Improving the Efficiency of Nondeterministic Independent And–parallel Systems

Enrico Pontelli, Gopal Gupta
DongXing Tang

Laboratory for Logic and Databases
Dept. of Computer Science
New Mexico State University
Las Cruces, NM, USA
{epontell,gupta,dtang}@cs.nmsu.edu

Manuel Carro
Manuel Hermenegildo

Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte
Madrid, Spain
{mcarro,herme}@fi.upm.es

Abstract

We present the design and implementation of the and-parallel component of ACE. ACE is a computational model for the full Prolog language that simultaneously exploits both *or-parallelism* and *independent and-parallelism*. A high performance implementation of the ACE model has been realized and its performance reported in this paper. We discuss how some of the standard problems which appear when implementing and-parallel systems are solved in ACE. We then propose a number of optimizations aimed at reducing the overheads and the increased memory consumption which occur in such systems when using previously proposed solutions. Finally, we present results from an implementation of ACE which includes the optimizations proposed. The results show that ACE exploits and-parallelism with high efficiency and high speedups. Furthermore, they also show that the proposed optimizations, which are applicable to many other and-parallel systems, significantly decrease memory consumption and increase speedups and absolute performance both in forwards execution and during backtracking.
Keywords: Independent And-parallelism, Or-parallelism, Implementation Issues, Memory Management, Performance Evaluation, Logic Programming.

1 Introduction

1.1 Logic Programming and Prolog

Logic programming is a programming paradigm where programs are expressed as logical rules [35, 8]. Logic programming languages have been shown to be suited to a wide range of applications, from compilers to databases and to symbolic applications, as well as for general purpose programming (see, e.g., [49]). Arguably, the most popular logic programming language nowadays is Prolog. Unlike conventional programming languages, Logic Programming languages disallow destructive assignment and include little explicit control information. Not only this allows cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for an evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can often be executed in any order without affecting the (declarative) meaning of the program.[1] In particular, these operations can be performed by the evaluator in parallel. Furthermore, the cleaner semantics also make logic languages more amenable to automatic compile-time analysis and transformation.

An important characteristic of logic programming languages is that they greatly facilitate exploiting parallelism in an *implicit* way. This can be done directly by the program evaluator, as suggested above, or, alternatively, it can be done by a parallelizing compiler, whose task then is essentially unburdening the evaluator from making run-time decisions regarding when to run in parallel. Finally, of course, the program can be parallelized by the user. In all cases, the advantage offered by logic programming is that the process is easier because of the more declarative nature of the language and its high level, which contribute in preventing the parallelism in the application from being hidden in the coding process. Furthermore, the parallelization process can be done quite successfully in an automatic way, requiring little or no input from the user. Clearly, implicit exploitation of parallelism can in many cases have significant advantages over explicit parallelization.[2] In that sense, Prolog offers a possible path for solving the new form of "(parallel) software crisis" that is posed to arise with the new wider availability of multiprocessors[3]—given systems, such as the one described in this paper, one can run Prolog programs written for sequential machines in parallel with little or no effort. For the rest of the paper we assume that the reader is familiar with Prolog and its execution model.

It must be pointed out that while the preferred target areas of Prolog are Symbolic and AI applications, our system, as any other Prolog system (parallel or not), can also be used for the execution of general purpose programs [49], retaining the advantages in performance of parallel execution. This is borne out from some of the benchmarks we have used in Section 5 of this paper.

1.2 Parallelism in Logic Programming

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs (and, thus, Prolog) [9].

1. *Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other [38, 1]. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel.

2. *Independent and-parallelism* arises when more than one goal is present in the query or in the body of a clause, and it can be determined that these goals do not "affect" each other in the sequential execution–they can then be safely executed (independently) in parallel [16, 28, 36, 24, 27].

[1]Data dependencies or side effects however do pose constraints in the evaluation order.

[2]This does not mean, of course, that a knowledgeable user should be prevented from parallelizing programs manually or even programming sequentially but in a particular way that makes it possible for the system to uncover more parallelism.

[3]For example, affordable (shared memory) multiprocessor workstations are already being marketed by vendors such as Sun (Sun Sparc 10–2000), SGI (Challenge), etc.

3. *Dependent and-parallelism* arises when two or more non-independent goals (in the sense above) are executed in parallel. In this case the shared variables are used as a means of communication. Several proposals and systems adhere to this execution paradigm. Some of them try to retain the Prolog semantics and behavior, either relying on low–level machinery [46] or on a mixture of compile–time techniques and specialized machinery [10]. Other proposals depart from standard Prolog semantics, mainly restricting or disallowing backtracking and using matching instead of general unification [50, 32, 34, 12, 2]. In general, these decisions simplify the architecture of the system.

1.3 ACE: An And-Or Parallel System and Execution Model

The ACE (And-or/parallel Copying-based Execution) model [21, 41] uses stack-copying [1] and recomputation [19] to efficiently support combined or- and independent and-parallel execution. ACE represents an efficient combination of or- and independent and-parallelism in the sense that it strives to pay for the penalties for supporting either form of parallelism only when that form of parallelism is actually exploited. It achieves this by ensuring that, in the presence of only or-parallelism, execution in ACE be essentially the same as in the MUSE [1] system—a stack-copying based purely or-parallel system, while in the presence of only independent and-parallelism, execution be essentially the same as in the &-Prolog [24] system—a recomputation based purely and-parallel system. This efficiency in execution is accomplished by extending the stack-copying techniques of MUSE to deal with an organization of processors into *teams* [10].

It is important to observe that reaching this goal goes far beyond solving a simple engineering problem in combining two existing systems. The experience of ACE showed that the combination of two forms of parallelism leads to question most of the design choices and requires new solutions to previously solved problems (e.g. memory management schemes). This allowed us to get a better insight in the issues to be tackled in implementing general parallel logic programming systems. Some of these fundamental issues are briefly sketched in Section 2.5.

The ACE system is an efficient implementation of the ACE model supporting the full Prolog language, that has been developed at the Laboratory for Logic, Databases, and Advanced Programming of the New Mexico State University, in collaboration with the CLIP group at the Technical University of Madrid, Spain. In this paper we will present briefly how some of the standard problems which appear when implementing and-parallel systems are solved in ACE. We then propose a number of optimizations aimed at reducing the overheads and the increased memory consumption. Finally, we present results from an implementation of the system which includes the optimizations proposed. The results show that ACE exploits and-parallelism with very high efficiency and excellent speedups. These results are comparable and often superior than those presented for other pure and-parallel systems. Furthermore, they also show that the proposed optimizations, which are applicable to many other and-parallel systems, significantly decrease memory consumption and increase speedups and absolute performance both in forwards execution and during backtracking. The ACE implementation belongs to the second generation of and-parallel systems, since it combines the techniques used in older, first generation systems (e.g. the first versions of &-Prolog [24]) with new innovative optimizations to obtain a highly efficient system.

As mentioned before, in this paper we are exclusively concerned with the analysis of the and-parallel component of the ACE system; for further details on the whole ACE system the interested reader is referred to [21].

2 Independent And-parallelism

As pointed out above, the main purpose of this paper is to illustrate the structure, features, and optimizations of the and-parallel engine developed for the ACE system, and evaluate its performance. In this section we explain the computational behavior of the and-parallel engine in more detail.

Much 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 [28, 36, 24, 46] are generally designed for shared memory platforms and based on the "marker model", and on derivations of the RAP-WAM/PWAM abstract machines, originally proposed in [28, 30] and refined in [24, 47, 48]. This model has been shown to be practical through its implementation in the &-Prolog system, which proved capable of obtaining quite good speedups with respect to state of the art sequential systems. Our design of the and-parallel component of ACE is heavily influenced by this model and its implementation in &-Prolog. However, in addition to supporting or-parallelism, ACE also incorporates a significant number of optimizations which considerably reduce the parallel overhead and result in better overall efficiency. These optimizations are fairly general, and are applicable to any and-parallel system whose implementation is based on the markers model. The election of having a shared memory space, in contrast to many other proposals which use distributed memory models, is now supported by the availability of commercial multiprocessors in the market and their relative ease of programming. In addition, shared memory machines offer a model for the implementor which is simpler and more portable than that offered by distributed memory architectures.

2.1 Introduction

As in the RAP-WAM, ACE exploits independent and-parallelism using a recomputation based scheme [19]—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).

For simplicity and efficiency, we adopt the solution proposed by DeGroot [16] 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 the presence of and-parallel computation with the previously mentioned *parbegin-parend* structure, where the different branches are assigned to different *and-agents* (and-agents are processing agents working in and-parallel with each other). Since and-agents are computing just different parts of the same computation (i.e. they are cooperating in building one solution of the initial query) they need to make available to each other their (partial) solutions. Doing this in a distributed memory machine would need a traffic of data which would impact negatively on performance. This is avoided in a shared memory machine by having *different* but *mutually accessible* logical address spaces. This can be seen in through an example: let us consider the following clause (taken from a program for performing symbolic integration):

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

The execution of the two subgoals in the body can be carried out in and-parallel. But at the end of the parallel part, the execution is sequential and it requires access to terms created in the stacks of different and-agents (see figure 2).

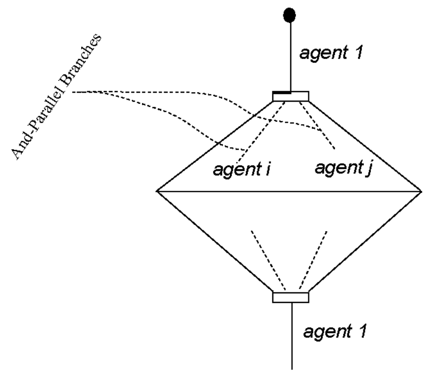


Figure 1: Computation Tree with Recomputation-based And-parallelism

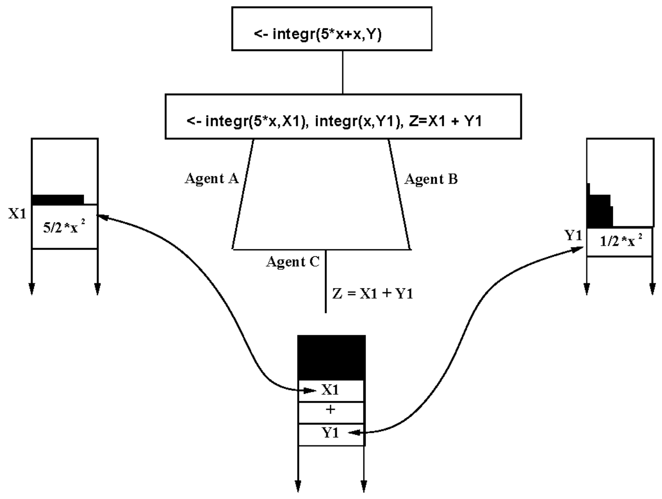


Figure 2: Need for Sharing of Data Structures

2.2 Conditional Graph Expressions

Since we are exploiting only *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 [16] and refined by one of us [28] (adapted also in the &-Prolog system [24]) 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:

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

where $\langle\, \mathit{conditions}\,\rangle$ 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 intuitive meaning of a CGE is quite straightforward: if, at runtime, the tests present in *conditions* succeed, then the subgoals $\mathtt{B}_1 \,\&\cdots\&\, \mathtt{B}_n$ can be executed in and-parallel, otherwise they should be executed sequentially. The notion of CGE can be further extended in different ways [24, 28]:

1. $\mathtt{B}_i$ can actually represent an arbitrary sequential conjunction of subgoals or a nested CGE.

2. we can explicitly add an *else* part to the CGE conditional, specifying eventually actions different from the plain sequential execution of the subgoals.

3. the ⟨*conditions*⟩ can be extended allowing more complex tests, whose purpose goes beyond independence analysis, e.g. they can be used to implement some form of granularity control [15, 26, 37] or more advanced notions of independence [13, 27, 5].

The ⟨*conditions*⟩ 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 take advantage of the and-parallel engines available. This process can be done manually by the programmer but is generally done automatically by specialized compile-time analysis tools (like the &-Prolog parallelizing compiler [3], which is also an integral part of ACE).

2.3 Forward Execution

Forward execution of a program annotated with CGEs is quite straightforward [28, 25]. Whenever a CGE is encountered, the conditions are evaluated and, if these conditions hold at runtime, the various subgoals in the CGE are made available for and-parallel execution (otherwise the subgoals in the CGE are run sequentially). Idle and-agents are allowed to pick up available subgoals and execute them. Only when the execution of those subgoals is terminated the continuation of the CGE (i.e., whatever comes after the CGE) is taken into consideration.

The execution of a parallel conjunction can thus be divided into two phases. The first phase, called the *inside phase*, starts with the beginning of the execution of the CGE and ends when the parallel conjunction is finished. Once the execution of the continuation is begun for the first time, the *outside phase* is entered.

At the implementation level, in order to execute all goals in a parallel conjunction in parallel, a scheduling mechanism used to assign parallel goals to available processors and some extra data structures are introduced to keep track of the current state of execution. The two main additional data structures are the *goal stack* and the *parcall frame.* Details of the structure of a Parcall frame are shown in Figure 3. In addition to parcall frames and goal stacks, an *input marker node* and an *end marker node* are used to mark the beginning and the end respectively of the segment in the stack corresponding to an and-parallel goal.

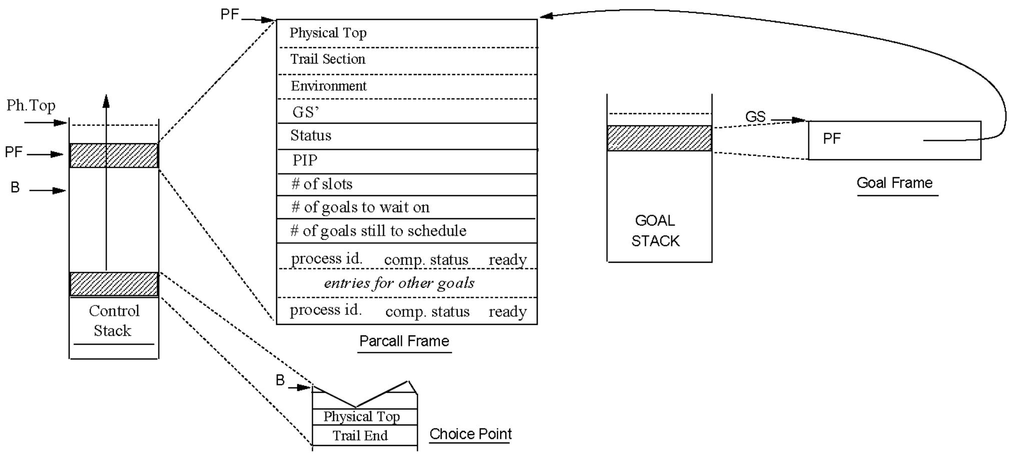


Figure 3: Additional data structures and related registers in ACE

During execution of and-parallel Prolog programs, when a parallel conjunction is reached that is to be executed in parallel (recall that a conditional parallel conjunction may be executed sequentially if the conditional test fails), a *parcall frame* is created.[4] The parcall frame is a descriptor of the parallel call, and it stores information regarding the parallel call as a whole and regarding the single subgoals belonging to the parallel conjunction.

In particular, the parcall frame contains:

- a slot for each goal in the parallel conjunction where information regarding the state of execution of that goal will be recorded (e.g. location of the execution, determinacy information, etc.);
- necessary information about the state of the execution of the parallel conjunction (e.g. number of subgoals still to be executed, status of the execution, connections with outermost parallel calls, etc.).

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.

In the original RAP-WAM proposal (and in DASWAM[5] as well), a descriptor of each subgoal (*goal frame*) is created and allocated in the goal stack. Idle and-agents will access remote goal stacks, extract a goal frame, and use the information to start a remote execution. In the current version of ACE a slightly different approach has been used. The goal stack contains pointers to the currently open parallel calls with work available for remote execution (as shown in figure 3). Experimental results [42] have shown that this approach is more efficient and results in considerable reduction in memory consumption.

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.

Execution of a parallel goal starts with the allocation of an *input marker* (to denote the beginning of a new stack section) and is completed by the allocation of and *end marker* (to denote the end of the stack section allocated for that goal). Markers are used to:

- separate the different sections of a stack belonging to execution of different subgoals—this is fundamental for garbage collection purposes;
- link (in the correct order) the different subgoals belonging to the same parallel call—this is fundamental to implement a correct backtracking semantics, as explained in the next section.

Markers store various sorts of information, like:

1. pointers to the various stacks (local stack, global stack) to allow garbage recovery;
2. links to logically preceding subgoal;
3. sections of trail associated with the present computation;

[4]This can be done in the control stack, as in ACE and DASWAM, or in the local stack, as in the RAPWAM and &-Prolog.

[5]DASWAM [47] is na abstract machine for an extension of &-Prolog (DDAS) which incorporate dependent and-parallelism.

The processor which completes the execution of an and-parallel subgoal belonging to a parallel call will proceed with the (sequential) execution of the parallel call continuation. When a solution for every subgoal in a parallel conjunction has been found so that the execution of the continuation can begin, we say that the parallel conjunction has been *closed* or *completed*.

2.4 Backward Execution

Backward execution denotes the series of steps that are performed following the failure of a test (e.g., an 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 ACE, where both or- and and-parallelism are exploited, backtracking should also avoid taking alternatives already taken by other or-agents.

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. The communication among the processors involved in such backtracking process in performed using "signals", which can be viewed as specialized messages carrying the information needed to perform backtracking correctly.

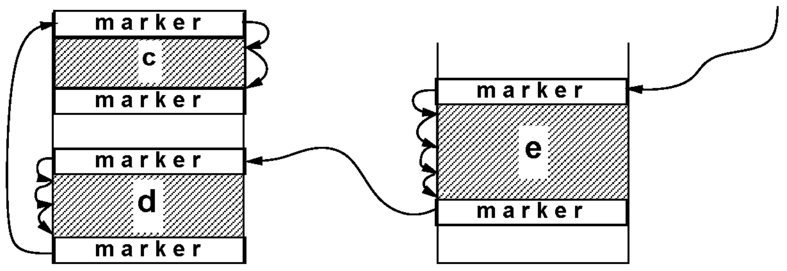
In any case, it must be ensured that the backtracking semantics is such that all solutions are reported. One such backtracking semantics has been proposed in [31]. Consider the subgoals shown below, where ',' is used between sequential subgoals (because of data-dependencies) and '&' for parallel subgoals with no data-dependencies (note, also, that ⟨*conditions*⟩ do not appear, because they are supposed to always hold):

```
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 initiated when the parallel call is in inside status). The latter behavior is a form of "intelligent" backtracking[6]. 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.

[6]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 [17, 7, 39, 18].

The presence of the markers allows the backtracking activity to move in the "right" direction. The markers represent the entry and exit point of each subgoal. This is further illustrated in figure 4.



... (c & d & e) , g

Figure 4: Backtracking over an And-parallel computation

If failure occurs in the inside phase, inside backtracking is used—and the whole parcall should fail, all the goals in the parallel conjunction being 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 the standard WAM.

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 WAM. 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. Thus, a *redo signal* has to be sent to the remote processor. 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 any more answers, the goal before the CGE will be backtracked over as in normal sequential execution.

Independent and-parallelism with the backtracking semantics described above was originally proposed in RAP-WAM [29], and has been efficiently implemented in the ACE system.

2.5 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 offer some very challenging issues:

Logical vs. Physical: the Warren Abstract Machine has been carefully designed to support efficient sequential execution of logic programs. In its design the WAM 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 existing at its creation time.

This is not anymore true when dealing with unrestricted scheduling of goals[7] [48]: 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 [28]).

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

Backtracking Policy: the backtracking semantics described earlier specifies *what* (and *when*) has to be done, but does not hints 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 and the current version of ACE use private backtracking, while DDAS [47] implements public backtracking. The choice of private backtracking in ACE has been mainly based on the issue of supporting stack-copying for or-parallelism. Private backtracking guarantees a "better" distribution of the computation between the processors, making it easier during stack-copying to decide which stack sections need to be copied. (Nevertheless, a prototype implementation of ACE with public backtracking is currently under development for comparison purposes.)

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 choicepoints. This makes trail management slightly less efficient than in traditional sequential implementations (two pointers, instead of one, pointing to the beginning and end of the relevant section of the trail, need to be saved in each choicepoint), but considerably simplifies both backtracking and management of or-parallelism.

Garbage Collection: the use of private backtracking allows recovery of a considerable amount of garbage "on-the-fly" during execution. Nevertheless, garbage collection remains more complicated, due to the lack of correspondence between logical order of backtracking and

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

physical distribution of the computation. Parts of computation which are not on the top of the stack may be backtracked over first, leaving behind *holes* in the stacks (this is the so–called "garbage slot" problem) that need to be properly tagged and eventually recovered.

Or-parallelism: last but not least, ACE has been developed from the beginning as a system in which and-parallelism and or-parallelism coexist and are concurrently exploited. The and-parallel engine was designed with this in mind, so that future extendibility of the system to also exploit or-parallelism is not compromised. The main aim in designing the and-parallel engine was to keep physical memory organization as clear and as close to the logical memory organization as possible. This is essential for efficient detection of areas to be copied during stack-copying. In particular, the adoption of a "segmented view" to the management of the trail stack and the use of private backtracking are meant to simplify the identification of the bindings to be installed/de-installed and to allow parallelization of the stack copying operation. While the above considerations have to be taken into account, the overall design of the and-parallel component of ACE is only marginally affected by them. This is because the and-parallel engine is a "basic" component of the system; an or-parallel agent is composed of a team of several and-parallel engines [21].

3 Signal Management

As mentioned in the previous sections, during execution the and-agents need to exchange messages. Each message implies a request sent to the destination agent for execution of a certain activity. The system supports two kinds of messages:

1. *redo* messages—used to request a remote backtracking activity. This is necessary whenever the logical path of the computation continues on the stack of a different agent.

2. *kill* messages—used to request a remote killing activity.

Some of the messages could be avoided by allowing an agent to freely perform backtracking on the stack of another agent, as in the public backtracking scheme. On the other hand, in the implementation of ACE we require the use of private backtracking, as mentioned previously.

Messages are sent and received asynchronously. The frequency at which an agent checks for the presence of messages can be tuned by modifying some system defined constants.

3.1 Kill Signals

In this section we discuss two methods for implementing the *kill* operation. As mentioned earlier, performing the kill is not easy since it is a global operation on the execution tree that may involve more than one processor.

The killing of a single subgoal G, once the corresponding processor has received a signal affecting it, involves the complete removal of all the information allocated on the stacks of the processor(s) during the execution of G. It consists of two actions:

- **garbage collection**: recovery of the stack space occupied by the killed computation;

- **trail unwinding**: removal of all the bindings generated during the killed computation.

None of these actions impose any sort of constraint on the order in which they must be performed (i.e. the various parts of the computation may be deallocated and unwound in any order, since they are mutually independent). However, care must be taken while mixing forward and backward execution (there might be data dependencies among the goals in the CGE and the previous ones). For instance, consider a goal:

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

in which the processor P1 that executed goal `a` also picks up goal `b`. Goals `c` and `d` are picked up by other processors. Suppose goal `b` fails, then processor P1 will send a kill to the processors executing `c` and `d`. While the kill of `c` and `d` is in progress, P1 cannot backtrack over `a`, and restart a new alternative—since the new alternative of `a` may reuse some memory locations that are successively modified by the trail unwinding process of `c` or `d`. In other words, the processor executing the goal preceding the parallel conjunction should not restart computation unless `c` and `d` are completely killed and the correct state restored to begin the new computation.

A kill phase is always started by a failing worker that reaches a parcall frame during backtracking. This covers two cases:

- the parcall frame is reached while there are other and-parallel subgoals of this parcall that are still active (i.e. the parcall frame is in *inside* status). In this case all the other subgoals of the parcall need to be killed.

- the parcall frame is reached after all the subgoals have detected at least one solution (i.e. the parcall frame is in *outside* status). In this case the backtracking semantics previously described is applied. When the CGE is re-entered after backtracking over the continuation, messages are sent to those and-parallel subgoals to the right whose computation was deterministic (i.e., these determinate goals do not offer any further alternatives, and hence such goals should be killed immediately rather than backtracked over). A redo message is sent to the rightmost non-deterministic subgoal.

In the second case, all the processors involved in the kill operation are free to return to their normal previous operations once the kill is completed (since the worker generating the kills is itself taking care of continuing the execution by sending *redo* messages to the non-deterministic subgoals). In the first case, instead, once the kill is completed, one of the workers involved in the kill needs to continue the main execution by backtracking over the computation preceding the parcall frame.

A kill can be serviced *lazily* or *eagerly*. Each approach requires a different kind of support from the underlying runtime system. The only data structures that are common to both the lazy approach and the eager approach described in this paper are those that are required to support sending/receiving of kill messages. Kill messages are realized by associating a kill-message queue with each worker. The kill-message queue of a processor should be accessible to all other processors—and consequently the access operations on these queues should be atomic.

3.1.1 Kill Steps

The process of killing a computation can be further subdivided into two distinct phases:

1. **Propagation phase:** in which the kill signal is propagated to all the and-parallel branches nested inside the and-branch of the subgoal being killed;

2. **Cleaning phase:** in which the space from killed computation is removed (garbage collection and trail unwinding).

The execution of the cleaning phase is relatively easy but requires the knowledge of the physical boundaries of the computation to be removed. The stack structure adopted to store the computation by any Prolog inference engine allows exclusively a bottom-up traversal of the computation tree (i.e. we can only visit the computation tree starting from the leaves and moving upwards, towards the root), which corresponds to scanning the stack from the top towards the bottom.[8] Thus, to scan the tree, we *at least* require pointers to the bottommost leaf nodes that represent the point from which the upward traversal to clean up should begin. Once this starting point is known, cleaning is a straightforward operation, which resembles in many aspects a backtracking process. As in backtracking, the worker performing the kill scans the stack, removing each encountered data structure and unwinding the part of trail associated with that part of the computation. The main differences with the backtracking process are:

- alternatives in the choice points are ignored—and the choice points are removed;
- parcall frames are treated as if they are in *inside* status, i.e. kills towards all the subgoals of the parcall frame are generated.

It is important to observe that the cleaning activity can be performed quite efficiently since parallel branches enclosed in a killed subgoal can be cleaned in parallel. Once the bottommost extreme of the computation to be killed has been detected, the cleaning step can be immediately applied. Figure 5 shows this process. The main issue—and the most difficult problem—is the actual detection of the location of the leaves from where the cleaning activity can be started. This is the purpose of the *propagation step* mentioned earlier and the rest of the section will deal with different approaches to tackle this problem.

In the following we present two approaches for propagating kills (with possible variations). These approaches are parameterized by:

1. **direction of the propagation:** two possible directions can be considered
   (a) *top-down*: kill signals are *actively* propagated from the root of the killed subgoal to the leaves;
   (b) *bottom-up*: kill is started from the leaves and pushed towards the root of the subgoal.

   Note that a *top-down* element is always present in any kill propagation mechanism since, after all, a kill is received by a subgoal and has to be propagated to its descendent parcall frames. The difference in the two approaches is related to how *eagerly* the top-down component is exploited.

2. **mode of propagation:** the propagation of the kill signals in the tree can be realized in two alternative ways:
   (a) *active*: the various workers are actively receiving (or seeking) and propagating the kill signals;
   (b) *passive*: workers lazily wait to receive a kill directed to them.

[8] A scan in the opposite direction would be very expensive, due to the variable size of the structures allocated on the choice point stack.

3.1.2 Lazy Propagation of Kill Messages

The main idea behind this propagation technique is to avoid sending kill messages (unless they are strictly necessary). This is realized by leaving to each processor the task of realizing when the computation that it is currently performing has been killed.

In the lazy approach to killing, a kill message is sent to a worker *only* when the bounds of the computation are known (i.e. the computation to be killed has already been completed). In this case the cleaning step can be immediately applied.

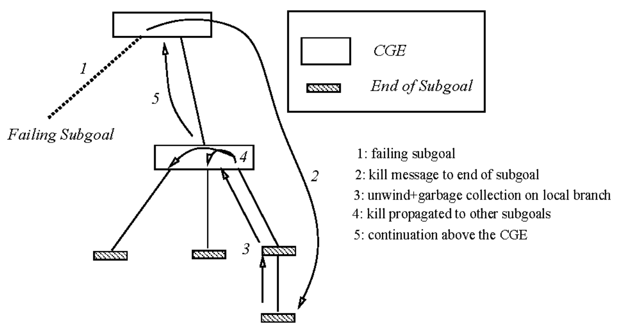

CGE
End of Subgoal
1
Failing Subgoal
5
4
2
3
1: failing subgoal
2: kill message to end of subgoal
3: unwind+garbage collection on local branch
4: kill propagated to other subgoals
5: continuation above the CGE

Figure 5: Cleaning Operation during a Kill

If the kill is issued when the branch to be killed is still computing, then a *suspended kill* is generated. A suspended kill is simply a kill operation that will be completed later; all the necessary information to execute the kill is stored in the slot describing the killed subgoal. The effects of this operation are:

1. since the worker which completes the execution of the subgoal will access the slot for updating the various fields of the slot (like recording its id for backtracking purposes), it will immediately realize that the computation that it has just completed has been previously killed and it will automatically start performing the cleaning operation (as explained above).

2. if the and-scheduler selects an and-parallel subgoal that is subsumed by another subgoal with a suspended kill then it will immediately discard the goal and look for new work. Key to this step is the presence of a representation of the computation tree which allows us to efficiently determine whether one subgoal is subsumed by another (i.e. one is a descendent of the other in the search tree).

3. periodically each worker checks whether its current computation is subsumed by one of the goals killed by a suspended subgoal. If this condition is satisfied then the worker will immediately interrupt the computation and start the cleaning phase.

The beauty of this approach lies in its simplicity. The scheme can also take advantage of many of the algorithms that have been developed for efficient backtracking (lazy kill is almost identical to backtracking). Furthermore, a worker is never distracted by kill messages during a useful computation, since the checks performed will affect its execution only if the worker is positioned on a killed branch of the tree. In this way the kill operation is postponed and performed only when no useful work is available.

The main disadvantages that we can identify in this approach are the following:

1. the implementation of this scheme relies on the availability of a representation of the computation tree which allows to determine efficiently whether a given subgoal is a descendent of another. It is an open problem whether this can be done in constant time.

2. the execution of the kill may be slower than in other schemes; this is due to the fact that cleaning is started by one processor from the bottommost end of a branch, making it an inherently sequential operation. Other approaches may offer a higher degree of parallelism during the cleaning up of execution.

3. more speculative work (which will be eventually undone) is being performed, so resources like processors and memory are used in what will be an unused computation.

A simplified version of this approach to kill has been implemented in the &-Prolog system. Detection of kill only occurs at the end of the subgoal execution. This leads to a very simple implementation involving little overhead. On the other hand it has the important drawback of being unsafe w.r.t. infinite computations [27].

3.1.3 Eager Kill

The disadvantages mentioned above seem to make the Lazy Kill approach not too advantageous, at least in principle. For this reason we propose a different approach, called eager kill, which is mainly (but not exclusively) a **top-down** approach (see Fig. 6(i)).

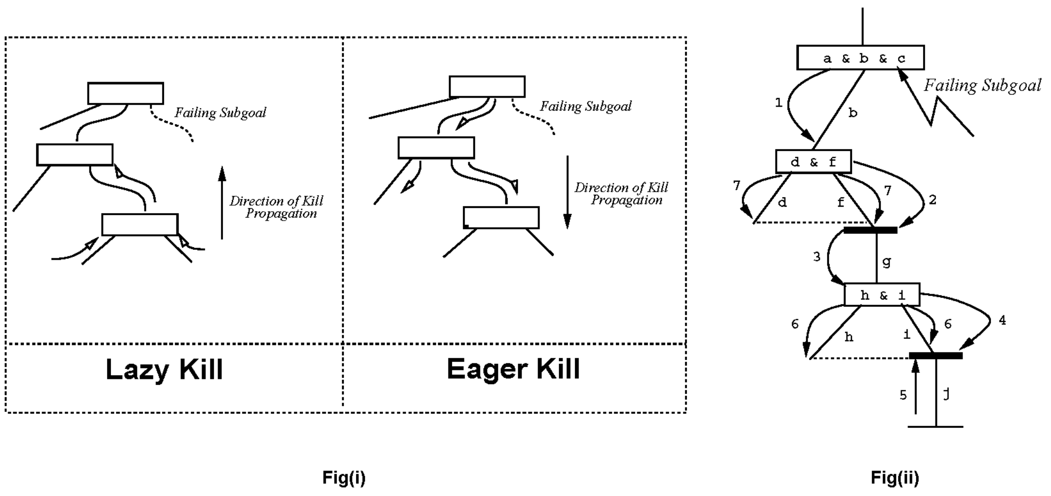


Figure 6: Eager and Lazy Kill

The main problem in this approach is the lack of information that will allow us to perform a top-down traversal of the tree (starting from a given node towards the leaves). As we will see later on, this can be accomplished with a small amount of information.

One of the main issues in killing is that of 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 kill of the part of the computation tree above the parcall may be started only once 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 situation may occur.

This introduces the issue of *scheduling* the worker on serving the kill messages. Using queues to store the kill messages and function calls to serve them introduces a specific and rigid ordering on the kill serving operations. The first kill to be received is the first to be served and, whenever the worker has left behind some busy-wait loops, it will return to them in the opposite order (w.r.t. the order in which it accepted the new kill messages). This approach may not be the optimal one. Any situation in which the same worker P_i is in charge of killing the computation preceding two different CGEs may lead to a potential delay (e.g. all the subgoals of one CGEs have been removed but P_i is suspended on the other CGE waiting for some subgoals to be killed). A better scheme can be realized by relaxing the implicit analysis order enforced by the recursive nature of the kill-serving function and by restructuring the killing activity as a loop which iterates as long as there is at least one parallel call on which such a worker is required to wait, but allowing a dynamic rescheduling of the killing activity.

Next, we present an example to illustrate our technique for eager propagation of a kill.

Example:

Let us consider the computation described in Figure 6(ii).

Assuming that processor P_i is the one which started the execution of `b`, then the initial kill message will be sent to P_i from the worker which failed in the computation of `c`. If P_i was looking for work by invoking the and-scheduler, then it will simply leave the and-scheduler and start serving the kill. Otherwise, at the next check for kill, it will suspend the current execution and move to serve the kill.

In the Eager kill approach, P_i has access to the first parcall frame generated during the computation of `b`. It positions itself on that parcall frame and, since this has already completed (i.e. it is in outside status), it starts the killing activity by sending a kill to the continuation of the parcall frame (step **2**). The continuation itself contains another parcall frame; the worker receiving this kill message will access the parcall frame (step **3**) and send another kill message to its continuation (since even this parcall frame is in outside status). The worker executing `j` will receive the kill and serve it, removing the whole computation of `j` and setting the appropriate bit in the parcall frame `(h & i)`. At this point the worker that is busy waiting on such parcall frame (busy waiting until the continuation has been killed) will kill all the subgoals of the parallel call (`h` and `i`, step **6**) and then continue further and remove `g`. Once `g` has been removed, a bit in the parcall frame `(d & f)` is set and P_i, which was in the meantime busy waiting on that parcall frame (busy waiting until the continuation has been killed), may proceed to send the kill messages to the subgoals of the parallel call (`d` and `f`, step **7**) and, once all of them have reported the end of the kill, it may proceed with the killing of `b`.

Once the whole branch has been removed and also `a` has reported the end of the kill, the worker P_i is free to restart the computation previously interrupted.

Note that both the Lazy and Eager schemes for propagating kill can be optimized further, however, we do not describe these possible improvements due to lack of space. More details can be found elsewhere [21]. The current version of ACE [20, 21] incorporates a hybrid kill management mechanism, where the lazy mechanism has been improved by making use of frequent check points (to verify the presence of kills along the current computation branch), together with a complete support of the workers scheduling mechanism mentioned above (to deal with multiple parallel calls failing concurrently).

4 System Optimizations

Innumerable optimizations can be applied to a recomputation-based and-parallel system like ACE. Some optimizations that have been implemented in the current version of the system deal with taking advantage of various forms of determinacy that arise during the computations. In the following text some of these optimizations are discussed. Last Parallel Call Optimization (LPCO), Shallow Parallelism Optimization, and Processor Determinacy Optimization are optimizations based exclusively on the run-time behavior of the program, while the Backtracking Families Optimization makes use of specific information produced by a suitable compile-time analysis tool. All of them are based on general optimization principles [43, 23], aimed at the exploitation of determinacy through *simplification* of the structure of the computation and *reuse* of parts of the computation. Some of these optimizations were suggested in [28] and left as future work.

4.1 Last Parallel Call Optimization

Last Parallel Call Optimization (LPCO) is a generalization of the Last Call Optimization [51]—adopted in most sequential implementations—to the case of parallel calls. Its intent is to merge, whenever possible, distinct parallel conjunctions. Last Parallel Call Optimization can lead to a number of advantages (discussed later). The advantages of LPCO are very similar to those for last call optimization [51] in the WAM. The conditions under which the LPCO applies are also very similar to those under which last call optimization is applicable in sequential systems.
Consider first an example: `?- (p & q).` where

```
p :- (r & s).          q :- (t & u).
```

The and-tree constructed is shown in Figure 7(i). One can reduce the number of parcall nodes, at least for this example, by rewriting the query as `?- (r & s & t & u)`[9]. Figure 7(ii) shows the and-tree that will be created if we apply this optimization. Note that executing the and-tree shown in Figure 7(ii) on ACE will require less space because the parcall frames for `(r & s)` and `(t & u)` will not be allocated. The single parcall frame allocated will have two extra goal slots compared to the parcall frame allocated for `(p & q)` in Figure 7(i). It is possible to detect cases such as the one above at compile time. However, our aim is to accomplish this saving in time and space at runtime. Thus, for the example above, our scheme will work as follows.

[9]Under the assumption that the two clauses are the only matching ones.

When the parallel calls `(r & s)` and `(t & u)` are made, the runtime system will recognize that they are the last parallel calls in their respective clauses and that the parallel call `(p & q)` is immediately above. Instead of allocating a new parcall frame some extra information will be added to the parcall frame of `(p & q)` and allocation of a new parcall frame avoided. Note that this is only possible if both `p` and `q` are determinate, i.e. they have at most one matching clause. The extra information added will consist of adding slots for the goals `r`, `s`, etc. In particular, no new control information needs to be recorded in the parcall frame of `(p & q)`. However, some control information, such as the number of slots, etc., need to be modified in the parcall frame of `(p & q)`. It is also necessary to slightly modify the structure of a slot in order to adapt it to the new pattern of execution.[10]

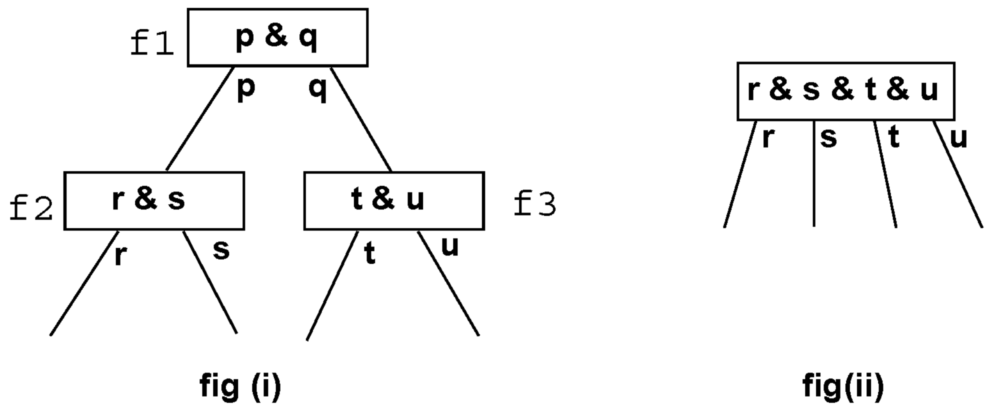


Figure 7: Optimization Schemes

It is important to observe that, if the goal `r` is to fail in inside mode, then in case (ii) (see Figure 7(ii)) killing of computation in sibling and-branches will be considerably simplified. In case (i) the failure will have to be propagated from parcall frame f2 to parcall frame f1. From f1 a kill message will have to be sent out to parcall frame f3. In case (ii) a linear scan of only one goal list is sufficient.

One could argue, as mentioned earlier, that the improved scheme described above can be accomplished simply through compile time transformations. However, in many cases this may not be possible. For example, if `p` and `q` are dynamic predicates or if there is not sufficient static information to detect the determinacy of `p` and `q`, then the compile-time analysis will not be able to detect the eventual applicability of the optimization. Our scheme will work even if `p` and `q` are dynamic or if determinacy information cannot be statically detected (because it is triggered only at runtime). Also, even more relevant, for many programs the number of parallel conjunctions that can be combined into one will only be determined at run-time [43].

In general, application of LPCO requires two *conditions* to be satisfied:

1. determinacy of the computation between two nested parallel calls;
2. non-existence of any continuation after the nested parallel calls (i.e., only the topmost parcall can have a continuation).

[10]For example, it is necessary to keep in each slot a pointer to the environment in which the execution of the corresponding subgoal will start.

These conditions are satisfied by a large number of programs (e.g., tail-recursive programs) [22]. Work is in progress to generalize this optimization, so that it applies to a wider range of programs [40, 42].

It is important to observe that the cost of verifying applicability of LPCO at run-time is absolutely negligible (comparison of two pointers). This is a further justification for keeping the optimization as a pure run-time operation.

4.2 Shallow Parallelism Optimization

The Shallow Parallelism optimization is aimed at reducing marker allocation by taking advantage of deterministic computations. Many programs involve the development of deep nestings of parallel calls, while the sequential subgoals (those which do not contain a further parallel call) are deterministic computations. The main idea is that once one of those deterministic computations has been completed, there is no need to keep any data structure alive (since on backtracking there will not be any alternatives available). For this reason the allocation of the input marker is *delayed* until the first choice point/parcall frame is allocated (in a fashion similar to *shallow backtracking* technique [6]). If the end of the computation is reached without allocating any input marker, then the end marker itself is not allocated (we simply need to record the boundaries of the current trail section in the descriptor of the subgoal). On backtracking no kill messages need to be generated for this kind of subgoals—we just need to unwind the trail section indicated in the corresponding slot of the parcall frame. This simple optimization allows savings in time and space since various data structures are not allocated and the number of messages sent during backtracking is reduced.

The cost of applying this optimization is minimal: a simple check at the time of choice-point creation. In all the benchmarks tested we were not able to observe any slow-down due to application of the Shallow Parallelism optimization.

The optimization is also illustrated in figure 8(i).

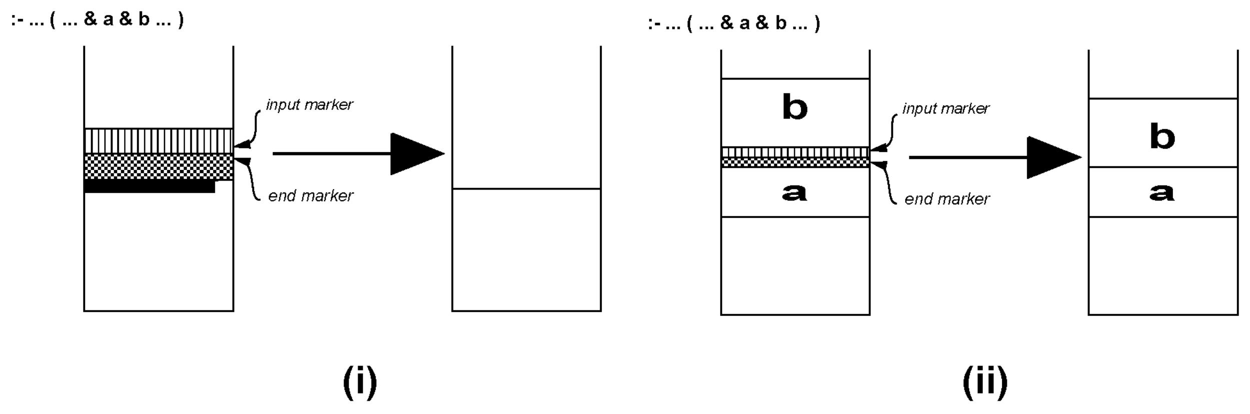


Figure 8: Shallow Parallelism and Processor Determinacy Optimization

4.3 Processor Determinacy Optimization

The aim of a general purpose and-parallel system is clearly to exploit the highest possible amount of parallelism, respecting the *no-slowdown* requirement (i.e., parallel execution should be guaranteed to run at least as fast as sequential execution[27]). Nevertheless the amount of parallelism exploited is often greater than the actual amount of computing resources available—which leads to situations in which the same computing resource (processor/agent/team of processors/etc.) will successively execute *different units* of parallel work, e.g. different subgoals of the same parallel execution. Thus, we get to a situation in which two potentially parallel pieces of computation are executed sequentially. In particular, two units of work can be actually executed contiguously and in the same order in which they would be executed during a purely sequential computation—if this is the case then all the additional operations performed that are related to the management of parallel execution (allocation of markers, etc.) represent pure overhead. The intent of the *Processor Determinacy Optimization* is precisely to reduce this sort of overhead as much as possible.

This saving is obtained by simply avoiding allocating any marker between the two subgoals and—in general—treating them as a unique, contiguous piece of computation (See Figure 8(ii)). Intuitively, the optimization acts at run-time to coalesce smaller goals in larger units of work taken care of by a single computing agents.

There are several advantages in doing this:

1. memory consumption is reduced, since we are avoiding allocation of the markers between consecutive subgoals executed on the same processor;

2. execution time during forward execution is reduced since the whole phase of creating an end marker for the first subgoal and an input marker for the second one (or a unique marker as happens in &-Prolog) is skipped;

3. execution time during backward execution is also reduced, since backtracking on the two subgoals flows directly without the need to perform the various actions associated with backtracking over markers (sending messages, etc.).

4.4 Backtracking Families Optimization

As we will show, the previously described shallow parallelism and determinate processor optimizations can achieve considerable performance improvements (see Section 5.2) and have the advantage of not requiring compile-time analysis (although compile-time knowledge can always be used to reduce or avoid the small run-time tests needed). However, it is interesting to explore whether these ideas can be extended to cover more cases if some compile-time analysis information is available. In particular, we concentrate on the following types of information: knowledge that a goal (and its whole subtree) is deterministic, and knowledge that a goal has a single solution. In addition, knowledge that a goal will not fail is also quite useful [27]. It is beyond the scope of this paper to address how this information is gathered – the reader is referred to related work in the area of abstract interpretation based global analysis [33, 4, 11, 44, 14]. We will address instead how such information can be exploited at the parallel abstract machine level.

We start by considering the case in which several parallel goals, perhaps not contiguous in the program text, but which are known to be deterministic, end up being executed on the same processor. As an example, consider the parallel call `(a & b & c)`, where `a, b` and `c` are known to be deterministic. If `a` and `b` are executed on the same processor, the situation is as in the previous section and clearly no markers need to be allocated between the two goals. But if `a` and `c` are executed on the same processor, one after the other, since they are known to be deterministic, no markers need to be allocated between them either. This is based on the fact that if a, b, c are known to be deterministic and independent `(a & b & c)` is equivalent to `(a & c & b)`, and to any other permutation, modulo side effects.[11] The advantage is clearly that the input marker of a can be simply shared by c.

The optimization can also be applied in cases where whole collections of related deterministic goals are created in loops, as in

```
    p:- (a & b & p).
    p.
```

We assume that `a, b, p` are known to be deterministic. An execution of p would generate goals of the form

```
(a & b & a' & b' & a'' & b'' ...  & p)
```

Since all these goals are independent and deterministic, no intermediate marker is needed whenever they stack one over the other in a given processor, i.e., only one marker would be needed per processor (assuming there are no other parallel conjunctions). Note that when p is to be backtracked from outside, all the goals `a, b, a', b'...` have to be backtracked over. However, the order in which this is done is not important. Thus, every segment formed by consecutively stacked goals can be backtracked (only untrailing is really required) in one step by simply unwinding down to the sole input marker. This saves time and space in forward execution, since fewer markers are needed, and time in backward execution, since fewer intermediate steps and messages are needed. We will call the set of parallel goals `a, b, a', b', a'', b''...` a *backtracking family*: a set of independent parallel goals, such that all of them are backtracked over in the same backtracking step in the sequential execution. The fundamental characteristic of the members of a backtracking family is that they have a "common choice point" which they backtrack to in case of failure.

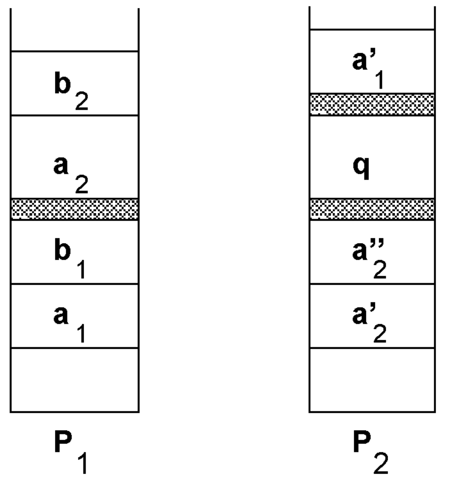


Figure 9: Backtracking families

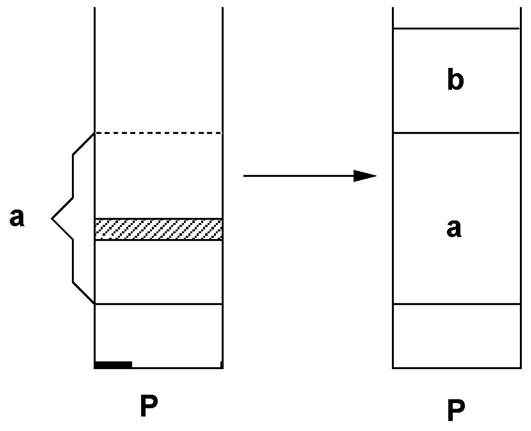


Figure 10: Eliminated choice points

[11]Note that side effects would have been taken into account beforehand by the parallelizer by imposing a synchronization among them — which, in the worst case, can lead to the sequentialization of the goals.

As mentioned before, unlike those proposed previously, this technique requires knowledge regarding goal determinism beforehand. In order to illustrate this, consider the situation in which, for (a & b & c), a is executed on processor P_1, and b in P_2. P_1 (which is deterministic) finishes first with a and picks up the goal corresponding to c (also deterministic). If b is deterministic, then there is no need for any marker between a and c (or saving the trail segment in the slot) since no intermediate backtracking is possible. This may be determined once b finishes, but then P_1 would have to wait for P_2, which is undesirable. On the other hand, if P_2 had finished before, then it could have stolen b, and the determinate processor optimization could have been applied.

Each backtracking family is given a unique identifier (e.g., the address of the "common choice point" that they would backtrack to). This identifier is also associated with each goal belonging to the family and stored in the input marker when the first goal of a family is picked up by a processor. When a goal is picked up by a processor, if it has a family identifier attached and it is the same as that of the current input marker in the processor, no new input marker needs to be allocated. It is clear that this has an implication on scheduling in the sense that picking up goals belonging to the same backtracking family as the last goal executed in a given processor is always preferable.

Markers are still necessary in principle between deterministic and non–deterministic goals, and between goals that do not belong to the same backtracking family. If we have (p_1 & q & p_2), where p is defined as above, and q generates non–deterministic computations, then the goals generated by p_1 can be stacked one over the other without markers. However, markers are needed to separate goals generated by p_1 and q, p_2 and q, and p_1 and p_2 (this is illustrated in Figure 9, where segments marked with $'$, $''$... correspond to different activations of the same goal, and goals marked with a subindex are offsprings of the corresponding initial call, p_1 or p_2).

However, note that the optimization is not necessarily restricted to deterministic goals, as might be implied by the discussion above. In fact, the fundamental characteristic of the goals of a backtracking family is that they have a "common choice point" that they backtrack to in case of failure of one of such goals. Thus, if a goal is deterministic in the end (i.e., it produces only one solution) it can also benefit from the proposed technique, even if it does create choice points and backtrack internally along the way, provided that it can be determined that such choice points will not provide additional solutions or that they will be discarded upon termination (for example, by executing a "cut"). In summary, goals which can be determined to have a single solution, independently of whether they create choice points during their execution and perform backtracking internally, are also eligible for forming a backtracking family with other such goals or with deterministic goals. Simple examples are (a, !) & (b, !), or even (a & b), !, where a and b may have non–determinism inside, but are made single solution by the presence of the cut (see Figure 10). As another, more elaborate example, consider sorting a list of complex items (i.e., not simple numbers) with quick-sort, where the tests performed in the partitioning predicate could be arbitrarily complex, leaving intermediate choice points and backtracking internally, but finally yielding only one solution. We believe it is possible to detect this "single solution" status in many cases through existing compile-time analysis techniques, even if this cannot be determined locally, as in the simple examples above.

In order to implement the proposed optimization the determinism information is passed to the low-level compiler through source program annotations. In the same way as we have assumed for the "&" annotations, these determinacy annotations can be provided by the user or generated by an automatic analyzer. Note that, while the annotation is static, its effect has a dynamic nature in general in the sense that, for a given program, the actual performance increase may differ from execution to execution due to different schedulings, which would result in different relative stackings of members of different families, and thus different actual numbers of markers allocated.

Regarding the benefits obtainable from the optimization proposed, it can clearly provide considerable savings in memory consumption, and, as a side effect of this, time savings due to the smaller number of markers which have to be initialized. In an ideal situation, in which all goals picked up by each processor belong to the same backtracking family only one marker would be allocated per processor (even while choice points internal to a parallel goal are allocated, used, and eventually discarded before the parallel goal finishes). Furthermore, and as mentioned before, backtracking is potentially also greatly sped up.

5 Preliminary Performance Results

The purpose of this section is to present the results obtained by executing some well-known benchmarks. They range from simple test programs to actual applications. The results for the following benchmarks are initially reported: Matrix Multiplication, Quicksort, Takeuchi, Tower of Hanoi, Boyer (a reduced version of the Boyer-Moore theorem prover), Listsum (a naive list-processing program operating over nested lists), Compiler (the PLM Prolog compiler written by P. VanRoy that is approximately 2,200 lines of Prolog code), POccur (a list processing program), BT_cluster (a clustering program from British Telecom, UK), Annotator (the annotator part of the ACE/&-Prolog parallelizing compiler that is about 1,000 lines), and Simulator (a simulator for simulating parallel Prolog execution that is about 1,100 lines, written by Kish Shen).

Table 1 illustrates the speedups obtained for the various benchmarks (all the figures have been generated on a Sequent Symmetry multiprocessors). The figures clearly indicate that the current implementation, even though not completely optimized, is quite effective. On many benchmarks, containing a sufficient amount of parallelism, the system manages to obtain linear speedups (e.g., for Matrix Multiplication and Hanoi). With more processors in the multiprocessor systems we believe we should be able to obtain higher speedups, provided the program contains enough parallel work.

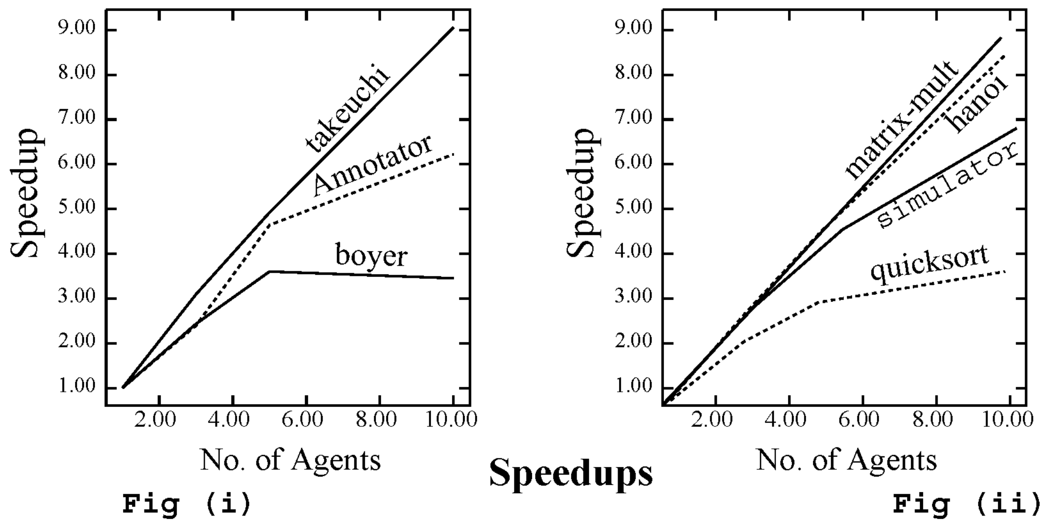


Figure 11: Speedups Curves for Selected Benchmarks on the Sequent Symmetry

Goals executed	ACE agents			
	1	3	5	10
matrix_mult(30)	5598	1954	1145	573
quick_sort(10)	1882	778	548	442
takeuchi(14)	2366	832	521	252
hanoi(11)	2183	766	471	231
boyer(0)	9655	5329	3816	2887
poccur(5)	3651	1255	759	430
bt_cluster	1461	528	345	215
annotator(5)	1615	556	392	213
compiler	29902	12522	6437	4801

Table 1: Execution times in msec. (Sequent Symmetry)

Goals executed	ACE agents		
	1	2	3
matrix_mult(40)	1379	784	498
annotator(12)	1025	550	400
compiler	5859	3317	2127

Table 2: Execution times in msec. (Sun Sparc 20)

Speedups for some benchmarks are shown in Table 1 and plotted in Figure 11. Note that for the *annotator*, *quick_sort*, and *boyer* benchmarks, the speedup curve flattens out because at some point all available parallelism is exhausted. Our implementation incurs an average parallel overhead of about 5%-30% over the version of SICStus Prolog it is based on. This parallel overhead is considerably reduced by triggering optimizations mentioned earlier.

As mentioned before, our benchmarks have been executed on a Sequent Symmetry. It is important to observe that tests made on other parallel machines have produced comparable speedups. Table 2 indicates some performance figures obtained on a 4 processor Sparc 10, which gives results comparable with those in Table 1 in terms of speedup, and serves as well to appreciate the relative speed of the two machines.

5.1 Shallow Parallelism

The shallow parallelism optimization has been incorporated in the ACE system. The results obtained have been extremely good. On average, an improvement of 5% to 25% in execution time over an unoptimized implementation is obtained due to this optimization alone. We can observe some of the results obtained in Table 3. In this table the execution times and relative percentage of improvement obtained on some common benchmarks are listed.

Observe that those benchmarks which show the best improvement under the Shallow Parallelism Optimization are those which contain a considerable amount of parallelism (nesting of over 1000 parallel calls) and in which the "leaves" of the computation tree are deterministic computations (*hanoi* and *takeuchi* are two such benchmarks). For other benchmarks the effects of the

Goals executed	ACE agents 1	3	5	10
matrix_mult(30)	5598/5214 (7%)	1954/1768 (10%)	1145/1059 (8%)	573/534 (7%)
takeuchi(14)	2366/1811 (23%)	832/586 (30%)	521/368 (29%)	252/200 (21%)
hanoi(11)	2183/1671 (23%)	766/550 (28%)	471/336 (29%)	231/180 (22%)
poccur(5)	3651/3197 (12%)	1255/1079 (14%)	759/662 (13%)	430/371 (14%)
bt_cluster	1461/1343 (8%)	528/480 (9%)	345/312 (10%)	204/189 (7%)
annotator(5)	1615/1422 (12%)	556/475 (15%)	392/322 (18%)	213/187 (12%)

Table 3: Unoptimized/Optimized Execution times in msec (% improvement in parenthesis)

optimizations are more limited—for example in the matrix multiplication benchmark the whole computation is deterministic but the determinism is not detected because of (i) the presence of nested (deterministic) parallel computations, and (ii) the presence of choice points whose remaining alternatives lead to failure. The "backtracking families" optimization discussed earlier, can be used in these cases, but requires compile-time analysis. The previously described benchmarks are quite deterministic in nature. Still, the shallow parallelism optimization gives surprisingly good results also on more complex benchmarks, involving backtracking across parallel subgoals. For example, running a program to solve a *map-coloring* problem (involving backtracking over parallel conjunctions), we obtained an average improvement in execution time of 14%. Clearly, since the main issue of this optimization is the avoidance of allocation of certain data structures, the computation will also gain considerable advantage in terms of memory consumption. Figure 12 illustrates the savings on the number of markers allocated obtained for some of the benchmarks (executing using a single processor). We go from an extreme case like *boyer* in which no saving at all is obtained, to some extremely good results, like for *takeuchi*, in which we save almost 50% of the total number of markers.

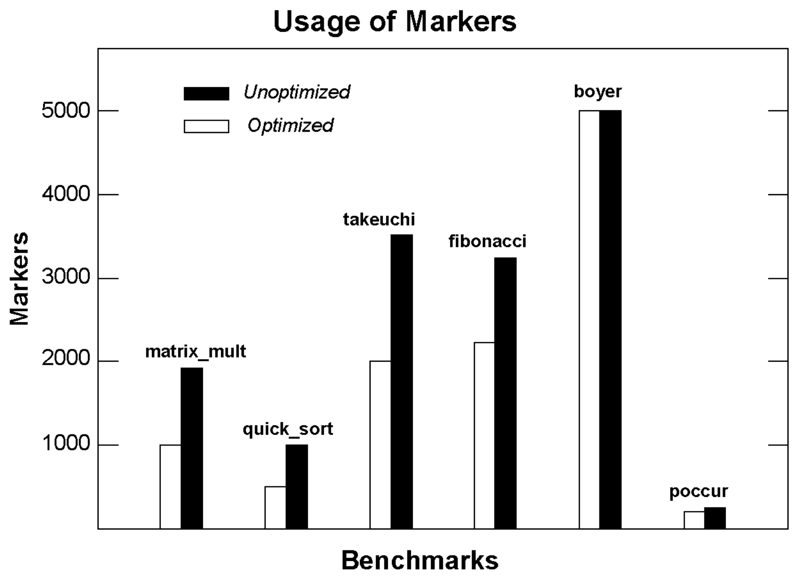


Figure 12: Number of Markers allocated (using one agent)

Finally, figure 13(i) shows the behavior of the optimization on the Takeuchi benchmark varying the number of processors used.

5.2 Processor Determinacy Optimization

Regarding forward execution, the results obtained with this optimization are very encouraging, as can be observed from Table 4. For many examples the optimization manages to improve the execution time from 4% to almost 20%.

Goals executed	ACE Execution	
	Unoptimized	Optimized
bt_cluster	1461	1391 (5%)
poccur(5)	3561	3418 (4%)
matrix_mult(30)	5598	5336 (5%)
listsum	2333	2054 (12%)
hanoi(11)	2183	1790 (18%)
takeuchi(14)	2366	1963 (17%)

Table 4: Unoptimized/Optimized Execution times in msec (single processor)

The variations in improvement depend exclusively on the number of parcall frames generated and on the effect of the marker allocation overhead on the overall execution time. For this reason we obtain considerable improvements in benchmarks like *takeuchi*, where we have deep nestings of parallel calls and the marker allocation represents the main component of the parallel overhead.

The optimization maintains its effects when we consider execution spread across different processors, as we can see in figure 13(ii), which shows the execution times of both the optimized and the unoptimized version for the Hanoi benchmark.

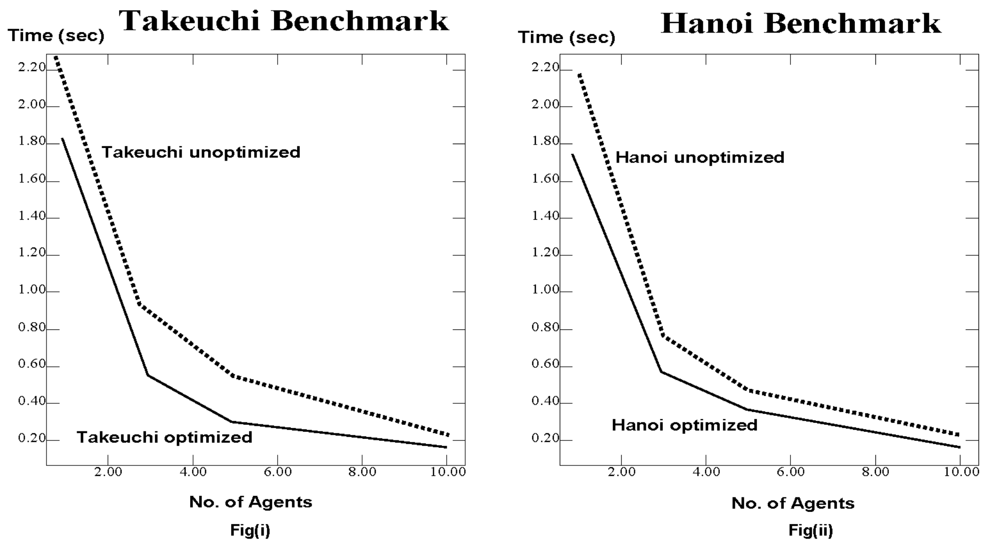


Figure 13: Shallow Parallelism Optimization and Processor Determinacy Optimization

The advantages are even more evident in terms of memory consumption: as we can see from Table 5 the number of markers allocated is cut in almost all the cases to half of the original value (measured during a single processor execution)—this because most of the examples analyzed have parallel calls of size 2 and the optimization allows avoiding the allocation of the marker between the two subgoals.

Goals executed	ACE Execution	
	Unoptimized	Optimized
bt_cluster	120	60
deriv(0)	174	87
poccur(5)	100	50
matrix_mult(30)	1798	899
listsum	3000	1500
takeuchi(14)	3558	2372
hanoi(11)	4094	2047

Table 5: Memory Consumptions (no. of markers)

5.3 Last Parallel Call Optimization

Table 6 illustrates the results obtained executing some of the benchmarks on ACE using LPCO.

Goals executed	ACE Execution with Shallow Parallelism Optim.			
	fw/no lpco	fw/lpco	bw/no lpco	bw/lpco
bt_cluster	890	843 (5%)	929	853 (8%)
deriv(0)	94	34 (64%)	131	38 (71%)
poccur(5)	3216	3063 (5%)	3352	3226 (4%)
annotator(5)	1327	1282 (3%)	1334	1281 (4%)
matrix_mult(20)	1724	1649 (4%)	1905	1696 (11%)
search(1500)	2354	1952 (17%)	8370	2154 (74%)

Table 6: Unoptimized/Optimized Execution times in msec (single processor)

The results are extremely good for programs with a certain structure. In particular, programs of the form `p(...) :- q(...) & p(...)`, where `q(...)` gives rise to a deterministic computation with a sufficiently deep level of recursion, will offer considerable improvement in performance. Interesting results are also seen by examining the effect of *inside failures* during execution: the use of LPCO allows further improvement. The presence of a single parcall frame considerably reduces the delay of propagating kill signals (kill signals are sent to sibling and-branches by a failed subgoal in a parallel conjunction to remove them from the computation). Table 7 shows the result obtained causing an inside failure during the execution of the matrix multiplication benchmark.

Also in terms of memory consumption, the combination of LPCO and Shallow Parallelism has proven to be extremely successful—while the LPCO cuts on the number of parcall frames,

Goals executed	ACE Execution	
	Unoptimized	Optimized
matrix_mult-inside-fail-first	5346	3100 (42%)

Table 7: Unoptimized/Optimized Execution times in msec (single processor)

the Shallow Parallelism optimization removes the allocation of input and end markers. Table 8 summarizes these results: each entry of the form $a/b \rightarrow c/d$ indicates that the number of markers needed went down from a to c and the number of parcall frames went down from b to d when the LPCO and shallow parallelism optimizations were applied. The second line of the table indicates the total percentage of improvement in stack consumption obtained by introducing the LPCO. Also these figures have been obtained by using a single processor during the execution.

Goals executed	ACE Execution				
	bt_cluster	*deriv*	*poccur(5)*	*serial*	*matrix_mult(40)*
Markers/Parcall	119/60 → 1/1	134/87 → 0/1	95/50 → 40/1	16/11 → 0/1	1638/1599 → 0/1
% Improvement	49%	45%	34%	43%	39%

Table 8: Memory Usage (Markers/Parcall-Frames for Unoptimized → Optimized)

5.4 Backtracking Families Optimization

Clearly, the practical advantages which can be obtained automatically with the backtracking families optimization strongly depend on the quality of the compile-time analysis performed or, if done manually, of the annotations provided by the user. The general issue of static analysis of determinism is beyond the scope of this paper. However, the potential of the optimization can still be assessed by making reasonable assumptions regarding the information that could be obtained based on the current state of the art in global analysis, and annotating the programs to encode this information. We have done this for a number of benchmarks and the results are shown in Table 9, which shows the number of markers allocated (without and with the optimization) when executing on 10 processors.

As expected, the number of markers in the optimized version actually differs much from run to run – a range over a large number of runs is given in this case. It can be observed that, as expected, the reduction in the number of markers allocated is quite significant, and larger than with the dynamic methods studied previously (which have the obvious advantage on the other hand of not requiring analysis). The results are graphically compared in figure 14.

As mentioned before, the advantage comes either from the knowledge that parallel calls that do create choice-points (and thus are not eligible for the shallow backtracking optimization dynamically) are in fact deterministic, or from the knowledge that goals that are not deterministic and are picked up by a processor out of contiguous order (and thus are not eligible for the determinate processor optimization) are in the same backtracking family. For example, it is quite simple to determine by global analysis (using the same information that the parallelizing compiler

Goals Executed	&–Prolog Execution	
	unoptimized	optimized
deriv(1)	261	[9]
deriv(2)	2109	[11]
deriv(3)	16893	[11]
deriv(4)	135165	[11]
boyer(0)	24	[3 – 8]
boyer(1)	747	[111 – 153]
boyer(2)	7290	[543 – 745]
boyer(3)	282168	[4770 – 6500]
quick_sort(50)	150	[13 – 15]
quick_sort(100)	300	[14 – 16]
quick_sort(150)	450	[15 – 17]
quick_sort(200)	600	[15 – 16]
poccur(1)	30	[9 – 10]
poccur(2)	60	[11 – 12]
poccur(3)	90	[11 – 13]
poccur(4)	120	[12 – 13]
poccur(5)	150	[12 – 13]
takeuchi(13)	1412	[19 – 23]
takeuchi(14)	4744	[21 – 29]
takeuchi(15)	10736	[21 – 30]
takeuchi(16)	21236	[23 – 28]

Table 9: Backtracking Families Optimization (memory consumption, 10 proc.)

uses to parallelize the benchmark) that the matrix benchmark is completely deterministic. This is the case also with most of the other examples in Table 9. However, the issue of determining precisely the exact extent to which this optimization is applicable in large programs using state of the art analysis technology remains a topic for future work.

It may be noticed that the number of markers allocated with no optimization is somewhat larger than in the previous tables. This is simply due to the fact that in order to split the work involved in system modification for testing the different optimizations proposed these tests were run on the current version of the &-Prolog system, rather than on ACE.

While due to the collaborative work between our two teams the two systems are currently quite similar, this version of &-Prolog differs slightly from the current implementation of ACE, and this justifies the slight difference in the number of markers used when no optimizations are implemented.

5.5 Kill Management Performance

Figure 15(i) shows the execution time obtained for a program that involves massive amount of killing. This program is for computing *fibonacci(16)*, where, after the computation is finished a failure is forced. As a result, the whole tree created during the Fibonacci computation needs to be unwound and removed. We expect that the time to kill, i.e., to unwind and remove the tree, should be approximately the same as the time it takes to construct the tree. Hence, the expected time for executing this program that fails at the end should be twice the time for

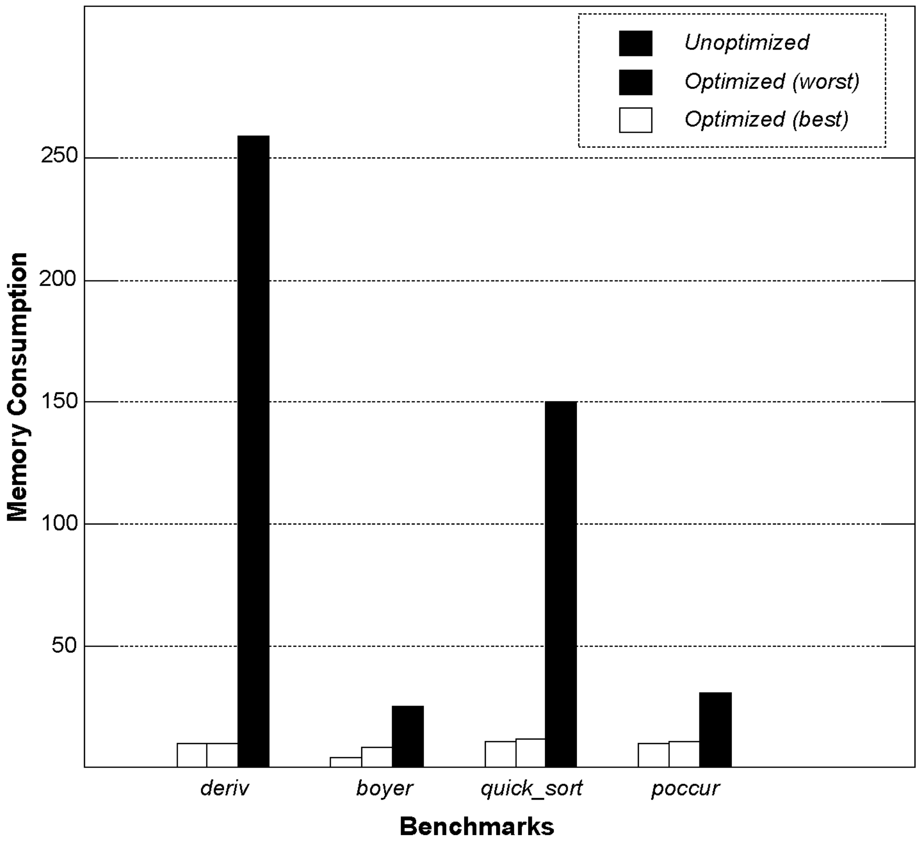


Figure 14: Improvements using Backtracking Families

successfully computing *fibonacci(16)*. The two curves (labeled "Actual time" and "Expected time" respectively) in Figure 15(i) show the actual execution time and the expected execution time obtained for different number of agents.

A third curve (labeled *Optimized Time*) shows the slight improvement obtained by introducing the determinate processor optimization (i.e., sequentializing consecutive and-parallel subgoals executed by the same agent thus reducing the number of input markers allocated and the number of parallel threads). The figure illustrates that with fewer processing agents (≤ 2) the actual time to process a parallel conjunction and then kill it is smaller than the expected time for this computation (in other words the time it takes to unwind and kill the tree is less than the time it takes to create it). This is because of the fact that with fewer processors each processor ends up sending signals to itself which can be handled faster. With larger number of processors the time to communicate kill messages becomes quite significant; as a result, and not too surprisingly, the actual time exceeds the expected time. The figure also shows that reducing the number of parallel threads via our optimization results in improved performance. This is also expected since by reducing the number of threads we effectively reduce the number of kill messages that will be exchanged.

The discussion so far suggests that killing adds a significant overhead to computation. However, killing leads to some advantages as well—in some cases failure of a computation can be detected quite a bit earlier, resulting in super-linear speedups. Figure 15(ii) illustrates this other extreme of the lazy approach to killing. The program, whose execution time is plotted against the number of processors, consists of activating two parallel threads. The first thread contains a huge amount of computation, while the second encounters an early failure. The early failure will not be immediately detected in sequential execution (it will be detected after the huge first thread is finished and the second thread is started), while in any parallel computation it will be immediately detected and propagated to the first thread avoiding the huge computation. This results in super-linear speedups as shown in Figure 15(ii).

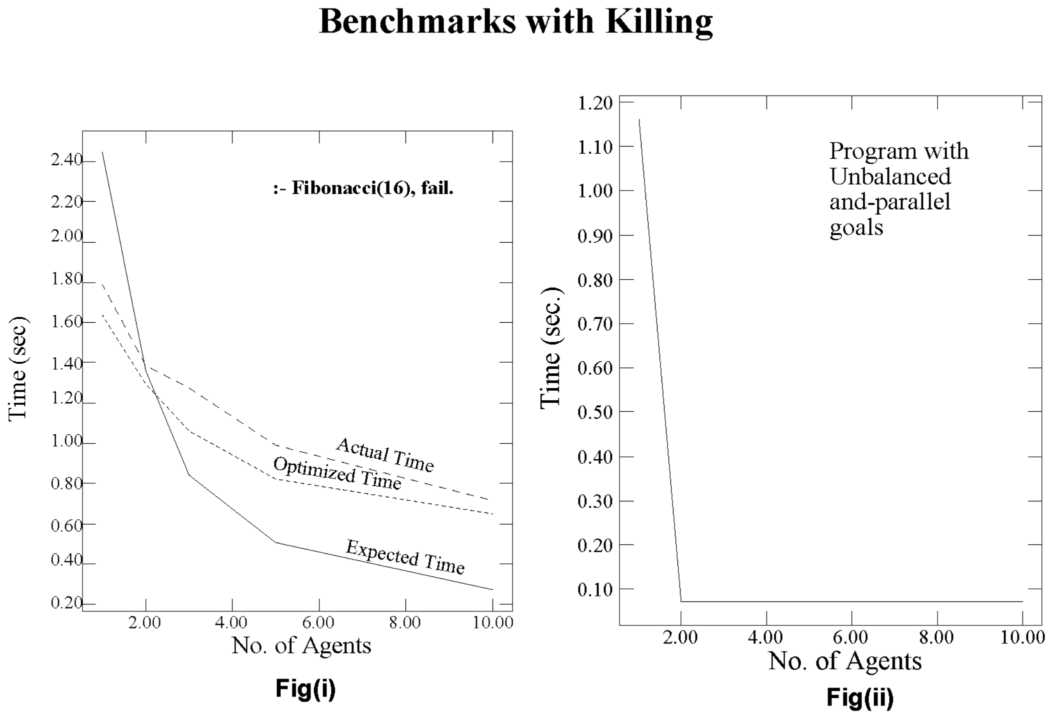


Figure 15: Benchmarks involving Killing

6 Conclusions

This paper describes some of the important features of the independent and-parallel component of the ACE system—a system which implicitly exploits both independent and- and or-parallelism from Prolog programs. We discussed the structure of the abstract machine and the organization of the execution, placing emphasis on the new ideas and optimizations introduced in the design. We presented results for a comparatively large suite of benchmarks, some over two thousand Prolog lines long. Our results show that our system is well-suited for parallel execution of symbolic applications coded in Prolog. These results also confirm our initial contention that ACE can exploit and-parallelism with the efficiency of and-parallel only systems such as &-Prolog: performance is quite close to that reported for the original version of &-Prolog described in [24], despite the fact that: (i) ACE contains a significantly richer implementation of signal management and backtracking than &-Prolog had at that time The version of &-Prolog reported in [24] did not implement outside backtracking because this restriction results in many simplifications in the backtracking machinery (backtracking was still allowed elsewhere, as, for example, within parallel goals). It was expected that it would not be a serious burden on the compiler to detect cases where backtracking across parallel goals might occur, which would then not be parallelized. However, for generality, and because ACE also exploits or-parallelism, support for full backtracking over parallel conjunctions is fully justified and has been shown herein not to result in too high an overhead; (ii) ACE supports or-parallelism. The results obtained with ACE are quite similar, as expected, to those obtained by &-Prolog, since the two models are based on similar principles. The optimizations introduced in ACE allow ACE to gain better execution efficiency and, in certain situations, better speedups (e.g. when LPCO allows to optimize computations containing heavy backtracking over parallel calls).

Apart from &-Prolog, very few "real" and-parallel Prolog systems have been realized. APEX [36] is another and-parallel system implemented by Lin and Kumar. Performance figures for APEX have been published for only some very simple benchmarks. For these benchmarks both &-Prolog and ACE appears to be superior or equivalent.

DDAS [47] is an extension of the RAP-WAM model to the case of dependent and-parallelism. Very recently a parallel implementation has been finished [45], but there are no performance figures available anywhere yet. Older articles on the DDAS used a high-level simulation, which does not allow realistic comparisons: the reported speedups were equivalent to those of ACE on most of the benchmarks, except few cases where DDAS gets better speedups, due to the fact that the simulator does not take into account certain overheads encountered in a real parallel implementations (e.g. need to use locks).

Our results also show that the optimizations that we proposed, that are also applicable to other parallel systems, can significantly reduce the memory and execution overhead in and-parallel systems. With these various optimizations ACE performance is the same as or better than that of the previously reported implementation of &-Prolog and the memory consumption greatly reduced with respect to that reported in previous studies [48].

References

[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, 1990.

[2] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In G.Levi and M.Martelli, editors, *Proc. of the 6th International Conference on Logic Porgramming.* MIT Press, 1990.

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

[4] F. Bueno, M. García de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Univ. Politecnica de Madrid, November 1994.

[5] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In Springer-Verlag, editor, *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994.

[6] M. Carlsson. On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog. In *Sixth International Conference on Logic Programming*, pages 3–16. MIT Press, June 1989.

[7] S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368, 1989.

[8] A. Colmerauer. Les gramaire de metamorphose. Technical report, Univ. D'aix-Marseille, Groupe De Ia, 1975.

[9] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

[10] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP*, 1990.

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

[12] S. Debray D. Gudeman, K. De Bosschere. jc: An Efficient and Portable Sequential Implementation of Janus. In *Procs. of the Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[13] M. García de la Banda, M. Hermenegildo, and K. Marriot. Independence in constraint logic programming. In *Proc. of the 1993 International Symposium on Logic Programming.* MIT Press, 1993.

[14] S. Debray, P. López-García, and M. Hermenegildo. Non-failure Analysis for Logic Programs. Technical Report CLIP14/94.0, Univ. Politecnica de Madrid, October 1994.

[15] S. K. Debray, N.-W. Lin, and M. Hermenegilo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation.* ACM Press, June 1990.

[16] D. DeGroot. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems*, pages 471–478. Tokyo, Nov. 1984.

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

[18] G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 1995. to appear.

[19] G. Gupta and M. Hermenegildo. Recomputation based Implementation of And-Or Parallel Prolog. In *Int'l Conf. on 5th Generation Computer Sys. '92*, pages 770–782, 1992.

[20] G. Gupta, M. Hermenegildo, and E. Pontelli. &ACE: A High-performance Parallel Prolog System. In *IPPS 95*. IEEE Computer Society, Santa Barbara, CA, April 1995.

[21] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.

[22] G. Gupta and E. Pontelli. Data–Parallel Execution of Prolog Programs in ACE. In *IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1995.

[23] G. Gupta and E. Pontelli. Optimization Principles for Parallel Implementation of Nondeterministic Languages and Systems. Technical report, Dept. of Computer Science, New Mexico State University, 1995.

[24] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

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

[26] M. Hermenegildo and P. Lopez-Garcia. Efficient Term Size Computation for Granularity Control. In L. Sterling, editor, *Proc. of the Twelfth International Conference on Logic Programming*. MIT Press, 1995.

[27] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[28] M. V. Hermenegildo. *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.

[29] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proc. 3rd ICLP, LNCS 225*, pages 25–40. Springer-Verlag, 1986.

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

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

[32] A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

[33] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 13/20:503–581, 1994.

[34] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, Weizmann Institute, 1992.

[35] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

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

[37] P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific Publishing Company, September 1994.

[38] E. Lusk and al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3), '90.

[39] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming.* MIT Press, June 1989.

[40] E. Pontelli and G. Gupta. Nested Parallel Call Optimization. Technical report, New Mexico State University, 1994.

[41] E. Pontelli and G. Gupta. An Overview of the ACE Project. In *Proc. of Compulog ParImp Workshop*, 1995.

[42] E. Pontelli and G. Gupta. On the Duality Between And-parallelism and Or-parallelism. In *Proc. of Euro-Par'95.* Springer Verlag, 1995.

[43] E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of Parallel Prolog Implementations. Technical report, New Mexico State University, 1994.

[44] P. Van Roy and A.M. Despain. High-performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25(1), 1992.

[45] K. Shen. Parallel execution of DASWAM. Private comunication.

[46] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-parallel Scheme. In *Proc. Joint Int'l Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[47] K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

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

[49] L. Sterling, editor. *The Second International Conference on the Practical Application of Prolog*. Royal Society of Arts, 1994.

[50] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[51] D. H. D. Warren. An Improved Prolog Implementation Which Optimises Tail Recursion. Research Paper 156, Dept. of Artificial Intelligence, University of Edinburgh, 1980.