A Segment-Swapping Approach for Executing Trapped Computations *

Pablo Chico de Guzmán[1], Amadeo Casas[2],
Manuel Carro[1,3], and Manuel V. Hermenegildo[1,3]
pchico@clip.dia.fi.upm.es, amadeo.c@samsung.com,
{mcarro,herme}@fi.upm.es

[1] School of Computer Science, Univ. Politécnica de Madrid, Spain.
[2] Samsung Research, USA.
[3] IMDEA Software Institute, Spain.

Abstract. We consider the problem of supporting goal-level, independent and-parallelism (IAP) in the presence of non-determinism. IAP is exploited when two or more goals which will not interfere at run time are scheduled for simultaneous execution. Backtracking over non-deterministic parallel goals runs into the well-known trapped goal and garbage slot problems. The proposed solutions for these problems generally require complex low-level machinery which makes systems difficult to maintain and extend, and in some cases can even affect sequential execution performance. In this paper we propose a novel solution to the problem of trapped nondeterministic goals and garbage slots which is based on a single stack reordering operation and offers several advantages over previous proposals. While the implementation of this operation itself is not simple, in return it does not impose constraints on the scheduler. As a result, the scheduler and the rest of the run-time machinery can safely ignore the trapped goal and garbage slot problems and their implementation is greatly simplified. Also, standard sequential execution remains unaffected. In addition to describing the solution we report on an implementation and provide performance results. We also suggest other possible applications of the proposed approach beyond parallel execution.
Keywords: Parallelism, Logic Programming, Trapped Computations, Backtracking, Performance.

1 Introduction

Extracting parallelism from sequential programs has become a key point for the practical exploitation of multicore technology. However, writing parallel application has shown to be a difficult, time-consuming, and error-prone process for developers. Consequently, the design of new language constructs that aim at easing the task of writing parallel applications and the development of language tools to uncover the parallelism intrinsic in sequential applications have drawn the interest of the research community. Traditionally, declarative languages have received much attention for both expressing and exploiting parallelism due to their comparatively clean semantics and expressive power. In particular, a large amount of effort has been invested by the community in the area of parallel

* Work partially funded by EU project IST-215483 *S-Cube*, MICINN project TIN-2008-05624 *DOVES*, and CAM project S2009TIC-1465 *PROMETIDOS*. Pablo Chico is also funded by a MEC FPU scholarship.

execution of logic programs [1], where two main sources of parallelism have been identified and exploited. Or-parallelism, efficiently exploited by systems such as Aurora [2] and MUSE [3], aims at executing different branches of the execution in parallel. On the other hand, and-parallelism schedules the literals of a resolvent to be executed in parallel. As an alternative to execution models specifically designed for executing and-parallel programs, efficient models to exploit and-parallelism based on the WAM were developed. The latter have the advantage of retaining the many optimizations present in the WAM which improve performance in the sequential execution parts — and, consequently, improve the overall performance. &-Prolog [4] (the first fully described such system) and DDAS [5] are among the best-known proposals in that class. In addition, other systems such as (&)ACE [6], AKL [7], Andorra [8] and the Extended Andorra Model (EAM) [9, 10] have tackled the challenge of increasing performance of applications by providing solutions that combine both kinds of parallelism. In this paper we will focus on *goal-level, independent and-parallelism*, a subclass of and-parallelism in which parallelism is exploited among goals which do not compete for resources (bindings to variables, I/O, databases, and others) at run-time.

Although previous systems that have exploited independent and-parallelism excelled at speeding up the execution of programs in multiprocessor systems [1], the difficulty of the machinery required to execute nondeterministic programs in parallel hindered their widespread availability. In particular, one of the most delicate aspects that these systems need to address is the management of trapped goals and stack unwinding, which are necessary to free garbage slots left by the nondeterministic parallel execution, resulting in a complex interaction between goal age, scheduling, and memory management [11, 12]. Dealing with these issues required low-level, complex engineering, such as special stack frames in the stack sets [4, 13].

Notwithstanding, non-determinism is an essential concept that arises in many core areas of computer science, such as artificial intelligence and constraint-based optimization, and is necessary in general problem-solving patterns, such as generate-and-test. In order to avoid complexity, recent approaches to independent and-parallelism focus more on simplicity than on ultimate performance, abstracting core components of the implementation out to the source level. In [14], a high-level implementation of goal-level IAP was proposed that showed reasonable speedups despite the overhead added by the high level nature of the implementation. Other recent proposals [15], with a different focus from traditional approaches, concentrate on providing machinery to take advantage of underlying thread-based OS building blocks. Unfortunately, these implementations have not completely removed to date the need for low-level machinery in order to solve the trapped goal and garbage slot problems or are only appropriate for coarse-grain parallelism.

In line with this trend towards simplicity, we propose in this paper a novel solution for trapped goals and garbage slots that is based on reordering the stack to generate a stack state that could have been generated by a sequential SLD execution. Although the implementation of this solution is involved, in return it does not impose constraints on the scheduler for parallel execution which can remain unchanged. As a result, the scheduler and the rest of the run-time machinery can safely ignore the trapped goal and garbage slot problems and as a result their implementation and maintenance are greatly simplified. Finally, it is worth mentioning that our approach does not affect the performance of standard sequential execution.

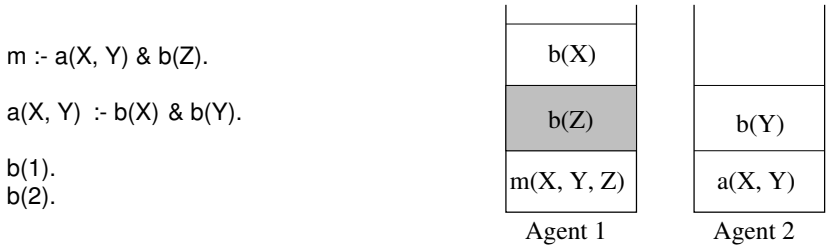


Fig. 1. Example of execution state in IAP with trapped goals.

In Section 2, we provide a brief introduction to the trapped goal problem and review some of the classical solutions that have been proposed to work around it. Section 3 focuses on the design and low-level details of our approach. Section 4 shows how this solution can be applied as well to solving the garbage slot problem. In Section 5, we present a performance evaluation of our approach, together with some data on the frequency of trapped goals in our implementation. Section 6 discusses how our technique can be applied to the implementation of execution strategies other than and-parallelism. Finally, Section 7 presents some conclusions.

For brevity, we assume the reader is familiar with the WAM [16, 17] and the RAP-WAM [4] architectures.

2 The Trapped Goal Problem

As mentioned before, one of the main challenges in IAP implementation is how to deal correctly with backtracking. The problem stems from the fact that in principle any of the available parallel goals can be selected for execution, and therefore they can be piled on the execution stacks in an order which differs from the one which would be generated by sequential execution. Since IAP implementations have been traditionally required to follow a right-to-left backtracking order, this clearly leads to a problem: it is possible that a goal to be backtracked over is *trapped* under a logically older goal which would hinder the application of the usual right-to-left backtracking order [11, 12]. We illustrate this with an example.

Figure 1 shows a possible state of the execution of a call to m using two agents.[4] When the first agent starts computing the first answer, goals a(X, Y) and b(Z) are scheduled to be executed in parallel. Let us assume that goal b(Z) is executed locally by the first agent and that goal a(X, Y) is stolen by the second agent for execution. Then, the second agent schedules goals b(X) and b(Y) to be executed in parallel, which results in goal b(Y) being locally executed by the second agent and goal b(X) taken by the first agent after finishing the computation of an answer for goal b(Z). In order to obtain another answer for predicate m, right-to-left backtracking requires computing additional answers for goals b(Z), b(Y), and b(X), in that order. However, goal b(Z) cannot be directly backtracked over since the execution of goal b(X) is stacked on top of it. Goal b(Z) has become a trapped goal.

Several solutions have been proposed to solve this problem. One of the original proposals makes use of *continuation markers* [4, 13] to *skip* over stacked goals. Even though

[4] Herein we use agent to refer to an executing thread attached to its own stack set.

this solution deals correctly with the trapped goal problem, it leads to a quite complex implementation, having to cope with a relatively large number of cases. In addition, it needs to store a good amount of additional information, which increases memory overhead. Another solution (also suggested in [4, 5, 18] and developed further and studied in [13]) is to allow public backtracking, i.e., to let an agent perform backtracking over a choicepoint that belongs to the stack set of a different agent. Unfortunately, this solution creates a difference between logical and physical views of the stacks, and adds the complexity of having to manage parallel accesses to the private stacks of each of the agents. More recently, a further solution to the problem was presented in [14], which is based on moving the execution of the trapped goals to the top of the stack before the agent starts to compute a new answer of the parallel goal. This solution simplifies the implementation, reducing the need for low-level machinery in comparison to previous approaches. However, garbage slots may still appear in the stacks. A common disadvantage of these approaches is that the parallel scheduler is forced to directly manage trapped goals. Also, they all share a relatively complex marker architecture. All of this keeps the complexity of these approaches still relatively high, affecting overall system maintenance, extensibility, and portability, as well as affecting standard sequential execution.

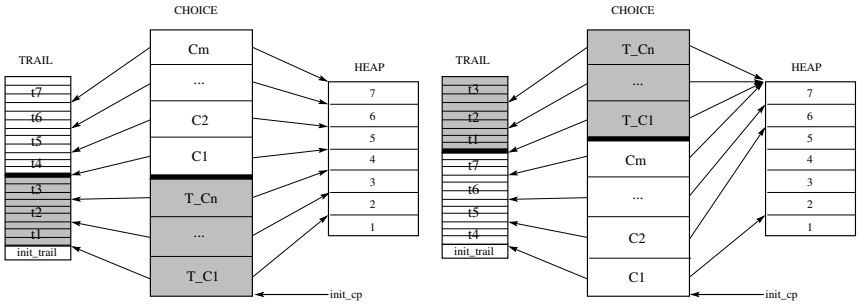
A completely different approach to solving the trapped goal and garbage slot problems is *restricted scheduling*: to keep track of goal execution order dependencies in order to restrict the set of goals that an agent is allowed to execute to only those that ensure that no goal under them will become trapped or garbage [11, 12]. An agent will not execute a goal `G` on a stack set if that stack set already contains a goal which could be backtracked over before goal `G`. While this solution shares with our approach the advantage of keeping stacks ordered, it complicates scheduling, adds overhead, and, above all, it comes at the cost of limiting the degree of parallelism in the system. In Section 5.2 we present a preliminary performance evaluation that shows that this effect can be quite significant in practice.

Finally, other systems with support for parallelism, such as Erlang [19], opted to create a new small stack set for each parallel goal. Note that Erlang, unlike Prolog, does not have support for backtracking. Therefore the problem we are tackling in this paper simply does not exist and the shape of the stacks is much simpler. The creation of multiple stacks (as needed) has also been suggested in the context of Prolog (as early as [11]), but the WAM multi-stack structure makes creating fresh stacks more expensive in time and memory.

Note that, while we have discussed so far approaches which keep the sequential solution order, trapped computations also appear in approaches to and-parallelism which give up on maintaining sequential execution solution order [20]. Therefore this paper is not as much a quest for efficiency as an attempt to find a simple solution (which minimizes changes to the scheduler while keeping the performance of the sequential execution) to a problem which seems unavoidable in and-parallel execution.

3 Reordering Stacks to Free Trapped Goals

In classical WAM implementations [16, 17], the order of the choicepoints corresponds to the chronological order in which backtracking has to be performed. This strong correspondence between the logical and the physical view of the choicepoint stack (and the corresponding heap and trail segments) is exploited to perform backtracking effi-



(a) Snapshot of a trapped goal execution. (b) Snapshot after choicepoints reordering.

Fig. 2. Example of choicepoint reordering before executing a trapped goal.

ciently, to reclaim all storage in the process in a very simple and fast way, and to pave the way to other optimizations. Unfortunately, trapped goals break this correspondence between logical and physical views and therefore some of the WAM assumptions do not hold anymore. As we saw before, this lack of correspondence appears in most previous approaches, in which the logical and physical views are separated. We propose to force this correspondence by explicitly reordering the stacks. The advantage is that this will maintain all the invariants of the sequential execution, which will in turn facilitate maintenance and make sequential optimizations easier to adopt.

3.1 An Example of Stack Reordering

Figure 2(a) shows the stack state of an agent which needs to compute a new answer of a goal that is currently trapped. `T_C`$_1$, ..., `T_C`$_n$ correspond to the choicepoints generated by the previous execution of such trapped goal. `C`$_1$, ..., `C`$_m$ correspond to the choicepoints that belong to computations younger than the one of the trapped goal. Pointers on the left of each choicepoint indicate the corresponding trail section associated to each choicepoint (`trail(choice_point)`), and show the limits of the logical effects that need to be undone when backtracking over each of the choicepoints. Pointers on the right of each choicepoint indicate the corresponding heap section of each of the choicepoints (`heap_top(choice_point)`), and show the limit of heap memory that can be reclaimed on backtracking.[5] In this case, it is possible to reinstate the correspondence between the logical and physical views by reordering the choicepoints in the stack. Figure 2(b) shows the stack set after reordering, which involves moving the choicepoints of the trapped goal `T_C`$_1$, ..., `T_C`$_n$ to the top of the stack, therefore creating a new backtracking execution order. Note that reordering the choicepoints needs a trail cell reordering in order to remove those logical effects generated by previous goal executions.

In addition, this choicepoint reordering operation requires updating the heap top pointers `heap_top(T_C`$_1$`)`, ..., `heap_top(T_C`$_n$`)` of each choicepoint to the current heap top of the agent's stack set, in order to protect the heap positions which belong to the trapping computations from backtracking over the trapped goal.[6] If these pointers are not reallocated, backtracking over the previously trapped goal (now on top)

[5] Similar pointers for the environment stack have been omitted from Figure 2(a).

[6] A similar idea was proposed in the context of tabling [21].

would set the H (heap) pointer to a location under the trapping goal heap area and forward execution would run over the heap area used by the trapping goal. For example, if heap_top(T_C$_n$) were still pointing to cell 4, backtracking over T_C$_n$ would set H = 4 and forward execution could overwrite heap cell 5, which belongs to another computation. By setting all heap pointers from heap_top(T_C$_1$) to heap_top(T_C$_n$) to point to the heap top, trapped cells remain protected and heap construction happens at the top of the heap. Given that younger heap cells point to older heap cells (i.e., top points to bottom in this figure), dangling pointers will not appear.

After reallocating pointers as shown in the previous paragraph, the heap section corresponding to the old trapped computation becomes unreachable. This is taken care of by updating the heap top pointer heap_top(C1) associated with choicepoint C1. It is made to point to the cell where the first choicepoint of the initially trapped computation was pointing (heap_top(T_C1)). This will reclaim the unused section on backtracking as backtracking over C$_1$ will set the heap pointer to the start of the heap area.

A similar operation needs to be performed for the environment stack to protect the environments of the trapping computation from backtracking. Note that it is not necessary to reorder the heap or the environment frame stack, and that the choicepoint stack reordering operation can be executed without requiring the agents to compete for mutual exclusion since this operation only affects locally the stack set of each agent.

3.2 Stack Reordering Algorithm

Figure 3 presents the algorithm that allows restarting the computation of a particular trapped goal. The procedure move_exec_top is supplied with a *handler* h as argument, which corresponds to a structure that is associated to the execution of each parallel goal, and stores the execution state of such computation. Let us use the example shown in Figure 2 to understand this procedure.

Fields initCP(h) and lastCP(h) of a particular handler h return the initial and the last choicepoint of the parallel computation associated with h. Lines 4 and 5 initialize local variables to point to the first choicepoint and the first trail cell of the trapped goal.

The first step in the algorithm (line 7) is to check whether the goal execution that needs to be restarted is currently trapped or not. If that is the case then the choicepoints of the trapped goal execution need to be moved to the top of the stack and the corresponding trail sections to the top of the trail (Section 3.1). In the case of the example shown in Figure 2, lines 8 to 12 of the algorithm copy the choicepoints T_C$_1$, ..., T_C$_n$ to an auxiliary memory location denoted by tg_cp and, similarly, the choicepoints C$_1$, ..., C$_m$ are copied over to yg_cp, the trail sections t1 to t3 are copied onto tg_trail, and finally trail sections t4 to t7 are copied over to yg_trail.[7]

We maintain a handler stack, HandlerStack, keeping the chronological order in which goals are executed. It is used by lines 14 to 18 to update the pointers initCP and lastCP of those handlers representing goals younger than the trapped one.[8] Lines 19 and 20 update the pointers initCP and lastCP of the trapped handler. Line 21 moves

[7] The amount of necessary auxiliary memory is usually negligible w.r.t. heap memory. Its size is the maximum between that of the trail/choicepoint stack section of the trapped goal and the trapping computations.

[8] Note that the complexity of this traversal is never worse than the choicepoint stack reordering.

```
1  void move_exec_top(Handler h)
2  begin
3
4      init_cp  = initCP(h);
5      init_trail     = trail(initCP(h));
6
7    if  (IS_YOUNGER_CP(CP(wam), lastCP(h))) then
8      MEM_ALLOC_COPY(tg_cp, init_cp, lastCP(h));
9      MEM_ALLOC_COPY(yg_cp, ONE_YOUNGER_CP(lastCP(h)), CP(wam));
10     MEM_ALLOC_COPY(tg_trail, init_trail,
11                        ONE_OLDER_TRAIL(trail(ONE_YOUNGER_CP(lastCP(h)))));
12     MEM_ALLOC_COPY(yg_trail, trail(ONE_YOUNGER_CP(lastCP(h))), trail(wam));
13
14     for all  handler OnTop(handler, h, HandlerStack) do
15     begin
16        initCP(handler) := initCP(handler)  − sizeof(tg_cp);
17        lastCP(handler) := lastCP(handler) − sizeof(tg_cp)
18     end for;
19     initCP(h) := initCP(h)  + sizeof(yg_cp);
20     lastCP(h) := lastCP(h) + sizeof(yg_cp);
21     MoveToTop(h, HandlerStack);
22
23     MEM_COPY(init_cp, yg_cp);
24     MEM_COPY(init_cp + sizeof(yg_cp), tg_cp);
25     MEM_COPY(init_trail, yg_trail);
26     MEM_COPY(init_trail + sizeof(yg_trail),    tg_trail);
27
28     for all  cp in  yg_cp do
29     begin
30        trail(cp)    := trail(cp)    − sizeof(tg_trail);
31     end for;
32
33     heap_top(CP(init_cp)) := heap_top(initCP(h));
34     frame_top(CP(init_cp)) := frame_top(initCP(h));
35
36     for all  cp in  tg_cp  do
37     begin
38        trail(cp)    := trail(cp)    + sizeof(yg_trail);
39        heap_top(cp) := heap_top(wam);
40        frame_top(cp) := frame_top(wam);
41     end for;
42   end if
43 end;
```

Fig. 3. Algorithm to perform choicepoints reordering in an agent's stack set.

the trapped handler to the top of the handler stack, corresponding with the new stack order.

The next step in the algorithm is to copy the choicepoints and trail sections back from the auxiliary memory locations `tg_cp`, `yg_cp`, `tg_trail` and `yg_trail` to the agent's stack and trail. This is performed in lines 23 to 26, which first move the choice-

points $C_1, \ldots, C_m$ back to the stack, followed by choicepoints $T\_C_1, \ldots, T\_C_n$, and then move trail sections t4 to t7 over to the trail, followed by trail sections t1 to t3.

Lines 28 to 31 iterate over the trail pointers of each of the initially trapping choicepoints trail(C_1), ..., trail(C_m) to ensure that they point to the updated location of their corresponding trail sections. Lines 33 and 34 update the heap and frame top pointers of the first trapping choicepoint in the stack C_1 to point to the original value of the heap and frame top pointers of the first choicepoint of the initially trapped computation T_C_1, to allow a proper release of the trapped goal memory when the execution backtracks over choicepoint C_1. After executing lines 23 to 26, init_cp is now pointing to C_1.

The algorithm then continues in lines 36 to 41 updating the trail pointers trail(T_C_1), ..., trail(T_C_n) for each of the choicepoints of the initially trapped computation to point to the new location of their corresponding trail sections, as well as their heap top and frame top pointers to point to the current heap top of the agent's stack set, and therefore protect the heap and the environment frame of choicepoints $C_1, \ldots, C_m$ from backtracking over the initially trapped goal. Now, the trapped goal is ready to be backtracked over using standard WAM machinery, after the auxiliary memory allocated in tg_cp, yg_cp, tg_trail and yg_trail is released.

Note that this algorithm assumes that choicepoints are not linked, but just stacked one over the previous one. In implementations where each choicepoint points to the previous one, reordering can boil down to pointer updating. Also, even for the case of memory reallocation, as will be shown in Section 5, backtracking over trapped goals does not occur very often, which reduces the impact of the overhead of the move_exec_top algorithm so that it does not significantly impact performance during execution of IAP.

3.3 Some Low Level Details

Our solution for the trapped goals problem requires considering two particular situations which have to be managed at a low level. The first one involves considering the *environment trimming* optimization, and the second one is related to "spurious" trail cells that may appear after the execution of the Prolog cut/0 operator.

Environment Trimming The *environment trimming* optimization reclaims, during forward execution, variables of the current environment when it is known that they are not going to be accessed again during forward execution. This is determined by comparing the ages of the current environment and the environment pointed to by the current choicepoint (using the environment_cp field of the choicepoint). In general, choicepoints protect local variables which have been created prior to the creation of the choicepoint. Under IAP execution, environment trimming may occur after a parallel call is performed. However, remote agents may generate choicepoints that protect some of these local variables. Unfortunately, environment trimming is not aware of the existence of these remote choicepoints, and unsafe environment trimming operations may then be performed. The simplest way to solve this problem is to insert a "void" choicepoint before any parallel call in order to protect all current local variables. A lighter solution would involve modifying the environment_cp field of the last choicepoint to protect all current local variables. The trail may be used to reinstall the original value of the environment_cp field before backwards execution is performed over this choicepoint.

Spurious trail cells Trail entries keep track of the cells with *conditional bindings*: those made to variables which appeared before a given choicepoint was pushed and bound after that choicepoint is pushed. These cells will not be reclaimed after backtracking over the last choicepoint of a goal execution, but their bindings need to be undone on backtracking. After executing a `cut/0` operator, some of these bindings become unconditional (because the choicepoint which was pushed between cell creation and cell binding is no longer active), and therefore they should not be part of the trail. Some Prolog systems remove these bindings as soon as a `cut/0` is executed. However, other systems (such as Ciao, on which our implementation is based) do not do so because these "spurious" trail cells do not affect the standard sequential execution: they always point to reclaimed memory. Unfortunately, spurious trail cells become a problem after the execution of the `move_exec_top` procedure, since the order of cells in the trail changes. In the case of environment frame variables, environment trimming or *last call* optimizations could make these spurious trail cells point to new, live environment stack sections. Using again the example in Figure 2, referencing a spurious trail cell belonging to choicepoint $\texttt{T\_C}_n$ could affect the environment frame of $\texttt{C}_1$. The problem is solved by invalidating those trail cells of choicepoints $\texttt{T\_C}_1, \ldots, \texttt{T\_C}_n$ which do not belong to the heap and the environment frame segments of these choicepoints (because they are not conditional).

4 Dealing with Garbage Slots

In addition to the trapped goal problem, unused sections of the stack may appear when executing nondeterministic parallel programs under IAP. Let us consider the goal `g :- a, (b & c), d.`. Let us assume that all goals in the body of `g/0` can return several solutions. There are several scenarios that may occur depending on which subgoal fails:

- If subgoal `a/0` fails, sequential backtracking is performed.
- Since subgoals `b/0` and `c/0` are mutually independent, if either one of them fails without a solution, backwards execution must proceed over subgoal `a/0`, because subgoals `b/0` and `c/0` do not affect each other's search space.
- If subgoal `d/0` fails, backwards execution must proceed over the right-most choicepoint of the parallel conjunction `b & c`, which could be trapped, and recompute the answers for all subgoals to the right of that choicepoint. Thus, backtracking within a conjunction of parallel subgoals occurs only if initiated by a failure from outside of the subgoals conjunction (also known as *outside backtracking*). Instead, if the backwards execution is initiated from within the subgoals in the parallel conjunction, backtracking proceeds outside of all these subgoals, i.e., to the left of the conjunction (also known as *inside backtracking*).

Inside backtracking requires canceling the execution of the parallel goals that belong to the same parallel goal conjunction. These goals could be trapped and they would then produce *garbage slots* in the trail, choicepoint stack, and heap. Traditionally, IAP implementations have solved this problem with methods closely related to those used to solve the trapped goal problem, with similar drawbacks, including complexity in the implementation of the parallel scheduler, as well as impacting its performance.

The solution for trapped goals that we have presented in Section 3.2 can be reused to avoid leaving garbage slots in the stack by executing the procedure `move_exec_top`

before canceling the execution of a trapped goal. By doing so, the corresponding trail cells and choicepoints of the canceled goal would be immediately reclaimed. The heap and environment frame stack would be reclaimed by garbage collection or upon backwards execution over the first choicepoint above the choice points of the canceled goal (choicepoint `C1` following the example in Figure 2). In our set of benchmarks, the trapped heap memory increases the memory use by 1% in the worst case.

5 Performance Evaluation

We present in this section some of the performance results obtained with the implementation of our proposed solution in the Ciao [22, 23] system. Such implementation is based on a previous high-level implementation of IAP [14], whose functionality has been augmented with the support to manage trapped goals and garbage slots.

All the benchmarks that are shown in this section were automatically parallelized with CiaoPP [24], using the annotation algorithms described in [25–27]. Finally, the actual performance results for each of the benchmarks were obtained after averaging ten different runs on a Sun UltraSparc T2000 (known as *Niagara* architecture) machine with eight 4-thread cores and 8Gb of memory running Solaris 10u1.

As we stated before, the aim of this paper is not so much to evaluate raw performance gains as it is to clarify up to which point the proposed technique is advantageous. In order to measure this, we will evaluate, on one hand, how often trapped goals appear in typical and-parallel computations and how much overhead the stack reorganization operations impose on the execution and, on the other hand, what speedups can be expected from executions which respect goal dependencies in order to avoid trapped computations.

We have used some deterministic and non-deterministic benchmarks selected from [28, 20], listed in Table 1. All these benchmarks can produce trapped goals (i.e., goals stacked out of order w.r.t. the sequential execution). Note that even if some of these benchmarks only return one solution, they are forced to fail in order to backtrack and explore all the search tree. For deterministic benchmarks, this means that backtracking is attempted on all parallel goals which are piled out-of-order in the stacks following the logical dependencies across the execution tree, even if they do not produce additional solutions.

5.1 Deterministic and Non-Deterministic Benchmarks

Table 2 presents the ratio of trapped goals vs. total parallel goals in the execution of each benchmark (column *Trapped*) and the percentage of the parallel execution time that is spent on the `move_exec_top` operation (column *Lost*). While this of course depends on the particular scheduling performed, it has been found to be quite stable in our current implementation. The evaluation is performed only up to 8 agents in order to make sure that every agent receives the full computing power of a core (threads in a core compete for shared resources, such as arithmetic units). The cases for one and two agents are also omitted since they do not generate trapped goals.

The first conclusion is that trapped goals do not appear very often in general, and their behavior depends largely on the nature of the benchmark itself. This scarcity favors our approach, whose cost grows with the number of trapped goals that need to be moved but otherwise does not pose overhead. This explains that the overhead imposed by the `move_exec_top` operation is very small in all of the benchmarks. These benchmarks create between 200 and 6000 choicepoints, and the precise number is related to the

Program	Description
fft	Fast Fourier transform.
fibo	22^{nd} Fibonacci number, executed sequentially from the 12^{th} downwards.
hanoi	Towers of Hanoi of size 14, executed sequentially from the 7^{th} downwards.
hanoi_dl	Towers of Hanoi with difference lists.
mmat	Multiplication of two 50×50 matrices.
pal	Recursively generates a palindrome of 2^{15} elements, switching to sequential execution when generating palindromes of length 2^7.
qsort	Use QuickSort to sort a list of 10000 elements, switching to sequential execution when the list to be sorted has 300 elements.
qsort_dl	QuickSort with difference lists.
iqsort	QuickSort with an irregular input list which makes the subgoals to be very different in size and favours the occurrence of trapped goals.
iqsort_dl	QuickSort with difference lists, sorting an irregular input list.
tak	Takeuchi function with arguments `tak(14, 10, 3).`
qsort_nd	Non-deterministic QuickSort (gives topological sortings) with an input list size 4000 elements, switching to sequential execution on 50 elements.

Table 1. Benchmark descriptions.

	3		4		5		6		7		8	
Program	Trapped	Lost	Trapped	Lost	Trapped	Lost	Trapped	Lost	Trapped	Lost	Trapped	Lost
fft	0.03	0.00	0.00	0.00	0.05	0.00	0.09	0.00	0.10	0.00	0.15	0.00
fibo	0.00	0.00	0.02	0.01	0.03	0.00	0.03	0.01	0.04	0.01	0.06	0.02
hanoi	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.00	0.04	0.00	0.04	0.00
hanoi_dl	0.00	0.02	0.00	0.03	0.03	0.05	0.04	0.05	0.03	0.05	0.04	0.07
mmat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.01	0.00	0.00
pal	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.01	0.00	0.03	0.00
qsort	0.02	0.00	0.02	0.00	0.07	0.00	0.11	0.00	0.06	0.00	0.09	0.00
qsort_dl	0.03	0.00	0.03	0.00	0.06	0.00	0.13	0.01	0.11	0.01	0.11	0.01
iqsort	0.03	0.00	0.09	0.01	0.18	0.02	0.26	0.02	0.27	0.02	0.35	0.03
iqsort_dl	0.03	0.00	0.08	0.01	0.15	0.02	0.20	0.02	0.28	0.03	0.36	0.03
tak	0.00	0.00	0.08	0.00	0.01	0.00	0.13	0.00	0.07	0.00	0.05	0.00
qsort_nd	0.02	0.00	0.07	0.00	0.14	0.00	0.21	0.00	0.33	0.01	0.39	0.01

Table 2. Trapped goal statistics.

amount of work that `move_exec_top` operation has to perform. The highest overhead (7%) is in `hanoi_dl`, which appears as an exceptional case. These results appear to support our thesis that it is debatable whether providing a very efficient but complex solution to the trapped goals problems is worth the effort. Instead, the proposed solution seems more practical since it greatly simplifies the parallel scheduler (with the added advantage, discussed later, that it can be reused for other purposes). This is even more so if we take into account that the frequency of trapped goals can be largely reduced by out-of-order backtracking with answer memoization [20], in which the traditional right-to-left order in backtracking is not maintained on parallel goal conjunctions. In this case the stack reorganization operation, although still necessary, is used even less frequently.

5.2 Avoiding Trapped Goals: the Impact of Goal Precedence

As mentioned in Section 2, a valid approach [12] to solving the trapped goal problem is to respect a notion of goal precedence during forward execution to completely avoid

	2		3		4		5		6		7		8	
Program	Trap	Prec	Trap	Prec	Trap	Prec	Trap	Prec	Trap	Prec	Trap	Prec	Trap	Prec
fft	1.75	1.74	2.06	1.75	2.69	2.68	2.68	2.69	2.87	2.68	2.97	2.67	3.02	2.68
fibo	1.91	1.72	2.62	2.51	3.18	2.50	3.98	4.10	4.51	3.98	5.48	5.14	5.98	5.13
hanoi	1.81	1.81	1.94	1.91	2.93	1.91	3.24	3.24	3.41	3.21	3.74	3.23	4.11	3.42
hanoi_dl	1.41	1.41	1.41	1.41	1.86	1.40	2.95	2.76	3.06	2.75	3.59	2.75	3.75	2.67
mmat	1.52	1.53	2.24	2.23	2.95	2.91	3.72	3.67	4.35	4.11	4.97	4.68	5.63	5.42
pal	1.81	1.82	2.27	1.83	2.59	1.82	3.18	1.82	3.29	3.17	3.60	3.18	3.96	3.03
qsort	1.79	1.78	2.25	1.78	2.51	2.27	2.69	2.29	2.84	2.29	3.42	2.29	3.73	2.29
qsort_dl	1.73	1.71	2.23	1.71	2.44	2.19	2.65	2.19	3.13	2.19	3.25	2.19	3.32	2.16
iqsort	1.33	1.33	1.33	1.33	1.67	1.33	2.27	1.33	2.43	1.33	2.80	1.33	3.02	1.33
iqsort_dl	1.29	1.29	1.30	1.29	1.64	1.29	2.13	1.29	2.68	1.29	2.88	1.29	3.19	1.29
tak	0.89	0.89	1.77	1.77	2.38	2.38	3.50	3.50	3.54	3.54	4.47	3.54	4.25	4.40
qsort_nd	1.53	1.53	1.59	1.58	1.92	1.59	1.93	1.59	2.01	1.59	2.34	1.59	2.54	1.66

Table 3. Speedup comparison: dependence analysis vs. trapped goals.

trapped goals. The low frequency of trapped goals previously found seems to suggest that this approach might be effective in practice.

In order to assess whether this is the case, we have developed a prototype implementation of IAP which schedules goals according to their precedence. Table 3 presents some of the speedups we obtained w.r.t. the Ciao sequential execution using this prototype (column *Prec*) and the speedups of our approach to handle trapped goals (column *Trap*), but adding the overhead of determining precedences: precedence dependencies are calculated but not used. The reason is that our dependency calculation algorithm may be suboptimal, and by applying it to both cases we obtain a conservative comparison.[9]

From the experimental results, the speedups obtained with a goal-precedence scheduler are in general reduced, with some benchmarks having a bigger difference (e.g., `iqsort` and `iqsort_dl`, probably due to an initial imbalanced split of the input list). In addition, the execution based on goal precedence of our prototype has been shown to be quite sensitive to the order in which the parallel goals are taken by remote agents, which makes the overall speed of the parallel execution less predictable. Finally, this solution is intended to match the behavior of standard sequential execution and is of no use in the case of strategies which use less strict execution strategies to increase the amount of search performed in parallel [20]. Therefore, we believe that avoiding trapped goals based on goal precedence has drawbacks which makes it not advantageous in practice.

6 Other Applications for Stack Reordering

So far, we have used `move_exec_top` to arrange the stack order so that it could have been generated by the standard sequential execution. However, other execution algorithms for logic programs can also benefit from this approach and take advantage of the `move_exec_top` operation. We show two examples: *swapping evaluation* [29] and *intelligent backtracking* [30].

Swapping Evaluation Swapping evaluation originates in the context of tabling [31]. Tabling records calls to goals to reuse their solutions and also to break infinite loops:

[9] Note that the observed overhead of the precedence analysis is rarely above 1%.

repeated calls (which generate loops) are suspended and other clauses for the looping predicate are tried in order to generate answers which allow the suspended computation branch to continue. The first call to a tabled predicate is named the *generator* and subsequent calls are named the *consumers*. Consumers read answers from a table where the generator inserted them. If the generator returns answers on demand, consumers can appear out of the scope of the generator execution. These consumers, named *external consumers*, suspend waiting for the generator to compute more answers, and fail when there are no more available answers.

External consumers change the standard SLD execution order. Assume `t/1` is tabled and has two solutions, `t(1)` and `t(2)`. In the query `?- t(X), t(Y)` goal `t(X)` is a generator and `t(Y)` is an external consumer. In an SLD execution, the answer sequence would be: {`X=1, Y=1`}, {`X=1, Y=2`}, {`X=2, Y=1`} and {`X=2, Y=2`}. Under tabled evaluation, `t(Y)` suspends and more answers of `t(X)`, the generator, are generated on backtracking. In this case, under tabled execution, the sequence of answers would be: {`X=1, Y =1`}, {`X=2, Y=1`}, {`X=2, Y=2`} and {`X=1, Y=2`}. With standard scheduling strategies (e.g., batched scheduling), the suspension of an external consumer can lead to massive memory consumption.

Swapping evaluation exchanges the role of the external consumer and its generator to avoid external consumer suspension. When `t(Y)` consumes the first answer, the execution tree of `t(X)`, which is trapped in the stack, is moved to the top of the stack so it can generate more answers. Swapping evaluation was originally implemented in XSB [32] and it is currently being ported to Ciao Prolog using the `move_exec_top` operation in order to untrap the execution tree of the generator.

Intelligent Backtracking Intelligent backtracking strategies are based on the idea of performing backtracking directly on the goal which generated the bindings that caused a failure. In the following example:

```
p(X,Y) :- a(X), b(Y), c(X).
a(1). a(2). b(1). b(2). c(2).
```

the execution of `c(X)` fails because `a(X)` unified `X` with `1`. Standard backtracking would retry `b(Y)` in a purposeless attempt to execute `c(X)` with a new binding for `Y`. Intelligent backtracking would change the backward execution order to allow backward execution over `a(X)` before backtracking over `b(X)`. Intelligent backtracking needs to keep track of the point where bindings were produced in order to safely detect the closest useful backtracking point. Intelligent backtracking could make use of the `move_exec_top` operation to change the backtracking order.

7 Conclusions

We have presented a new algorithm to solve the trapped goal problem in which the stack is reordered to generate an execution state that could have been generated by the sequential execution. Using this algorithm simplifies the implementation of the scheduler for parallelism and does not affect the performance in case of standard sequential execution. Our approach has been implemented in the Ciao system, and we have performed an experimental evaluation of its effectiveness. We have also compared our approach to that based on keeping track of goal dependencies in order not to generate trapped goals and found that the restriction in the degree of parallelism brought about by the

dependency-based approach makes this solution less advantageous. On the other hand, the use of the `move_exec_top` operation imposes only a limited overhead and does not restrict parallelism. Finally, the stack reordering operation presented in this paper represents semantically a change in the backtracking execution order, which we believe could be successfully applied to the implementation of tabling, swapping evaluation, or intelligent backtracking.

References

1. Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.: Parallel Execution of Prolog Programs: a Survey. ACM Transactions on Programming Languages and Systems **23**(4), pp. 472–602 (July 2001)
2. Lusk, E., Butler, R., Disz, T., Olson, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B., Haridi, S.: The Aurora Or-parallel Prolog System. New Generation Computing **7**(2/3), pp. 243–271 (1988)
3. Ali, K.A.M., Karlsson, R.: The Muse Or-Parallel Prolog Model and its Performance. In: 1990 North American Conference on Logic Programming, pp. 757–776. MIT Press (October 1990)
4. Hermenegildo, M., Greene, K.: The &-Prolog System: Exploiting Independent And-Parallelism. New Generation Computing **9**(3,4), pp. 233–257 (1991)
5. Shen, K.: Overview of DASWAM: Exploitation of Dependent And-parallelism. Journal of Logic Programming **29**(1–3), pp. 245–293 (November 1996)
6. Pontelli, E., Gupta, G., Hermenegildo, M.: &ACE: A High-Performance Parallel Prolog System. In: International Parallel Processing Symposium, IEEE Computer Society Technical Committee on Parallel Processing, pp. 564–572. IEEE Computer Society (April 1995)
7. Janson, S.: AKL. A Multiparadigm Programming Language. PhD thesis, Uppsala University. (1994)
8. Santos-Costa, V.M.: Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I. PhD thesis, University of Bristol. (August 1993)
9. Warren, D.: The Extended Andorra Model with Implicit Control. In Sverker Jansson, ed.: Parallel Logic Programming Workshop, Box 1263, S-163 13 Spanga, SWEDEN. SICS (June 1990)
10. Lopes, R., Costa, V.S., Silva, F.: A Novel Implementation of the Extended Andorra Model. In Ramakrishnan, I.V., ed.: Practical Aspects of Declarative Languages, Third International Symposium. Volume 1990 of Lecture Notes in Computer Science., pp. 199–213. Springer (March 2001)
11. Hermenegildo, M.: An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel. PhD thesis, U. of Texas at Austin. (August 1986)
12. Hermenegildo, M.: Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In: 4th. ICLP, pp. 556–575. MIT Press (1987)
13. Pontelli, E., Gupta, G.: Backtracking in independent and-parallel implementations of logic programming languages. IEEE Transactions on Parallel and Distributed Systems **12**(11), pp. 1169–1189 (November 2001)
14. Casas, A., Carro, M., Hermenegildo, M.: A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In García de la Banda, M., Pontelli, E., eds.: 24th International Conference on Logic Programming (ICLP'08). Volume 5366 of LNCS., pp. 651–666. Springer-Verlag (December 2008)
15. Moura, P., Crocker, P., Nunes, P.: High-level multi-threading programming in logtalk. In Warren, D., Hudak, P., eds.: 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08). Volume 4902 of LNCS., pp. 265–281. Springer-Verlag (January 2008)

16. Warren, D.: An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 (1983)
17. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
18. Shen, K., Hermenegildo, M.: Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In: Proceedings of EuroPar'96. Number 1124 in LNCS, pp. 635–640. Springer-Verlag (August 1996)
19. AB, E.: Erlang Efficiency Guide. 5.8.5 edn. (October 2011) From `http://www.erlang.org/doc/efficiency_guide/users_guide.html`.
20. P. Chico de Guzmán, Casas, A., Carro, M., Hermenegildo, M.: Parallel Backtracking with Answer Memoing for Independent And-Parallelism. Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue **11**(4–5), pp. 555–574 (July 2011) http://arxiv.org/abs/1107.4724.
21. Demoen, B., Sagonas, K.: CHAT: the copy-hybrid approach to tabling. Future Generation Computer Systems **16**, pp. 809–830 (2000)
22. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla-(Eds.), G.: The Ciao System. Ref. Manual (v1.13). Technical report, School of Computer Science, T.U. of Madrid (UPM) (2009) Available at `http://www.ciaohome.org`.
23. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming, (2012) http://arxiv.org/abs/1102.5497.
24. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), pp. 115–140 (2005)
25. Muthukumar, K., Bueno, F., de la Banda, M.G., Hermenegildo, M.: Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. Journal of Logic Programming **38**(2), pp. 165–218 (February 1999)
26. Cabeza, D.: An Extensible, Global Analysis Friendly Logic Programming System. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain. (August 2004)
27. Casas, A., Carro, M., Hermenegildo, M.: Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In: LOPSTR'07. Number 4915 in LNCS, pp. 138–153. Springer-Verlag (August 2007)
28. Casas, A.: Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages. PhD thesis, University of New Mexico (UNM), Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM 87131-0001 (USA). (September 2008)
29. P. Chico de Guzmán, Carro, M., Warren, D.S.: Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling. Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue **10 (4–6)**, pp. 401–416 (July 2010)
30. Pereira, L., Porto, A.: Intelligent backtracking and sidetracking in horn clause programs - the theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa (October 1979)
31. Warren, D.S.: Memoing for logic programs. Communications of the ACM **35**(3), pp. 93–111 (1992)
32. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3), pp. 586–634 (May 1998)