel schemes, where such usage would increase with an increasing number of workers.

It is also apparent that our approach is able to effectively recover memory on backtracking, as in a sequential system, as shown by Figure 9. This result is further supported by data for the system running programs with dependent and-parallelism, presented in [21].

We now compare the memory utilisation in the parallel case with that of the sequential case. We can observe that the total memory utilisation of the control and local stacks is about the same in the parallel case as in the sequential case for atlas, but much greater for orsim and boyer. The usage of the trail is also generally greater, but this is due mainly to the additional

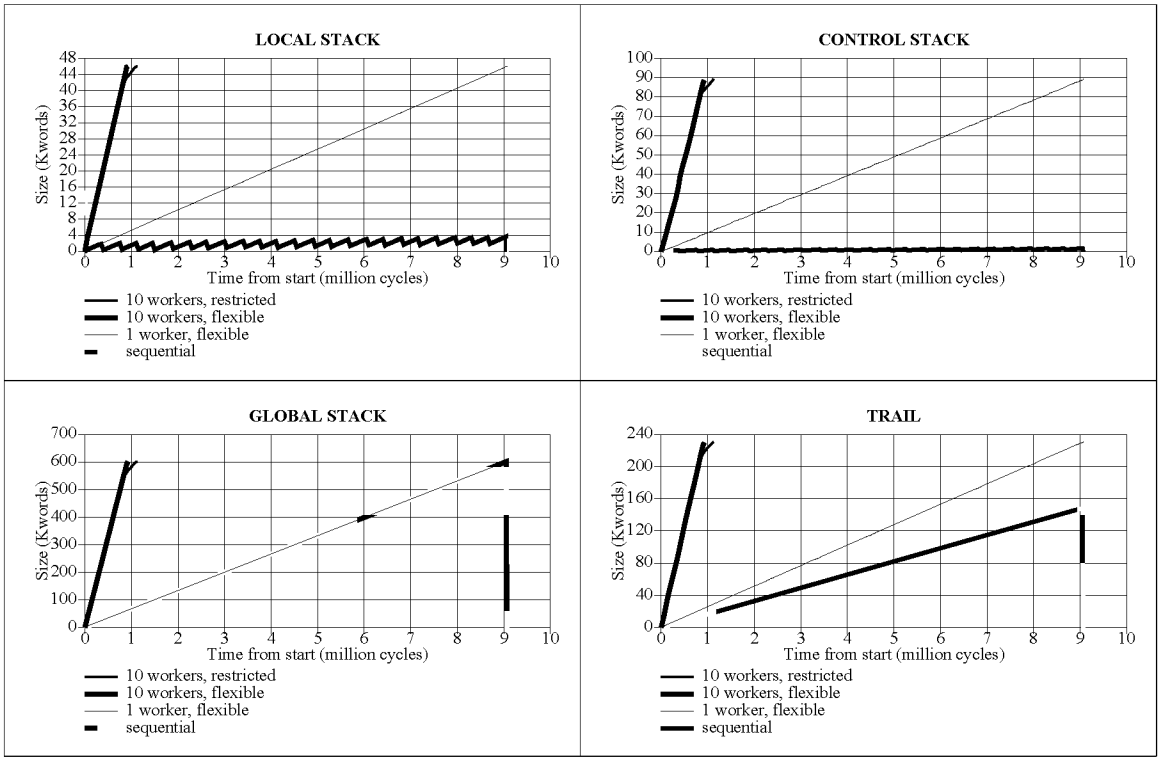

Figure 7: Memory usage for orsim

cost of variable age tags. The usage for the global stacks, however, is very similar in all cases. In fact, in the graphs the lines representing the various configurations in the global stack often merge into each other, because their usage is so close. The fact that the global stack usages are similar for the sequential and parallel cases suggests that the variable age tag method is able to preserve the optimisations of the two stack model, without globalising many more variables than in the sequential case, although at the cost of larger trail utilisation. Turning to the combined memory usages of all the stack sets, the parallel system uses between from 30% more memory for orsim to about 18 times more memory for boyer than the sequential system. Both boyer and orsim are close to worst cases from the point of view of total memory utilisation for our approach for a number of reasons: they are highly deterministic and recursive, with much fine grain parallelism which includes the last goal in tail recursive clauses. This, as we will see later, is what creates the large control and local stack utilisations. Furthermore, in boyer the global stack is not very heavily used, which makes the control and local stack figures more directly affect the total.

While the memory consumption is high in absolute value, specially in the case of boyer, it is not that high from the point of view of standard measures often used for parallel systems.[5] These measures are based on integrating the memory demand in time. Thus, the fact that in our approach memory utilisation is spread evenly among workers means that *individually* they do not use excessively more memory than the sequential case and, although the overall memory consumption is higher, that memory is used for a shorter period. In the case of orsim, the

[5]Thanks to David H. D. Warren for pointing us to these measures.

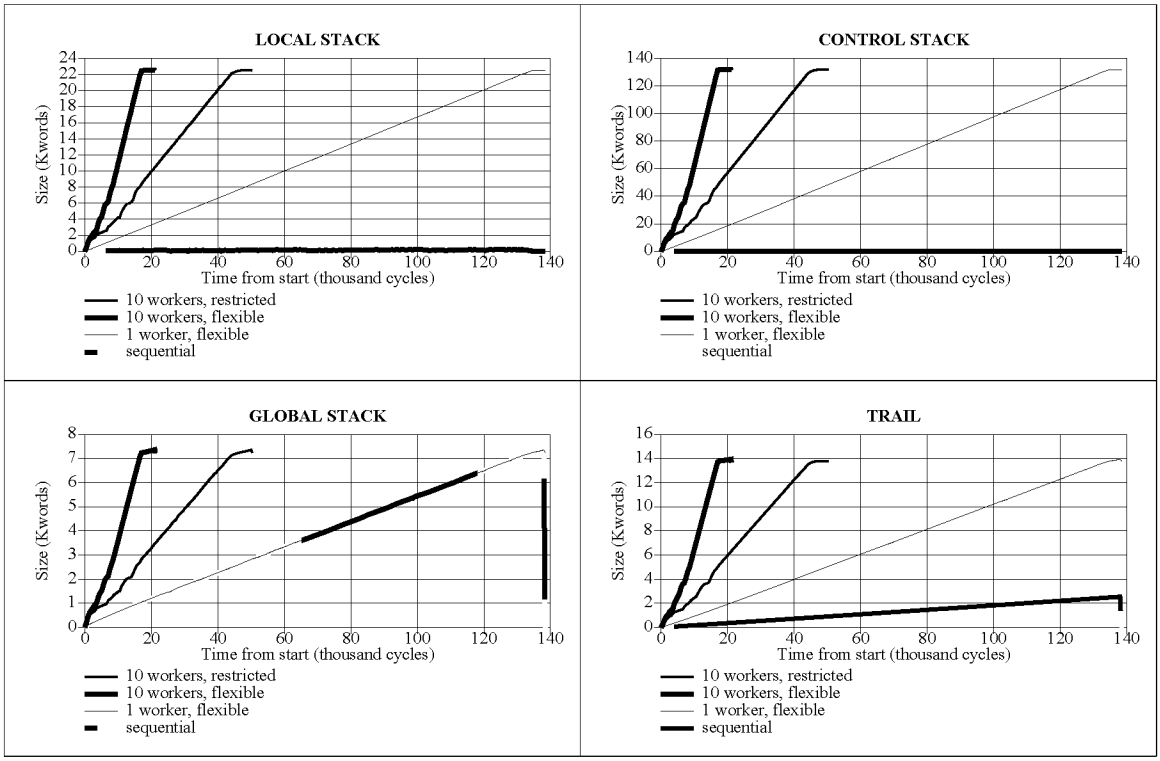

Figure 8: Memory usage for boyer

memory usage of each worker using the above mentioned measure is actually much smaller than the sequential usage. For boyer, the demand for memory is actually not much worse in the parallel case than in the sequential case: although boyer uses about 18 times more memory in parallel at the end of the execution (figure 8 suggests that this ratio grows steadily from the start of execution), it also runs about 6.5 times faster, so the actual demand is generally less than twice as much as in the sequential system. Moreover, as the total amount of memory used does not seem to increase with increasing number of workers, whereas the speedup does increase, the integrated memory demand would decrease with increasing number of workers, and eventually (in this case probably at around 20 workers) be less than that of the sequential case. Of course, this measure of memory usage assumes that a parallel machine has more real memory than the sequential machine. In fact, ideally, a machine with N processors should have N times as much real memory.

However, independently of the fact that the memory demand achieved can be considered reasonable, we feel there are ways the situation can be improved. For this purpose we will discuss the reasons behind the additional consumption in the different stacks, reason about how much it is ideally possible to achieve, and propose some solutions.

In sequential execution, last call optimisation allows the local stack's memory to be reused. When this last goal is executed in and-parallel, we cannot expect it not to use more space, as what was executed sequentially and which would thus allow the re-usage of the same space, is now executed in parallel and thus must use separate spaces. This is a similar situation to or-parallelism, except that in that case the separate usage occurs all the time (instead of being

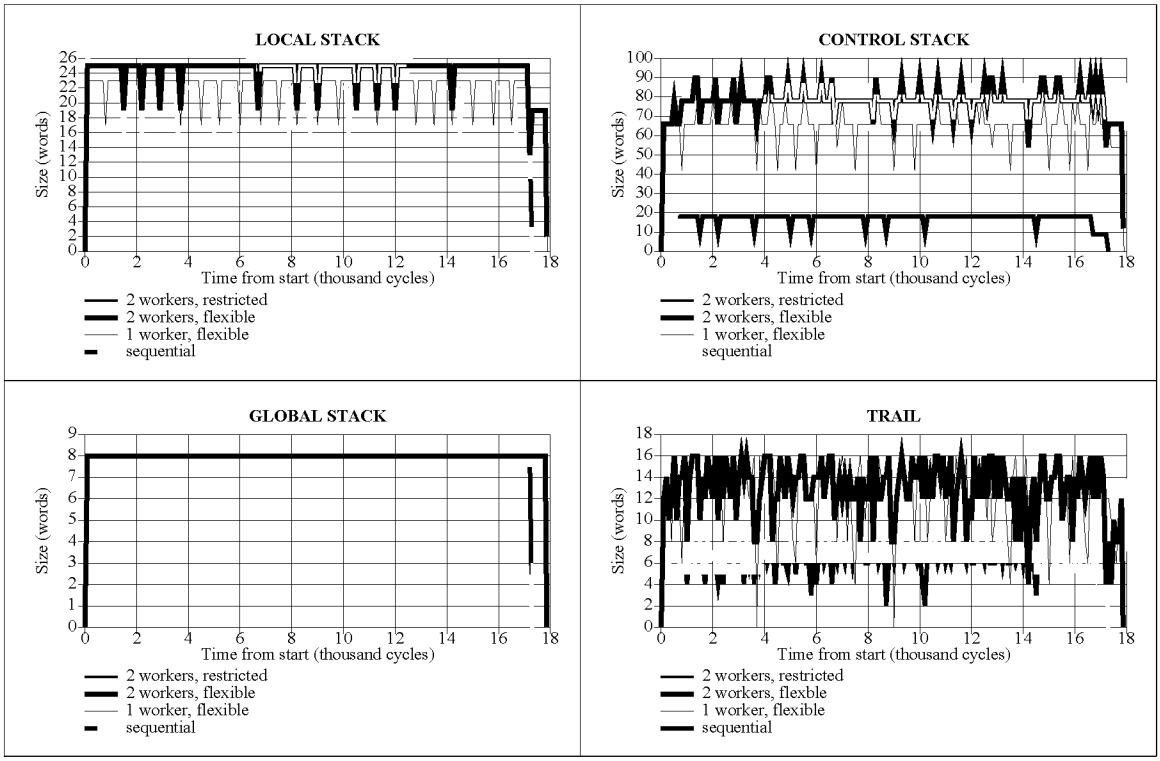

Figure 9: Memory usage for atlas

just a special case), and for all stacks. However, it is reasonable to expect the ideal usage for *each* worker's local stack to be as low as the sequential case. This is not currently achieved by our systems. Thus improvement certainly seems possible, possibly by applying some forms of last call optimisation even in parallel (currently nothing special is done). This could be done for example by having the last goal to be picked up in the parallel call deallocate the environment when after it has constructed its arguments and by preventing parallelisation when it is detected that local execution will occur (*i.e.* parallel goals will end up executed on the current stack set by the current agent). Garbage collection can also be used to recover this memory.

Regarding the control stack, and for deterministic, fine grain cases such as boyer, this area is lightly used in the sequential execution because few if any choice points are allocated. In contrast, for the parallel case, it is to be expected that a marker scheme would use more memory: many markers are allocated on the control stack, because many parallel goals are generated. On the other hand our figures represent an upper bound on consumption since they assume that the markers contain all the information to support dependent and-parallelism. For a purely independent and-parallel system, the markers are actually smaller, and thus the usage of the control stack would also be smaller. Furthermore, our measurements do not include any of the many optimisations which are possible in which the sizes of markers allocated are reduced. These optimisations include allocating smaller markers when it is known that the parallelism exploited is independent; allocating only minimal markers when the goal executed is deterministic (and does not fail), when the same worker is executing successive sibling and-tasks; reduction of the sizes of markers during garbage collection, *etc.* We are actively researching ways to detect and

apply some of these optimisations.

Finally, the trail usage figures are quite interesting. The test for trailing is different in the parallel and sequential case. In the sequential case, a binding is trailed if one or more choice points were allocated between the creation of the variable being bound and the binding. The trail test can be implemented by keeping track of the last choice point and comparing its age to that of the variable. A similar scheme can be used in the parallel case, except that the last choice point does not always correspond to the one in the sequential case. The last choice point at the start of execution of each and-goal in a CGE is the last choice point before the entry into the CGE. This can mean that less bindings will be trailed in the parallel case, as is reflected in the trail usage for orsim: recall that variable tags are being used, and so the parallel trail usage should be twice that of the sequential one if the same number of bindings are trailed. The actual parallel usage is less than twice, showing that less bindings are being trailed. On the other hand, the storage corresponding to a goal can spread across many stack sections, and the last choice point may be far removed from the section in which the variable is bound. In the implementation measured we simply trail a binding if the variable was created outside the current section, in order to avoid more complex tests. Thus, the number of bindings trailed can also be more than in the sequential case, as is shown by the trail usage of boyer. It should be reasonably easy to design a more sophisticated trailing test so that less bindings would be trailed in such cases. Note that for atlas, the number of trailings appears to be more or less the same in the sequential and parallel case, as the trail is used more or less twice as much in the parallel case.

9 Dealing with Signals

In and-parallel execution, events that takes place on one task can affect the behaviour of other tasks. For example, under the "restricted" intelligent backtracking scheme of &-Prolog [13], when a goal in a CGE fails, all sibling and-goals are "killed". In DDAS [21, 20], there is even more interaction between and-goals because of the dependent and-parallelism.

Such communications between tasks can be implemented by allowing tasks to send signals to each other. For example, when a task is told to undo its computation (referred to as **roll back**), a 'kill' or 'redo' signal is sent to the task. A 'kill' signal informs the task that receive the signal that it is to be killed. A 'redo' signal, which is needed in DDAS, means that after undoing the computation, 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 in how memory can be recovered.

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 is not 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 is composed of many 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. Each slot in the P_Call Frame represents one and-goal in the CGE, contains a pointer to the start of their respective first sections, and the ID of the worker that executed the first section. When a signal is sent to a task, it is sent to the worker that executed the first section, along with the address of where

the section is. The worker is interrupted by the signal, and process the signal before returning to doing its original work.

Once a worker receives a signal for a task that it started, the signal must be propagated to the other stack sections of the task. This is done by following the pointers in the various markers to the other stack sections. For both 'kill' and 'redo', 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 (especially in DASWAM) apply to the same tasks, and the system filters 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 is undone before those representing the earlier work. The exception is that work done by sibling and-goals can be rolled back in parallel. Note that if a stack section is being remotely backtracked upon when a signal is received, the roll back does not occur until the remote backtracking has finished — this prevents multiple backward execution on the same segment.

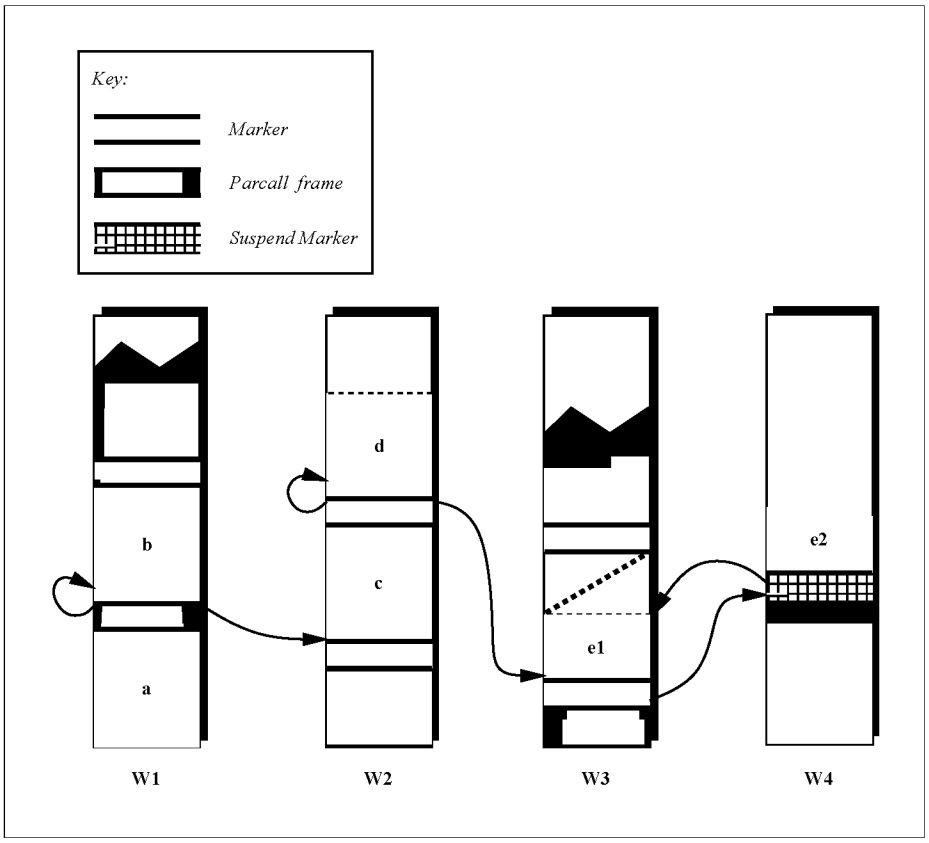

Figure 10: Example stack state before rollback

The actual mechanism used can best be illustrated by an example. Figure 10 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 segment "a", a CGE is encountered, and two descendant and-tasks, executing segment "b" (on W1) and "c" (on W2), are started. At some point, segment "b" is completed, and a new segment started on top of it. Segment "c"

encounters another CGE, spawning segments "d" (executed locally on W2) and "e" (executed on W3). Segment "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 segments: "e1" on W3, and "e2" on W4. e1 has been partially remotely backtracked, and segment "e2" is in the process of forward execution. At this point, the task associated with segment "a" receives a 'kill' signal.

The rollback has to undo the states of segments "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 dangerous 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 segment "e2" is still running, it might access its ancestral stack segments "e1", "c" and "a". Thus the kill signal is propagated to the youngest child segments before the killing starts. In this example, "b", "e2" and "d" are rolled back, when "e2" has been rolled back, "e1" is rolled back. Segment "c" is rolled back when both its descendant segments ("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 that unlike remote backtracking — where the backtracking worker can perform backtracking on another worker's stack, here there are opportunities for parallelism: *e.g.* segments "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.

9.1 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 10, 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.

This scheme is able to handle multiple roll backs, and is thus more flexible than the globally synchronised scheme of APEX [14].

10 Conclusions

We have studied aspects of memory management in the context of non-deterministic and-parallel systems. We have presented a memory management scheme that allows more flexible scheduling of tasks for than previous proposals, and shown that it offers several advantages over such proposals. We also discussed how cuts and roll backs can be handled in our scheme. Although we have concentrated on WAM-based environment stacking models, we believe most of our findings should also apply to other stack-based approaches such as goal stacking models. We have shown that the mechanisms presented are also useful for a number of other purposes, such as the efficient support of suspension in the context of non-deterministic and-parallel goals. We are actively researching many of these possibilities that the scheme has opened up for us, which include the support of dependent and-parallelism (for example as in the DDAS scheme), the efficient parallelisation of constraint logic programming systems (where suspension is often heavily used), more efficient support of side effects, *etc.* Although our results show that by some measures the demand on memory of our current implementations is still close to, and sometimes even better than, that of a sequential system, we can still sometimes use significantly larger raw amounts of total memory than such systems. We are actively researching ways in further reducing the total amount of memory used, basically through determinacy analysis and compilation of marker creation. In addition, a garbage collector that can deal with the special features of our scheme will also be very useful.

References

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

[2] K. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.

[3] U. C. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Volume 3*, pages 841–850, 1988.

[4] A. J. Beaumont and D. H. D. Warren. Scheduling Speculative Work on Or-parallel Prolog Systems. In *Logic Programming: Proceedings of the Tenth International Conference*, pages 135–149. The MIT Press, 1993.

[5] 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.

[6] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

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

[8] 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.

[9] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[10] 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.

[11] 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.

[12] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 253–268. The MIT Press, 1990.

[13] 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.

[14] 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. MIT Press, August 1988.

[15] 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. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988, Vol. 3*, pages 819–830. Institute for New Generation Computer Technology, 1988.

[16] E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[17] K. Muthukumar and M. V. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 80–97. The MIT Press, June 1989.

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

[19] P. V. Roy and A. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

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

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

[22] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 135–151. The MIT Press, 1991.

[23] R. Y. Sindaha. The Dharma scheduler – Definitive scheduling in Aurora on Multiprocessors Architecture. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 296–303. IEEE Computer Society Press, Dec. 1992.

[24] H. Tebra. *Optimistic And-Parallelism in Prolog.* PhD thesis, Vrije Universiteit te Amsterdam, 1989.

[25] E. Tick. Memory Performance of Lisp and Prolog Programs. In E. Shapiro, editor, *Third International Conference on Logic Programming*, pages 642–649. Springer-Verlag, 1986. Published as Lecture Notes in Computer Science 225.

[26] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming. The MIT Press, 1989.

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

[28] 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.

[29] 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.

[30] R. Yang, A. J. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. Technical report, Department of Computer Science, University of Bristol, 1993.