&ACE: The And-parallel Component of ACE
(A Progress Report on ACE)

Enrico Pontelli
Gopal Gupta

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

Manuel Hermenegildo

Facultad de Informática
Universidad Politècnica de Madrid
28660-Boadilla del Monte
Madrid, Spain
herme@dia.fi.upm.es

Abstract

ACE is a computational model for full Prolog capable of concurrently exploiting both Or and Independent And-parallelism. In this paper we focus on the specific implementation of the And-parallel component of the system, describing its internal organization, some optimizations to the basic model, and finally presenting some performance figures.
Keywords: Independent And-parallelism, Or-parallelism, implementation issues.

1 Introduction

The ACE (And-Or/Parallel Copying-based Execution) model [6] uses stack-copying [1] and recomputation [5] to efficiently support combined Or- and Independent And-parallel execution of logic programs. ACE represents an efficient combination of Or- and independent And-parallelism in the sense that penalties for supporting either form of parallelism are paid only when that form of parallelism is actually exploited. Thus, in the presence of only Or-parallelism, execution in ACE is *exactly* as in the MUSE [2] system—a stack-copying based purely Or-parallel system. In the presence of only independent And-parallelism, execution is *exactly* like the &-Prolog [7] system - a recomputation based purely And-parallel system. This efficiency in execution is accomplished by introducing the concept of *teams of processors* and extending the stack-copying techniques of MUSE to deal with this new organization of processors. The purpose of this report is to describe in full details the independent and-parallel component of the ACE system, as it has been developed in the last few months in the Laboratory for Logic, Databases, and Advanced Programming of the New Mexico State University in collaboration with Universidad Politécnica de Madrid, Spain. The report is organized as follows. The first part gives a general introduction to ACE, first describing separately the techniques adopted for Or-parallelism and Independent And-parallelism, and then showing how they are merged in the ACE framework. Section 2 introduces the model of independent and-parallelism adopted, focusing on the structure of the operational semantics (forward and backward execution). Section 3 presents the details of the and-parallel engine, describing its components and how they interact with each other. Section 4 presents some advanced issues, such as management of kill and redo signals, and some optimizations implemented in the system. Section 5 concludes the report, presenting some results obtained by executing a set of well-known benchmarks in and-parallel.

1.1 Or-Parallelism in ACE

ACE exploits Or-parallelism by using a stack-copying approach (like MUSE [2]). In this approach, a set of *or-agents* (workers in the case of MUSE, teams in the case of ACE—as explained later) maintain a *separate* but *identical* address space (i.e. they allocate their data structures starting at the same logical addresses). Whenever an *or-agent* $\mathcal{A}$ is idle, it will start looking for unexplored alternatives generated by some other or-agent $\mathcal{B}$. Once a choice point p with unexplored alternatives is detected in the computation tree $\mathcal{T}_\mathcal{B}$ generated by $\mathcal{B}$, then $\mathcal{A}$ creates a local copy of $\mathcal{T}_\mathcal{B}$

and restarts computation by backtracking over p and executing one of the unexplored alternatives. The fact that all the or-agents maintain an identical logical address space allows to reduce the creation of a local copy of $\mathcal{T}_\mathcal{B}$ to a simple memory copying operation (as shown in figure 1). This whole operation of obtaining work from another agent is named *sharing* of or-parallel work.

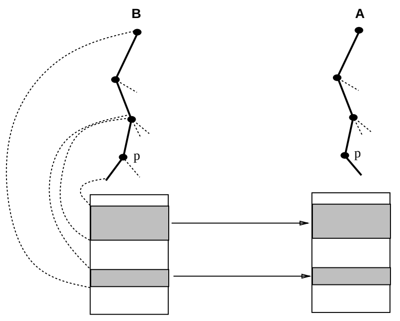

Figure 1: Stack-copying based Or-parallelism

In order to reduce the number of sharing operations performed (since each sharing operation may involve a considerable amount of overhead), unexplored alternatives are always searched starting from the bottommost part of the tree; during the sharing operation all the choice points in between are shared between the two agents (i.e. at each sharing operation we try to maximize the amount of work shared between the two agents).

Furthermore, in order to reduce the amount of information transferred during the sharing operation, copying is done *incrementally*, i.e., only the difference between $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_\mathcal{B}$ is actually copied.

1.2 And-Parallelism in ACE

ACE exploits Independent And-parallelism using a recomputation based scheme [3], 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). Figure 2 sketches the structure of the computation tree created in the presence of and-parallel computation: a *parbegin-parend* structure is introduced, and the different branches are assigned to different agents. Since we are exploiting only *Independent* And-parallelism, only independent subgoals are allowed to be executed concurrently by different *and-agents*. Dependencies are detected at the run-time by executing some simple tests introduced by the *parallelizing compiler*. In ACE we have adopted the technique originally designed by DeGroot [4] and refined by Hermenegildo [8] (adopted also by &-Prolog [7]) of annotating the program at compile time with *Conditional Graph Expressions (CGEs)*. This will be explained in details in a later section.

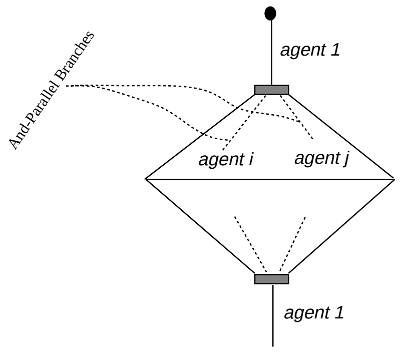

Figure 2: Computation Tree with Recomputation-based And-parallelism

Since the and-agents are computing just different parts of the same computation (i.e. they are cooperating in building one solution o the initial query) they need to have *different* but *mutually accessible* logical address spaces.

1.3 And-Or Parallelism in ACE

In ACE parallelism is exploited at two different levels. At the higher level we mapped the notion of *or-agent* to the notion of *team of processors* (i.e. an or-agent is a set of processors) and the different or-agents interact to exploit or-parallelism. At the lower level we mapped the notion of *and-agent* to the notion of processor inside a team (i.e. each processor is an and-agent) and the various and-agents inside the same team interact to exploit the and-parallelism present along the or-branch computed by the whole team. This is illustrated in figure 3.

This organization allows us to:

1. minimize the amount of changes to be done to the basic and-parallel engine;

2. clearly draw the boundaries between the different components of the system.

Each processor in the ACE system is basically an and-parallel engine, capable of carrying on its own computation and interacting with a certain number of other processors (those belonging to the same team). The only new features that need to be added are the followings:

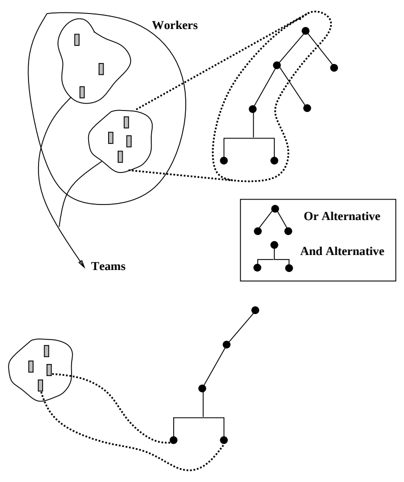

Figure 3: Workers Organization in the ACE system

- a mechanism to keep track of the amount of or-parallel work produced by the computation of a team;
- a mechanism to allow interaction of one team with another in order to guarantee synchronization and execution of sharing operations;
- extended backtracking, allowing calls to the or-scheduler whenever a team backtracks over a shared choice point.

A sharing operation involves copying (part of) the computation tree generated by a team to another team. The operation is not straightforward since the computation tree is spread over the address spaces of the different processors belonging to the team. Furthermore, we want to perform copying incrementally, i.e., transfer only what is strictly necessary, and this selective operation is complicated by the arbitrary ordering of the different sections of computation on the stacks of the various team members. To make the whole process more effective, we have introduced in ACE some *sharing conditions* that a choice point should satisfy in order to allow its sharing with other teams (i.e., giving away its untried alternatives for or-parallel processing to other teams) . A choice point p satisfies the *sharing conditions* iff the whole computation on its left (in the and-tree exploited by the team) has already been completed. Figure 4 shows an example of this: choice point p satisfies the conditions, while the choice point q does not, since there is a branch on its left which has not completed yet.

In ACE, during a sharing operation, only the choice points satisfying the sharing condition are

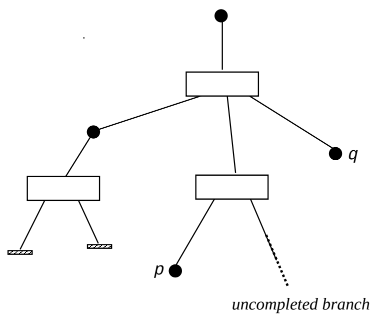

Figure 4: Sharing assumptions in ACE

actually taken into consideration. This simplifies both the copying operation (it is easier to detect which parts of the computation need to be transfered), the scheduling activity (we are guaranteed that everything on the left is terminated and successful and we do not have arbitrary intermixing of shared and private parts in the computation tree), and the management of side-effects and extra-logical predicates.

Different approaches to incremental copying have been studied and heuristics to choose the most appropriate in each situation have been developed. The interested reader is referred to [6] for a detailed discussion of this topic.

2 Independent And-parallelism

The main purpose of this report is to illustrate the structure and the features of the And-parallel engine developed for the ACE system. ACE exploits independent and-parallelism (i.e. only subgoals that do not share any variables are executed in and-parallel), following the model designed by DeGroot [3] and successively refined by Hermenegildo in his &-Prolog system [7]. This section explains the basic principles behind this computational model; the first section illustrates the notion and the meaning of *Conditional Graph Expressions (CGEs)*, while the second and third section illustrates the computational behaviour of an engine capable of executing programs annotated with CGEs. It should be pointed out at the outset that our design of the and-parallel component of ACE is very heavily influenced by RAP-WAM and its implementation in &-Prolog. In fact, much of the machinery is the same except for a few minor modifications and some optimizations.

2.1 Conditional Graph Expressions

A conditional graph expression (CGE for simplicity) is an expression of the form:

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

where

$\langle$ *conditions* $\rangle$ - is a conjunction of simple tests on variables appearing in the clause (typical tests are *ground*, that verifies whether the argument is instantiated to a ground term, and *independent*, that verifies whether the arguments share any variables with arguments of other goals);

& - 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 [7]:

1. $\mathtt{B}_i$ can actually represent an arbitrary sequential conjunction of subgoals (and the system should deal with its execution in the appropriate way);

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

A standard Prolog program need 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 or by using some specialized compile-time analysis tools (like the &-Prolog parallelizing compiler [9]).

2.2 Forward Execution

Forward execution of a program annotated with CGEs is quite straightforward. Whenever a CGE is encountered, the conditions are evaluated and, if the evaluation is successful, the various subgoals in the CGE are made available for and-parallel execution. Idle and-agents are allowed to pick up available subgoals and executed 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. Forward execution of a CGE resembles the *parbegin-parend* construct introduced in various concurrent imperative programming languages.

2.3 Backward Execution

Backward execution denotes the series of steps that are performed following a *failure*—due to unification or lack of matching clauses. 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 presence of CGEs, standard backtracking should be upgraded in order to deal with computations which are spread across processors. &ACE has a complete implementation of such a backtracking scheme.

As long as backtracking occurs over the sequential part of the computation no particular problems occur—we just use plain Prolog-like backtracking. Problem occurs when backtracking involves a CGE. Two cases are possible:

- backtracking occurs **inside** one of the branches $\mathtt{B}_i$ of the CGE, and no solutions are found inside $\mathtt{B}_i$. In this case we can observe that the whole CGE does not have any solution (since the subgoals are known to be independent). This allows the removal of the whole CGE and propagation of backtracking to the computation preceding the CGE itself. Figure 5 depicts this case.

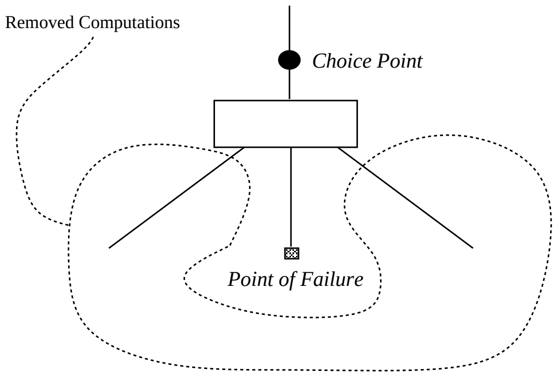

Figure 5: Inside backtracking

- backtracking occurs in the continuation of the CGE (or, anyway, outside a CGE), and there are no alternatives between the point of failure and the parallel call. This situation is illustrated in figure 6. In this case backtracking should try to mimic Prolog backtracking, by searching for a new alternative moving from the rightmost branch of the CGE to the leftmost one. If a new successful alternative is found in the branch B_i, then all the subgoals B_j (with $i < j \leq n$) are re-executed in parallel. This is called *recomputation-based* and-parallelism, as mentioned earlier, since the subgoals on the right are completely recomputed for each new solution found on the left. If no successful alternatives are found in any of the subgoals of the CGE, then the whole CGE is removed and backtracking is propagated to the preceding computation.

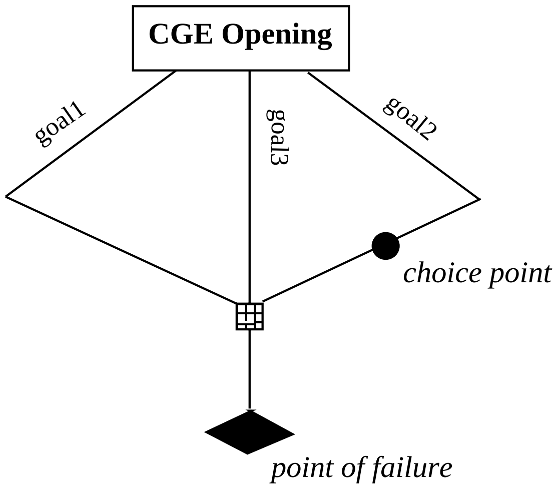

Figure 6: Outside Backtracking

This scheme permits us to obtain all the solutions of a CGE in the same order in which they are produced in a corresponding sequential execution (of course we are not considering the case in which some of the or-alternatives inside the CGE have been taken by some other or-agent for execution).

3 System Organization

The purpose of this section is to describe in detail the internal organization and the activity of the and-parallel engine of ACE (named &ACE for the sake of simplicity). ACE has been implemented on the top of *SICStus Prolog* ans as such it inherits the basic structure of the SICStus WAM architecture [10] together with most of its features and optimizations.

The following section is organized as follows. In subsection 3.1 the internal memory layout of &ACE and the data structures allocated during the execution are analyzed. Subsection 3.2 presents a brief overview of the few new instructions added to the SICStus instruction set. Subsection 3.3 deals with the issue of scheduling and-parallel work between the various and-agents. Finally, subsection 3.4 gives an overview of a typical &ACE execution.

3.1 Data Structures

3.1.1 Memory Layout

An and-agent in &ACE keeps a memory organization similar to the one used by a SICStus WAM. Figure 7 presents the memory map of a &ACE and-agent.

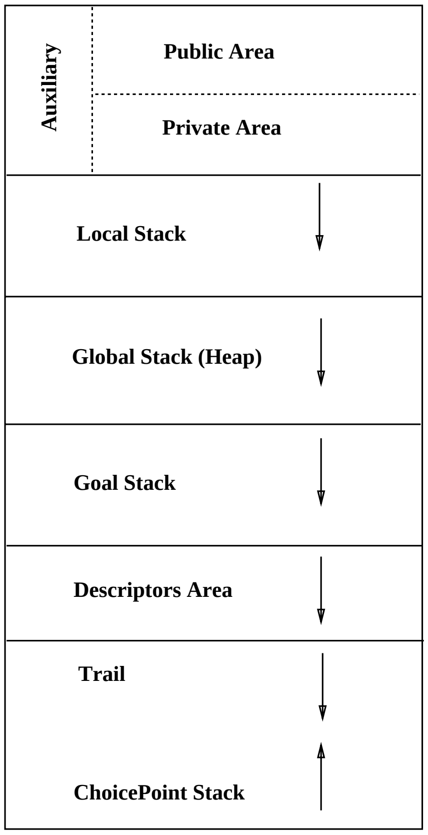

Figure 7: Memory Organization of an and-agent

Focusing on the new areas:

Auxiliary Area: it is used to store various information required during the execution. It is subdivided into two subareas:

- **Public Area:** this is accessible by all the members of the team, and it is used to store information used to perform and-scheduling (pointers to the top of the goal stack, amount of and-parallel work available, etc.);

- **Private Area:** this is accessible only by the specific and-agent and it contains essentially the various registers of the abstract machine (current choice point, current environment, program counter, etc.) and any additional information required by the local execution of the and-agent.

Goal Stack: whenever a CGE is encountered during the execution and the *conditions* are successfully evaluated, parallel execution is activated. The and-agent executing the CGE creates a record (called *goal frame*) for each subgoal of the CGE and stores it in its goal stack. Idle and-agents looking for work will extract a goal frame from the goal stack of a remote agent and use the information stored in it to start the local execution of the subgoal[1].

Descriptors Area: this area, managed as an heap, is used to allocated descriptors of the subgoals executed by the considered and-agent. This is required, as explained in section 4.1.2, to properly manage out-of-sequence kill signals.

The other areas are unchanged in their structure w.r.t. SICStus WAM. Note that the architecture of an &ACE agent is very similar to that of an agent in &-Prolog.

3.1.2 Data Areas

Execution in the WAM is characterized by allocation and deallocation of certain data structures on the stacks of the abstract machine. SICStus WAM uses:

- the *heap* to allocate *complex terms*;
- the *local stack* to allocate *environments*;
- the *choice point stack* to allocate *choice points*;
- the *trail* to save the addresses relative to *conditional bindings.*

&ACE makes use of some additional data structures (also present in RAP-WAM), required to support the and-parallel execution of subgoals.

Parcall Frame: parcall frames are allocated on the choice point stack and are used to identify the execution of a CGE. A parcall frame contains, in addition to other information:

- pointer to the environment existing at the time of execution of the CGE (required to allow a proper evaluation of the arguments of each subgoal);
- various counters to keep track of the subgoals of the CGE that have been executed, that are still executing, or that are still waiting in the goal stack;
- the lock required to guarantee mutual exclusion in the access of common areas (e.g. the parcall frame itself);
- a *slot* for each subgoal of the CGE is allocated in the parcall frame, collecting information regarding the execution of the corresponding subgoal (id of the processor which started the subgoal, current status of the execution, id of the processor which completed the execution, etc.).

Goal Frame: goal frames are allocated on the goal stack and are used to maintain the basic information required to start remotely the execution of such subgoal. Typical information stored in a goal frame are:

- a pointer to the code of the subgoal;
- pointer to the parcall frame relative to the CGE from which the subgoal has been taken.

On the goal stack the goal frames are organized in a double-linked list. This is required since it may be necessary to remove goal frames that are not lying on the "physical" top of the goal stack (e.g. during a kill operation).

Input Marker: input markers are allocated on the choice point stack and they are used to denote the beginning of the execution of a subgoal on a remote processor. An input marker is allocated by the processor that steals the subgoal and it indicates the beginning of a new "section" of the choice point stack. Typical information stored in an input marker are:

- a reference to the parcall frame and to the slot relative to the subgoal;
- a pointer to the environment in which the arguments of the subgoal are to be evaluated (i.e. the environment existing at the time in which the parcall frame has been allocated);

[1]the structure of a goal frame is illustrated in a later section

- a time stamp (used for killing purposes, as explained in section 4.1.2);
- a continuation pointer indicating the piece of code to be executed when the subgoal successfully terminates.

End Marker: end markers are allocated on the choice point stack and are used to identify the completion of a subgoal part of a parallel execution. It is used to denote the "logical" end of a stack section and to save the value of some registers which are needed in case of backtracking (like the registers pointing to the current active parcall frame). Since most of the information stored in an end marker is analogous to that required in an input marker (at least the part necessary for register management) and since an input marker can be allocated only after an end-marker, the two structures can be partially merged—in &ACE implementation the actual input marker allocation phase consists only of filling some empty fields of the previous end marker. This allows a considerable reduction in terms of space and time.

Trail Marker: trail markers are allocated on the choice point stack. To properly understand the role of the trail markers we need to analyze the two views of the choice points stacks in the &ACE system. Operations on the choice point stack may access the stack following two paths:

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

For example, allocation of a new data structure (e.g. a new choice point) needs to use the physical path (new structures can be allocated only on the real top of the stack), while backtracking should follow the logical path. Intermixing of different computations may occur because of backtracking on trapped subgoals—when the trapped subgoal is reactivated, the new data structures are allocated on the top of the stack—as shown in figure 8: backtracking on the trapped subgoal Y lead to the allocation of a new choice point on the top of the stack; at this point physical and logical paths are different.

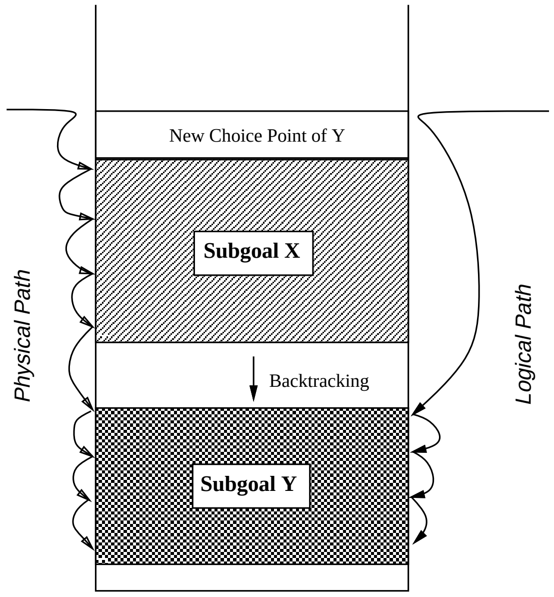

Figure 8: Backtracking on a trapped subgoal

Keeping track of the two paths is quite simple, since each data structure allocated on the choice point stacks contains

- a pointer to the previous structure along the logical path;
- a pointer to the previous structure along the physical path[2].

A trail marker is needed in order to keep track of the logical path in some special cases of backtracking. Specifically, a trail marker is required when both the following conditions are satisfied:

1. we are currently backtracking on a trapped subgoal (i.e. a subgoal which is not lying on the top of the choice point stack);
2. during backtracking on the trapped subgoal we take the last alternative from a choice point and its execution leads to allocation of new data structures.

In this situation a link between the old structures and the new ones is required (mainly for

[2]actually this pointer is unnecessary if we know the size of the structure, but keeping it explicitly allows to save time during the various operations

trail management purposes); this link is represented by the trail marker. As the name suggests, the most relevant pieces of information stored in a trail marker are the current trail pointers.

Section 3.4 illustrates how these data structures are used during an actual execution.

Every structure allocated on the choice point stack includes also some additional information required for

- *trail management*: because of the lack of a unique view of the choice point stack (as mentioned before, we have two different views of it, the logical and the physical one), trail management becomes slightly more complicated. It is not anymore sufficient to associate with each choice point structure the current top of the stack but we need to associate an identification of the *trail section* that need to be unwound during backtracking. A trail section is identified by a pair of pointers (top and bottom of the section).

- *garbage collection*: garbage collection over the choice point stack is complicated by the arbitrary intermixing of computations that may occur due to backtracking over trapped subgoals. Whenever traditional garbage collection is not possible (i.e. we are backtracking on a trapped subgoal) an explicit call to a garbage collection routine is performed (and only if there is evidence that garbage collection will be useful). A bit tag has been added to each data structure on the choice point stack to help this explicit garbage collection activity.

3.2 Instructions Set

A limited amount of changes have been done to the original SICStus WAM instruction set. Some new instructions have been added and some of the old ones have been appropriately updated. In general, all the old instructions are unchanged except for:

- some additional storing/restoring of the heap pointer;

- update of the `neck` instruction[3] to deal with some of the new fields present in the choice point structure.

[3]used in SICStus Prolog to switch from shallow to deep backtracking

The new instructions introduced are required to support the execution of the CGEs and the synchronization between parallel executions. These new instructions are as follows:

1. `pcall`: a certain number of instructions have been added to allow creation of a new parallel call (pcall), allocation of a parcall frame, and activation of a parallel execution.

2. `check_goals`: this instruction typically follows a pcall and is used to support reactivation of parallel execution during outside backtracking;

3. `pop_wait`: this instruction is used to allow the creator of the parallel call to locally execute some of the subgoals of the CGE;

4. `hook`: this instruction represents the "join" of the parallel call. It is executed at the end of a subgoal and allows the and-agent to update the status of the subgoal, allocate the end marker and switch to a different execution.

5. `sch`: this instruction represents the entry-point to the and-scheduler.

The `pcall` and `pop_wait` instructions are inherited from RAP-WAM [8], while the others have been added specifically to support ACE features.

3.3 Scheduling

The and-scheduler is quite simple in its structure. The scheduling algorithm is activated by an idle and-agent and it looks for and-parallel work as follows:

1. if any subgoal is available on the local goal stack, then it will be preferred to any remote work;

2. otherwise other processors are scanned. Priority is given to the processor that signals the highest amount of and-parallel work available.

No restrictions on the subgoal to be selected are imposed. In the future we plan to extend the scheduling algorithm in order to choose work in a "better" way—e.g. selecting subgoals that will allow an 'easier' backtracking and a reduced intermixing of executions.

3.4 System Activity

This section briefly illustrates the steps followed by &ACE during a typical execution.

3.4.1 Forward Execution

As long as CGEs are not encountered (i.e. `pcall` instruction is not executed) forward execution is exactly the same as in SICStus WAM. When a `pcall` is met, a new parcall frame is allocated, initialized, and all the subgoals but the leftmost one are loaded on the goal stack (the leftmost subgoal is directly executed by the same and-agent that created the parcall frame). Once the lock on the goal stack has been released, all the subgoals become available for parallel execution. At the end of the execution of the leftmost subgoal, the and-agent which started the parcall execution will perform a `pop_wait`, executing locally any further subgoal of the same parcall still available on the local goal stack. Whenever a subgoal is picked up for execution from a goal stack, an input marker is allocated and execution is started. At the end of the execution an end marker is allocated and the information in the parcall frame is appropriately updated. The last and-agent reporting termination of a subgoal of a parcall frame automatically becomes in charge of carrying on the continuation of the CGE. This clearly means that the agent that will complete the execution may be different from the one which started it. An and-agent completing a subgoal while there are still other subgoals running for the same parcall will automatically enter and-scheduling and start searching for new work. Subgoals executed by the various and-agents are uniquely time-stamped (by using a global counter shared between the various and-agents).

3.4.2 Backward Execution

In addition to being able to perform the standard sequential backtracking, backward execution involves:

- being able to perform inside and outside backtracking, as mentioned in section 2.3;
- being able to perform killing of a computation;
- being able to perform communication between and-agents, in order to report situations in which a subgoal should be killed or backtracked over.

Backtracking is performed in the usual way, by moving downwards in the choice point stack and analyzing the data structures encountered. In the case of &ACE this moving downwards is performed following the logical path (not the physical one). The following situations may occur:

1. a *choice point* is encountered: in this case a new alternative is taken and explored. If the new alternative is the last one, the choice point is removed; furthermore if the current subgoal is trapped (i.e. the current point on the logical path does not match the current top of the physical choice point stack), and the new alternative is the last one of the choice point, then a trail marker needs to be allocated to keep track of the new section of trail that we are opening.

2. an *end marker* is encountered: we are entering an outside backtracking situation. If the subgoal below the end marker is the rightmost of the CGE then the end marker is skipped and we keep going down on the same logical path. Otherwise the rightmost subgoal of the CGE is detected and backtracking is transferred to it.

3. an *input marker* is encountered: if the parcall frame referred by the input marker is still in inside status (i.e. we are still looking for the first solution to the CGE) then kill signals are propagated to the other subgoals of the CGE. Immediately after this the agent quits backtracking and moves to the scheduling phase (the agent which created the parcall will carry on the backtracking). If the parcall frame is in outside mode, then reaching an input marker simply means that we have completely explored a subgoal without finding further solutions—backtracking is transferred on the subgoal on its left (using a redo signal to the agent which completed such subgoal).

4. a *parcall frame* is encountered: the situation is exactly as in the previous case (encountering an input marker) since the parcall frame is used as "input marker" for the leftmost subgoal of the CGE.

4 Advanced Issues and Optimizations

4.1 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[4]. We decided to disallow this for the following reasons:

- this would make the integration with Or-parallelism unnecessarily complex;

- the use of explicit messages makes the current status of the execution more "evident", which allows an or-agent looking for work to decide which kind of actions are to be performed in order to appropriately install the stolen alternative;

- we desire to keep the overall design simple and clean;

- it is not clear whether this would provide a real advantage, since locking and/or synchronization may become necessary on certain areas of the stack.

Messages are sent and received asynchronously (i.e. a message can be received and served some time after it has been sent but the sender is free to continue its execution immediately after sending the message). The frequency at which an agent checks for the presence of messages can be tuned up by modifying a system defined constant. The examples presented in the last section of this report have been executed allowing the maximum delay between successive message checks (i.e. checks for the presence of messages have been performed only during critical phases of the execution, like opening of new parallel calls, termination of a branch, etc.).

4.1.1 Redo Signals

A redo signal is generated exclusively during outside backtracking and it is used to transfer the current backtracking activity to a subgoal located on the stack of a different agent. To be more precise, the message is sent to the agent which completed the execution of the subgoal on which backtracking has to be performed. Once received, the agent will locate the end marker relative to that subgoal, remove it and start backtracking from that point (following the logical path relative to such subgoal). Figure 9 illustrates this process.

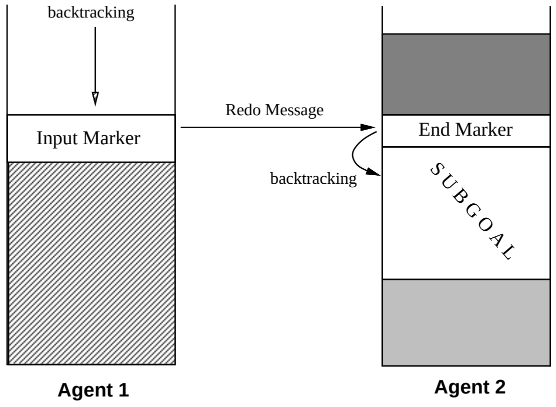

Figure 9: Transferring backtracking activity using Redo Messages

4.1.2 Kill Signals

The management of kill signals is more involved than redo signals. A kill can be generated in two occasions:

1. during inside backtracking a kill message for each subgoal of the CGE to be removed;

2. during outside backtracking a kill message is (or, should be[5]) sent to those subgoals which have shown a deterministic behaviour—since they don't have alternatives there is no sense in trying to search for other solutions.

The second case is the simpler one to handle, since we are guaranteed that the branch that we are killing has already been completed. The message can be directed to the agent which completed the computation, which in turn will take care of removing the computation and unwind the relevant part of the trail. The first case is more complex. A kill can be generated for computations that are still active and we may desire to have it propagated as fast as possible (a kill prunes computation that have turned speculative). Clearly, if the branch to be killed has already been completed there are no problems, since we have a pointer to the end of the computation and this offers a starting point to unroll and unwind the computation. It is, in fact, important to observe that the organization of

[4]in some situation this kind of activity is performed in &ACE

[5]see section 4.2

the computation in a WAM-like architecture allows traversing the computation tree from the bottom to the top (by following the logical paths of the various subgoals) but not vice-versa—we do not have any way of traversing the tree going from the top towards the leaves.

Two approaches [11] have been considered:

1. a *lazy* approach, in which the killing of a branch is delayed until certain checking points are reached (like termination of a branch). Clearly this approach is far from optimal, but it is easy to implement and limits the amount of overhead.

2. a *direct* approach in which the kill signal is progressively propagated inside the branch towards the lower levels of the tree. This is approach guarantees a faster propagation of the killing, but

   (a) requires a minimal support to allow moving top-down in the tree (like pointers linking nested parcall frames);

   (b) requires locking of the parcall frames to allow a correct propagation;

   (c) requires a level of synchronization during the kill action.

The current implementation makes use of the lazy approach (slightly optimized to allow a more frequent detection of kills). We are planning to introduce the direct method in near future, since it will offer better performance. The kind of approach adopted for handling kills seems to affect the interaction with the Or-parallel components of ACE only marginally. A complete treatment of the more advanced algorithm for killing can be found in [11].

4.2 Shallow Parallelism

Innumerable optimizations can be done to a recomputation-based and-parallel system like &ACE. In this framework we have avoided many of them trying to keep the design of the and-parallel engine sufficiently simple, necessary condition for the later introduction of the or-parallel features.

One optimization that has been implemented in the current version of the system deals with taking advantage of deterministic computations. Many of the classical benchmarks proposed for and-parallelism 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 of keeping 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 the shallow backtracking technique); if we reach the end of the computation without allocating any input marker, then the end marker itself is not allocated and we simply record the boundaries of the current trail section in the slot of the parcall frame relative to the current subgoal. On backtracking no kill message needs to be generated for this kind of subgoals (we just need to unwind the trail section indicated in the slot of the parcall frame). This simple optimization allows to save time and space since

- some input and end markers will not be allocated at all on the stack;
- allocation and initialization of data structures on the choice point stack is expensive;
- during backtracking certain kill messages will be avoided.

Figure 10 shows the results of adopting the optimization on the Takeuchi benchmark. Even on a relative small computation time the difference between the unoptimized and the optimized version is significant (on a single processor we have observed an improvement of around 25% for certain benchmarks).

5 Performance Results

The purpose of this section is to present the results obtained by executing some well-known benchmarks on &ACE. The results for the following benchmarks are initially reported:

Matrix Multiplication: a classical program for multiplying a $N \times N$ matrix by its transposed matrix;

Quicksort: quicksort of a $N \times 50$ elements vector;

Takeuchi: a well-known benchmark developed at ICOT;

Hanoi: a tower of hanoi solver collecting the moves in a list;

Boyer: a reduced version of the Boyer-Moore theorem prover.

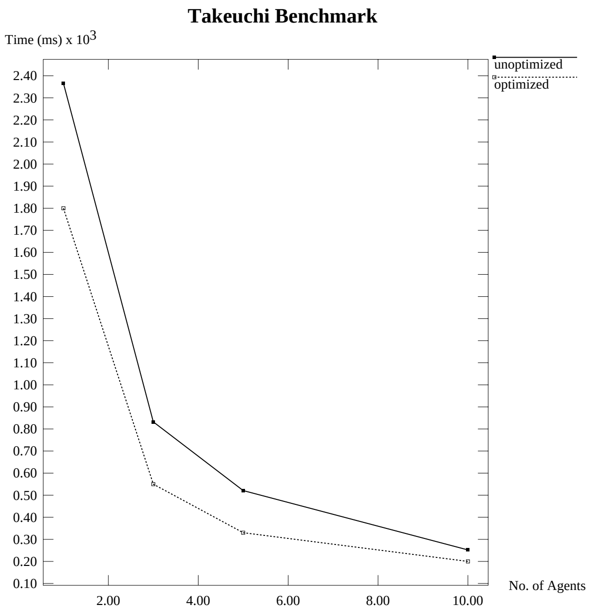

Figure 10: Improvement on execution time using the determinate execution optimization

Compiler: The Aquarius Prolog compiler (approximately 2,200 lines of Prolog code).

Poccur: A list processing program.

BT_cluster: A clustering program from British Telecom.

Annotator: The &-Prolog annotator (1,000 lines).

Table 1 indicates, for each benchmark, the execution time (in ms.) and the speed-up obtained. The execution times are given in the format α/β, where α is the time obtained without the shallow-parallelism optimization and β is the time obtained using the optimized version. The speed-up figures are with respect to the optimized execution.

Table 1 illustrate the speedups obtained for the various benchmarks. The figures clearly indicates 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 (like matrix multiplication and hanoi). The fact that certain benchmarks shows higher speedups is due to the amount of and-parallel work available in the benchmark and the granularity of the and-parallel work exploited.

Speedups for some benchmarks are shown in Table 1 and plotted in Figure 11. The largest benchmark is the Aquarius Prolog Compiler (approximately 2,200 lines of Prolog code). Note that for the *compiler*, *quicksort*, and *boyer* benchmarks, the speedup curve flattens out because at some point all available parallelism is exhausted.(e.g., in the case of compiler, the speed-up was measured while the compiler was compiling a program of 8 clauses, thus the maximum speedup that can be obtained will be 8; if a larger program was compiled higher speedups will be obtained). Our implementation incurs an average parallel overhead of about 10% over Sicstus Prolog. Some of this parallel overhead is avoided by triggering optimizations mentioned earlier that are based on recognizing determinacy of goals. These optimizations yield, depending on the program, an improvement of 5% to 25% over the unoptimized version. Some improvement data is shown in Table 1 (each entry in Table 1 shows the time in millisecond before the optimization and after the optimization; the number in parenthesis gives the speed-up obtained; for compiler benchmark the unoptimized figure is not shown). As is obvious, improvements due to optimization can be substantial; in some cases superlinear speedup is obtained.

6 Conclusions

This report describes some of the most 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 machine and the organization of the execution, putting emphasis on new ideas and optimization introduced in the design. The results obtained show good performance and good speedups. The results, if compared to those reported for &-Prolog [7], present some slow-downs which are due to the

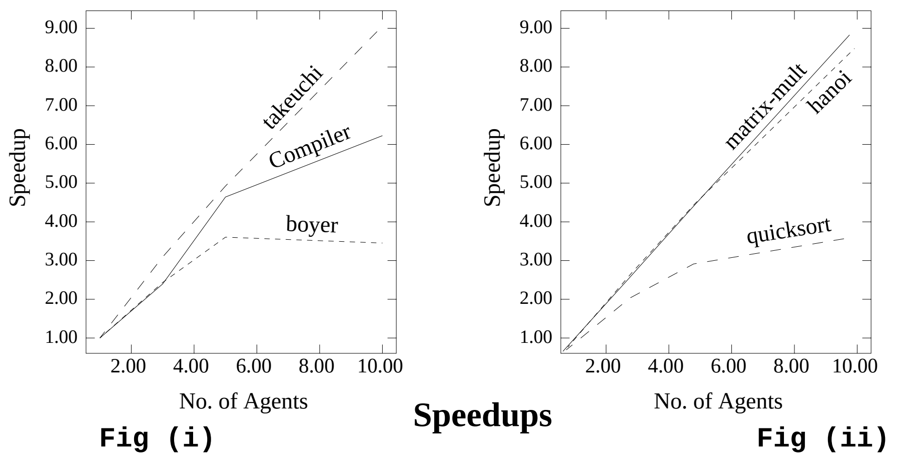

Figure 11: Speedups Curves for Selected Benchmarks

Goals executed	&ACE agents 1	3	5	10
matrix_mult(30)	5598/5214	1954/1768 (2.95)	1145/1059 (4.92)	573/534 (9.76)
quick_sort(10)	1882/1536	778/632 (2.43)	548/455 (3.38)	442/373 (4.12)
takeuchi(14)	2366/1811	832/586 (3.09)	521/368 (4.92)	252/200 (9.06)
hanoi(11)	2183/1671	766/550 (3.04)	471/336 (4.97)	231/180 (9.28)
boyer(0)	9655/9290	5329/3829 (2.43)	3816/3199 (2.90)	2887/2687 (3.46)
compiler	—/29902	—/12522 (2.39)	—/6437 (4.65)	—/4801 (6.23)
poccur(5)	3651/3197	1255/1079 (2.96)	759/662 (4.83)	430/371 (8.62)
bt_cluster	1461/1343	528/480 (2.8)	345/312 (4.30)	/189 (7.11)
annotator(5)	1615/1422	556/475 (2.99)	392/322 (4.42)	213/187 (7.60)

Table 1: Unoptimized/Optimized Execution times in msec (Speedups are shown in parenthesis)

fact that &ACE contains a complete implementation of signal management and full backtracking[6]. However, with various optimizations &ACE performance is the same as or better than &-Prolog's (note, however, that these optimizations, such as shallow parallelism, can also be incorporated in &-Prolog). Of course, not all benchmarks that can be run on &ACE can be run on &-Prolog because of absence of a complete implementation of backtracking in the latter. The benchmarks we have chosen above do not involve outside backtracking and can be run on both systems.

[6] The main reason why &-Prolog did not implement backtracking is that its designers hoped that they will be able to optimize away the non-determinism present in parallel conjunctions at compile-time. However, because ACE exploits or-parallelism inside and-parallel goals, in absence of enough processing resources it should be able to try other alternatives inside and-parallel goals via backtracking. Hence support for backtracking over parallel conjunctions is essential for ACE.

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] K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1991. 19(6):445–475.

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

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

[5] 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.

[6] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. in *1994 Int'l Conf. on Logic Progr.*, MIT Press, June 1994.

[7] 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.

[8] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[9] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming.* MIT Press, June 1991.

[10] Swedish Institute of Computer Science. Industrial Sicstus Prolog Internals Manual, 1989.

[11] T. DongXing, E. Pontelli, G. Gupta, and M. Carro. Last Parallel Call Optimization and Fast Backtracking in And-parallel Logic Programming Systems Technical Report NMSU-CSTR-9403, Dept. of Computer Science, New Mexico State University, March 1994.