Efficient Backtracking in And-Parallel Implementations of Non-deterministic Languages

Enrico Pontelli & Gopal Gupta*
Laboratory for Logic, Databases, and Advanced Programming
Dept. of Computer Science
New Mexico State University
{epontell,gupta}@cs.nmsu.edu

Abstract

We consider the problem of efficiently supporting backtracking in *independent and-parallel* non-deterministic systems. We consider this problem in the context of logic programming, although the solution proposed is sufficiently general to be applicable to any non-deterministic language or system. Efficient implementation of backtracking is a difficult task, the primary reason being that backtracking is an inherently sequential activity. Due to this complexity, most existing and-parallel systems either do not support backtracking over parallel calls or they perform very poorly on non-deterministic benchmarks. We propose an implementation model for efficiently supporting backtracking. Our model employs various optimizations, as well as a novel memory organization scheme in which processors are allowed to traverse each others' stacks to achieve this efficiency.

The solution developed has been implemented in the ACE Parallel Prolog system. The performance of our implemented system is analyzed on a variety of non-deterministic benchmarks. The results achieved are remarkable: speedups achieved during forward execution are not lost in heavy backtracking activities and frequently super-linear speedups are obtained thanks to a semi-intelligent backtracking scheme.
Keywords: Backtracking, and-parallelism, logic programming, non-deterministic systems.

1 Introduction

Non-determinism arises in many areas of computer science. Artificial intelligence and constraint-based optimization are two such areas where non-determinism is commonly found. By non-determinism we mean the potential existence of multiple solutions to a problem. Search problems, generate-and-test problems, constrained optimization problems, etc., fall in this class. Non-determinism has also been incorporated in many programming languages: logic programming languages (e.g., Prolog), constraint programming languages (e.g., Chip [13]), concurrent constraint languages (e.g., AKL [24]), rule based languages (e.g., OPS5 [2]), and, recently, even in some traditional imperative languages [1].

Non-determinism present in a problem offers a rich source of parallelism. A problem is usually expressed as a *goal* to be achieved/proved/solved together with *rules* (or clauses) that specify how a given goal can be reduced to other subgoals. Given a (sub-)goal, there may be multiple ways of reducing it (non-determinism). On applying a rule, a (sub-)goal may reduce to a number of smaller (conjunctive) subgoals, each of which need to be solved in order to prove the original (sub-)goal. Two principal forms of parallelism can be exploited in problems that admit non-determinism:

*Ongoing work is supported by NSF Grants CCR 96-25358, HRD 96-28450, INT 95-15256, and and by NATO Grant CRG 921318

1 *Or-parallelism:* the different potential solutions can be searched in parallel—i.e., given a subgoal, there may be multiple rules that can be used to reduce it.

2 *And-parallelism*: while looking for a specific solution, the different operations involved can be executed in parallel (i.e., conjunctive subgoals may be reduced in parallel).

Or-parallelism is a direct result of non-determinism, while and-parallelism is the more "traditional" form of parallelism, commonly exploited in standard (deterministic) programming languages (e.g., Fortran [32]).

A system that exploits parallelism from problems with non-determinism may look simple to implement at first, but experience shows that indeed it is quite a difficult task. A naive parallel implementation may lead to very slow executions and may incur a severe (parallel) overhead compared to a corresponding state-of-the-art sequential system. Excessive parallel overhead may cause a naive parallel system to run many times slower on one processor compared to a similar sequential system. Reduction of parallel overhead results in improved *sequential efficiency* (performance of the parallel system on one processor) and *absolute efficiency* (overall execution time) of the system. It should be noted that the objective of parallel execution is to obtain better *absolute performance* and not just better speed-ups.[1] In the past, various implementations have achieved excellent parallel speed-ups but poor absolute performance—and often the good speed-ups were actually due to the parallelization of the additional overhead, more than the parallelization of the actual computation.

We can identify two major sources of inefficiency: scheduling and non-determinism. The parallelism present in these frameworks is typically very irregular and unpredictable; thus, parallel implementations of non-deterministic languages typically rely on dynamic scheduling. Most of the work for partitioning and managing parallel tasks is performed during run-time. These duties are absent from a sequential execution and represent parallel overhead. In the context of and-parallelism, one of the most delicate aspects is the management of non-determinism, expressed as backtracking over parallel calls. The task is very complex—indeed, as pointed out by Tick [29], this complexity is one of the reasons behind the limitations (committed-choice behaviour) imposed on the languages developed during the Japanese Fifth Generation project.

This paper overviews the issues involved in supporting non-determinism and backtracking in an *independent and-parallel* system and presents a simple and effective solution to the problem. The solution is based on the basic *point-backtracking* scheme, a generalization of the right-to-left Prolog backtracking proposed by Hermenegildo and Nasr [15], and builds on a novel memory organization scheme and on a collection of optimizations, which allow us to reduce the parallel overhead incurred during backtracking to a minimum. To our knowledge, this scheme is novel. Its simplicity guarantees the possibility of efficient implementation, and its generality allows its application to different and-parallel systems and frameworks.

In the rest of the paper, we take logic programming systems as representatives of non-deterministic systems. In particular, we illustrate the application of our model to an actual and-parallel implementation of Prolog, called ACE [11, 21, 23]. In the next section we discuss the forms of parallelism available in logic programming. In section 3 the basic execution model and the main implementation issues for independent and-parallelism are presented. Section 4 describes the complex problems arising from supporting backtracking in presence of and-parallelism, and describes our solution and its implementation in the ACE system. Section 6 analyzes the performance of the resulting system on a variety of benchmarks (deterministic as well as non-deterministic). Finally, section 7 presents conclusions and future work. The reader is assumed to have some familiarity with the concepts of logic and parallel logic programming.

[1] Where the speedup is defined as time taken by one processor to solve the problem divided by time taken by n processors for the same task.

2 Parallelism in Logic Programming Languages

Logic programming is a programming paradigm in which programs are expressed as logical implications. An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages, they disallow destructive assignments and explicit control information. Not only does this allow a cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for a runtime evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the declarative meaning of the program. In particular, these operations can be performed *in parallel.* An important characteristic of logic programming languages, thus, is that parallelism can be exploited in an *implicit* or a *semi-implicit* way. This can be done directly by the runtime evaluator, as suggested above, or, alternatively, it can be done by a parallelizing compiler. The task of the parallelizing compiler is essentially to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user (through suitable annotations or simple language extensions). In all cases, the advantage offered by logic programming is that the process is easier thanks to the more declarative and high level nature of the language, and requires minimal or no input from the programmer.

Two forms of parallelism are commonly identified in this context, *or-parallelism* and *and-parallelism.* Or-parallelism (ORP) arises when multiple rules define some relations and a call unifies with more than one rule head—the corresponding rule bodies can be executed in *or-parallel* fashion. Or-parallelism is thus a way of efficiently searching for solution(s) to a goal. And-parallelism (AP) is generated by the parallel execution of a conjunction of subgoals. Two major types of and-parallelism are typically identified, depending on whether and-parallel execution of subgoals accessing the same unbound variables is permitted or not: *(i) independent-and parallelism (IAP)*: arises when conjunctive subgoals that do not share any unbound variables are executed in parallel. During IAP no synchronization is required between parallel threads[2] (as in &-Prolog [14] and ACE [11]); *(ii) dependent-and parallelism (DAP)*: arises when conjunctive subgoals that share unbound variables, and compete in creating bindings for them, are executed in parallel.

2.1 Or-parallelism *vs* And-parallelism

Execution of programs containing non-determinism—i.e., the potential to produce multiple solutions via the use of alternative clauses—has typically been the exclusive domain of or-parallelism. This also stems from the fact that supporting non-determinism (through backtracking) in and-parallel systems is extremely complex. Backtracking is an inherently sequential process which requires, in order to produce a correct semantics, the traversal of the computation structure in a well-defined order. The distributed nature of an and-parallel computation makes this process challenging. Indeed, various classes of and-parallel logic languages, such as committed choice languages (e.g., Parlog, GHC, Concurrent Prolog) completely ruled out the issue of backtracking in presence of and-parallelism [25]. In and-parallel systems proposed to date, the heavy overheads incurred in the management of backtracking over and-parallel computations (that often completely waste the advantages achieved from parallelism) appear to be unavoidable—see [27, 12]. As a matter of fact, it is very rare to see an and-parallel logic programming system being evaluated using non-deterministic benchmarks. On the other hand, the current lack of combined and- and or-parallel systems (the first prototypes are just now appearing [11]) implies that execution of programs with non-determinism requires sacrificing and-parallelism, even if and-parallelism is present in large quantities. Furthermore, the adoption of IAP may lead to situations where backtracking can lead to super-linear speedups w.r.t. sequential execution. Given the and-parallel call

[2] With the exception of guaranteeing correct order of execution of side-effects and other order-sensitive instructions.

`?- a & b & c`, if `c` fails, then the whole call has no solutions and can be terminated; in contrast, for the same situation, a sequential execution will explore all alternatives of `a` and `b` before reaching the same conclusion. Nevertheless, producing an *efficient* and-parallel implementation capable of exploiting these features is an extremely complex task (see also [23]). In fact this form of semi-intelligent backtracking (called *inside backtracking* in [14, 15]) requires complex synchronization and message passing mechanisms [23].

3 Independent And-Parallelism

Considerable amount of work has been done to date in the context of independent and-parallel execution of Prolog programs. Practical models and systems which exploit this type of parallelism [14, 18, 27, 16] have been generally designed for shared memory platforms and are based on the *marker model* and on derivations/variations of the RAP-WAM/PWAM abstract machines—originally proposed in [15] and refined in [14, 27, 28]. This model has been shown to be practical, through its implementation in the &-Prolog and DASWAM systems, and proved capable of obtaining significant speedups with respect to state-of-the-art sequential systems (e.g., SICStus Prolog).

Our design of the and-parallel component of ACE is influenced by this model and its implementation in &-Prolog. Nevertheless, the memory organization of ACE differs from the marker model. The marker model builds very complex memory patterns, storing large quantities of additional information. We believe the organization adopted in ACE can provide better performance than the models directly based on the usage of the marker scheme, as comparisons between existing implementations demonstrate. ACE is one of the first systems to implement full-blown backtracking over parallel calls; it tackles all the necessary issues (e.g., *signal management*) and covers all possible cases that can arise.

The independent and-parallel component of ACE (also known as *&ACE*) represents the second generation of independent and-parallel systems. Compared to the first generation IAP systems, &ACE possesses the following novel features [21]:

- a simplified abstract machine instruction set;
- novel memory organization which removes the need of most of the additional data structures used in the previous execution models;
- detailed execution model for backtracking over and-parallel calls.

The ideas developed for &ACE have been realized in a concrete implementation of the engine and tested on a large pool of benchmarks (both artificial and real-life applications) with excellent results [21].

3.1 Execution Model

As in the most of the proposed schemes for independent and-parallelism [14, 18, 27], ACE exploits independent and-parallelism using a recomputation based scheme [11]—no sharing of solutions is performed at the and-parallel level. This means that for a query like `?- a,b`, where `a` and `b` are nondeterministic, `b` is completely recomputed for every solution of `a` (as in Prolog). This is in contrast with some of the previous works that were based on the notion of *goal reuse* [6, 16]—goal reuse, in fact, has been shown to create situations where Prolog semantics cannot be guaranteed.

For simplicity and efficiency, we adopt the solution proposed by DeGroot [9] of *restricting* parallelism to a nested *parbegin-parend* structure. This is illustrated in Figure 1, which sketches the structure of the computation tree created in presence of an and-parallel computation. The different branches are assigned to different *and-agents* (and-agents are processors working in and-parallel with each other).
Since and-agents just compute different parts of the same computation (i.e., they cooperate in building each

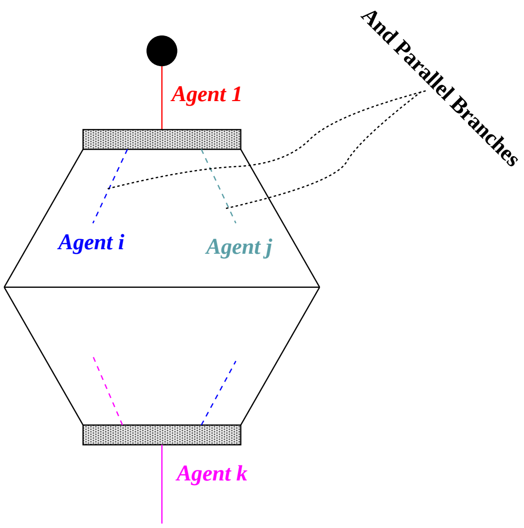

Figure 1: Computation Tree with Recomputation-based And-parallelism

solution of the initial query) they need to make the (partial) solutions they compute available to each other. This can be seen through an example: let us consider the following clause

$$\texttt{integr(X + Y,Z)} \leftarrow \texttt{integr(X,X}_1\texttt{), integr(Y,Y}_1\texttt{), Z = X}_1 \texttt{ + Y}_1$$

the execution of the two `integr` subgoals in the body can be carried out in and-parallel. But at the end of the parallel part the execution is sequential and requires access to terms created in the stacks of different and-agents. Since we are exploiting *independent and-parallelism*, only independent subgoals are allowed to be executed concurrently by different and-agents. In order to ensure this, in ACE we have adopted the proposal originally designed by DeGroot [9] and refined by various researchers [14, 22, 27] of annotating the program at compile time with *Conditional Graph Expressions (CGEs)*. A conditional graph expression (CGE for simplicity) is an expression of the form:

$$(\langle\, conditions \,\rangle \;\Rightarrow\; \texttt{B}_1 \,\&\cdots\&\, \texttt{B}_n)$$

where $\langle\, conditions \,\rangle$ is a conjunction of simple tests on variables appearing in the clause, which check for the independence of the goals, and "&" denotes *parallel conjunction* (the symbol "," is used to denote sequential conjunction). The intuitive meaning of a CGE is quite straightforward: if, at runtime, the tests present in *conditions* succeed, then the subgoals $\texttt{B}_1 \,\&\cdots\&\, \texttt{B}_n$ can be executed in and-parallel, otherwise they should be executed sequentially. The $\langle conditions\rangle$ can also be omitted, if they always evaluate to true, i.e., if it can be determined that the goals inside the CGE will be independent at runtime for any possible binding.

A standard Prolog program needs to be annotated with CGEs in order to exploit and-parallelism. In the ACE system a compiler has been developed to perform automatic generation of CGEs. The compiler is based on abstract interpretation techniques and was originally developed by Hermenegildo's group in Madrid [3]. This compiler has been modified by us to integrate advanced features, like granularity control and detection of dependent and-parallelism [22]. In any IAP system it is common to distinguish between two phases during the execution of a program:

1. *forward execution*: this phase deals with the selection of subgoals and execution of the resolution steps. The set of conjunctive subgoals that are executed in and-parallel is termed a *parallel call* (or *parcall*);

2. *backward execution*: this phase deals with the sequence of steps taken on failure or when a new solution needs to be found. It requires extension of backtracking to the case of parallel execution.

3.2 Forward Execution in &ACE

Each and-agent has a structure very similar to a standard *Warren Abstract Machine (WAM)* [31]—the standard model for sequential Prolog implementation.

During forward execution the various clauses are applied to reduce a given goal. In presence of IAP, special actions are required whenever a CGE is encountered. The execution of a parallel conjunction can be divided into two phases. The first phase, called the *inside phase*, starts with the opening of the CGE and the distribution of the subgoals between the different and-agents, and ends when the parallel conjunction is finished—i.e., each subgoal has reported a successful solution. Once the CGE is completed, then the *continuation*, i.e., the sequential code which follows the CGE, is begun and the *outside phase* is entered.

At the implementation level, in order to execute all goals in a parallel conjunction in parallel, a scheduling mechanism is used to assign parallel goals to available processors and some extra data structures are introduced to keep track of the current state of execution.

During execution of and-parallel Prolog programs, when a parallel conjunction is reached that is to be executed in parallel, a descriptor is created. This descriptor of the parallel call, called *parcall frame*, stores information regarding the parallel call as a whole (e.g., pointer to the environment of the call, counter of active subgoals, etc.) and regarding the single subgoals belonging to the parallel conjunction (e.g., status of each subgoal). The parcall frame refers to a descriptor of each subgoal (called *slot*), created in a separate heap space. The lifetime of the parcall frame corresponds to the lifetime of all the subgoals belonging to the parallel call; thus, it is created once the parallel call is started and it is removed once all the solutions of all the subgoals in the parallel call have been reported. Parcall frames are allocated on a dedicated stack. The creation of the parcall frame identifies the beginning of the parallel call. The next step is allowing the idle and-agents to pick up subgoals belonging to the parallel call for remote execution. A *Goal Stack* is associated to each agent for this purpose. During the creation of a parallel call, descriptors of the parallel goals (*goal frames*) are created and pushed on the goal stack of the current processor. Processors can pick up a goal for execution from the goal stacks of other processors as well as their own goal stack, once they become idle.

The execution of a subgoal in parallel call uses a *segment* in each of the stacks belonging to an and-agent. In most of the existing models for independent and-parallelism these segments are physically delimited using special data structures (*markers*). ACE avoids the use of physical markers by maintaining an implicit decomposition of the stacks in logical segments. The processor which completes the execution of the last active and-parallel subgoal belonging to a parallel call will proceed with the (sequential) execution of the parallel call continuation. Figure 2 shows the new data structures used in ACE to support IAP.

Memory management is further complicated by the need of keeping two separate views of the various stacks (especially the choice-point stack). Operations on the choice point stack, as well as other stacks, may access the stack following two paths:

- the *logical* path, in which the data structures relative to the current execution appear to be contiguous on the stack;
- the *physical* path, in which the data structures relative to the current execution are potentially spread over different parts of the stack, intermixed with (parts of) sections corresponding to the execution of other subgoals.

For example, allocation of a new data structure (e.g., a new choice point) needs to use the physical path (new structures can be allocated only on the real top of the stack), while backtracking should follow the logical path. Intermixing of different computations may occur because of backtracking on trapped subgoals—when the trapped subgoal is reactivated, the new data structures are allocated on the top of the stack—as shown in Figure 3: backtracking on the trapped subgoal Y lead to the allocation of a new choice point on the top of the

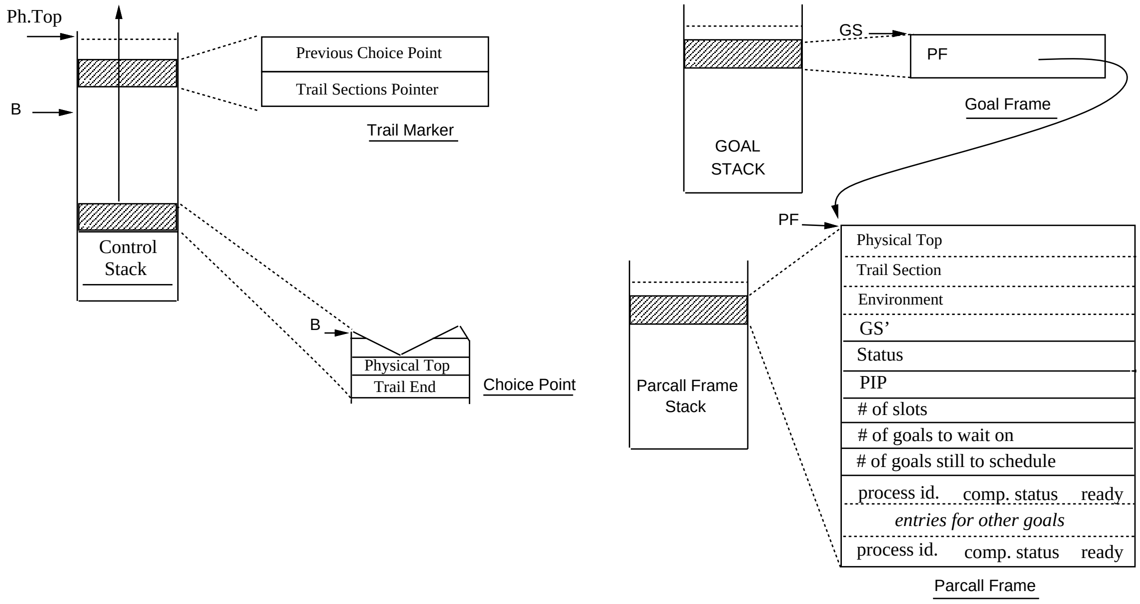

Figure 2: Data Structures and Related Registers in &ACE

stack; at this point physical and logical paths become different. Keeping track of the two paths is quite simple in &ACE, since each data structure allocated on the choice point stacks contains a pointer to the previous structure along the logical path as well as a a pointer to the previous structure along the physical path.[3]

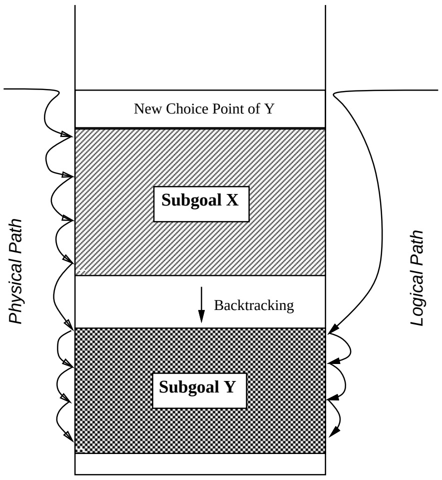

Figure 3: Backtracking on a Trapped Subgoal

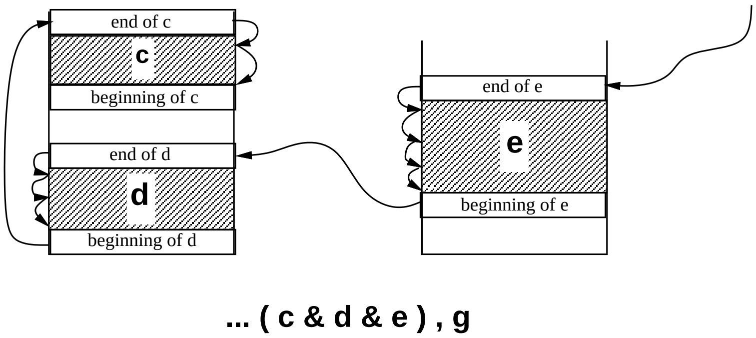

Figure 4: Backtracking over an And-parallel computation

4 Backtracking in IAP

Backward execution denotes the series of steps that are performed following the failure of a test (e.g., an

[3]Actually the physical pointer is unnecessary since size of the structures on this stack is known, but keeping it explicitly saves time during the various operations, and the increase in memory consumption is insignificant.

arithmetic test), a unification failure, or the lack of matching clauses for a given call. Since an and-parallel system explores only one or-branch at a time, backward execution involves backtracking and searching for new alternatives in previous choice points. In the presence of CGEs, sequential backtracking must be modified in order to deal with computations which are spread across processors. As long as backtracking occurs over the sequential part of the computation plain Prolog-like backtracking is used. However, the situation is more involved when backtracking involves a CGE, because more than one goal may be executing in parallel, one or more of which may encounter failure and backtrack at the same time. Thus, unlike in a sequential system, there is no *unique backtracking point*. Furthermore, the choice-point which offers a cure to a failure may lie within a segment belonging to an agent different from the one that encountered the failure, creating unsafe situations where different agents may be operating on the same stack segment. The communication among the processors involved in such backtracking process is performed using "signals", which can be viewed as specialized messages carrying the information needed to perform backtracking correctly. In any case, it must be ensured that the backtracking semantics is such that all solutions are reported, and in correct order. One such backtracking semantics has been originally described in [15] and represents the most natural generalization of standard Prolog backtracking (it has some elements in common with the scheme described by Conery [6]).

4.1 Basic Execution Model

Consider the goal:

```
a, b, (c & d & e), g, h
```

Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails: If subgoal `a` or `b` fails, sequential backtracking occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails, backtracking must proceed to `b`, because `c`, `d` and `e` do not affect each other's search space—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and re-compute all goals to the right of this choice point. If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right" (also known as *outside backtracking*—since it is initiated when the parallel call has already reached outside status). If initiated from within, backtracking proceeds outside of all these goals, i.e., "to the left" (also known as *inside backtracking*—since it is initiated when the parallel call is in inside status). The latter behavior is a form of "intelligent" backtracking.[4] When backtracking is initiated from outside, once a choice point is found in a subgoal g, an untried alternative is picked from it and then all the subgoals to the right of g in the parallel conjunction are restarted.

In the presence of the stack segments, the backtracking activity has to "move" in the correct direction. This means that the different segments need to be properly *linked*, in order to allow backtracking to flow from one segment to the *logically* preceding one. In the marker model, the markers represent the entry and exit point of each subgoal. In ACE, as discussed later, segments are directly connected without the need of any additional data structure. Thus, the logical structure that the backtracking algorithm has to see is the one depicted in Figure 4.

If failure occurs in the inside phase, inside backtracking is used—and the whole parcall should fail, since all the goals in the parallel conjunction are independent. To realize this, the failing processor should send a *kill signal* to all processors that have stolen a goal from that parcall, to undo any execution for the stolen goal. After all processors finish undoing the work, the goal before the CGE will be backtracked over as in standard sequential execution.

[4] However, such "intelligent backtracking" cannot be used in the presence of side effects, and additional synchronization mechanisms must be used to ensure a proper execution.

On the other hand, after a parallel conjunction completes, if a goal in the continuation of the CGE fails, then backtracking proceeds into the conjunction in *outside* mode. Outside backtracking is from right to left in the CGE similar to the backtracking in sequential execution. The only difference is that a goal to be backtracked over may have been executed by a remote processor, if another processor stole the goal. In most of the systems this situation is dealt with via special messages, called *redo signals*, which are used to request a backtracking activity to another agent. ACE allows each agent to freely move in the segments created by other agents, thus avoiding the need of redo signals. If a new solution is found during backtracking, the goals to the right of this goal in the parallel conjunction have to be re-executed. If outside backtracking fails to produce additional answers, the goal before the CGE will be backtracked over as in normal sequential execution.

4.2 Challenges

The above brief description of the system's activity may give the false impression that engineering an and-parallel implementation is relatively simple. However the design and implementation of proper mechanisms to support and-parallelism raise some very difficult issues.

Logical vs. Physical: The Warren Abstract Machine takes full advantage of the direct correspondence between logical structure of the execution and its physical layout in the abstract machine's data areas, as well of as the ordering in memory of these areas. This allows a considerably simpler execution of very complex operations—e.g., backtracking becomes very efficient, since a choice point on the stack is capable of completely identifying the execution state that exists at its creation time.

This is not anymore true when dealing with unrestricted scheduling of goals[5] [28]: the computation can be arbitrarily spread on the stacks of different agents, and the physical order of computations on each stack can be completely different from the logical one (e.g., a subgoal $\mathcal{A}$ which appears to the right of a subgoal $\mathcal{B}$ may appear on the stack in the opposite order [15]).

This lack of matching between logical and physical view of the execution creates considerable difficulties. Positioning on a choice point is not sufficient to get a view of a state of the execution (as in sequential computations). This correspondence has to be explicitly recreated, using additional data structures.

Backtracking Policy: the backtracking semantics described earlier specifies *what* (and *when*) has to be done, but does not state how backtracking on distributed computations can be implemented. Two approaches are feasible:

1. *Private Backtracking:* each agent is allowed to backtrack only over parts of the computation that are lying in its own stacks. This simplifies the memory management, but requires implementation of synchronization mechanisms to transfer the backtracking activity between agents.
2. *Public Backtracking:* each agent is allowed to backtrack on the stacks of other agents. This avoids the additional costs of communication between agents, but makes garbage collection and the overall memory organization more complex.

&-Prolog implements private backtracking. Experiments performed in the ACE project have shown that public backtracking behaves considerably better. Private backtracking requires exchanging large number of messages between agents, and most of these are served asynchronously, forcing agents to suspend waiting for other agents to react. Public backtracking considerably facilitates backtracking. Our experience shows that the locking of choice points during backtracking has very rarely created a bottleneck in real practical applications, and even in those cases public backtracking has produced better performance then private backtracking. For these reasons we have decided to adopt public backtracking in the ACE system.

[5]We use the term *unrestricted scheduling* to specify that no restrictions are imposed during scheduling on the selection of the next piece of work.

Synchronization: any reasonable implementation of inside backtracking—i.e., an implementation which interrupts the execution of a goal as soon as it is determined to be a useless computation—requires communication between the different processors operating on a given parallel call. This is typically realized in the form of kill messages sent by the failing processor to all other subgoals in the parallel call. As described in the next section, this imposes complex global synchronization requirements, and is considerably difficult to implement.

Trail Management: one of the main problems in managing backtracking in an and-parallel system is detecting the parts of the trail stack that need to be unwound (i.e., detecting bindings that have to be removed to restore the proper computation state). The current model used by ACE is based on a *segmented view* of the stack, where a segment is defined by a section of the stack between two consecutive choice-points. This is described in detail in the next few subsections.

Garbage Collection: the use of public backtracking allows recovery of a considerable amount of garbage "on-the-fly" during local backtracking. Nevertheless, garbage collection remains more complicated, due to the lack of correspondence between logical order of backtracking and physical distribution of the computation. Parts of computation which are not on the top of the stack or are accessed remotely may leave behind *holes* in the stacks that need to be properly tagged and eventually recovered.

4.3 Synchronization During Backtracking

One of the main issues in killing computations (e.g., when the whole parallel call fails) is the unwinding of the trail, i.e., removal of the bindings generated during the computation that is killed. In the presence of and-parallel computation, this operation needs to be carefully performed in order to avoid race conditions. If, for example, we are trying to kill the computation containing `a,(b & c)`, where `a`, `b` have been executed by P_i and `c` by P_j, then a synchronization point needs to be introduced at the level of the parallel call. Without such point, it may happen that P_i returns to its original computation (the one interrupted by the kill message) before P_j completes unwinding `b`. This may lead to P_j overwriting memory space which has been just allocated on the heap by P_i. This essentially means that, in the presence of a parcall, the killing of the part of the computation tree above the parcall may be started only *after* all the subgoals of the parcall have been completely killed. This seems to be the least restrictive requirement to impose (unless a previous global analysis deduces precise information about the bindings made in the computation before the parcall). Essentially the worker P that is in charge of continuing the killing above the CGE will wait for all the subgoals to be completely removed. During this waiting period P needs to keep checking for other kill messages received, since, otherwise, a deadlock may occur.

4.4 Existing Proposals

The issue of dealing with backtracking in and-parallel system has been tackled by various researchers [15, 5, 10, 14]. The execution model presented at the beginning for ACE originated from the scheme presented by Hermenegildo and Nasr [15], which in turn extends on the work by Conery [6] and DeGroot [9]. In particular, Hermenegildo and Nasr emphasized the importance of taking advantage of the independence of the subgoals to produce a semi-intelligent backtracking scheme: fundamentally important for compensating the cost incurred when backtracking over distributed computations.

Nevertheless, many of the models presented did not take sufficiently into account the interaction between the backtracking algorithm and the execution model adopted to support IAP. Pontelli et al. [23] and Shen [27] presented rather complete schemes describing integration of a semi-intelligent and-parallel backtracking policy into actual and-parallel implementations. As is clear from these presentations, the complexity of the resulting system is considerable, due to the need of complex synchronizations and interactions between processors during the execution. The cost of the resulting implementations have been proved to be high, either making backward execution over and-parallel calls inconvenient [23] or imposing additional overheads on the whole execution [27], forward phase included.

Proposals have been made to attempt improving the execution behaviour in presence of potentially non-deterministic and-parallel computations. Hermenegildo has presented various static analysis schemes whose results can be applied to improve management of and-parallel executions (e.g., [8]). Hermenegildo and Carro [23] described also an optimization scheme, called *backtracking families* which allows to take advantage of special features of the computation to improve backtracking. Related optimizations have been presented by Gupta and Pontelli [12]. In the context of dependent and-parallelism, Shen presented [26] an approach to improve backtracking by grouping subgoals in a parallel call with respect to the common variables. However, the scheme does not apply to IAP systems.

In spite of the number of proposals, it is evident that not enough effort has been put towards developing a *practical* solution that is capable of exploiting, as often as possible, features of the computation to reduce the overhead induced by backtracking over a parallel call. We believe the solutions described next represents a viable approach to this problem.

5 Backtracking in &ACE

5.1 Introduction

In typical (sequential) implementations of Prolog, backtracking is used to accomplish different objectives:

- *Alternatives:* backtracking moves back in the choice-point stack in search of a choice-point with unexplored alternatives. Once one of these is detected, one of the alternatives is taken, installed and forward execution reactivated;

- *Physical Memory:* the installation of a new alternative implies that the computation between the choice point with unexplored alternatives and the point where backtracking was started is not needed any longer. Thus the scan of the choice-point stack can be used to immediately reclaim the memory used by the discarded part of the computation;

- *Logical Memory:* discarding a part of the computation implies not only garbage collecting the memory used, but also making sure that all the *logical* effects of such computation are removed. In the case of logic programming, the effects of a computation are mainly represented by the collection of variable bindings generated during the computation. For this purpose, Prolog implementations maintain the *trail stack*, which is used to record each (conditional) binding performed. During the scan of the choice-point stack, the trail is concurrently visited and all the affected bindings are removed.

Of these three phases, clearly only two of them are *essential* for the computation, the selection of the alternative and the logical removal of the discarded computation. The first is fundamental for allowing the computation to restart, while the second is needed to avoid the new computation from being affected by the discarded one.

Most of the models presented in the literature do not make the above distinctions and try to accomplish all the three tasks concurrently. As a result the resulting models are very complex; recovering and reusing memory which is "owned" by different processors is challenging as it requires synchronization between the processors (that is, for example, what the marker-based model by Hermenegildo and Shen [28] proposes).

It is our opinion that in the context of an and-parallel implementation the three tasks above have to be kept separate and dealt with independently. In particular, the task of performing complete on-line garbage collection during backward execution appears to be in conflict with the requirement of being able to quickly switch to the new alternative. In the case of the ACE system, we decided to remove this requirement. Garbage collection is performed on-line only for the part of the computation which is directly controlled from the backtracking processor. This, combined with the lower memory requirements of the ACE model and the presence of an off-line garbage collector (performed by a separate, low priority thread), allows to reduce the burden during the backtracking phase.

5.1.1 Searching the Alternative

In and-parallel executions, the computation is spread across the stacks of different processors. Thus searching for a choice-point with unexplored alternatives require additional effort. In particular:

- the search should be able to move from one stack to another whenever needed;
- the search should be able to detect the *boundaries* of each subgoal, in order to decide when to switch do a different subgoal (which may lie in a different stack);
- the search should have knowledge of the ordering between the different fragments of the computation spread between the processors—i.e., given a subgoal, detect who is its predecessor.

In the marker model [14, 27] this is achieved by using additional data structures (the *markers*) which delimit each subgoal and connect the different subgoals in the correct order via pointers. Thus, during backtracking:

- backtracking proceeds in the local stack until a marker (*initial marker*) is encountered; the marker denotes the beginning of the current subgoal and contains a pointer to the marker which terminates the preceding subgoal (*end marker*);
- backtracking moves to the detected end-marker and proceeds;

The need of traversing all the different markers imposes considerable overhead. Furthermore, markers need to be properly created and linked during forward execution.

In ACE this is avoided, as explained in subsection 5.3.1, by immediately linearizing the parallel computation during forward execution and removing the need of allocating markers for each subgoal[6].

5.1.2 Managing the Trail

Managing the trail represents another challenging task. During sequential execution a single pointer (stored in the choice point) is sufficient to immediately detect the collection of bindings that need to cleared. In the case of and-parallel computations this is not the case. The trail itself is spread across different processors and, apparently, multiple accesses to different trails are needed to achieve the goal of clearing the bindings. For example, in the marker model the markers are used to denote sections of the local trail that are associated to the goal they represent. Thus, untrailing is achieved by traversing the various markers and removing bindings goal by goal. The possibility of having trapped subgoals implies that untrailing may leave behind many "holes" in the trail stacks, which will be recovered only once all computations on top have been deallocated.

In ACE, as described in the successive subsections, this problem has been solved in a different way. The trail is not anymore a stack but it's seen as a linked sequence of trail segments. A single pointer (and no traversals of the computation) is needed to detect the collection of bindings to remove. The process of untrailing is almost as efficient as in the sequential case. Furthermore, once untrailing is completed, the part of the trail used can be immediately reused by other computations. In addition to this, the technique illustrated below allows to considerably reduce the number of bindings that need to be untrailed in case of backtracking over a

[6] Only one marker per parallel call is created and only during backtracking.

parallel call (especially for the case of inside backtracking). Related to the issue of trail unwinding, we also need to remember the need to synchronize on a parallel call during the backtracking process as was discussed earlier.

5.2 Compile-time Support for Backtracking in IAP

In the ACE system we make use of a sophisticated compiler [3, 22] (originally developed by the CLIP group in Madrid and successively modified by us) that statically annotates the program with both IAP and DAP. The compiler makes use of abstract interpretation to statically infer various properties regarding program's execution. In particular the abstract interpreter used in ACE [3, 22] is capable of providing various information like variable sharing, freeness, independence, granularity and predicate modes. All these information have been combined to support the process of variable protection during program execution, discussed in the next section.

Given a parallel call (`b`$_1$ `& ... & b`$_n$) let us denote by *vars(*`b`$_i$*)* the variables present in the goal `b`$_i$. To improve backtracking we are interested in having as much knowledge as possible regarding the existence and the nature of bindings for each of these variables. The result of the static analysis is used to improve the "independence" of the parallel call from the rest of the computation. In particular, we would like this independence to allow a processor to backtrack to the computation preceding the parallel call without having to worry about the computation performed by the other processors operating on the parallel call. In the previous implementation schemes this is not possible, since parallel subgoals may generate bindings for variables created before the parallel call—thus backtracking cannot proceed outside of the parallel call until all those bindings have been properly removed. This means that independence of subgoals applies for forward execution but does not hold for parallel backtracking.

Let us call *external variables* w.r.t. a parallel call P all those variables which appear in the parallel call P but have been created before the parallel call itself. The intuition behind our approach is to try to improve the backward independence by allowing bindings to external variables to be posted only before or after, but not during, the parallel call. This way, a failure of one subgoal in the parallel call will not prevent the processor from backtracking outside of P, since no bindings to external variables have been posted by any of the other subgoals. Let us consider the following points:

`... ( ... & b`$_i$`[X] & ...) ...`

1. parallel subgoals are guaranteed to be independent, thus for each variable in $\bigcup_{i=1}^{n} vars(\mathtt{b}_i)$, there exist at most one goal `b`$_i$ which will be able to affect the binding status for such variable;

2. a variable `X` accessed by a goal `b`$_i$ which is free after the parallel call can be safely ignored by the management of the parallel call (e.g., no trail/untrail, etc.);

3. a variable `X` which has a ground binding before the parallel call is again not going to need any special treatment.

4. any variable which does fall in one of the previous cases is likely to receive a binding during the execution of the parallel subgoal.

The two distinct cases 2 and 3 can be approximated using abstract interpretation (sharing, freeness, and groundness detection). The ACE compiler is capable of detecting these two cases with good precision [3, 22].

There are two other cases that are not covered. The first one is characterized by the presence of an external variable which is free before the parallel call and bound after it. This case, again, is detectable with good precision via abstract interpretation, using in particular freeness information. In this case, the ACE compiler has been modified to generate a new class of WAM instructions (`init_protected`, `close_protected`) which are used to initialize a new variable—which is going to be used by the parallel subgoal instead of the original external one, and to transfer the binding from the new variable to the external one at the end of the parallel call. A sequence of `close_protected` instructions is generated by the compiler at the end of the parallel call.

The remaining case is the one in which an external variable has a partial binding before the parallel call (i.e., it is bound to a structure which may contain free variables inside) and the binding can be affected by the parallel call. The abstract interpreter approximates this case as a complement of the other cases mentioned before. The compiler attempts to deal with this situation in two ways:

- a *depth-k* analysis [17] is applied with respect to the external variables in order to attempt guessing the structure of the existing variable binding. All the variables that can be detected are treated as in the case previously discussed (generating pairs of `init_protected/close_protected` instructions);
- if depth-k analysis is insufficient to approximate the complete structure of the binding of an external variable, then a sequence of special instructions (`protect_unknown`) is generated to refer to the "unknown" parts of the binding. The behaviour of these instructions is illustrated in the next section.

5.3 Run-Time Support

5.3.1 Linearization

The linearization scheme has been adopted in the ACE system to allow a faster execution of outside backtracking, as well as to lower the burden on the forward phase of execution.

Linearization is aimed at partially recreating the sequential structure of the execution that was lost during the parallel computation. This goal is achieved by

- linearizing the sequence of choice points generated during the parallel computation (see Fig. 5);
- connecting the different sections of trail into a linear structure;

The advantage of linearization is that outside backtracking can be implemented essentially as sequential backtracking, without the need of traversing complex representations of the computation tree (as in the marker-based model). Choice points are connected to form a linear list which is traversed by backtracking in search for unexplored alternatives. Computations which do not generate alternatives are immediately exposed and do not require any additional step during backtracking. This allows to keep outside backtracking comparable in efficiency to sequential backtracking, and allows to trigger many determinacy-based optimizations [12].

The implementation of choice-point linearization is relatively simple. Choice points are linked in a single-linked list, where each choice-point directly indicates its predecessor. This modification to the organization of choice-points in the SICStus-WAM was performed independently from the use of linearization and does not introduce any overhead. This organization seems necessary to allow for IAP execution, considering the need discussed in the previous sections of maintaining separation between physical and logical organization of the computation. Each slot in the parallel call is used to keep track of the beginning and the end points of a parallel computation (as pointers to the choice-point stack). Whenever a subgoal is completed, its first choice-point is linked to the last choice-point of the subgoal on its left, and its last choice-point is made the predecessor of the first choice-point of the goal on the right. If either of these two links cannot be immediately established—because either the subgoal on the left has not been completed yet, or the subgoal on the right has not been started yet—then appropriate information is attached to the neighboring slots and the link will be completed by the neighboring subgoals.

The use of choice-point linearization allows to support IAP without the need of input and output markers. This implies considerable reduction in memory consumption—as discussed in details in [12]—as well as some

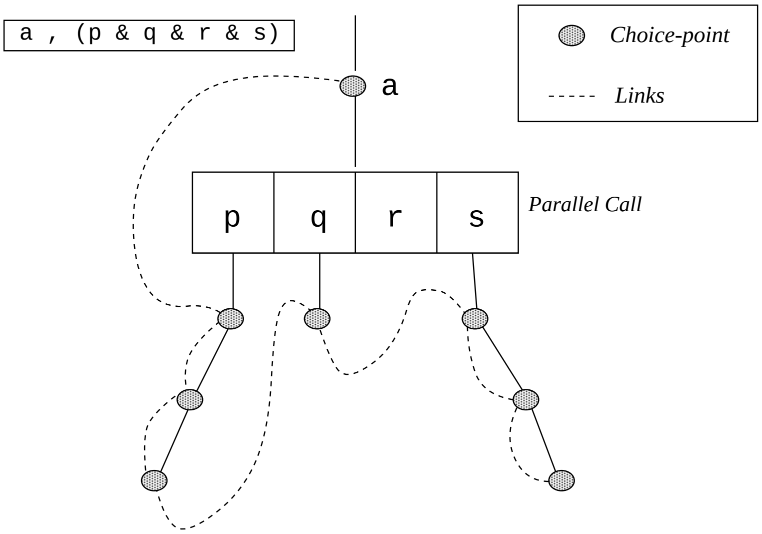

Figure 5: Choice-points linearization

improvements in execution time during forward execution, thanks to the avoidance of the marker creation operations. The cost for updating the links between the choice-points is negligible (two pointers per subgoal).

We must also point out that the linearization of the choice-point stack is relevant in the context of combined exploitation of and- and or-parallelism [11]. In fact, the leftmost part of the computation that includes all the subgoals that have already been "linked" together represents the part of the computation that is eligible for or-parallelism—i.e., it satisfies the *sharing conditions* [11] (also adopted in other proposals [7]).

The second aspect of linearization, as mentioned above, deals with the management of the trail stack. One of the main reasons behind the speed of sequential backtracking is the ability of detecting the collection of bindings to be removed just using a single pointer, as illustrated in figure 6.

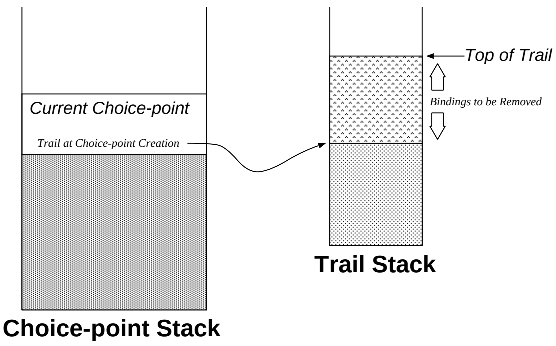

Figure 6: Untrailing in Sequential Execution

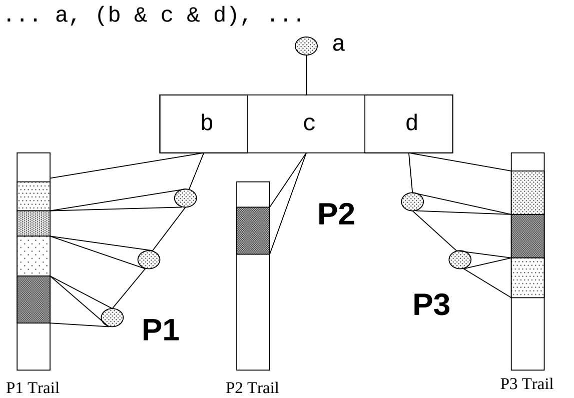

Figure 7: Trailing in IAP Computations

In the case of IAP the trail is split into multiple segments associated to the different components of the parallel computation, as show in figure 7. Each choice-point B identifies a segment of the trail which contains the bindings generated between the time of creation of B and the time at which the successive choice-point is allocated. Since we adopt public backtracking, i.e., any processor can backtrack in the space of any other processor, the choice-points may belong to different stacks and refer to trail sections present in different processors areas. The situation, as show in figure 7, is further complicated by the fact that computations can be deterministic (see subgoal `c` in the figure): this opens the problem of finding a place to keep track of the bindings generated by the deterministic subgoals (as no choice-point is immediately available for this purpose). The same problem applies for the trail segment generated from the beginning of a subgoal until the creation of the first choice-point (as for subgoals `b` and `d` in figure 7).

In ACE we have adopted the following solution:

- the trail is represented by a linked list of segments of variable size;
- a segment is associated to each subgoal at its creation time, and this solves the problem of both deterministic goals—all the bindings will be placed in the subgoal original segment—and the initial bindings (before creation of the first choice-point).
- as in the case of choice-points, segments are linked during the completion of a subgoal to create a single linear structure.

During backtracking, this allows the determination of the sequence of bindings to untrail just by keeping a pointer to the linear structure of trail segments.

In particular we have experimented with different implementations of this scheme. The first one was based on keeping a simple linked list. This introduces a small penalty due to the need of identifying the links between the different segments. The second implementation reconstructs the linear organization of the trail by adding one level of indirection in the trail access and by maintaining a segment table (in the same fashion as in the segmented memory management schemes adopted in operating systems). The third scheme is similar to the first, with the difference that there are no explicit links between the segments. The switch from one segment to the other is obtained by pushing the trail pointer itself at the beginning of each trail segment. During untrailing the trail pointer is modified as any other variable; the modification will place in the trail pointer the address of the next segment to explore. Not surprisingly this last implementation behaves far better than the other two, reducing the overhead of backtracking to a negligible amount.

5.3.2 Variable Protection

Goal of variable protection is to allow fast inside backtracking to take place. The intuition behind variable protection is to extend the notion of independence to the level of variable bindings. Thus we want to make computations independent of each others at the level of public bindings created. The goal of this is to allow inside backtracking to leave the computation without having to suspend on the parallel call and without having to worry about undoing the public bindings created by other processors. Variable protection allows to avoid making the external bindings public until all the subgoals have successfully completed. This is realized as follows:

- creating local copies of variables that are known to be free (and to be bound by a parallel subgoal). Thus if `X` is known to be free before a parallel call and we know that one of the subgoal may produce a binding for it:

  `...` ⟨`X` *free here*⟩ `( ... & p(.. X ..) & ...)` ⟨*variable* `X` *bound here*⟩ `...`

  then the variable protection will simply replace the variable `X` in the parallel with a brand new variable, introducing an explicit binding at the end of the parallel call:

  `... ( ... & p(... X1 ...) & ...), X = X1 ...`

  Now, since X1 is a variable seen only by `p`, in case of an inside failure no untrailing is immediately necessary. This means that backtracking can proceed outside of the parallel call without the need of synchronizing on it—i.e., without having to wait for all the processors to complete the untrailing of their local bindings. Furthermore, the binding `X=X1` is very fast, since it just requires copying one pointer into the variable `X`. This protection is implemented by the pair of WAM instructions `init_protected` and `close_protected`—the first one renames the variable while the second one posts the binding to the external variable. This pattern has been frequently observed in and-parallel benchmarks.

- variable protection becomes more complex if a variable accessed by an and-parallel subgoals is, before the parallel call, neither free nor ground. In this case the scheme adopted in the previous situation does not apply, since replacing `X` by `X1` will disallow `p` from accessing the partial binding for `X`.

  We have explored two alternatives. The first one consists of creating a new copy of the term in `X` with all the variables properly renamed to new ones:

  ```
  ...  copy_term(X,X1), (...  & p(..  X1 ..)  & ...), X=X1, ...
  ```

  and performing the explicit binding of `X` and `X1` at the end of the parallel computation. This approach may potentially introduce undesired overhead during forward execution, due to the time spent in creating a copy of the term and performing the final unification.

  The second alternative we have considered consists of adapting the variable representation mechanism adopted to support execution of DAP in ACE [20]. This scheme allows to create a special class of variables, characterized by the fact that an attribute can be attached to the dereference link outgoing from the variable; the attribute is automatically checked during dereferencing and a mode flag is set depending on the value of the attribute. This scheme was used to implement the concept of shared variables for ACE: the mode flag was set whenever a dependent variable was dereferenced from one of its consumer goals. The attribute was linked to an information stored in the subgoal descriptor, which was sufficient to determine whether the goal was currently producer or consumer of the shared variable.

  The same mechanism has been adapted to create a protected environment for the variables during execution of a parallel subgoal. The access to a annotated variable within a subgoal is unrestricted. The access from outside the subgoal can occur only in two cases:

  - the subgoal has been completed and we have entered the continuation of the parallel call;
  - the parallel call has failed and an alternative from a previous choice point is currently being explored.

  The two cases are easily identified by having the filter on the protected variables connected with the termination status of the subgoal. If the filter notifies that the subgoal is completed, i.e., with no parallel call failure, then the filter is immediately removed, thus making the bindings completely public. If the filter notifies that the subgoal was not completed, then the binding is immediately removed.

  This second approach can be implemented very efficiently. We have modified the compiler to produce code to generate the initial annotations to the non-free, non-ground variables which are passed to the parallel subgoals. The binding process requires only one additional operation, performed only when an annotated variable is encountered for the first time. The publication of the binding is applied only the first time an annotated variable is encountered outside of the parallel call.

  The implementation of this mechanism is realized through the use of the `protect_unknown` instruction.

The variable protection mechanism allow to remove the need of suspending backtracking on a parallel call in order to wait for all the processors to complete backtracking. Thanks to the independence between the computation which precedes the parallel call and the parallel call itself, backtracking can directly proceed. This allows to considerably speed-up the inside backtracking phase.

6 Performance Results

The implementation scheme presented in the previous section is aims to provide efficient support for both forward and backward execution in IAP systems. The concrete implementation of these ideas has been realized in the context of the ACE parallel Prolog system. Nevertheless we strongly believe in the generality of our approach and in the possibility of applying similar ideas to obtain fast parallel implementations of other non-deterministic systems (logic programming, constraint programming, rule-based programming, search-based systems, etc.).

The performance of the forward execution phase of ACE has been described in various works [21, 19, 23] and it has been shown to be highly competitive with the fastest existing IAP and DAP implementations of Prolog available. No other system, to our knowledge, has provided this level of speed-ups consistently on a very large pool of real-life applications and with such a low parallel overhead (on average 5% to 10% with respect to the original SICStus sequential engine used for the implementation). Figure 8 presents the speed-up curves obtained on some benchmarks by the ACE system.

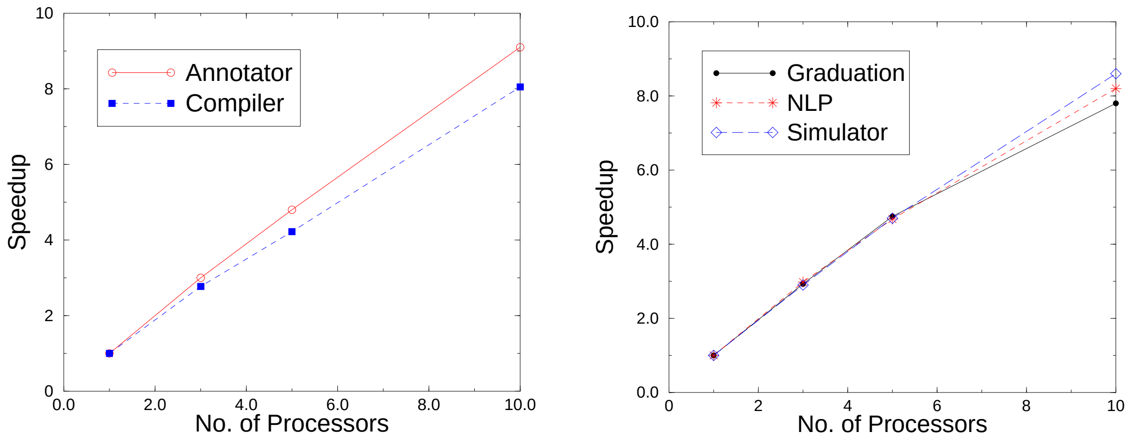

Figure 8: Speedups for Deterministic IAP Benchmarks

With respect to outside backtracking, the performance offered by ACE is particularly good. We have run various benchmarks which involve creation of parallel calls and outside backtracking over them. Figure 10 presents the comparisons in execution time with respect to the corresponding sequential execution for some of them. Some of the benchmarks involved finding additional alternatives during outside backtracking, with relative reactivation of forward execution, while others simply used backtracking to produce a global failure of the execution. In almost all the case analyzed we have observed a global overhead (forward+backward execution) less than 7%, which is a remarkably good result. The only few cases where outside backtracking gave rise to higher overhead are those where subgoals have a very fine grain and failures tend to emerge later in the execution—which is a rather rare situation in most applications, as discussed by Touati and Despain in their empirical study of the WAM [30]. In these rare cases, the situation can be easily lifted by using other optimization schemes, like the *Last Parallel Call Optimization* and the *Nested Parallel Call Optimization* [12]. Figure 9 shows the improvement in speedup obtained on some of these strange benchmarks.

We did not have access to any system with similar features, nor results on backward execution efficiency have been analyzed or presented for other recent IAP systems. Thus we were not able to compare these results with others.

The situation for inside backtracking is considerably more complex. In the original implementation of ACE

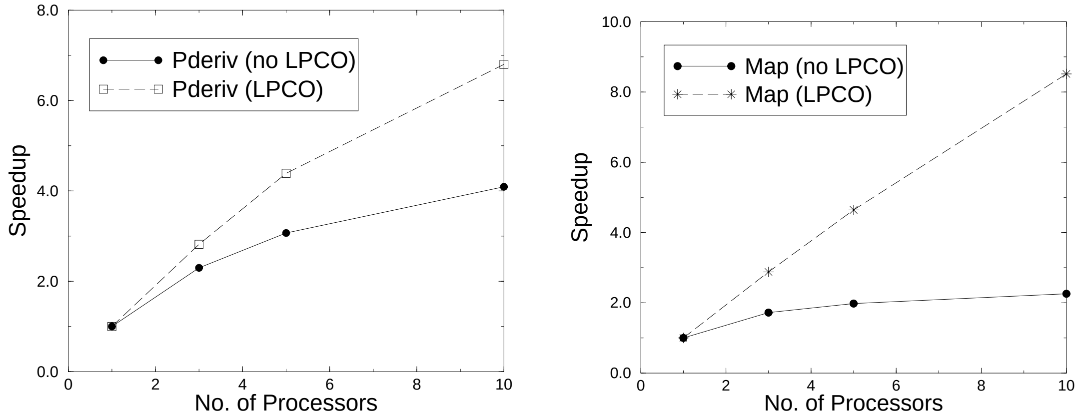

Figure 9: Improvements in Backward Speedup using LPCO

[21] we introduced a direct implementation of the inside backtracking scheme generalizing what was proposed by Hermenegildo and Nasr [15]. The implementation and some preliminary results are discussed in [23]. This system, was the first reported implementation of this scheme in the context of IAP/DAP (successively Shen [27] presented performance results for a parallel implementation of DAP; his system also includes analogous backtracking scheme, though he did not present any performance figure for backtracking). The scheme for inside backtracking introduces excessive overhead and may result in a considerable slow down with respect to sequential execution—mainly due to the need of synchronizing over parallel calls. This has also been confirmed by experiments conducted by the CLIP group in Madrid [4].

The current version of ACE replaced the older scheme with the mechanisms presented in the previous sections, compile-time detection of variable status, linearization and variable protections. Once again, the only system we had available for the comparison purposes was the older version of ACE—DASWAM (the parallel implementation of DAP by Shen) did not report results on this matter, and &-Prolog unofficially reported similar problems to those noticed in the older version of ACE. Furthermore, we believe that the implementation of the traditional scheme for inside backtracking realized in the ACE system was stretched to achieve the best possible performance and was very gracefully integrated with the other optimizations available in the system (LPCO, processor determinacy, shallow parallelism, etc. [12]).

Table 1 presents the performance results achieved on a representative pool of benchmarks which perform inside backtracking. The benchmarks are modifications of traditional IAP benchmarks; these modifications attempt to create examples representative of the various possible backtracking situations. All the modifications are "natural", i.e., they are not forced to create situations favorable to the implementation. All the failing situations are reasonable and, we believe, cover all the most relevant cases (deep failure, shallow failure, ground arguments, free variables, and partial bindings).

The *Hanoi* benchmark has been modified in order to produce a very deep failure, thus forcing inside backtracking to climb through a large number of nested parallel calls. The *Queens* is a representative of a goal which repeats the construction of the parallel call after inside backtracking. *Serialize* has been modified in two ways, one called $Serialize_{deep}$ creates a deep failure with additional choice points in between, while $Serialize_{flat}$ creates a failure in the higher levels of the nesting of parallel calls. In the case of *Mandelbrot*, the speeds drop suddenly when at least 5 processors are used: this is caused by the presence of exactly four parallel calls before the one which is actually failing. With 5 processors the failure, which is present almost at the beginning of the call, is immediately detected, allowing inside backtracking to quickly remove the whole

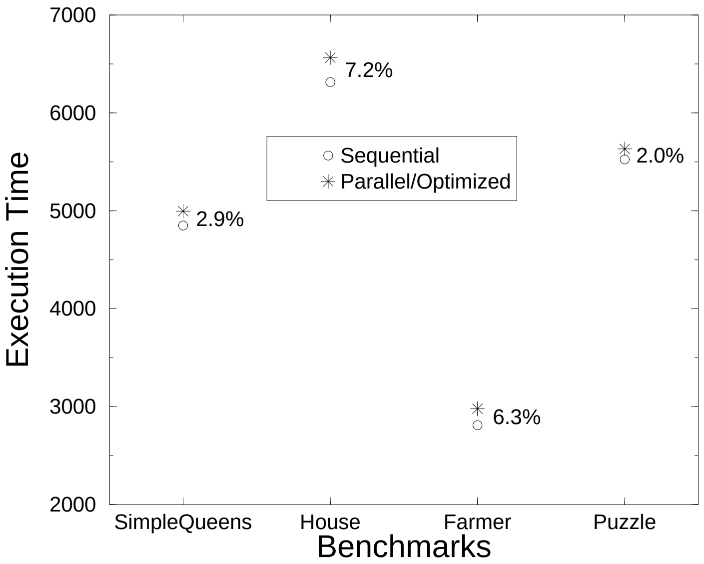

Figure 10: Outside Backtracking vs. Sequential Backtracking

nesting of parallel calls.

Benchmark	&ACE Processors 1		2		5		*Max. Improvement*
	Old	New	Old	New	Old	New	
Hanoi	5690	5500	3850	3700	2990	2300	30%
Compiler	2750	2730	38370	2730	11350	2730	1405%
	2750	2730	5114	2730	5114	2730	87.3%
BT_Cluster	4900	4900	3900	900	3750	900	433%
Annotator	3860	3860	1990	1960	1129	80	1411%
Queens	4012	3987	2954	1922	3914	905	430%
Mandelbrot	1800	1800	1000	890	600	100	600%
$Serialize_{deep}$	2705	2691	2703	1901	2812	1911	47.1%
$Serialize_{flat}$	1643	1643	2109	1207	2391	1211	97.4%

Table 1: Improvement with Fast Backtracking

The performance results are excellent. In all examples (also in further benchmarks not presented her due to lack of space) we have observed considerable improvement in execution speed. The removal of the need of synchronizing on a parallel call during inside backtracking allowed us to achieve super-linear speedups in various examples, e.g., *Compiler* and *Annotator* in table 1. This originates from the ability of detecting a failure faster than a sequential execution.

The comparisons between execution times of selected benchmarks are presented also in figure 11. The results presented shows the behaviour of the execution up to 5 processors. The speedups in most benchmarks consistently improve with higher number of processors—we have not shown that as it does not add much to our discussion on performance of backtracking. Detailed analysis of the ability to exploit parallelism of ACE has been presented elsewhere [21, 19, 23].

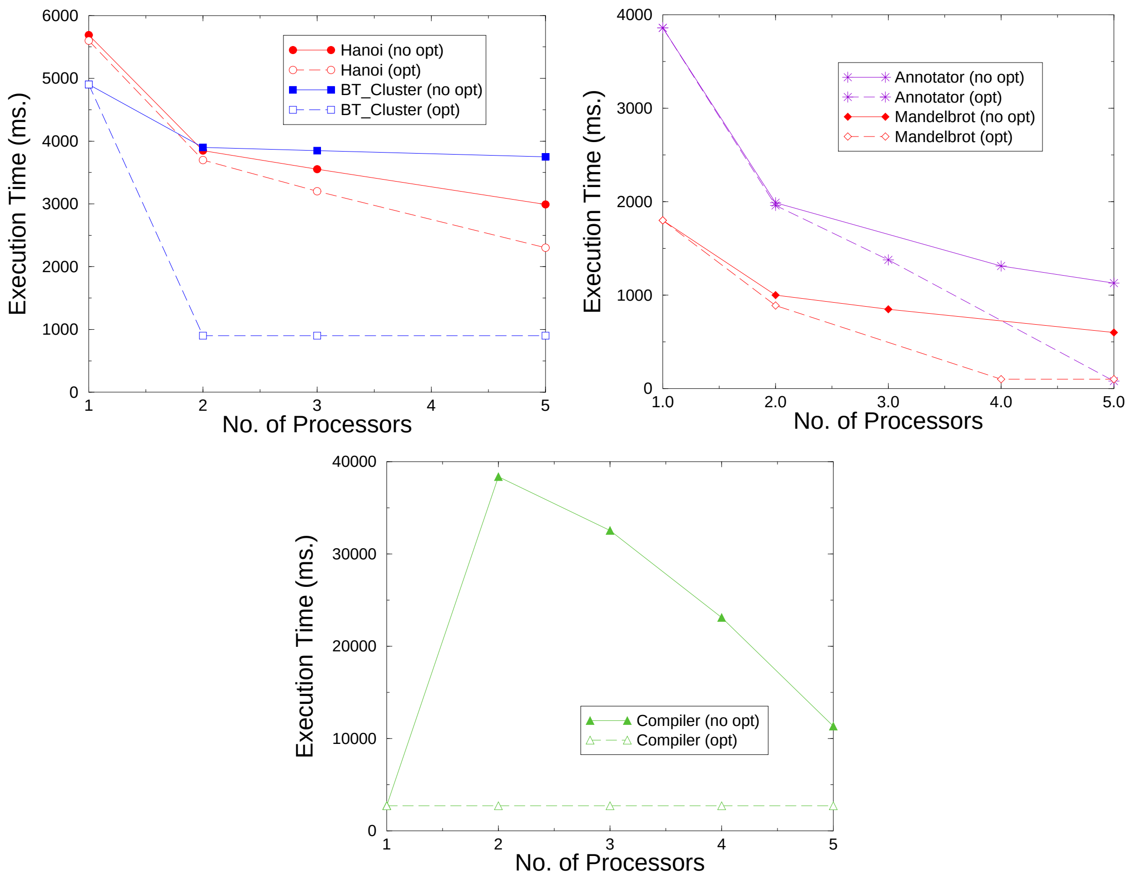

Figure 11: Comparison of Execution Times

7 Conclusions

In this paper we have analyzed the problem of efficiently performing backtracking in systems that exploit independent and-parallel. In particular, we have focused on the instance of this problem in the area of parallel logic programming—although our solution is general enough to be applied to other non-deterministic systems.

We have proposed an implementation model which allows to efficiently perform backtracking, respecting the correct semantics of the language and introducing a very limited amount of overhead. Furthermore, our scheme is capable of achieving super-linear speedups in situations where one parallel goal is unsuccessful—a possibility identified by other authors but difficult to realize in practice.

The execution model described has been implemented in the ACE parallel Prolog system, and we have described the performance of the system on various representative benchmarks involving backtracking over/during parallel calls. The performance results achieved are remarkable.

We believe the performance of the system can be further improved by making a more intense use of static analysis. In particular, we are currently exploring the possibility of using non-failure analysis and determinacy analysis [22, 8] to reorganize the parallel calls and drive the backtracking process in the best possible way (e.g., actually reusing side-effect free deterministic goals). We are also studying how to improve the performance of backward execution in the context of DAP, following the initial ideas by Shen [26].

Acknowledgements

The research presented has benefited from discussions with M. Carro, M. Hermenegildo, K. Shen, and V. Santos Costa, all of whom we would like to thank.

References

[1] Apt, K., and Schaerf, A. Search and Imperative Programming. In *Symposium on Principles of Programming Languages* (1997), ACM Press.

[2] Brownston, L. *Programming Expert Systems in OPS5: an introduction to rule-based programming.* Addison-Wesley, 1985.

[3] Bueno, F., de la Banda, M. G., and Hermenegildo, M. Effectiveness of Global Analysis in Strict Independence-based Automatic Program Parallelization. In *Int. Logic Programming Symposium* 1994, , MIT Press.

[4] Carro, M., and Hermenegildo, M. Issues on parallel backtracking in iap. (personal communication), 1994.

[5] Chang, J.-H., and Despain, A. M. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *Int. Symposium on Logic Programming* 1985, IEEE Computer Society.

[6] Conery, J. S. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[7] Correia, E., Silva, F., and Costa, V. S. The SBA: Exploiting Orthogonality in And-or Parallel System. In *Int. Symposium on Logic Programming* (1997), MIT Press.

[8] Debray, S., Lopez-Garcia, P., and Hermenegildo, M. Non-failure Analysis for Logic Programs. In *Int. Conference on Logic Programming* (1997), MIT Press.

[9] DeGroot, D. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems* (1984).

[10] Dembinski, P., and Maluszynski, J. AND-parallelism with Intelligent Backtracking for Annotated Logic Programs. In *Int. Symposium on Logic Programming* (1985), IEEE Computer Society.

[11] Gupta, G., Hermenegildo, M., Pontelli, E., and Costa, V. S. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94* (1994), MIT Press.

[12] Gupta, G., and Pontelli, E. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *Int. Parallel Processing Symposium* (1997), IEEE Computer Society.

[13] Hentenryck, P. V. *Constraint Handling in Prolog.* MIT Press, 1988.

[14] Hermenegildo, M., and Greene, K. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.* (1990), MIT Press.

[15] Hermenegildo, M. V., and Nasr, R. I. Efficient Management of Backtracking in AND-parallelism. In *Int. Conf. on Logic Programming* (1986), Springer-Verlag.

[16] Kale, L. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Int. Conf. on Logic Programming* (1987).

[17] King, A., and Soper, P. Depth-k Sharing and Freeness. In *Int. Conf. on Logic Programming* (1994), MIT Press.

[18] Lin, Y. J., and Kumar, V. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Int. Conf. and Symp. on Logic Programming* (1988), MIT Press.

[19] Pontelli, E., and Gupta, G. Parallel Symbolic Computation with ACE. *Annals of AI and Mathematics.* (to appear).

[20] Pontelli, E., and Gupta, G. Implementation mechanisms for dependent and-parallelism. In *Int. Conf. on Logic Programming* (1997), MIT Press.

[21] Pontelli, E., Gupta, G., and Hermenegildo, M. &ACE: A High-performance Parallel Prolog System. In *IPPS 95* (1995), IEEE Computer Society.

[22] Pontelli, E., Gupta, G., Pulvirenti, F., and Ferro, A. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In *International Conference on Logic Programming* (1997), MIT Press.

[23] Pontelli, E., Gupta, G., Tang, D., Carro, M., and Hermenegildo, M. Efficient Implementation of Independent And Parallelism. *Computer Languages 22*, 2/3 (1996).

[24] Saraswat, V. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, (1989).

[25] Shapiro, E., Ed. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

[26] Shen, K. Improving the Execution of the Dependent And-parallel Prolog DDAS. In *Proc. of PARLE 94* (1994).

[27] Shen, K. Initial Results from the Parallel Implementation of DASWAM. In *Int. Conf. and Symp. on Logic Programming* (1996), MIT Press.

[28] Shen, K., and Hermenegildo, M. Divided We Stand: Parallel Distributed Stack Memory Management. In *Implementations of Logic Programming Systems*, Kluwer Academic Press, 1994.

[29] Tick, E. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming* (1995).

[30] Touati, H., and Despain, A. An Empirical Study of the Warren Abstract Machine. In *Int. Symp. on Logic Programming* (1987),IEEE Computer Society.

[31] Warren, D. H. D. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[32] Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.