ACE: <u>A</u>nd/Or-parallel <u>C</u>opying-based <u>E</u>xecution of Logic Programs

Gopal Gupta[†], **Manuel Hermenegildo**[‡], **Enrico Pontelli**[†] and **Vítor Santos Costa**[§]

Abstract

In this paper we present a novel execution model for parallel implementation of logic programs which is capable of exploiting both independent and-parallelism and or-parallelism in an efficient way. This model extends the stack copying approach, which has been successfully applied in the Muse system to implement or-parallelism, by integrating it with proven techniques used to support independent and-parallelism. We show how all solutions to non-deterministic and-parallel goals are found without repetitions. This is done through recomputation as in Prolog (and in various and-parallel systems, like &-Prolog and DDAS), i.e., solutions of and-parallel goals are not *shared*. We propose a scheme for the efficient management of the address space in a way that is compatible with the apparently incompatible requirements of both and- and or-parallelism. We also show how the full Prolog language, with all its extra-logical features, can be supported in our and-or parallel system so that its sequential semantics is preserved. The resulting system retains the advantages of both purely or-parallel systems as well as purely and-parallel systems. The stack copying scheme together with our proposed memory management scheme can also be used to implement models that combine dependent and-parallelism and or-parallelism, such as Andorra and Prometheus.

1 Introduction

Recently, stack copying has been demonstrated to be a very successful alternative for representing multiple environments in or-parallel execution of logic programs. In this approach, stack frames are explicitly copied from the stack(s) of one processor[1] to that of another whenever the latter processor needs to share a branch of the or-parallel tree of the former. In practice, by having an identical logical address space for all processors and allocating the stack(s) of each processor in identical locations of this address space, the copying of stack frames can be reduced to copying large contiguous blocks of memory from the address space of one processor to that of the other—an operation which most current architectures perform quite efficiently—without requiring any sort of pointer relocation. The chief advantage of the stack copying approach is that program execution in a single processor is exactly the same as in a sequential system. This considerably simplifies the building of parallel systems from existing sequential systems, as was shown by MUSE [2, 1] which was built using the sequential SICStus Prolog System.

Similar arguments can also be made for the design of independent and-parallel systems based on *program annotation* (i.e. using Conditional Graph Expressions) and *recomputation of subgoals* [7] (i.e. non-deterministic and-parallel subgoals are recomputed and not shared), as proven by the experiences of &-Prolog [16] and DDAS [23]. Briefly, a program annotated for independent and-parallelism contains expressions of the form

$\cdots, (< conditions > \Rightarrow\ literal_1 \& \cdots \& literal_n), \cdots$

where $literal_1$, ..., $literal_n$ will be executed in (and-) parallel only if the $< conditions >$ are satisfied.

A long standing goal of parallel logic programming systems designers has been to obtain more general systems by combining different forms of parallelism into a single framework. In particular, one would expect that independent and-parallelism and or-parallelism, that have been exploited so well in Prolog, could naturally be exploited together. In fact, this is a hard problem, as the difficulties (e.g. supporting full Prolog) faced by several previous proposals [14, 26, 22] do show. Recently an abstract model, called the Composition Tree [12], has been designed that allows efficient realization of systems that combine both forms of parallelism while supporting full Prolog. In this paper we design a novel model, a realization of the C-tree, exploiting or- and independent and-parallelism, which subsumes both the stack copying approach (for or-parallelism) and the subgoal recomputation approach (for and-parallelism).

The resulting and-or parallel system, called ACE, is in the same category as PEPSys [26], ROPM [22] and the AO-WAM System [14]. However, our system is arguably better than the above systems in many re-

[†]Laboratory for Logic, Database, and Advanced Programming, Dept. of Computer Science, New Mexico State University, Las Cruces, NM, USA.

[‡]Facultad de Informática, U. Madrid (UPM), Madrid - Spain.

[§]Dept. of Computer Science, University of Bristol, Bristol, UK.

[1]Throughout the paper we will often refer to the "stack" of a "processor" meaning the *memory areas* that a *computing agent* is using.

spects, the chief ones being ease of implementation, sequential efficiency, and better support for the full Prolog language, in particular being able to incorporate side-effects in a more elegant way. These advantages are due to several factors. One of them is that ACE *recomputes* independent and-parallel goals, rather than *sharing* their solutions (solution sharing was adopted in all the previously proposed models [14, 26, 22]). Recomputation means that, given a goal `a(X) & b(Y)`, where the two subgoals `a` and `b` are independent, the solutions for the subgoal `b` will be recomputed for every solution found for subgoal `a`. Recomputation has important advantages (they are are discussed at length in [12, 25]), and was fundamental in the design of the C-Tree model. In [12] we presented the C-tree, along with a few preliminary ideas on how to realize the C-tree using an environment representation technique based on stack copying as well as binding arrays. In this paper we show how the complete independent and- and or-parallel system based on C-Tree can be constructed using stack-copying.

ACE subsumes both MUSE [2] and &-Prolog [16] in terms of execution behaviour. One of our aims is to have ACE subsume performance characteristics of MUSE and &-Prolog as well, namely, their low parallel overhead, their considerable speedups for interesting applications, and their support for the full Prolog language. To accomplish this we need to carefully address the many issues that arise in combining both forms of parallelism. These issues are:

- Synchronization between independent and-work and or-work: that is, deciding when should the alternatives created by goals working in independent and-parallel be made available for or-parallel processing. In ACE we lay down a set of *sharing requirements* that a choicepoint should satisfy before a processor can pick an or-parallel alternative from it.

- Memory management: for or-parallel execution in MUSE the stacks of one processor *should not be* visible to the other processors (except during copying), while in independent and-parallel execution in &-Prolog the stacks of one processor *should be* visible to all other processors. In ACE processors are organized into *teams* to get around these conflicting requirements for or- and independent and-parallelism respectively.

- Scheduling: ACE can use the existing schedulers of MUSE and &-Prolog for scheduling or- and independent and-parallelism respectively. However, in addition, it should also balance the amount of resources allocated for exploiting or- and independent and-parallelism.

- Efficient implementation of copying: while MUSE copies stacks of a single processor at a time, ACE needs to copy stacks of multiple processors. Therefore, developing optimized copying techniques is even more fundamental for ACE.

- Implementation of Prolog's extra-logical features (such as cuts and side-effects): Both MUSE and &-Prolog have developed techniques for supporting full Prolog. In ACE we need to extend these techniques to support sequential Prolog semantics in presence of both or- and independent and-parallelism. Here we can benefit from the principles designed for the C-tree abstraction [10].

In designing the solutions to these problems, our aim is to obtain a full Parallel Prolog system that will have low sequential overheads and good parallel speedups. Also, we try to follow the techniques that have been used for &-Prolog and MUSE as much as possible, as they have proven to be effective in practice.

Our perspective so far of ACE is as concretizing the C-tree framework for combining independent and- and or-parallelism using stack copying. Once ACE is fully described, it will be apparent that ACE can be seen in quite a different perspective. In this new perspective, ACE generalizes the principle of copying, from the copying used in MUSE to obtain or-parallelism between sequential computations, to *copying to obtain or-parallelism between and-parallel computations*. In the paper we show that this principle, *Generalized Copying*, not only gives a way to understand ACE, but it applies, and should be useful, to combine or-parallelism with many forms of and-parallelism, such as parallelism between determinate and-goals as exploited in Basic Andorra [6], or with dependent and independent and-parallelism as exploited in DDAS [23].

The paper is organized as follows. We first present the ACE model. Although the C-tree abstraction is implicit to our reasoning, it is not needed for understanding the rest of the paper. We use the stack copying approach to give a more intuitive feel for the model. We then enter the more specific problems of memory management, and how copying can be implemented between stacks sets. We give a brief overview of the new scheduling problems that arise, and present and discuss two schemes for the optimization of copying in ACE. We also propose a scheme to support cut and side-effects in ACE. We finally discuss the effectiveness of ACE and show how our scheme can be generalized to dependent and-parallel systems. Throughout the paper, we assume some familiarity with the implementation of Prolog, &-Prolog, and MUSE.

Like in &-Prolog, we assume that programs are annotated to express the and-parallelism using basic Conditional Graph Expressions (CGEs) before execution commences. The &-Prolog parallelization tools [20] will be used to automatically generate such annotations from standard Prolog code. Alternatively, programs can always be annotated by the user.[2]

2 The Stack Copying Approach to And-Or Parallelism

In ACE, the multiple environments that are needed to implement or-parallelism are supported through explicit stack copying. We first summarize the stack copying approach (as used by the MUSE system). In a stack-copying or-parallel system several processors explore different alternatives in the search tree independently (modulo side-effect synchronization). The execution of each processor is identical to a sequential Prolog execution. Whenever a processor P1 exhausts its branch and wants to share work with another processor P2 it selects an untried alternative from a choice point `cp` in P2's stack. It then copies the entire stack of P2, backtracks up to the choice point `cp` in order to undo all the conditional bindings made below `cp`, and starts executing one of the untried alternatives. In this approach, provided there is a mechanism for copying stacks, the only cells that need to be shared during execution are those corresponding to the choice points. Shared choice points are thus copied from P2's private memory to shared memory where they can be accessed from both P1's and P2's private memory via pointers[3] (these choice points are said to have been made *public*, following MUSE's terminology).

If we consider the presence of and-parallelism in addition to or-parallelism, then, depending on the actual types of parallelism appearing in the program and the nesting relation between them, a number of cases can be distinguished. The simplest two cases are of course those where the execution is purely or-parallel or purely and-parallel. Trivially, in these situations standard or-parallel and independent and-parallel execution respectively apply, modulo the memory management issues, which will be dealt with later. We next discuss the interesting cases where both forms of parallelism are present in the computation.

2.1 And "Under" Or

And "under" or refers to cases where or-parallelism present inside and-parallel goals is not exploited [25]. Thus, only alternatives in those choice points that are not nested inside any CGE, i.e. not created during processing of and-parallel goals, are made available for or-parallel processing. The two cases are, first the case in which the goal that gives rise to or-parallelism is not preceded by any CGE; and second, the case in which this goal is in the continuation (but not inside) of some CGE.

The first case is illustrated in Figure 1. Consider the tree, shown in Figure 1.(i), that is generated as a result of executing a query `a` which during its execution calls a clause containing a CGE `(true => b(X) & c(Y))`. In Figure 1.(i) processor P1 has started execution of goal `a`, left an untried alternative ("embryonic branch") `a2`, and then entered the CGE. And-parallel execution can remain identical to the standard subgoal recomputation approach (like in &-Prolog), hence a processor P2 can simply pick up the execution of goal `c`. Or-parallel execution can also remain identical to pure stack-copying. If processor P3 wants to pick up the `a2` branch left behind by P1, it can simply copy the portion of the tree from the root to the embryonic node, and continue with the untried alternative (Figure 1.(ii)). This resembles a standard stack-copying execution (as in MUSE).

Figure 1.(iii) and Figure 1.(iv) present the second case, when a processor selects an untried alternative from a choice point created during execution of a goal g_j in the body of a goal which occurs after a CGE. In other words, there has been and-parallelism above the selected alternative, but all the and-tasks are finished. A processor selecting such an alternative will have to copy all the stack portions in the branch from the root to the CGE, the portions of stacks corresponding to all the and-tasks inside the CGE, and those of the goals between the end of the CGE and g_j. All these portions have in principle to be copied because the untried alternative may need access to variables in all of them. In Figure 1.(iii), processor P1 started execution of the goal creating a CGE `(b & c)`, and fully executes `b`. Processor P2 executed the goal `c` in and-parallel. Both have finished execution of the CGE (leaving no choice points behind) and then processor P1 has taken the continuation `d` and left an untried alternative `d2`. This alternative can be picked up by another processor P3. The processor P3 has therefore to copy the portion of the tree from the root to the CGE, the portions inside the CGE, and the portion of the continuation up to the embryonic node. The processor P3 can then start execution of the `d2` alternative (Figure 1.(iv)).

2.2 Or "Under" And

In "Or Under And" the untried alternatives of choice points created within and-parallel goals in CGEs are also made available for or-parallel processing. One could simplify, and disallow or-parallel processing of such alternatives, trying them sequentially via backtracking instead, but there is experimental evidence that a considerable amount of or-parallelism may be lost [25]. Therefore, ACE does support or-under-and parallelism. When an alternative created within and-parallel goals in a CGE is selected, one

[2] In &-Prolog unrestricted dependency graphs can be expressed (i.e. more general than those possible with CGEs), by combining the "&" operator and synchronization builtins. However, since such graphs can be handled in a similar way to that given in the description that follows, the discussion will be limited for simplicity and without loss of generality to CGEs.

[3] To be precise, shared choice points are not copied but a *record* representing the choice point is created in the shared area.

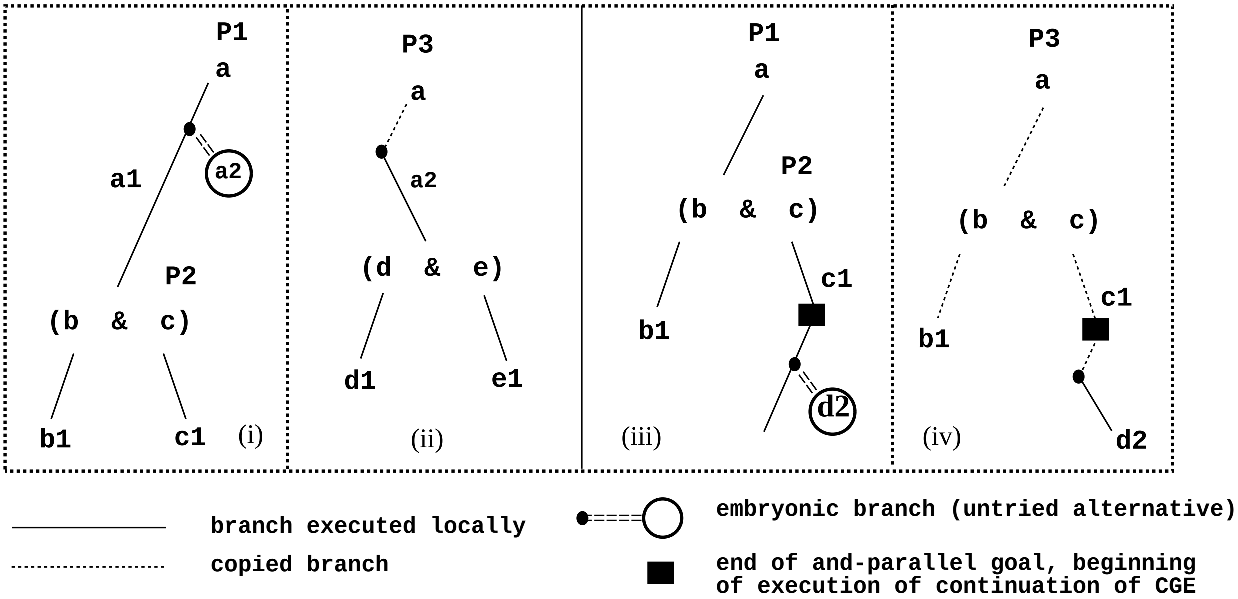

Figure 1: And Under Or

needs to carefully decide which portions of the stacks to copy. Our guiding principle is the following: copy all branches that would be copied in an equivalent or-parallel (MUSE in this case) execution, and recompute all those branches that would be recomputed in an equivalent pure and-parallel computation. As far as the and-parallel execution is concerned, we want to be as close as possible to the recomputation approach hence implementing the PWAM "point backtracking" strategy [19] used in &-Prolog. As we will see, our strategy results in copying only parts that &-Prolog reuses during backtracking and recomputing those that &-Prolog (and also MUSE and Prolog) recomputes.

Consider a CGE ($true => g_1 \& \ldots g_i \ldots \& g_n$) that is encountered during execution, and whose goal g_i has an untried alternative in one of the choice points in its search tree. Assume a processor picks up this untried alternative for or-parallel processing. Then this processor will have to copy all the stack portions in the branch from the root to the CGE including the *CGE descriptor*[4] (called *C-node* in [12] and *parcall frame* in &-Prolog [18]). It will also have to copy the stack portions corresponding to the goals $g_1 \ldots g_{i-1}$ (i.e. goals to the left of g_i). The stack portions up to the CGE need to be copied because each different alternative within g_i might produce a different binding for a variable, `X`, defined in an ancestor goal of the CGE. The stack portions corresponding to goals g_1 through g_{i-1} have to be copied because execution of the goals following the CGE may need to access some of the bindings generated by the goals $g_1 \ldots g_{i-1}$. The stack portions corresponding to the goals $g_{i+1} \ldots g_n$ *need not be copied*, because these goals would be recomputed. The issue is further illustrated with a simple example.

[4] The CGE descriptor records the control information for the CGE and its independent and-parallel goals for exploiting and-parallelism.

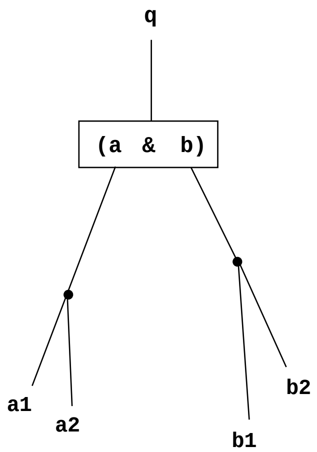

Figure 2: Execution tree with alternatives inside the CGE

Figure 2 shows the and-or tree for the query `q` containing a CGE `(true => a(X) & b(Y))`, each of whose goals leads to two solutions. For sake of simplicity, we have only shown the path from root of the tree to the CGE.

Execution in ACE begins with processor P1 executing the top level query `q`. When P1 encounters the

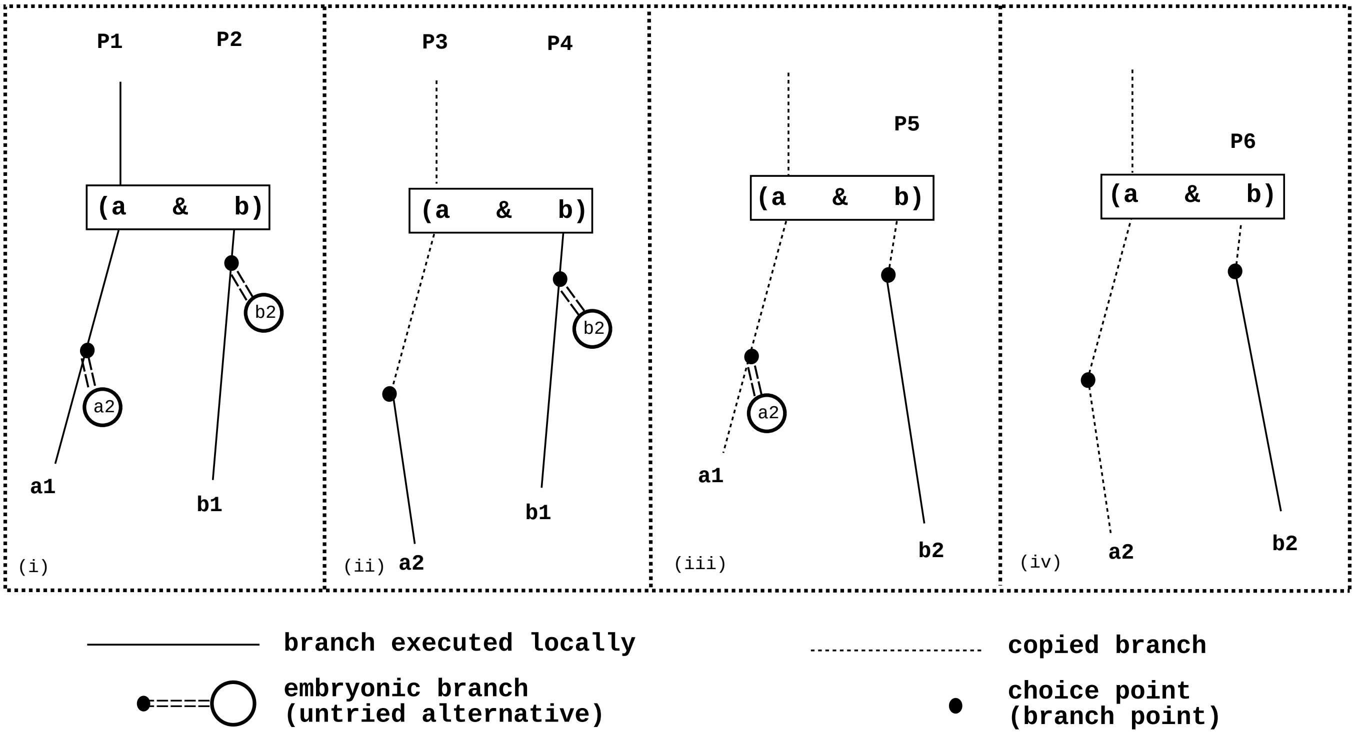

Figure 3: Or Under And

CGE, it picks the subgoal `a` for execution, leaving `b` for some other processor. Let us assume that processor P2 picks up goal `b` for execution (Figure 3.(i)). As execution continues P1 finds solution `a1` for `a`, generating a choice point along the way. Likewise, P2 finds solution `b1` for `b`.

Since we allow for full or-parallelism within and-parallel goals, a processor can steal the untried alternative in the choice point created during execution of `a` by P1. Let us assume that processor P3 steals this alternative, and sets itself up for executing it (before P3 can steal the alternative, P1 has to move the choicepoint into the shared area). To begin execution of this untried alternative P3 copies the stack of processor P1 (Figure 3.(ii) shows this process; see index at the bottom of Figure 3 for explanation of the symbols). P3 then simulates failure to remove conditional bindings made below the choice point, and restarts the goals to its right (i.e. the goal `b`). Processor P4 picks up the restarted goal `b` and finds a solution, `b1`, for it. In the meantime, P3 finds the solution `a2` for `a` (see Figure 3.(ii)).

Note that before P3 can commence with the execution of the untried alternative and P4 can execute the restarted goal `b`, they have to make sure that any conditional bindings made by P1 after the selected choice point, as in Muse, and that any bindings made by P2 while executing `b` have been cleared. The former can be implemented by either (i) P4 copying `b` from P2 and completely backtracking over it[5]; or, (ii) P3 (or P4) getting a copy of the trail stack of P2 and resetting all the variables that appear in it (see later).

At this point, two copies of `b` are being executed in or-parallel, one for each solution of `a`. Note that the process of finding the solution `b1` for `b` leaves a choice point behind. The untried alternative in this choice point can be picked up for execution by another processor. This is indeed what is done by processors P5 and P6 for each copy of `b` that is executing. These processors copy the stack of P2 and P4 respectively, up to the choice point. The stack portions corresponding to goal `a` are also copied (Figures 3.(iii), 3.(iv)) from processors P1 and P3, respectively. The processors P5 and P6 then proceed to find the solution `b2` for `b`.

Note that if there were no processors available to steal the alternative (corresponding to solution `b2`) from `b` then this solution would have been found by processors P2 and P4 (in their respective copies of `b`) through backtracking as in &-Prolog. The same would apply if no processors were available to steal the al-

[5] An optimization could be that P4 choses not to backtrack over `b` or recompute it again, rather P4 simply copies `b` and reuses it. This optimization is only valid if `b` has not yet generated a solution (or at least, execution of the continuation of the CGE, which may bind variables conditionally in `b`, should not have begun). Some problems may also arise with extra-logical predicates in `b`, and in general only the part before such an extra-logical predicate can be copied into P4.

ternative from `a` corresponding to solution `a2`.

In the above example, all other operations that are performed during and-parallel execution remain the same as in &-Prolog. Thus, execution of the continuation of the CGE can begin only after at least one solution has been found for all goals in the CGE. Also, backtracking in the CGE takes place just as in &-Prolog, i.e. goals to the right should be completely explored before a processor can backtrack inside a goal to the left.

We place a restriction (called the *sharing requirement*) on choice points inside a CGE that can be made available for or-parallel processing: given a goal g_i in a CGE, choice points arising in it can be made available for or-parallel processing only if the goals to the left of g_i in that CGE have reached a solution. If the CGE containing g_i is nested inside another CGE, then all goals to the left of the goal leading to the inner CGE should also have found a solution, and so on. Thus, in the example above (Fig. 3(i)), the alternative `b2` of `b` cannot be picked up by any team[6] until the solution `a1` has been found. The sharing requirement serves two purposes:

(i) as far as or-parallel scheduling is concerned it keeps us very close to the scheduling strategy employed by MUSE;

(ii) it avoids (a form of) speculative or-parallelism, because if the goal to the left (for which a solution had not been found yet) failed, the work would have gone wasted.

We could go one step further, and stipulate that choice points inside CGEs will be made available only if *all* goals in the CGE have found at least one solution. Although this will keep us closer to &-Prolog and enable us to do a limited form of intelligent backtracking (the kind that is also present in &-Prolog), this will overly restrict the amount of or-parallelism. So this restriction is not adopted, although its lack might result in extra work in some situations. For instance, in the example above (Figure 3), if `b` were a failing goal, i.e. a goal without any solutions, then trying multiple alternatives of `a` in or-parallel would result in wasted work: `b`'s failure would be discovered multiple times since `b` is recomputed for every alternative of `a`.

3 Memory Management in ACE

One of the main features of stack-copying based or-parallel systems which greatly facilitates stack copying is that each processor has an identical logical memory address space. This enables one processor to copy (part of) the stack of another without relocating any pointer. In the presence of and-parallelism this feature may be hard to ensure, as each goal in a CGE may be executed in and-parallel by a different processor. In other words, as far as and-parallel execution is concerned, all the participating processors should work on separate segments of a common address space, whereas for or-parallel execution each processor should have an identical but independent logical address space (so that stack portions can be copied without any pointer relocation). Thus, the requirements for or- and and-parallelism seem to be antithetical to each other. The problem can be resolved by dividing all the available processors into *teams* such that and-parallel work can only be shared between processors of the same team, and or-parallel work can only be shared between teams. All processors in a team thus share the same logical address space, but each team has its own independent logical address space (which must be identical to the address space of all other teams to allow copying without any pointer relocation).

To implement and-parallelism, the address space of each team is divided up into k *memory segments* (as happens in &-Prolog), where k is the maximum number of processors allowed in any given team. The memory segments are numbered from 1 to k. Each processor of the team allocates its stack set (heap, local stacks, trail etc.) in one of the segments. The sizes of the k different memory segments in the address space of a team are not required to be the same. However, once one team's address space has been divided into segments using some scheme for division, the address spaces of all other teams should be divided into segments in an identical way, so that during copying of stacks no pointer relocation is needed[7] (Figure 4).

Processors belonging to other teams are allowed to join a different team as long as there is a memory segment available for them to allocate their stacks in the new team's address space.

Consider the simple scenario where a choice point, belonging to a team T1 and *outside the scope of any CGE*, is picked by a team T2. Let i be the memory segment number in T1 in which this choice point lies. For simplicity, we assume that the root of the Prolog execution tree also lies in memory segment i. T2 will thus copy the stack from the ith memory segment of T1 into its own ith memory segment. Since the logical address space of each team is identical and is divided into identical segments, no pointer relocation is needed. Failure is then simulated and the execution of the untried alternative of the stolen choice point begun.

Now consider the more interesting scenario where a choice point, created by a team T1 and which lies within the scope of a CGE, is picked up by a processor in a team T2. Let this CGE be $(true \Rightarrow g_1 \& \ldots \& g_n)$ and let g_i be the goal in the CGE whose sub-tree con-

[6] The idea of viewing a set of workers as a *team* will be analyzed in details later on.

[7] This constraint may be relaxed quite a bit, since identical division of address space needs to be done only for those teams that will *share computation*, and then only for the parts that are shared.

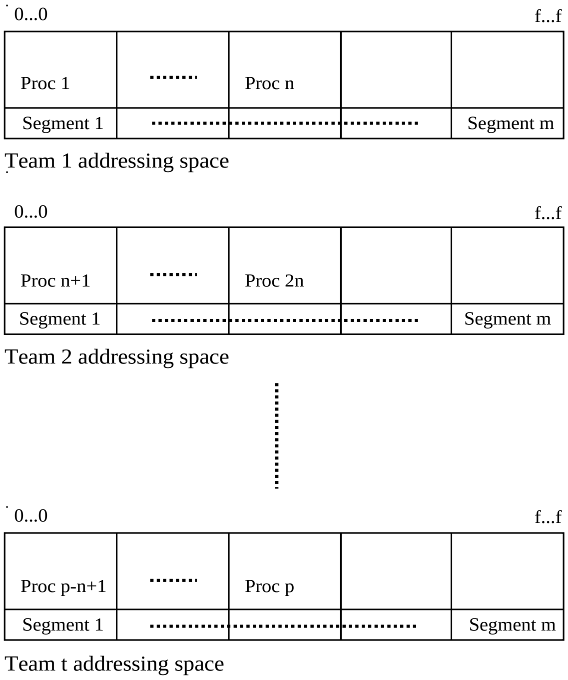

Figure 4: Address Space in Muse

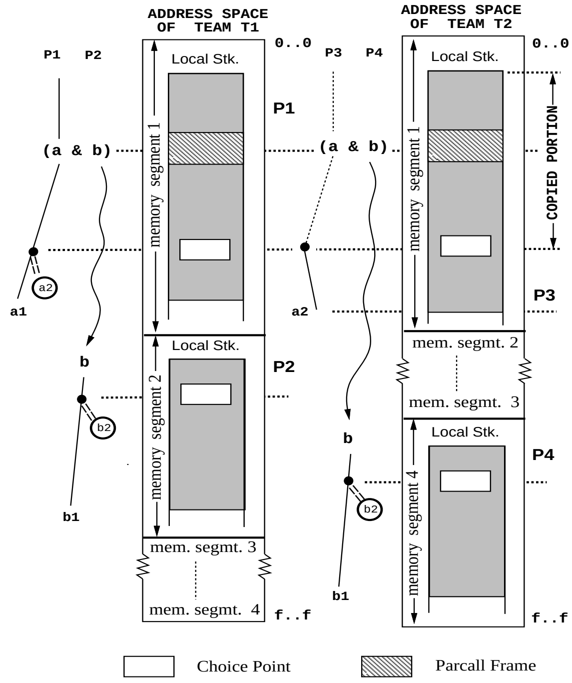

Figure 5: Illustration of Stack Copying

tains the stolen choice point. T2 needs to copy the *stack segments* corresponding to the computation from the root up to the CGE and the *stack segments* corresponding to the goals g_1 through g_i. Let us assume these stack segments lie in the memory segments of team T1 numbered $i_1, \ldots, i_k$. They will be copied, at the same position, into the memory segments numbered $i_1, \ldots, i_k$ of team T2. (section 7 describes a strategy for incremental copying). Failure would then be simulated on g_i. We further need to remove the conditional bindings made during the execution of the goal $g_{i+1} \ldots g_n$ by team T1. Let $i_{k+1} \ldots i_l$ be the stack segments where $g_{i+1} \ldots g_n$ are executing in team T1. As before, we can either copy the trail stacks of these segments and reinitialize (i.e. mark unbound) all variables that appear in them and then discard the copied trails, or we can copy the stack segments corresponding to goal $g_{i+1} \ldots g_n$ themselves in the appropriate memory segments of T2 and then backtrack over them. Once removal of conditional bindings is done the execution of the untried alternative of the stolen choice point is begun. The execution of the goals $g_{i+1} \ldots g_n$ is reinitiated (since we are following a recomputation approach) and these can be executed by other processors which are members of the team (some of this re-computation can be avoided, as mentioned earlier). Note that whereas copied stack segments occupy the same memory segments as the original stack segments restarted goals can be executed in any of the available memory segments (clearly if T2 decides to copy the computations done by team T1 for goals g_{i+1} through g_n to save recomputation or for untrailing, as mentioned earlier, then the corresponding stack segments will have to be copied in the same memory segments, i.e. i_{k+1} through i_l, of T2).

Returning to the earlier example (fig. 3), for execution to proceed as shown there, each pair of processors (P1, P2), and (P3, P4) would have to be in the same team (respectively teams T1 and T2). Each of processors P5 and P6 will also have to be in a separate team (respectively teams T3 and T4). Assuming that P1 starts the execution of query `q` in memory segment numbered 1, and P2 starts the execution of `b` in memory segment numbered 2 (in the address space of team T1), then P3 would be forced to copy the stack segment corresponding to `a` in memory segment number 1 of its address space. Assuming that only the trail stack of `b` is copied (to reset conditional bindings), P4 is free to execute `b` in any memory segment of T2 (which will be a segment other than memory segment 1, because only one processor in a team can use a memory segment at a time). Suppose P4 has its stacks located in segment 4 of the address space of team T2; then, it will execute `b` in memory segment number 4. When P5 and P6 steal the alternative corresponding to solution `b2` then each of them will copy stack segments corresponding to `a` to memory segment 1 of their respective address spaces, and the stack segment corresponding to `b` to memory segments 2 and 4 of their respective address spaces.

The copying of stacks by team T2 from team T1 corresponding to figures 3.(i) and 3.(ii) is further illustrated in figure 5. To keep the figure simple only the local stacks are shown. In reality, the heap and the trail will be copied too. Also note that copied choice points are transferred to a shared area to which the choice points in the local address space now point, as in MUSE.

The shared memory area is not shown in Figure 5. Note that because goals in a CGE are recomputed, parcall frames and any other structure used to support and-parallelism (as the various markers used by the PWAM [18]) are copied rather than shared (see Figure 5)

Note also that each memory segment in a team's address space has a complete set of stacks for a processor to work on corresponding to the "stack set" of &-Prolog [16]. Thus, the segmented memory management proposed can also be viewed as each team having a number of stack sets on which different processors ("agents") can work on. This view allows the immediate application of standard memory management techniques developed for independent and-parallelism [17] within each team.

This leads to a layering of the parallelism exploitation in ACE: at the lower layer, within each team, the computation is purely and-parallel, as in a group of "stack sets" in &-Prolog to which a number of "agents" are attached; at the higher layer, among the teams, the computation is purely or-parallel (as in MUSE). Thus, it is easy to see that in the presence of only and-parallelism our system would be as efficient as &-Prolog, while in the presence of only or-parallelism it will be as efficient as the MUSE system. Also notice that the amount of stack copying that will be done, in the presence of both and- and or- parallelism, would be identical to that done in the MUSE system provided, of course, that the scheduling strategy is the same.

However, the set up time for executing the untried alternative choice points that fall within the scope of a CGE may be different than in MUSE, due to the spreading of the computation across different stacks. On the other side, the actual copying operation may result even faster than in MUSE, since ACE can take advantage of having multiple processors transferring in parallel parts of the computation tree. Note that these differences only appear when both and- and or-parallelism are being exploited *simultaneously*.

An interesting property of ACE also related to memory management is that it adapts quite naturally to a hybrid multiprocessor in which parts of the address space are shared among subsets of processors, as, for example, in a system containing multiple shared-memory multiprocessors connected by a message passing or broadcast local network [4]. In this kind of system each shared-memory multiprocessor would naturally be a candidate for constituting a team with its own memory address space, and the various teams would then be spread over the different multiprocessors and communicated by the message-passing or broadcast local network. And-parallelism would be exploited within the shared-memory system while or-parallelism would be exploited over the distributed network of these shared memory systems. The argument above has been based on locality of addressing space issues, but a perhaps even more important factor involved is that of the access time depending on location. It also makes sense from this point of view to keep processors in a team, which communicate more often, within the fast communication area and put different teams, which communicate less often, at a larger distance from the point of view of communication. Similar principles apply and a similar approach can be taken for implementing and-or parallel systems on general NUMA (Non Uniform Memory Access) Machines even if they have a global addressing space.

4 Scheduling

The need to schedule work arises at two independent levels: (i) and-parallel work at the level of processors within a team and, (ii) or-parallel work at the level of teams. Thus a processor can steal and-parallel work only from members of its own team and an idle team can steal an untried alternative from a choice point. This suggests that separate schedulers can be used for managing the and-parallel and or-parallel work respectively. Schedulers developed for &-Prolog and for MUSE can be used for this purpose.

For example, and-parallel work can be managed exactly as in the &-Prolog scheduler: idle processors steal available goals from *goal-stacks* of other processors in their team.

Or-parallel work can be managed as in MUSE: idle teams request work from other teams (as in MUSE, it will be convenient to share as much work as possible). A distinction can be made between the *public* part and the *private part* of the execution tree: the choice points in the public part have been made sharable, while those in private part have not been made sharable yet. Execution in a team continues normally as in an and-parallel system (as described above), until the team is interrupted by another that is looking for work. At this point all choice points in the private part that satisfy the sharing requirement are made sharable or public. The requesting team picks an alternative from one of these choice points for or-parallel processing.

Finally, one has the new problem of balancing the number of teams and the number of processors in each team, in order to fully exploit all the and- and or-parallelism available in a given Prolog program. In order to solve this problem dynamically processors can migrate from one team to another or start new teams. An idle processor first looks for and-parallel work in its own team. If no and-parallel work is found, it can decide to migrate to another team where there is work, provided (a) it is not the last remaining processor in that team, and (b) there is a free memory segment in the memory space of the team it joins. If no such team exists it can start a new team of its own, perhaps with idle processors of other teams, provided there is a free address space available for the new team. The new team can now steal or-parallel work from other teams. Some of the 'flexible scheduling' techniques [8] that are being developed for deciding when a processor should switch teams etc. in the Andorra-I system can also be used in ACE.

5 Implementation of ACE

The discussion so far has aimed at providing a general, high-level description of the ACE execution model. In this section we will present a number of practical issues which arise in the implementation of the model and propose a number of solutions for efficiently resolving them. The two main issues we study are related to memory management and how efficient copying through only copying parts of the stack sets between teams can be implemented, and how full Prolog should be supported in ACE. (Further details on implementation of ACE, that are not included here for lack of space, can be found in [13]).

5.1 Goal Execution Order in CGEs

Memory management is a complex problem in the implementation of parallel logic programming systems, one that is closely related to scheduling [17].

Memory management is simplified in MUSE because each processor manipulates a separate Prolog stack set. In contrast, in ACE a team manipulates multiple stack sets that may have to be copied when teams fetch work from other teams. Furthermore, depending on and-scheduling, only parts of such stack sets may be needed: the order in which stack frames are pushed on the processor's stack may not obey the order in which they would have been pushed in a sequential Prolog implementation, and thus a stack segment may contain "trapped" stack frames (actually, whole "stack sections") that are not part of the computation surrounding it [17]. As a result of this, when copying stack segments we may copy sections that are unrelated to the branch we need. We can completely avoid copying these non-relevant parts, but then many small fragments of the stack will have to be copied making the copying operation somewhat inefficient[13], and, in any case, the hole created by the trapped goal would remain in the copying stacks because copying is address-to-address to avoid pointer relocation. Incremental copying is also made difficult by this potential lack of order. We explain these practical issues in more detail next.

Ideally, we would prefer that a parallel stack-based system implementing Prolog semantics would obey the *seniority constraint*: Given two stack frames, f_1 and f_2, corresponding to two nodes in the Prolog search tree, then f_1 should be allowed to appear above f_2 (there might be other intervening nodes between f_1 and f_2) if and only if f_1 will appear above f_2 in the stack in standard sequential execution of Prolog. Thus frames of descendent nodes in the execution tree must appear on top of frames of their ancestors. The reason why this is helpful is that, while backtracking, if we reach a frame f then we know that f is on top of the stack and that the frames corresponding to all its descendents have been backtracked over and reclaimed, thus considerably simplifying memory management. If the seniority constraint is not obeyed, then holes may appear in the stack both in and- and or-parallel systems, and "trapped goals" may appear in and-parallel systems [17].

In fact, the seniority constraint may impose severe constraints on parallel systems. Enforcing this constraint for an independent and-parallel system such as &-Prolog (or ACE) may severely constrain the way and-parallel goals can be scheduled. Given a CGE `.....(true => a & b & c)......` if a processor picks the goal `c` for and-parallel processing, then following this constraint will effectively shut it out from picking any goals (after it has finished processing `c`) to the left of `c`, or goals from CGEs created during execution of `a` or `b`, or goals from ancestor CGEs that are to the left. The seniority constraint is obviously too severe in this case and indeed systems such as &-Prolog [16] and Aurora [9] do not obey it. Rather they let holes (and trapped goals, in the case of &-Prolog) be created in the stack, that will be reclaimed when everything above the hole gets reclaimed (see Figure 6). This creates many problems in ACE, because now when we copy a stack set, we may copy many trapped goals that are not part of the current alternative stolen. These trapped goals may need to be identified before execution can begin in the copying team. This problem and our solutions are further discussed in the next section in the context of techniques for *incremental copying* of stacks[8].

5.2 Incremental Copying in ACE

An optimization that significantly improves the performance of stack-copying or-parallel systems, like MUSE, is *incremental copying*, i.e., when a processor copies a stack of another, then only those parts are copied in which the two processors differ. This is illustrated in Figure 7 (only local stacks are shown). Suppose processor P1 is working on branch 1, and P2 on branch 2. At this point both P1 and P2 have a common stack up till the branch node `a` (modulo conditional bindings). Suppose now that after exploring branch 2, P2 decides to pick an alternative from P1

[8]Note that goal-recomputation, as used in ACE, actually helps in maintaining seniority constraints, because every time we recompute goals, we execute them on top of existing solved goals that are to the left, thus righting the order somewhat.

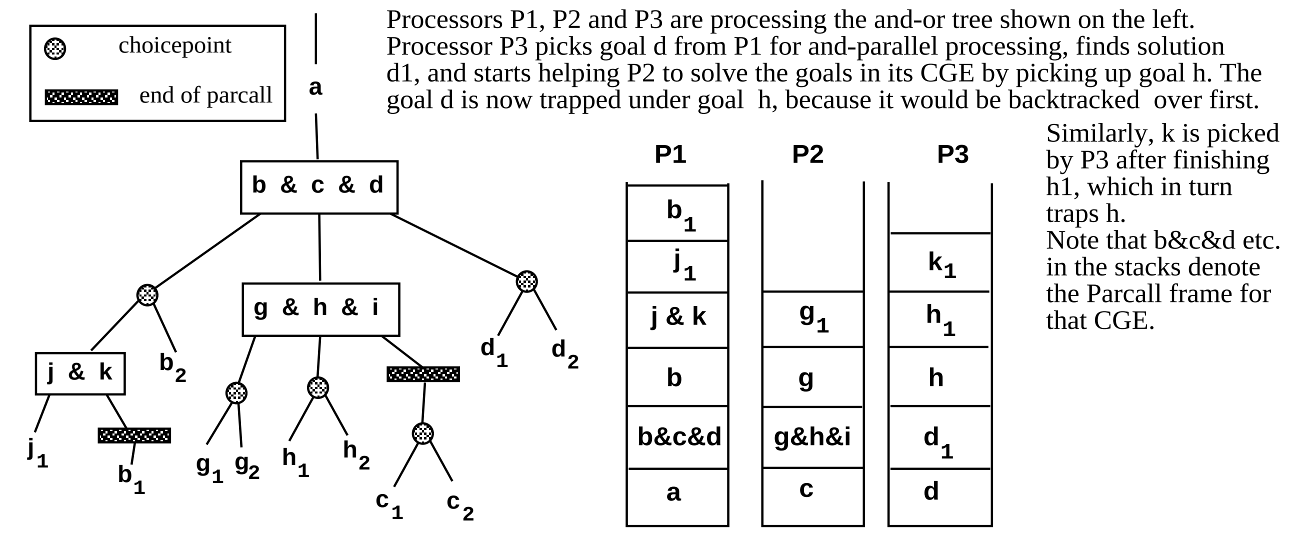

Figure 6: Trapped Goals in And-parallel Execution

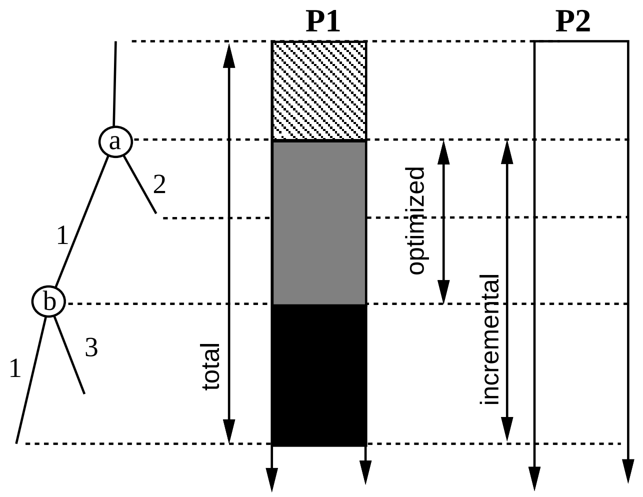

Figure 7: Incremental Copying in MUSE

(along branch marked 3) in node b. To do so it backtracks up to node a and steals the second alternative from b in P1. Therefore, before P2 can proceed, it needs to create on its stacks the state that existed in P1 at the time the choicepoint corresponding to b was created. To do so it copies P1's stacks. The copying and restoring of state can be done in three ways [3] (Figure 7):

(i) Total: copy the entire stacks of P1 (everything from the root to the bottom most node along branch 1), then backtrack until choicepoint b is reached. Thus, the hatched, gray, and the black shaded segments of P1's stack in Figure 7 will be copied;

(ii) Incremental: copy only frames below choice point a (those above a are already on P2's stack), then backtrack until choice point b. Thus, only the gray and black segments in P1's stack in Figure 7 will be copied);

(iii) Optimized incremental: copy only the stack segments between choice points b and a because those above a are already on P2's stack, and those below b are not needed for execution. Thus, only the gray shaded portion of P1's stack in Figure 7 is copied. The exception is that the entire trail stack below a is copied, so that the parts of the trail stack below choice point b can be used for removing conditional bindings.

Clearly, option (i) involves unnecessary copying[9] because there are copied parts that are immediately

[9] Experiments on the Sequent Symmetry have shown that for memory chunks larger than 4K the copying time is proportional to the size of the memory chunk being copied [13].

backtracked over and reclaimed. Option (ii) also does unnecessary copying, unless the black shaded part of the stack in Figure 7 is very small in size. In MUSE the difference between Incremental and Optimized is almost irrelevant, since in most of the cases there will be hardly anything on the stack below the choice point (assuming the stack is growing downward as in Figure 7) from which the new alternative is taken. This is a consequence of the scheduling policy adopted by MUSE, in which alternatives are always taken from the bottommost choice point (known as dispatching-on-bottommost [2]).

In ACE, however, things are different due to the presence of and-parallelism. Referring again to the and-or tree shown in Figure 6, suppose that a team, T2, was working on alternative $\mathtt{g}_2$ of goal `g` in the inner CGE (which it stole from a team, T1, earlier). It finds a solution, looks for more work, and decides to pick an alternative $\mathtt{h}_2$ from node `h` (corresponding to solution $\mathtt{g}_1$ from T1). T2 and T1 have a common stack up to the CGE labeled `(g & h & i)`. The stack frames leading up to choicepoint `g` are also present in both. Applying the idea of incremental copying, T2 will have to copy the difference between T1 and itself. As before, there are two ways of copying incrementally: (i) blindly copying the difference between corresponding stacks (of different processors of the two teams) on T2's stacks (Incremental Copying); (ii) copying only those parts which will be useful for T2, i.e. leaving out the parts that will be immediately backtracked over (e.g. the frames corresponding to $\mathtt{h}_1$, `i`, $\mathtt{c}_1$ and $\mathtt{d}_1$) copying only the trail for such parts. (Incremental Copying is illustrated in Figure 8).

While in MUSE Incremental copying (rather than Optimized incremental copying) results in very little space being copied that gets immediately backtracked over, in ACE this may not be the case, as ACE supports and-parallelism and follows the *sharing requirement* to make nodes public. Consider the following scenario: suppose as before that T2 tries to steal alternative $\mathtt{h}_2$ from choicepoint `h`, and that T1 had not yet found a solution for goal `i`. In this case, T1 will not make the choice points of all the branches to the right of `i` public (that is, choice points created from the goal `d` in the example). This is for two main reasons. Firstly, and as mentioned when presenting the sharing requirement, work available from these choice points will be very speculative, as `i` may yet fail (possibly after computing for a long time) and all the work in copying these branches and picking work from them may therefore be wasted. Second, making these choice points public will lead to mixing of public and private parts of the logical search tree[10]. For instance, if execution of `i` (or the CGE's continuation) in T1 was to lead to further choice points, they will initially be private and hence not be visible to other teams, although choice points of goals to the right of `i` (such as `d`) will be public. Thus, during normal backtracking through a CGE private choice points might be encountered after a processor has backtracked into the public area (or worse yet, if another team steals alternative from `d` and later backtracks, it will not see parts of `i` at all since they were never copied because of being private to T1). This mixing of public and private areas of the logical search tree, thus, will result in complications in scheduling. Hence, choice points in goals to the right of an incomplete goal in a CGE are never made public. As a consequence of this, Incremental copying will end up copying all the private goals, that may form a large part of the tree, and immediately descarding them (by backtracking over them), which is clearly a waste.

[10]Note that they are already mixed in the physical stacks.

Therefore, it makes sense to use Optimized copying for ACE, although it is more complicated to implement.

It is not very clear, on the other side, which incremental copying approach (Incremental or Optimized) will reduce the synchronization time between teams T1 and T2. However, it is obvious that Incremental copying is the simpler of the two: in the case of Incremental copying T1 has to synchronize for the duration of the copying of the difference, while in the Optimized case we first have to figure out the limits of copying for each processor stack in T1 (which may require a complete traversal of the and-parallel tree computed by T1) and then do the copying. Optimized incremental copying for ACE for the and-or tree shown in Figure 8 is illustrated in Figure 9.

In ACE we propose to use both Incremental and Optimized incremental copying depending on the situation. The following heuristic tries to balance the excessive unnecessary copying (Incremental copying) and the excessive synchronization time (Optimized copying) by dynamically detecting which of the two options may give the best results. If the choicepoint from which the alternative is being stolen is outside the scope of any CGE, or it is in the scope of some CGEs and all these CGEs have found a solution (i.e. each subgoal of each CGE has already found a solution) in the team from which the alternative is being stolen, then Incremental copying will be adopted. Otherwise, Optimized incremental copying will be used. As mentioned, Optimized copying requires traversal of all CGEs in which the choicepoint is nested and obtaining the addresses of *input-markers* and *end-markers* [18, 16, 13] for each goal in these CGEs. From these addresses and the information in Parcall frames we determine the useful part of the various stacks to be copied. Finally, note that all processors in a team can cooperate to speed up detecting the areas to be copied and copying of stack segments from one team to another.

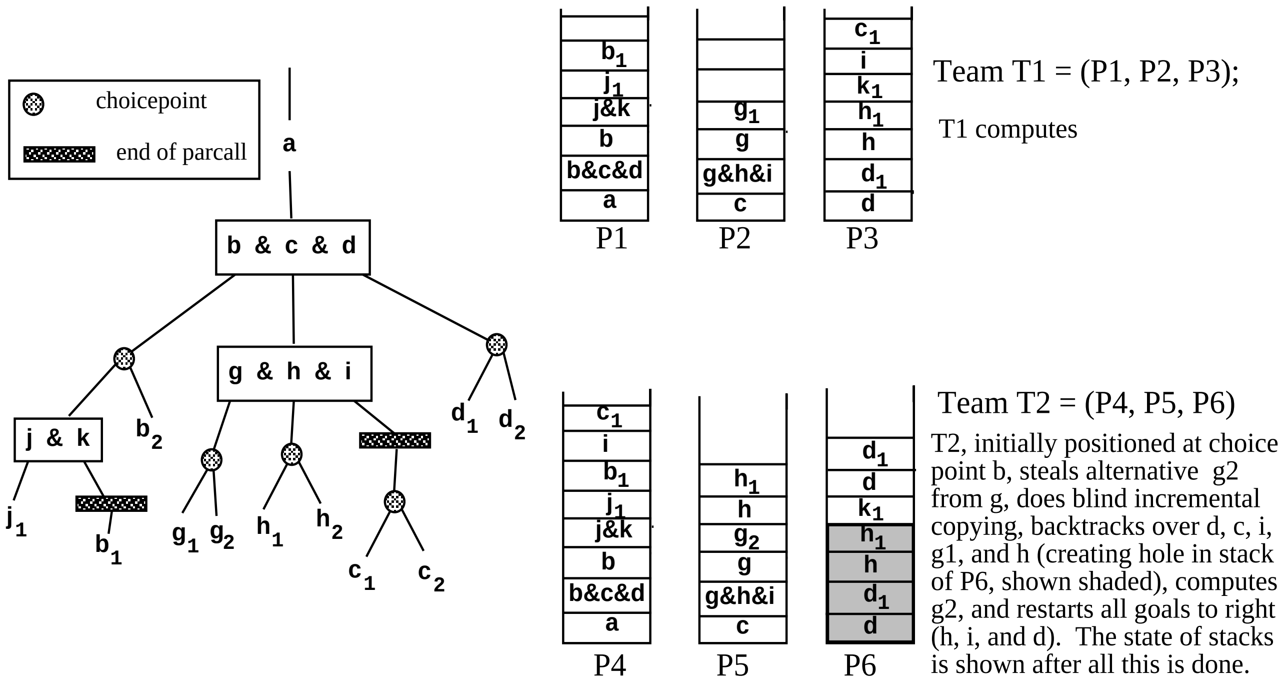

Figure 8: Incremental Copying in ACE

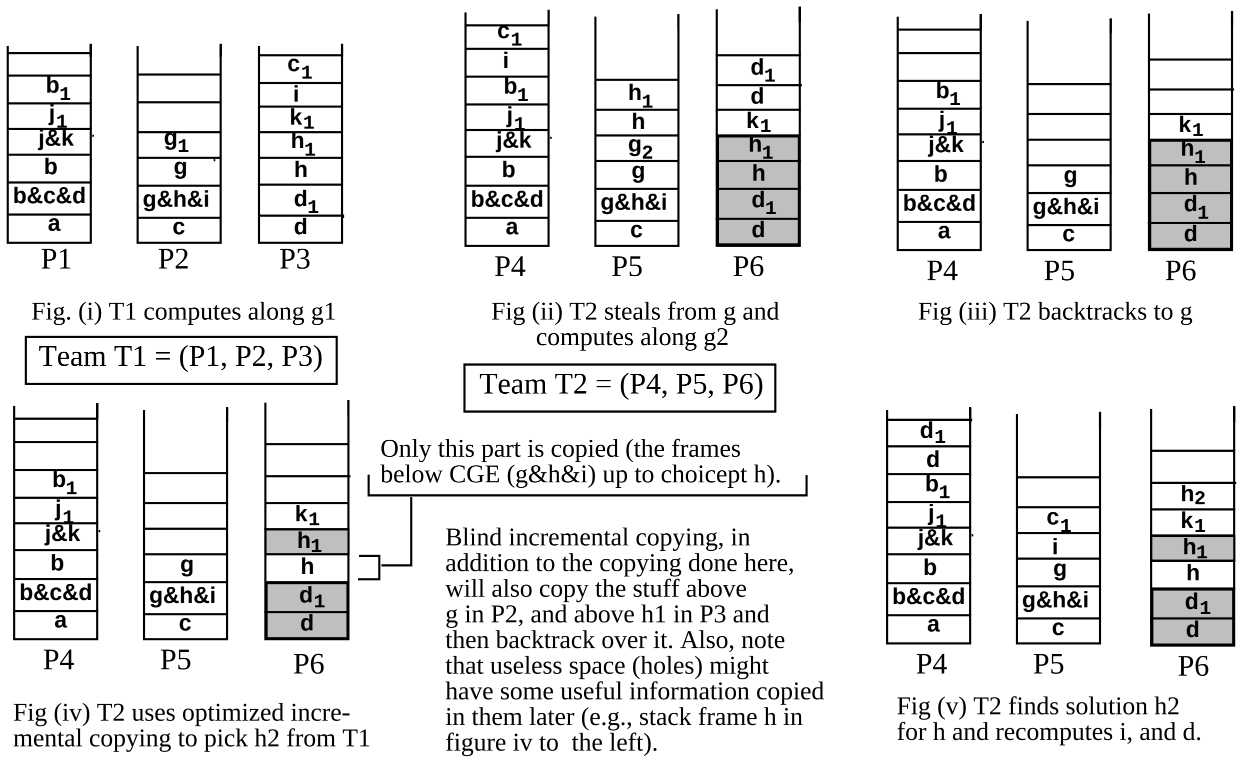

Figure 9: Optimized Incremental Copying in ACE

5.3 Implementing Side-effects and Cuts in the ACE Model

One advantage of an and-or parallel model that recomputes independent goals is that since it closely mirrors traditional Prolog execution it can quite easily support full Prolog, i.e. support the execution of order sensitive predicates such as side-effect predicates (e.g. read, write, assert, retract, and calls to dynamic predicates) and cut (!).

Essentially, a side-effect predicate (`sep` for brevity) should be executed only after the `sep` "preceding" it (preceding in the sense of left-to-right, top-to-bottom Prolog order) has finished execution. If the preceding `sep` has not been executed, the current `sep` should suspend, and resume after the execution of the preceding `sep` is over. However, given a `sep`, determining the `sep` that "precedes" it is akin to solving the halting problem, and therefore the knowledge that the preceding `sep` has finished has to be approximated.

For example, consider supporting `seps` in purely or-parallel systems [15]. Here, the preceding `sep` is assumed to be finished if the or-branch containing it has finished execution. In other words, a `sep` is executed only when the branch containing it becomes the leftmost in the or-parallel tree. Likewise, in purely independent and-parallel system, such as &-Prolog, a `sep` encountered in an independent and-parallel goal g in a CGE C is executed only after all the independent and-parallel goals to the left of g in the CGE C have finished execution. If the CGE C containing the goal g is nested inside a goal h, which is an independent and-parallel goal in another CGE D, then all the independent and-parallel goals in CGE D that are to the left of goal h should have finished, and so on.

We can combine the conditions for executing `seps` in a purely or-parallel system with those for a purely and-parallel system to generate the conditions for executing a `sep` in an and-or parallel system such as ACE. Given a CGE ($cond \Rightarrow g_1$ & ... & g_i & ... & g_n), where we assume that the parallel execution of goal g_i leads to a side-effect, the conditions under which this side-effect will be executed are given below. Note that the goal g_i is being recomputed in response to solutions $s_1 \dots s_{i-1}$ that will be found for goals $g_1 \dots g_{i-1}$ respectively. Let $b_1 \dots b_{i-1}$ be the or-branches in respective search trees of goals $g_1 \dots g_{i-1}$ that lead to these solutions. The conditions are as follows:

(i) The or-branch that contains the `sep` in the search tree of goal g_i should become leftmost [11].

(ii) The computation of solutions $s_1 \dots s_{i-1}$ should have finished; and the or-branches $b_1 \dots b_{i-1}$ should be leftmost in the search tree of their respective goals $g_1 \dots g_{i-1}$.

(iii) If the CGE containing g_i is nested inside another CGE then conditions (i) and (ii) must recursively hold for the inner CGE with respect to the outer CGE. If the CGE is not nested inside other CGEs, then the or-branch in which it appears should be leftmost with respect to the root of the whole computation tree.

[11] with respect to the equivalent or-parallel tree

In the rest of this section we present a concrete technique for determining when a `sep`'s turn for execution has come during and-or parallel execution. The techniques make use of techniques developed for &-Prolog [7, 21, 5], MUSE [3], and Aurora [9]. For simplicity, and without loss of generality, we assume that when a processor reaches a `sep` it repeatedly performs the above check until it succeeds (thus the processor busy-waits rather than suspends). However, suspension would be used in practice[12] so that the processor that encountered the `sep`, rather than busy-waiting and wasting cpu-cycles, can do useful work elsewhere.

5.3.1 Side-Effects in ACE

Note that while verifying the above conditions to check if a side-effect can be executed, processors need to access shared choice points recorded in the shared memory (to do the leftmost check). This can be expensive, especially in a non-shared memory or a hybrid multiprocessor system. One can reduce the number of accesses to shared memory by first requiring a processor that has reached a side-effect to:

(a) check if all goals to the left of g_i in the current CGE, and those to the left in all the ancestor CGEs have produced a solution (first part of condition (ii), and condition (iii))

(b) check if the side-effect is in the leftmost branch, and the solutions to preceding goals in all the CGEs are in leftmost branches (condition (i), second part of condition (ii), and condition (iii)).

Note that check (a) does not require access to the shared area, it is performed wholly within the address space of the team executing the side-effect. Check (b) will be made only after check (a) succeeds, thus reducing the number of accesses to shared area. The above decomposition also neatly separates the and-parallel and the or-parallel components of the check. Both checks (a) and (b) must be implemented efficiently, particularly check (a) since it is going to be performed more often[13].

[12] Implementation of suspension does not present problems in &-Prolog. Techniques for implementing suspension more efficiently in MUSE by storing the difference between the suspended branch and the one that the processor switches to have recently also been developed by the MUSE group. These techniques can be adapted for ACE.

[13] Indeed, check (a) can be implemented quite efficiently since the appropriate information about the status of and-parallel goals is maintained in the CGE's descriptor, and therefore, performing check (a) involves a simple look-up of the corresponding parcall frame(s).

The presence of the sharing requirements allows to separate the side-effect checks for or and and-parallelism in a different way. In fact the sharing requirements guarantee that all the branches on the left of a public choice point are completed (otherwise the choice point would not satisfy the requirements during the sharing operation). Because of this we need not perform the check (a) in the public part of the tree. Furthermore, in the private part of the tree the check (b) is unnecessary since no sharing operations have been performed (the side effect is for sure in the leftmost branch). Thanks to these observations, if P is the bottommost public node in the current branch, then we can organize the side-effect check as follows:

(1) apply check (a) only to the subtree rooted at P;

(2) apply check (b) only to the public part of the tree above P.

Two main algorithms have been proposed to handle side-effects in independent and-parallel systems (like &-Prolog): synchronization blocks[7, 21], and visiting each ancestor CGE and checking if goals to the left have finished [5]. Either one of these can be used for performing check (a).

To check if a given node is in the leftmost branch of a given subtree, we need access to the left sibling nodes of the immediate ancestor choice point nodes (given a node, if the choice point node above it doesn't have any left siblings, the node is in leftmost branch of the subtree rooted at that choice point). However, the sibling-nodes of a choice point are not directly accessible to a team doing the check, therefore we have to use some other technique to determine this. The technique that we use parallels the technique proposed for MUSE [3].

We use the fact that part of the choice point in ACE is shared, and hence the fields in the shared part of choice-point are visible to all teams. Each shared choice-point in ACE includes a *teamsbitmap*, (from MUSE's *workersbitmap*). The teamsbitmap indicate which teams are exploiting alternatives of that node. When the ith alternative is picked by a team from a node, the ith bit in the teamsbitmap is set. When the subtree corresponding to the ith alternative has been completely explored and backtracked through, the ith bit is reset. In the *alt-number* field in the private part of the choice-point, a team also records the *alternative number* which it picked from this choice point. Note that the alt-number field will occupy the same memory address in the address space of each team that is working below this choice point.

The algorithm for verifying leftmostness is thus as follows: the team goes up the execution tree; whenever it reaches a shared choice-point it looks at the corresponding teamsbitmap; if there are other teams that are working on the alternatives of this node, the corresponding images of the choice-point are checked in the address spaces of these teams to see if the current branch is leftmost. This is done by a a simple arithmetic comparison of the alt-number fields in these choice points. Note that while checking for leftmostness of side-effect goals and solutions of goals to the left in a CGE etc. we are only concerned with determining leftmostness of nodes in the subtree of a goal in the CGE (local leftmostness), and not in the whole program search tree (global leftmostness).

Several optimizations are necessary to make this algorithm efficient. Firstly, if two teams share a choice-point N, they will also share all ancestor nodes. Thus, one needs to compare two teams only once for the youngest node they share. Secondly, as in the Aurora schedulers (and proposed as an optimization in MUSE), one can keep track of the current node up to where a worker or team is leftmost.

Finally, we can completely avoid accessing any remote choice point by storing in each shared frame a bitmap which indicates for each alternative in the choice point whether there is at least one active team working on that alternative.

5.3.2 Implementing Cut in the ACE Model

The effect of a cut is to prune all branches to the right of the path from the place where the cut is executed to the node where the clause containing the cut was introduced (cut level). Hence, because a cut can only cut up to the current CGE, condition (iii) is always trivially satisfied. The treatment of cuts is similar to that of side-effects except that in the case of cut some action can be taken (i.e. some pruning can be done) without the cut becoming leftmost in the entire tree [15]. Basically, a cut can be immediately exercised in the subtree in which that cut is leftmost. Other branches can be pruned only after the cut becomes leftmost in the entire tree. Thus, in ACE when a cut is encountered pruning can be immediately done up to the point where conditions (i) and (ii) above succeed. To prune other parts the team has to wait until condition (i) and (ii) are satisfied right up to the root node, i.e. the cut becomes a global leftmost. Note that pruning a choice-point consists of clearing that choice-point and signaling any teams exploring alternatives to the right to terminate execution. Termination of execution by a team means that all the processors in the team abandon execution and backtrack. The efficient techniques used to deal with cuts in or-parallel systems (like those of MUSE [3] and Aurora [9]) can be adapted to ACE.

6 Efficiency and Generality of the ACE Model

We believe that an implementation of the ACE model will be quite an efficient realization of an or-and independent and-parallel system. This is primarily because, as may already be evident, in the presence of only or-parallelism ACE will be as efficient as MUSE, while in the presence of only independent and-parallelism it will be as efficient as &-Prolog. Therefore, it appears clear that having an ACE system would be, at least, as powerful and efficient as having both a MUSE and an &-Prolog system, in the sense that now a single system will run or-parallel only programs and and-parallel only programs with similar performance as the MUSE and &-Prolog systems respectively. ACE should also combine speedups from programs where both or- and independent and-parallelism are available, hence performing even better than the best of MUSE or &-Prolog for such applications. Note that with respect to MUSE, the parts that are copied in and-or parallel execution in ACE for a given program are exactly those that will be copied by MUSE in an equivalent purely or-parallel execution of the same program, but, whereas MUSE will copy one large stack segment at any given time, by exploiting independent and-parallelism, ACE may spread this segment over many memory segments in the address space of the team. This may in principle add some overhead to the copying cost (since many small segments rather than one large segment may have to be copied). However, because each team has multiple processors, the copying of multiple segments can be done in parallel. With respect to &-Prolog, ACE does not introduce any new overheads. The only inefficiency present in the ACE model is with respect to memory consumption, but that cannot be avoided if we want to use stack-copying for representation of multiple environments. Given that memory is inexpensive, we hope that this will not be such a big bottleneck.

Another important point that should be noted is that the approach outlined in this paper for implementing and-or parallel systems, while presented in terms of combining the types of parallelism present in MUSE and &-Prolog, is actually quite general, and can be applied to implement other systems that exploit and- and or-parallelism, such as Andorra-I [6], Prometheus [24], and IDIOM [11]. It is quite easy to see how Andorra-I, a system that exploits or-parallelism and determinate dependent and-parallelism, can be implemented (the implementation of Andorra-I by Yang, Santos Costa, and Warren is based on binding arrays) using stack-copying. In Andorra-I there is no or-parallelism within and-parallel goals since only deterministic goals can be processed in and-parallel (thus it reduces to the case described in section 2), thus and-parallel execution can be performed by each team locally. Or-parallelism will be implemented using stack-copying and the memory-management scheme described above. Likewise, Prometheus [24], a system that exploits or-parallelism and non-determinate dependent and-parallelism (with no coroutining) by extending CGEs, can be easily implemented using the ACE scheme. In fact, since the DAS-WAM abstract machine on which Prometheus is based is itself based on that of &-Prolog no extra measures need to be taken apart from those needed to support dependent and-parallelism, which are for the most part orthogonal to the issues dealt with by ACE. IDIOM, which adds independent and-parallelism to Andorra-I, can also be implemented using the ACE approach. Its implementation can be thought of as a combination of the ACE and Andorra-I implementations, and, again, is straightforward to derive.

7 Conclusions

In this paper, we presented ACE, a model capable of exploiting both non-deterministic and-parallelism and or-parallelism. We have discussed both high-level and low level implementation issues and shown how using recomputation the scheme can incorporate side-effects and support Prolog as the user language easily.

We have shown how ACE subsumes two of the most successful approaches for exploiting parallelism in logic programming (MUSE and &-Prolog).

We have argued how the resulting system has a good potential for low sequential overhead, can be implemented in a reasonably easy way by extending existing systems, and retains the advantages of both purely or-parallel systems as well as (even non-deterministic) purely and-parallel systems. A collaborative implementation of ACE on Sequent and other multiprocessors is under way at New Mexico State University and University of Madrid (UPM).

8 Acknowledgements

The research presented in this paper has benefited from discussions with Khayri Ali, Manuel Carro, Roland Karlsson, Kish Shen, and David H.D. Warren, all of whom we would like to thank. Part of this work was done while Gopal Gupta was a member of David Warren's group at Bristol and supported by UK SERC grant GR/F 2740. Manuel Hermenegildo was supported in part by ESPRIT/CICYT project PEPMA. Ongoing work is supported by NSF Grants CCR 92-11732 and HRD 93-53271, Grant AE-1680 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, by NATO Grant CRG 921318, by ESPRIT/CICYT project PARFORCE, by a fellowship from the European Commission to V. Santos Costa and by a fellowship from Phillips Petroleum to Enrico Pontelli.

References

[1] K.A.M. Ali. Or-parallel Execution of Prolog on the BC-Machine. In *Fifth International Con-*

ference and Symposium on Logic Programming, pages 253–268. MIT Press, 1988.

[2] 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.

[3] 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.

[4] Gordon Bell. Ultracomputers: a Teraflop Before its Time. *Communications ACM*, 35(8):26–47, 1992.

[5] 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.

[6] 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.

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

[8] Inês Dutra. A Flexible Scheduler for the Andorra-I System. In *ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, Computer Science Department, University of Bristol, June 1991.

[9] E. Lusk et al. The aurora or-parallel prolog system. *New Generation Computing*, 7(2,3), '90.

[10] G. Gupta and V. Santos Costa. Cut and Side-Effects in And-Or Parallel Prolog. In *4th IEEE Symp. on Parallel and Distr. Processing*, '92.

[11] G. Gupta, V. Santos Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. In *Int'l Logic Prog. Symp.*, pages 152–166. MIT Press, '91.

[12] 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.

[13] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. Ace: And/or-parallel copying-based execution of logic programs. Technical Report, 1993.

[14] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 N. American Conf. on Logic Prog.*, pages 332–349. MIT Press, 1989.

[15] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[16] 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.

[17] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Proc. 4th ICLP*, pages 556–575. MIT Press, 1987.

[18] 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.

[19] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *3rd ICLP*, pages 40–55. Springer-Verlag LNCS 225, 1986.

[20] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. Journal of Logic Prog., To appear.

[21] 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.

[22] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89*, pages 313–331. MIT Press, 1989.

[23] 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.

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

[25] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *Proc. 1991 International Logic Programming Symposium*. MIT Press, 1991.

[26] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *IEEE Int'l Symp. on Logic Prog.*, pages 436–448, 1987.