CUTS AND SIDE-EFFECTS IN AND-OR PARALLEL PROLOG *

GOPAL GUPTA AND VITOR SANTOS COSTA †

▷ Practical Prolog programs usually contain extra-logical features like cuts, side-effects, and database manipulating predicates. In order to exploit implicit parallelism from real applications while preserving sequential Prolog semantics, a parallel logic programming system should necessarily support these features. In this paper we show how Prolog's extra-logical features can be supported in an and-or parallel logic programming system. We show that to support extra-logical features an and-or parallel logic programming system should *recompute* the solutions to independent goals instead of *sharing* them. We describe an abstraction called the Composition Tree for representing and-or parallel execution with recomputation. We introduce the notion of "local-leftmostness" in the Composition Tree and use it for deriving complete and efficient methods for supporting extra-logical predicates in and-or parallel logic programming systems based on the Composition Tree abstraction. ◁

1. INTRODUCTION

One of the most important features of Logic Programming Languages is that they can be implicitly executed in parallel without any intervention from the programmer. There are three prominent forms of implicit parallelism in Logic Programming Languages:

*This paper is an expanded version of the paper "Complete and Efficient Methods for Supporting Cuts and Side-effects in And-Or Parallel Prolog" that appeared in the Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing held in Arlington, Texas, in Dec. 1992.

Address correspondence to Laboratory for Logic, Databases, Advanced Programming, Department of Computer Science, New Mexico State University, Las Cruces, NM 88003.
†Universidade do Porto, Rua do Campo Alegre 823, 4150 Porto, Portugal

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

1. *Or-parallelism:* arises when multiple clauses define some predicate and a goal unifies with more than one clause head—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently executing a nondeterministic procedure; it speeds up the search for a solution to the top-level query by exploring alternative solutions in parallel.
2. *Independent and-parallelism:* arises when arguments of a subgoal are bound to structures that do not share any variables with those of another conjunctive subgoal. Such *independent* goals can be executed in parallel.
3. *Dependent and-parallelism:* arises when two mutually dependent goals are executed in parallel to cooperatively produce an answer.

In this paper we focus on or-parallelism and independent and-parallelism only.

There are several successful systems that either support purely or-parallel execution (such as Aurora [21] and Muse [2]), or purely independent and-parallel execution (such as &-Prolog [18]). This success prompted researchers to explore the combination of both forms of parallelism. As a result many models for combining these two forms of parallelism were proposed [24, 5, 14, 13, 15].

The success of systems that exploit only one form of parallelism can in part be explained by the fact that not only they yield good parallel speed-ups over a large number of applications, they also support *full* Prolog semantics. By full Prolog we mean the complete Prolog language including the extra-logical features—read, write, cut, assert, retract, var, etc.—whose evaluation is sensitive to execution order. These systems guarantee that order sensitive predicates are executed in the "correct" order, i.e., the order in which they will be executed in sequential execution, even during parallel execution. Moreover, these systems do not sacrifice any parallelism in supporting these extra-logical features because they employ a dynamic scheme, rather than a static one which would partition the logic program into sequential and parallel parts, or which would partially sequentialize the logic program. There are two reasons why we want to support full Prolog semantics even during parallel execution: (i) Existing Prolog programs can be executed in parallel and speeded up; (ii) User programs can be developed in a sequential environment and then executed in parallel to obtain faster execution. Thus, not only user programs behave in a more predictable way, their debugging and development also becomes easier. It may appear that a program with extra-logical predicates cannot achieve any speed-up but that has already been shown to be untrue by or-parallel systems such as Aurora [21] and Muse [2] and by and-parallel systems such as &-Prolog [18] and Andorra-I [8]. The reason speed-ups can be obtained is because the other logical operations between two extra-logical predicates can be executed in parallel, and hence if the extra-logical predicates are spaced far-between, good speed-ups can still be obtained. For example, many Prolog applications read their input, perform some computation, and then output a result. A parallel logic programming system can improve performance by speeding up the computation, and still preserve the interactions that would be performed by a sequential execution. Speed-ups will be harder for Prolog applications that perform side-effects during the computation, and in some cases to obtain good speedups it will be necessary to rewrite the parts of the program where most computation is performed. Even in these cases, transparent support of the extra-logical operations allows programmers to concentrate only on the parts of the program where they can obtain speed-ups, instead of having to completely rewrite their programs.

Ideally, systems that combine or- and independent and-parallelism should also support full Prolog semantics without sacrificing any parallelism, since purely or- and independent and-parallel systems can do so. However, and-or parallel systems that have been devised so far have either taken the static approach (i.e., taken measures at compile-time to guarantee correct execution order for extra-logical predicates), or they have completely ignored the problem of supporting extra-logical features of Prolog. Thus, while the ROPM [24] and the AO-WAM [14] systems have ignored the problem, the PEPSys system divides the program into sequential and parallel parts, where extra-logical predicates are allowed only in the sequential parts [25, 5]. Both approaches either require some intervention from the programmer, or lose some parallelism which could otherwise be exploited had a dynamic approach been used. In this paper we show how extra-logical features can be supported in a system combining or-parallelism and independent and-parallelism (henceforth, we will call such a system an and-or parallel system) using the dynamic approach so that neither any user intervention is required nor any parallelism is unnecessarily lost. Our techniques apply to only those and-or parallel systems that use Conditional Graph Expressions (CGEs) [19] for representing independent and-parallelism.

Our ability to support extra-logical predicates of Prolog using a dynamic approach crucially rests on our decision to *recompute* the solutions of independent goals rather than *sharing* them.[1] We have developed an abstract model for representing and-or parallel computation with re-computation, called the Composition Tree [12, 15]. We have also shown how the environment representation techniques of purely or-parallel systems can be adapted for use in and-or parallel systems based on the composition-tree [12, 15, 13]. In this paper we show how Prolog's extra-logical features and full Prolog Semantics can be supported in the Composition Tree (*C-tree* for brevity) abstraction without sacrificing any parallelism. The techniques we present are an elegant generalization of the effective techniques that have been used for supporting full Prolog on purely and-parallel and or-parallel systems.

The rest of this paper is organized as follows: in Section 2 we discuss how Full Prolog is supported in current purely and-parallel and purely or-parallel systems; in Section 3 we describe the C-tree data-structure and a general method for supporting side-effects in parallel execution models that are based on the C-tree. In Section 4 the methods for supporting side-effects on C-tree based model are described. Section 5 presents a specific implementation of these methods for a parallel system where the entire C-tree is assumed to be accessible to (or shared among) all processors of the multiprocessor system. Section 6 presents methods and algorithms for implementing cuts in the C-tree. Section 7 concludes the paper. Throughout this paper familiarity with the basic concepts and techniques used in purely or-parallel such as Aurora [21, 6] and purely and-parallel systems such as &-Prolog [19, 18] is assumed.

[1] It should be noted that recomputation of independent goals was first introduced in &-Prolog [19, 18]. Recomputation vs reuse for and-or parallel systems was first studied in [27]. See also [26].

2. EXTRALOGICAL PREDICATES

Prolog programs include pure predicates and *extralogical predicates*. By extralogical predicates we mean side-effect predicates, including I/O predicates, such as `read` and `write`, database predicates, such as `assert` and `retract`, and meta-logical predicates such as the cut (`!`), `var` and `call`. Meta-logical predicates such as `var`, do not cause problems in combined independent and- and or-parallel systems; however, `var` will cause problems if we were to have dependent and-parallelism as well [9]. Extralogical predicates have been classified as *soft* and *hard* [10]. A soft extralogical predicate (such as read/1 and write/1) is one that does not influence the behaviour of other goals outside the goal that contains the extralogical predicate. A hard extralogical predicate (such as `assert` and `retract`) can also influence the behaviour of goals outside. An extensive discussion of hard and soft extralogical predicates can be found in [10]. However, we assume that predicates corresponding to goals whose execution may be affected by hard extralogical predicates (i.e., goals whose set of matching clauses may be modified at runtime) are declared as `dynamic` from beforehand by the user (most modern Prolog systems will indeed require the user to declare such predicates as dynamic). Calls to dynamic predicates can then also be regarded as extralogical. As a result, the distinction between soft and hard extralogical predicates is no longer needed and all extralogical predicates can be treated uniformly (this approach was first adopted in the Aurora or-parallel system [21]).

In general, to get the same external behaviour as sequential Prolog the execution of an extra-logical predicate has to be suspended until one can be sure that all other extra-logical predicates 'preceding' it have finished ('preceding' in the sense of left-to-right, top-to-bottom, sequential Prolog execution). In the case of cut, however, one can do better. Since the execution of cut does not affect the computed answer, but only the shape of the search tree, a cut can be executed immediately and parallel execution can proceed with goals following the cut. However, not all the actions that are normally associated with a cut can be taken immediately, if the sequential semantics of Prolog is to be preserved. Only those actions can be carried out which are safe; unsafe actions have to be postponed until all extra-logical predicates before this cut have finished (see later).

2.1. Extra-logical Predicates in Purely Or-parallel Systems

In or-parallel execution several or-branches, or alternatives, may be explored at the same time. To maintain sequential Prolog semantics, a side-effect predicate should be executed only after all others preceding it have finished. However, detecting in advance precisely when all preceding side-effects have finished is akin to solving the halting problem, and therefore this knowledge has to be approximated. The most widely used approximation is to execute a side-effect predicate only when the or-branch containing it becomes the leftmost branch in the whole or-parallel search tree [16]. Thus, if a side-effect predicate is encountered by a processor, the processor checks if the node containing that side-effect is in the leftmost branch of the or-parallel tree. If so, the side-effect is executed, otherwise the processor suspends until the branch becomes leftmost. After suspending, the processor may either busy-wait until that branch becomes leftmost (as done in the initial Muse system), or it can search for work elsewhere (as done in Aurora and in later versions of Muse), leaving this suspended side-effect for some other processor to execute.

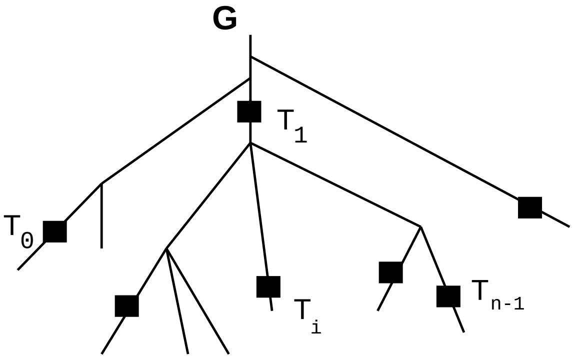


Figure 1. Side-effects in Or-Parallelism

Figure 1 shows a search-tree for a goal G. During execution, the side-effects $T_0, T_1, \ldots, T_n$ will be executed. The side-effect T_0 is placed in the leftmost branch, and will therefore be executable as soon as it is called. In contrast, the side-effect T_1 can only execute when the two search-tree branches to its left have been finished.

It can be easily shown that a dynamic scheme outlined above for ordering the execution of side-effect predicates is superior to a naive static scheme that simply sequentializes a program (more sophisticated static schemes that employ compile-time analysis may fare somewhat better). Consider for example the following definition of predicate `p` which contains predicates $\texttt{se}_1$, $\texttt{se}_2$, and $\texttt{se}_3$ that lead to side-effects (directly or indirectly):

```
p :- g1, se1, h1.
p :- g2, se2, h2.
p :- g3, se3, h3.
```

An or-parallel system using the static approach will declare this predicate sequential. hence at runtime the choice point for `p` will be labeled sequential, and its alternative tried sequentially, i.e., one after the other. Sequentialising this predicate to obtain Prolog semantics (i.e., trying the alternatives in choice-point for `p` in sequential fashion) will lead to loss of or-parallelism because goals $\texttt{g}_1$, $\texttt{g}_2$ and $\texttt{g}_3$ cannot be executed simultaneously (although goals that are generated by their clauses can be executed in or-parallel), rather they will be executed sequentially since the alternatives of `p` will be tried sequentially. If goals $\texttt{g}_1$, $\texttt{g}_2$, $\texttt{g}_3$ are substantially large, the loss of parallelism may be substantial. However, in the dynamic approach, the goals $\texttt{g}_1$, $\texttt{g}_2$, $\texttt{g}_3$ can be executed simultaneously, except that side-effect $\texttt{se}_3$ is executed only after $\texttt{se}_2$ and $\texttt{h}_2$ have finished, which in turn is executed only after $\texttt{se}_1$ and $\texttt{h}_1$ have finished. This will result in loss of parallelism between $\texttt{h}_1$, $\texttt{h}_2$ and $\texttt{h}_3$ but that will also be lost in the static approach, and unless one can determine the nature of $\texttt{h}_i$s (i.e., if they do or do not contain any further side-effects), devising a technique that will execute them in parallel while preserving Prolog sequential semantics is hard. Clearly, although the dynamic scheme involves some runtime overhead, it is better than the static approach in that it exploits more parallelism.

Various techniques have been proposed for efficiently determining when a branch becomes leftmost [1, 20, 3, 6, 29, 4] for different underlying or-parallel models. Arguably, the techniques developed for the Aurora system [21] have been the most well researched and successful [16, 6, 29, 17]. To determine if a branch containing a node is leftmost in the entire tree, the following technique is used. For each node the system dynamically keeps track of the root node of the largest subtree whose leftmost branch contains the node. This root node is called the *sub-root node* [29][2]. If the sub-root node is identical to the root node of the or-tree, then the node is in a leftmost branch. It should be noted that the algorithms developed for supporting side-effect for Aurora, as well as those developed in this paper, do not crucially depend on the whole search tree being shared amongst all processors. The fact that the tree is shared just makes the implementation of algorithms such as subroot node detection easier.

2.2. Extra-logical Predicates in Purely And-parallel Systems

We now analyse side-effects in purely and-parallel systems that use CGEs, such as &-Prolog. In such systems supporting side-effects in accordance to Prolog's sequential semantics can take advantage of the fact that at any given time only one solution exists on the stacks and the order in which solutions are explored is exactly like in the sequential execution of Prolog. If an and-parallel goal in a CGE reaches a side-effect, it must make sure that all the sibling and-branches to its left have finished execution. This is because an and-branch to its left may fail, and in such a case, in the equivalent sequential Prolog execution the side-effect would never be reached and hence would never be executed. Therefore, to preserve sequential Prolog semantics, in a purely and-parallel system a side-effect should be executed only after all sibling and-branches to its left have finished execution. If the CGE containing the branch with the side-effect is nested inside another CGE, then all sibling and-branches of the inner CGE which are to its left must also have finished, and so on. If any of these and-parallel goals have not finished, the execution of the side-effect must be suspended until they do. Figure 2 gives an example of side-effect execution in and-parallelism.

The central question is now how to detect that all and-parallel goals to the left have finished. Two major approaches have been proposed (there are other approaches as well, but they are closely related to the two mentioned below):

(i). Add synchronization code to CGEs using counting semaphores. Processors wait on these counting semaphores before they can execute a side effect. The semaphores are organized in such a way that correct ordering of side-effects results [10, 22].

(ii). A processor on reaching a side-effect predicate suspends. The suspended processor repeatedly performs the following operation: It checks if all sibling and-branches to the left of the node containing the side-effect have finished. If they have finished, it recursively checks if all the sibling and-parallel branches to the left of the goal that gave rise to current CGE have finished [7, 22]. The side-effect would be executed the moment the suspended pro-

[2] Also, note that the parent node of the sub-root node is called the *leftjoin node*.

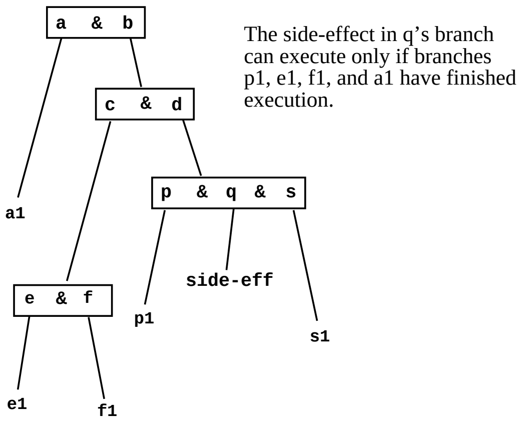


Figure 2. Side-effects in &-Prolog

cessor detects completion of all these sibling and-parallel branches to the left (in figure 2 this corresponds to waiting until goals `a`, `c`, and `p` are finished).

Systems such as &-Prolog introduce a limited amount of intelligent backtracking: if a goal in a CGE completely fails, the entire CGE fails, and all siblings goals in the CGE are killed. If siblings to the left of the failing goal contain side-effects, problems may arise. A solution is to only kill siblings to the left if they cannot contain side-effects [7], or kill them only after they have executed all their side-effects.

Having shown the advantage of using a dynamic technique for supporting side-effects over a static technique for purely or-parallel systems, let us also show the same for purely independent and-parallel systems. In an and-parallel system a static technique will enforce the correct order of execution of side-effects by replacing parallel conjunctions with sequential conjunction (the static technique is available as an option, for instance, in the &-Prolog compiler [23]). Thus given a CGE $(cond \Rightarrow p_1 \ \& \ p_2 \ \& \ p_3)$ where goals p_1 and p_2 both contain side-effects $\mathtt{se}_1$ and $\mathtt{se}_2$ respectively and p_3 is pure:

p_1 :- $\mathtt{g}_1$, $\mathtt{se}_1$, $\mathtt{h}_1$.
...
p_2 :- $\mathtt{g}_2$, $\mathtt{se}_2$, $\mathtt{h}_2$.
...

then in the static approach the CGE will be re-written as $(cond \Rightarrow (p_1, p_2) \ \& \ p_3)$. Clearly, $\mathtt{g}_1$ and $\mathtt{g}_2$ can no longer be executed in parallel, and hence if they involve large amounts of computation substantial parallelism may be lost. However, the dynamic approach will not replace parallel conjuncts by sequential ones and will execute $\mathtt{g}_1$ and $\mathtt{g}_2$ in parallel. In the dynamic approach side-effect $\mathtt{se}_2$ will be executed only after the goal p_1 has completely finished. Thus, the dynamic approach will not be able to execute $\mathtt{h}_1$ and $\mathtt{h}_2$ in parallel, but neither would the static approach. Again, although, the dynamic approach will incur some run-time overhead, the amount of parallelism gained may be significant.

3. AND-OR PARALLELISM WITH RECOMPUTATION

In the presence of both and- and or-parallelism, it is possible to *reuse solutions* as argued for example in [14]. That is, given two non-deterministic and-parallel goals, say `a(X) & b(Y)` (where `&` denotes parallel conjunction), then rather than computing `b` in its entirety for every solution found for `a`, it would be much better if we first computed all solutions for `a` and `b` separately and then combined these solutions through a join[3]. Although this may apply to pure logic programs, it does not apply very well to logic programs which contain side-effects and other extralogical features (as it is so often the case with Prolog programs). For example, consider the case where, within `b`, the value of a variable is read from the standard input and then some action taken which depends on the value read. The solutions for `b` may be different for every invocation of `b` (where each invocation corresponds to a different solution of `a`). Hence solution sharing would yield the wrong results in such a case. The simple solution of sequentializing such and-parallel computations results in the loss of too much and-parallelism, because if `a(X), b(Y)` falls in the scope of some other goal which is being executed in and-parallel then that goal has to be sequentialized too, and we have to carry on this sequentialization process right up to the topmost level and-parallel conjunction. If, however, the goals are recomputed then this sequentialization can be avoided, and parallelism exploited even in the presence of cuts and side-effects. Hence, there is a strong argument for recomputing non-deterministic and-parallel goals, especially if they are not pure, and even more so if we want to support Prolog as the user language.[4] Recently, techniques have been proposed for exploiting and- and or-parallelism that recompute independent goals [12]. These techniques are based on an abstraction called the Composition Tree [15, 12]. Techniques have also been proposed for extending the environment representation methods, such as Binding Arrays [30] and Stack Copying [2], for obtaining practical and-or parallel systems based on the Composition Tree abstraction (The PBA model [15], and the ACE model [13]).

The *Composition Tree* essentially uses the idea of recomputing independent goals of a parallel conjunction. Consider the CGE `(true => a & b)`, the classical and-or tree for which is shown in figure 1(i). With recomputation, for every alternative of `a`, the goal `b` is computed in its entirety. In C-tree based models, each separate combination of `a` and `b` is represented by what we term as a *composition* node (c-node for brevity). Thus, each composition node corresponds to a different solution for the parallel conjunction, i.e., a different "continuation".

We explain how the Composition-tree (C-tree for brevity) is incrementally constructed, using the example in figure 1. Consider the conjunctive goal `a & b`, where `a` and `b` are independent. Both `a` and `b` have three alternative solutions.

A C-Tree corresponding to the execution of `(true => a & b)` is shown in figure 1(ii). Figure 2 explains how part of the C-tree is built. Note that in the figures solid lines represent computation. First, the non-determinism of goals `a` and `b` needs to be represented: a choice point is used to represent the fact that several alternative solutions can be extracted from `(true => a & b)`. The children of this choice-point

[3] For the sake of simplicity most examples we have chosen have parallel conjunctions with only two goals. The discussion here, as well as the techniques presented in the rest of the paper, also hold for cases where the parallel conjunction contains more than two goals.

[4] Recomputation of and-parallel goals was first considered in [19], but perhaps for different reasons.

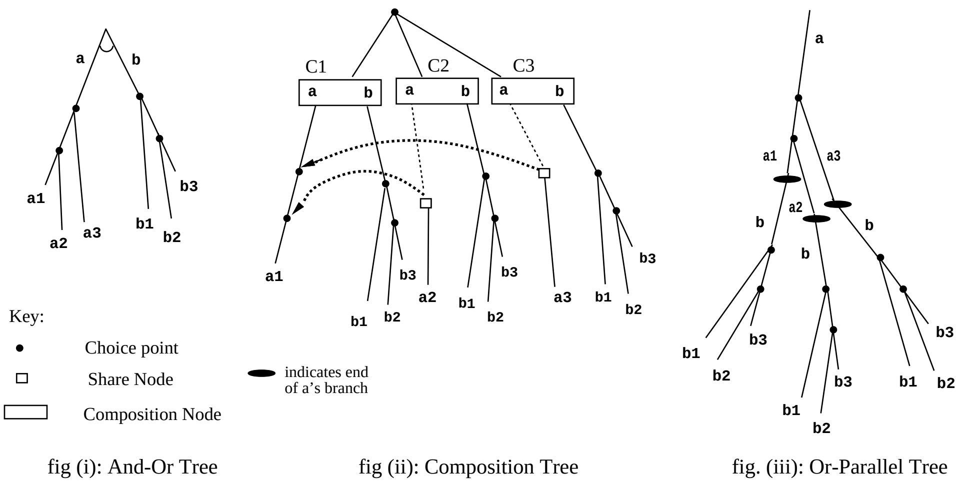


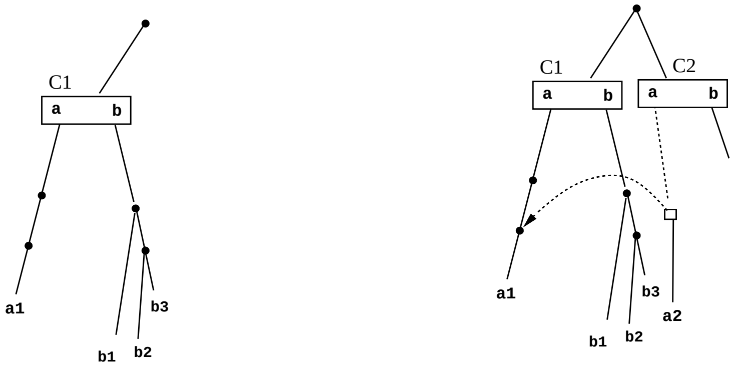
Figure 3. Composition Tree

are c-nodes, where each c-node corresponds to a combination of solutions of goals `a` and `b`. Let us suppose processors P1 and P2 begin execution of the top level query. They first create a top choice-point to represent non-determinism present in the CGE, and place as its first alternative the c-node `C1` under which they will execute `a` and `b` in and-parallel, just as in &-Prolog. This is represented in figure 2(i). As the goals `a` and `b` are executed, new choicepoints that lead to alternative solutions of goals `a` and `b` are created. Or-parallelism can be exploited by executing alternatives of these choicepoints in parallel. A new group of processors can, therefore, move in and pick up untried alternative of goal `a`. We have chosen to follow Prolog and recompute independent goals to the right, hence the goal `b` will be recomputed from scratch (Figure 2(ii)). The execution of the alternative of `a`, and the recomputation of goal `b` is done under a new c-node (node C2 in Figure 2(ii)). Also, note that part of the computation for `a` is logically shared between the two groups of processors[5]. The dotted arrow represents this sharing. The dotted arrow is emanating from what we call a *share-node* and points to the choicepoint, nodes above which are logically shared. The actual implementation of share nodes varies with the data structures being used to represent parallel execution, but in general share nodes must give access to the shared branches. The share node should also be accessible from the corresponding c-node and vice versa (depicted by the dotted line connecting the share-node to the c-node).

The final *Composition-tree* for the above query might appear as shown in figure 1(ii). Note that a third group of processors has created another c-node (C3 in Figure 1(ii)), to exploit the last alternative in goal `a`. Note also that the c-nodes serves purposes very similar to the Parcall frames and markers of the RAP-WAM [19].

The C-tree can represent or- and independent and-parallelism quite naturally: execution of goals in a c-node gives rise to independent and-parallelism while par-

[5]However, in an actual implementation, depending on the environment representation technique used, it may or may not be physically shared as well (see later).



(i) A Group of Processors Moves to Exploit And-Parallelism

(ii) New Group of Processors Exploits the Available Alternative in goal a

Figure 4. Constructing the Composition Tree

allel execution of untried alternatives gives rise to or-parallelism.

Both Prolog and a pure C-Tree based model will always recompute goals to the right. Therefore, they have the same search space, leading to a topological similarity between the C-tree and the purely or-parallel tree of the query [15, 12]. This similarity can be seen by comparing figure 1(ii) and figure 1(iii), that represents the purely or-parallel search tree for the program above[6]. Recomputation means that branches that are "shared" in the purely or-parallel tree (i.e., that are "common", even though different binding environments may still have to be maintained—we will refer to such branches and regions for simplicity simply as "shared") are also shared in the C-tree, and vice-versa. Due to sharing the subtrees of some independent and-parallel goals may be spread out across different composition nodes. Thus, the subtree of goal `a` is spread out over c-nodes C1, C2 and C3 in the C-tree of figure 1(ii), the total amount of program-related work being essentially what Prolog would perform.

A more complex case for nested CGEs is shown in Figure 3. In this case, goal `g` leads to a choicepoint from which another group of processors steals the alternative `g2`. As in the example above, a new c-node labeled G2 is created in which the second solution for `g` (i.e., `g2`) is found and the recomputation of subgoal `h` done. Once `g2` and `h` finish in G2, the continuation of `(f & g & h)`, i.e. `m`, is executed. Once `m2` is finished, effectively a solution for goal `c` in c-node C1 (Figure 3) has been found, and therefore, the group of processors must now recompute `d` for this solution of `c`. However, c-node C1 was reserved for computation arising out of the first solution of `c`, namely, the one that will give `m1`. One solution is to create a new c-node, labeled C2 in figure 3 to compute `d`. From this example one can notice that when an alternative is stolen from a choicepoint nested inside a number of c-nodes, then new c-nodes need to be created to exploit and-parallelism from each c-node

[6] To see the topological similarity, in Figure 1(ii) join all the share nodes to the choicepoint they point to and consider only the solid lines below the c-nodes—the or-tree shown in Figure 1(iii) will be obtained.

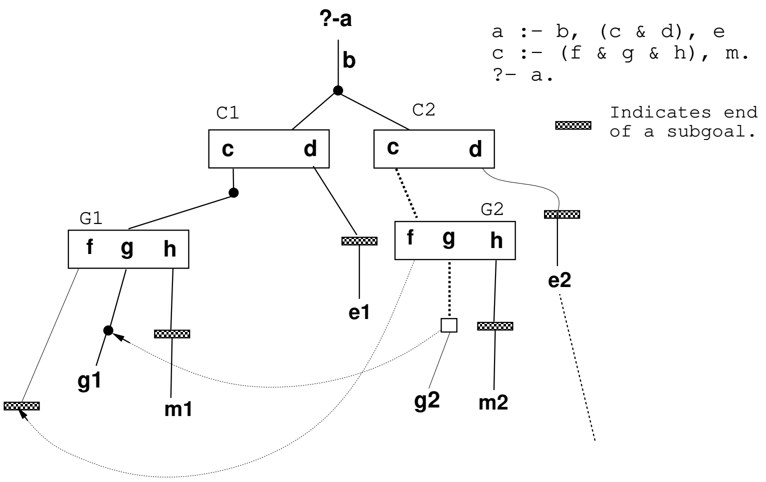


Figure 5. A more complex Composition Tree

whose scope includes that choicepoint. Thus, in this example both c-nodes G1 and C1 have to be recreated as c-nodes C2 and G2 respectively when an alternative is stolen from choicepoint in goal `g`. However, these new c-nodes need contain only information relevant to the restarted goals, because during backtracking if a share node is encountered, the processor will follow the dashed arrow and will start backtracking from the node pointed to by this share node. Thus, if a processor backtracks over `e2`, reaches c-node C2, then backtracks over `m2`, then `h`, and then `g2`, eventually reaching the share node below `g` in c-node G2, then it will immediately skip to the choicepoint above `g1` pointed to by this share node, and will continue backtracking from there. So there is no need to keep control information for goals `c` and `g` in share-nodes C2 and G2 respectively since that information, if kept, will never be used.

Throughout the previous explanation, we have commented that "groups" of processors would move to take alternatives and work in and-parallel. We now formalize this notion. In an and-or parallel system that uses the C-tree abstraction, processors are divided into *teams*.[7] Or-parallelism is exploited between different teams while and-parallelism is exploited among the processors in the teams. Note that the number of c-nodes that get created in the C-tree depends on the number of teams available for execution. Thus, in the figure 1(ii), three c-nodes were created since three teams were available (Each team works on a c-node and assigns independent and-goals to its member processors for execution). If there were five teams available the constructed C-tree will be as shown in figure 4 (Note that the equivalent or-tree shown to the right in figure 4 is unchanged). In the example we have chosen, a maximum of nine c-nodes may be created (since each of `a` and `b` lead to three solutions each). Thus, the amount of parallelism exploited is determined by the processing resources available. If enough processors are not available (e.g., for the C-tree in Figure 1(ii)), the remaining solutions are found by backtracking.

[7] The notion of teams was first proposed and exploited in Andorra-I system [8].

We have so far defined the C-tree as an abstraction of and-or parallel execution. Several implementations to the C-tree are possible. One can have the entire C-tree as physically shared and hence accessible to processors of all the teams (e.g., the PBA model [15, 12]). It is also possible to take a completely opposite view in which each team has its own copy of the C-tree resulting in the C-tree being logically shared but not physically shared (e.g., ACE model [13]). In any case, the C-tree must be *logically shared* for teams to share work. Indeed, to find or-work teams must be able to look at choicepoints with unexploited alternatives in the C-tree. As in or-parallel models, for a team to select work the team needs to move up and down in the C-tree, and pick an alternative from a choicepoint with work. Moreover, teams also need to know which tasks are available for and-parallel execution. Usually descriptors for the and-tasks are stored in the global stack, and they will be physically shared (in a sharing implementation) or just copied (in a copying implementation) when the team moves to the new work.

In the rest of this paper we show how side-effects can be supported in the C-tree abstraction. We first present a most general algorithm that is based only on the idea that the tree is logically shared. We then give concrete algorithms for supporting side-effects and cuts for a model that assumes that the entire C-tree is shared. The techniques for supporting side-effects developed for models where C-tree is physically shared can be readily adapted for models where the C-tree is not physically shared.

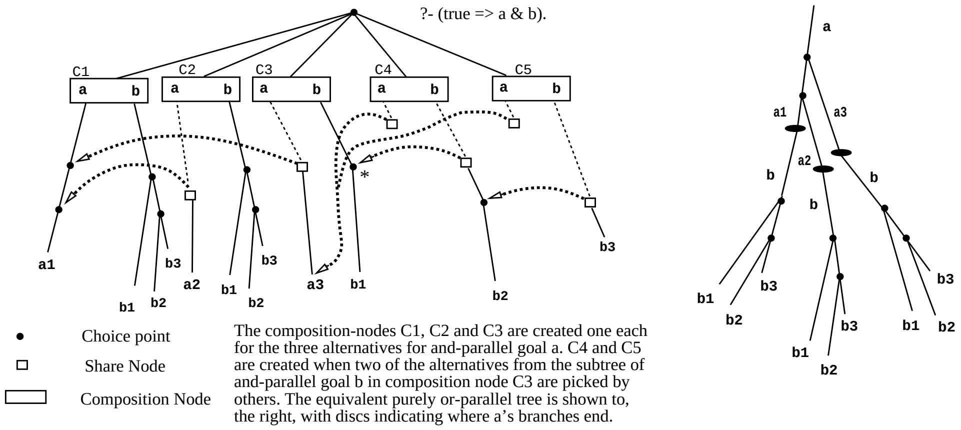


Figure 6. Composition Tree with 5 teams

We now define some terminology which we will repeatedly use in this paper. Given a node n in an and-branch b, the other and-branches which correspond to the same composition node (or c-node) as that of b are termed as *sibling and-branches of* n. The composition node is termed as *immediate ancestor c-node* of n, and the and-parallel goal corresponding to n is termed as the *and-parallel goal* of n. Thus, if we consider the node marked with an asterisk in figure 4, then c-node `C3` would be its immediate ancestor c-node, the subgoal `b` in `C3` its and-parallel goal, and the branch `a3` of `C3` its sibling and-branch.

4. SIDE-EFFECTS IN AND-OR PARALLEL SYSTEMS WITH GOAL RECOMPUTATION

In an and-or parallel system that uses goal recomputation, each or-parallel environment exists separately—they will only share those parts which will also be shared in equivalent purely or-parallel computation. There is no sharing of independent and-parallel goals (unlike the and-or parallel models that reuse solutions). This simplifies the problem of supporting side-effects considerably. Also, a dynamic scheme for supporting side-effects and cuts in and-or parallel systems should fare much better than a static scheme (the reasons must be obvious from discussions for pure cases in Sections 2.1 and 2.2). In the rest of this section we outline such a dynamic approach.

Recall that there is a 1-1 correspondence between the C-tree and the purely or-parallel tree of a query (figures 1 and 4). In the topologically equivalent purely or-parallel tree, given a CGE `(true =>` g_1 `&` ...`&` g_n`)`, then the or-parallel subtree of the goal g_i would appear immediately below that of g_{i-1}. Moreover, for every success branch of g_{i-1} the tree for g_i is duplicated (this corresponds to doing recomputation). We define the notion of "locally leftmost" to study the relationship between the *leftmostness* property in a C-tree and in a purely or-parallel tree. A node is *locally leftmost in a subtree* if it is in the leftmost branch in that subtree (i.e., all branches to the left of this branch have been backtracked over either due to failure or due to attempts to find more solutions after success). A subtree is *locally leftmost* in another subtree that contains it if the root-node of the smaller subtree is *locally leftmost* in the bigger subtree.

A sufficient condition for executing a side-effect according to Prolog semantics can now be given. When a side-effect is encountered in a C-tree, *the side-effect can be executed as soon as the associated node is in the leftmost branch of the topologically equivalent purely or-parallel tree.*

Consider a node n in the subtree of goal g_j in the CGE `(true =>` g_1 `&` ...`&` g_n`)`. Let b_1, b_2...b_{j-2}, b_{j-1} be the left sibling and-branches of node n. Node n is in the leftmost branch of the topologically equivalent pure or-parallel tree *if and only if*: (a) it is locally leftmost in the subtree of its and-parallel goal g_j; (b) branches b_{j-1}...b_1 are locally leftmost in the subtree of their respective and-parallel goals g_{j-1}...g_1; and (c) the c-node of n is similarly locally leftmost in the subtree of its and-parallel goal, its sibling and-branches to the left are locally leftmost in the subtrees of their respective and-goals, and so on, upto the root the search tree. Given that the C-tree is topologically equivalent to the purely or-parallel tree, if a node containing a side-effect satisfies the above condition, then we can safely assume that this side-effect is leftmost, that all side-effects before this side-effect have finished, and hence this side-effect can be safely executed. If not, the side-effect has to suspend, and wait until the condition is satisfied. Note that because we check for *local leftmostness* in the subtree of each and-parallel goal to the left, *we do not have to globally order the c-nodes explicitly.* Instead, global ordering is implicitly realized from the local ordering of c-nodes and their nesting.

We can now give the conditions for execution of a side-effect in a C-Tree based model. Given a CGE one of whose and-goals leads to a side-effect predicate, a side-effect can be executed:

(i). if the node containing the side-effect is locally leftmost in the subtree of its and-parallel goal; and,

(ii). if the sibling and-branches to its left are locally leftmost in subtrees of their respective and-parallel goals; and,

(iii). If the CGE is nested inside another CGE, then the c-node corresponding to the inner CGE must recursively satisfy rules (i), (ii) and (iii); otherwise, if the CGE is not nested inside another CGE, then the c-node corresponding to this CGE must be in the leftmost branch in its tree.

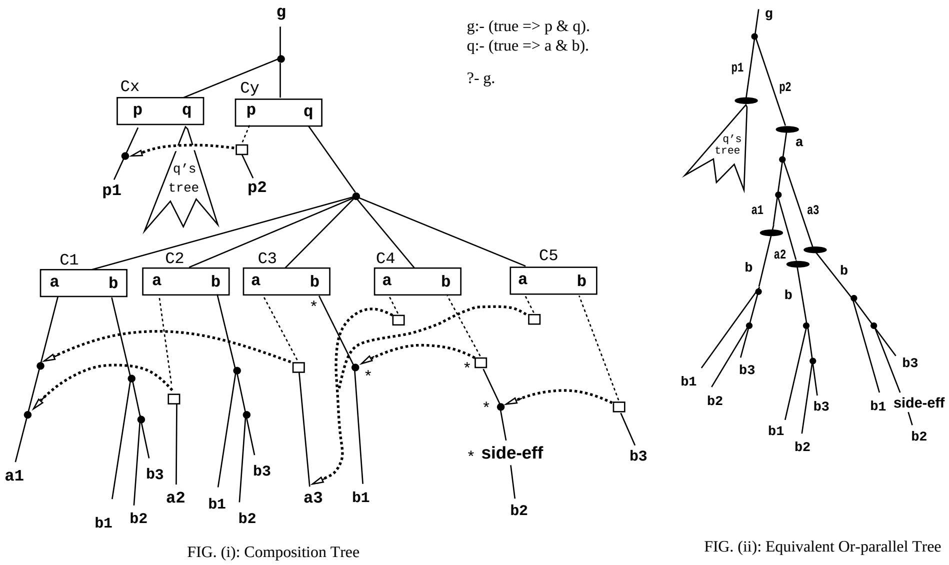


Figure 7. Executing Side-effect in Models with Goal Recomputation

The corresponding algorithm is shown below: n is the node containing the side-effect. The first step in the algorithm is to obtain the and-parallel goal, g, of n (g is represented by the root of the subtree corresponding to the and-parallel goal), followed by obtaining the immediate ancestor c-node of n (if n is not in scope of any CGE, the immediate ancestor c-node will be set to NULL). This code assumes that the function `tip_node` returns the leaf node of the finished branch for an and-parallel goal g_i. The implementation of the `WAIT` operation and of the functions `locally_leftmost`, `tip_node`, and `finished` depends on the environment representation scheme used. An implementation and ensuing optimizations for the model where C-tree is physically shared among all processors is discussed in Section 5.

```
check_leftmost(n) {
    g = and-goal(n);
    current-c-node = immediate-ancestor-c-node(n);
    while (current-c-node != NULL)
     { if  not(locally_leftmost(n, g))   WAIT;
       else
          foreach goal g_i to the left of g in the CGE
            if (not finished(g_i, current-c-node)
            OR  not check_leftmost_upto(tip_node(g_i, current-c-node), g_i))
               WAIT;
       n = current-c-node;
       g = and-goal(n);
       current-c-node = immediate-ancestor-c-node(n);
     }  /*end while*/
    if (locally_leftmost(n,ROOT))      /*ROOT is the global root*/
     execute side-effect; }
```

It is important to note that the subtrees of sibling and-branches to the left of the node which contains the side-effect may be spread over many composition nodes. That is its different or-branches may be parts of different and-parallel computations (c.f., figures 1 and 4). Nodes of such a subtree are accessed with the help of the *share* nodes which contain a pointer to their corresponding choice points (dotted arrows in figure 1 and 4).

Note that the property of a node being *"leftmost"* in g_j requires that execution of the sibling and-branches corresponding to goals $g_1 \ldots g_{j-1}$ is finished, otherwise "leftmostness" is meaningless. Thus, the condition for supporting side-effects in purely and-parallel systems, described above, is naturally included in our "leftmostness" property. The "leftmostness" property is illustrated further in Figure 1. In the example shown in figure 1, before the side-effect `side-eff` in the second clause of `b` corresponding to branch `a3` can execute, the branch containing this side-effect should become leftmost in the purely or-parallel tree (Figure 1(ii)). In the C-tree this corresponds to the branch containing the `side-effect` (marked by asterisks in the figure) becoming locally leftmost in the subtree of goal `b` spread across composition nodes `C3` and `C4`. Also, branch `a3` (the left sibling branch of the branch containing `side-eff`) should become locally leftmost in the subtree spread across composition nodes `C1`, `C2`, and `C3`. This is because c-node `C4`'s subgoal `a` has a share node which points to the terminal node of branch `a3`. In addition, since c-nodes for the parallel conjunction `a & b` are nested within that of `p & q`, the branch `p2` should become locally leftmost in the subtree of goal `p` spread across c-nodes `Cx` and `Cy`.

Note also that we have not given details on the implementation of the functions `finished()`, `immediate-ancestor-c-node()`, `locally_leftmost()`, and `and-goal()`. In general, the implementation of these instructions will depend on the actual model we use to represent the C-Tree, such as copying in ACE or the Paged Binding Array, and we give an example in the following section. The `check_leftmost_upto()` is similar to `check_leftmost` but it will stop when `current_c_node` equals the second argument, instead of stopping when it reaches NULL. The `WAIT` operation can either be implemented as a busy-wait or, preferably, as suspension (i.e., the processor suspends and goes elsewhere in the tree to do some other useful work). If suspension is used then on resumption the leftmostness check may be begun afresh, or if the appropriate information regarding the point at which suspension took place, is recorded at the time of suspension, the check can resume from the point where it was suspended.

To clarify further, consider a program whose composition tree is shown in Figure 2(i). Suppose there are only two points where a side-effect is present in the program (shown as `s1` and `s2` in the C-tree). Suppose two teams of processors, `T1` and `T2` respectively, executing c-nodes `C1` and `C2` in parallel, reach the side-effect `s1` and `s2` respectively at the same time. Given the state of the computation (Figure 2(i)), side-effect `s1` will suspend because its sibling branch to the left is not complete yet. Side-effect `s2` will also suspend because although it is locally leftmost in the subtree of goal `b` its left sibling and-branch is not locally leftmost in the subtree of goal `a` (note that the subtree of goal `a` is spread over c-nodes `C1` and `C2`). When branch `a1` completes (Figure 2(ii)), side-effect `s1` can be executed immediately. However, side-effect `s2` cannot be executed because `a2` is still not locally leftmost in `a`'s subtree. After team `T1` has found all solutions for goal `b` (note that the continuation of the CGE is executed after every solution that is found for `b`) under c-node `C1`, it will backtrack into branch `a1`. As soon as `T1` reaches the lowest choicepoint in branch `a1` (labeled `ch` in Figure 2) while backtracking, branch `a2` becomes the leftmost branch in the subtree of goal `a`, and side-effect `s2` can be executed (figure 2(iii)). Figure 2(iv) shows the equivalent or-parallel tree.

5. ALGORITHMS FOR SUPPORTING SIDE-EFFECTS

In this section we present concrete algorithms for supporting side-effects and cuts in a parallel logic programming system that is based on the C-tree, and in which the entire C-tree is assumed to be physically shared among all processors of the multiprocessor system (thus the target architecture for such a system is a shared memory multiprocessor). The PBA model [15, 12] is an example of such a system. As mentioned earlier, in such models the concept of teams of processors is used to implement and-parallelism. The various teams—each team consisting of one or more processors—work in or-parallel to search for multiple solutions, while within a team, the member processors work co-operatively, in and-parallel to generate a specific solution. In this section we describe in detail how side-effects and cuts can be efficiently implemented in the models that assume the C-tree to be physically shared. These techniques can be readily adapted for models where the C-tree is assumed not to be physically shared[8].

To incorporate side effects in a model in which the C-tree is physically shared we should be able to check the "locally leftmost" property of nodes efficiently. This is indeed possible due to the presence of *c-nodes* and *share* nodes. If the entire C-tree is accessible, a given and-parallel goal can access the sibling and-branches to its left via the c-nodes (that are shared). C-nodes also keep track of the individual activities of each and-parallel goal, hence it is easy to determine when all sibling

[8] Outline of algorithms for supporting side-effects in a model that does not assume a physically shared C-tree can be found in [13].

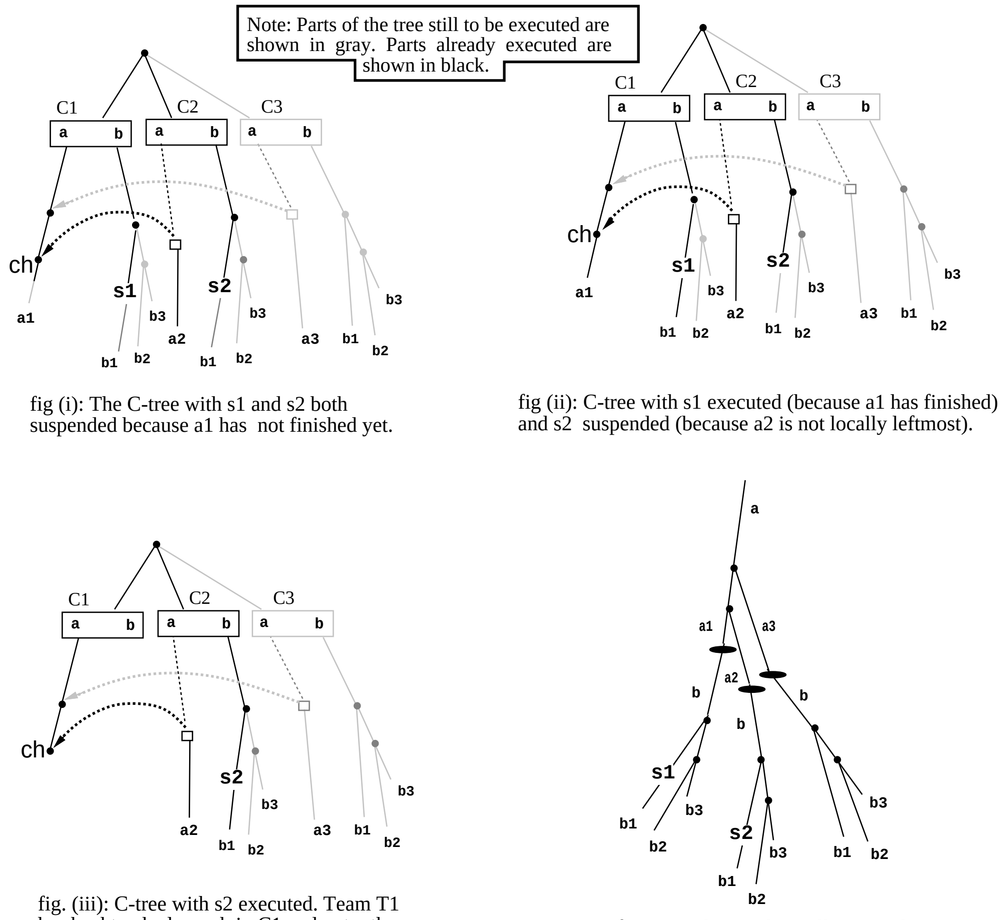


fig (i): The C-tree with s1 and s2 both suspended because a1 has not finished yet.

fig (ii): C-tree with s1 executed (because a1 has finished) and s2 suspended (because a2 is not locally leftmost).

fig. (iii): C-tree with s2 executed. Team T1 has backtracked over b in C1 and upto the choicepoint marked ch in branch a1 so that branch a2 is now locally leftmost in a's subtree.

fig. (iv): The topologically equivalent Or-Parallel Tree for the computation.

Figure 8. Side-effect in Models with Goal Recomputation (cont'd)

goals to the left of an and-parallel goal have finished. Moreover, in shared C-tree model a marker node called the *end-marker* has to be added to the stack whenever the end of an and-parallel goal in a CGE is reached. It marks the tip of the and-branch of an and-parallel goal of a CGE. A pointer to the end-marker of a goal in a CGE is also recorded in the slot corresponding to that and-parallel goal in the c-node (in the `end-marker-ptr` field). This information is used by the function `tip_node`. Until a solution is found for the goal, the end-marker-ptr field in the c-node for that goal contains NULL. The implementation of `tip_node`, `finished`, and `locally_leftmost` in algorithm `check_leftmost` therefore becomes:

tip_node(g, c-node)
{ return(c-node(g)→end-marker-ptr); }

finished(g, c-node)

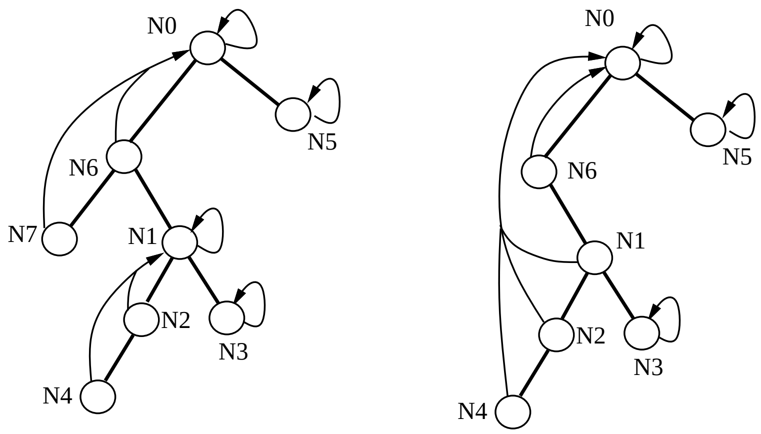
```
    { return(c-node(g)→end-marker-ptr != NULL ); }

locally_leftmost(n, g)
    { return(subroot-node(n) == g); }
```

In the code above `c-node(g)` denotes the record where the status information regarding goal `g` is kept. Thus, `c-node(g)→end-marker-ptr` returns the pointer to end-marker node for the goal corresponding to `g`. Also note that the immediate ancestor composition node of `n` can be obtained from the CN (composition-node) register, which keeps track of this information (like the PF register of RAP-WAM [19]). Finally, note that in the C-tree, the subroot-node for a choice-point is always bounded by its immediate ancestor c-node, i.e., the current subroot-node is reset whenever a new and-task is started.

We further need to address how to efficiently keep track of the subroot node of a given node within the search tree of an independent and-parallel goal. As we said, the subroot-node for a node must be found within the search tree starting from the node and going at most up to the first CGE, which is a pure or-tree, and therefore this can be done in a way similar to that used by purely or-parallel systems, such as Aurora (an extensive discussion can be found in [17, 16]). We describe the technique used by Aurora's Dharma scheduler [29]. Other schedulers, such as the Bristol scheduler [6] use similar ideas. Each node n has a field called the *subroot-node* field which contains a pointer to the highest node r in whose subtree n is in the leftmost branch. For example, in figure 1(i) the subroot node for `N6` would be pointing at `N0`. When a node m spawns children nodes, the first child node sets its subroot node field to that of the subroot node field of m, while others set it pointing to themselves. When a node is determined to be a subroot node of a node k, a bit is set in the parent node of that subroot node (i.e in the leftjoin node of k). Referring to figure 1, the processor expanding node `N1` would set the bit for `N6`. When a processor backtracks to a node which is a leftjoin node of another node, after exploring a branch and reclaiming it, and it finds this bit set, it resets this bit. In the example, after backtracking from `N7`, the processor would reset the bit. The next time a processor checks for "leftmostness" of a node, whose leftjoin node's bit is reset, it recomputes its subroot node because its subroot node may have moved higher up. This is shown in figure 1(ii), where the processor expanding node `N4` needed to recompute its root-node because `N6`'s bit has been reset, thus finding that its true root-node is `N0`. To recompute the subroot-node, each node has to know whether or not it is the leftmost child of its parent node. This information is kept by keeping a *sibling chain* [6] which links the various children nodes of a node from left to right. Each node also has a pointer to its leftmost child node. When a processor determines a subroot node for a node, it sets the subroot-node field of all nodes in the path from that node to the subroot node to point to that subroot-node.

5.1. Optimising the Basic Algorithm for a Shared Memory Space Model

It must be apparent from the previous discussion that essentially we are trying to determine "global" leftmostness of a side-effect (i.e., if the node is in the leftmost



(i): An or-subtree adorned with subroot node information. Each node also has a pointer to its subroot node.

(ii): Subroot node information after removal of branch N6-N7.

Figure 9. Subroot Node

branch with respect to the entire tree) by determining local-leftmostness in all the intervening CGEs between the root-node and the node containing the side-effect. We determine local-leftmostness by requiring the nodes (choice points and c-nodes) to keep track of their subroot nodes in the subtrees corresponding to their immediate ancestor and-parallel goals. Thus, the global-leftmostness check consists of lots of local-leftmostness checks. One can optimize the determination of global leftmostness by having direct access to the subtree node of a node with respect to the entire tree (i.e., by keeping track of the subroot node of a node in the topologically equivalent purely or-parallel tree). This will make the global leftmost check quite simple (just a simple comparison to see if the subroot node is identical to the root of the entire tree). However, the overhead is that the subroot node information for each node has to be updated.

To implement this we use a new field in the c-nodes. We call this field c-node(g)[n]$\rightarrow$subroot-node, with one element per goal in the corresponding CGE. We also add a register SRB, "SubRoot Node," that points to the subroot node of the current processor's node. The algorithm for updating the subroot node can now be extended in the way described below.

Essentially, we follow the algorithm for the purely or-parallel case, except where c-nodes are concerned. In the case of a c-node, we have to handle the three new situations: first, when a c-node for a CGE is set up; second, when an and-task finishes (a solution to an and-goal of CGE is found); and, third, when a CGE is backtracked into. The extensions to the algorithm are thus:

- *Creation of C-node:* There are two cases, first, when the very first c-node is created for a CGE, and second, when a c-node is created as a result of a team stealing a choice point from below another c-node. In the first case, the leftmost and-goal of the c-node inherits the subroot node of the parent node of the c-node and the subroot nodes of other and-goals are set to point to themselves. That is, c-node$\rightarrow$ g_0(subroot-node) := SRB, where SRB is pointing to subroot node of c-node's parent node; the others are set to point to themselves. When the first choice-point in an and-goal is created, its subroot node field is assigned from the subroot-field of the goal entry in the corresponding c-node. In the second case, the c-node is created as a result of stealing an alternative from a choice-point below an existing c-node. There are three types of goals whose subroot nodes have to be computed: (i) the goal g_j from where the alternative was selected; (ii) the goals g_i to the left, i.e., $i < j$; (iii) the goals g_i to the right, that is, $i > j$. The goals g_i such that $i < j$ have already been executed by other teams, hence the stealing team need not set c-node$\rightarrow$ g_i(subroot-node) for these goals. The goal g_j corresponds to the new alternative just picked. This alternative can never be the leftmost, hence the subroot node for the corresponding computation will be the first alternative created, and its SRB should be set to itself (note that the share-node for g_j will be its leftjoin node). Finally, for each goal g_i such that $i > j$ (goals to the right) c-node$\rightarrow$ g_i(subroot-node) should be set pointing to itself.

- *End of an And-goal:* Whenever an and-goal computation terminates, its nearest right and-sibling uses the subroot node information of the terminated and-goal to update its own subroot node information. That is, if a solution is found for and-goal g_i then c-node$\rightarrow g_{i+1}$(subroot-node) := SRB, where SRB points to the subroot node of the end-marker node of g_i.
- *Backtracking into a CGE:* When backtracking takes place through a CGE, the and-siblings to the right of the task to where one backtracks must be restarted; the subroot-node fields in the c-node corresponding to these restarted goals are set pointing to themselves.

Verifying if a side effect can be executed is a simple question of verifying if its and-or subroot node is the root of the whole tree. Also, optimizing access to the and-or subroot node can be made according to the principles described for the several Aurora or-schedulers. Therefore, in the best case we can now verify leftmostness in constant time (if we are pointing directly at the subroot node), whereas in the worst case we are still linear on the number of ancestor nodes.

Note that now the overhead of determining (global) leftmostness is identical to that in Aurora [17, 29]. This is not surprising since the steps (i), (ii) and (iii) above are really keeping track of the subroot node information in the equivalent purely or-parallel tree. Note that with respect to purely and-parallel computation our optimized algorithm using the concept of subroot-node permits us to check if a side-effect can be executed (hopefully) more efficiently compared with algorithms presented in [7, 22]. Our optimized algorithm applied to purely and-parallel systems can be viewed as an optimization of the scheme presented in [7].

6. PRUNING IN RECOMPUTATION BASED MODELS

Implementing cuts in and/or-parallel systems follows the same principles as implementing side-effects except that some of the actions (i.e., part of the pruning) can be taken immediately (figure 4). The desired effect of a cut is that all branches to the right, up to the node where the cut was introduced, get pruned. However, a problem can arise if the cut is not leftmost: for example, another cut to the left of this cut might eventually succeed and prune this cut itself, as shown in figure 1. This will happen when the scope of the second cut (that is, the choice-point up to which it will prune) is below the scope of the first cut, causing some branches to get pruned incorrectly (see figure 1 or see [16] for more details). Note that the branches pruned by a cut that lie in the subtree in which the branch containing the cut is leftmost, can be immediately removed. Pruning of other branches may need to wait until after the branch containing the cut becomes globally leftmost.

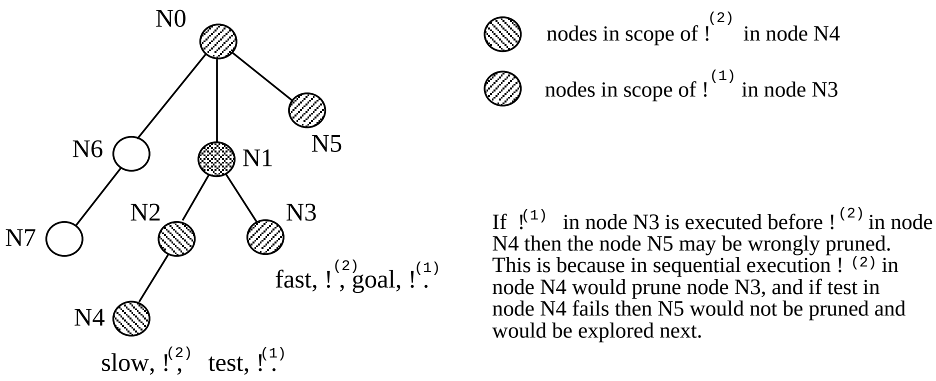


Figure 10. Executing a Cut in an Or-tree

6.1. Pruning in Purely Or-Parallel Models

In the purely or-parallel case[9], pruning at a choice point node n is performed by a processor p by discarding the right alternatives of the node n, verifying if n is the last node to prune, and if not, moving to n's parent node to carry on further pruning. The problem thus is knowing when can processor p move up and carry on pruning without being pruned itself (i.e., without the pruning of the branch that p is working on) by a cut in a branch to the left. One simple solution is to wait until the branch containing n is locally leftmost with respect to n's parent node: in this way p can never prune branches incorrectly. However, in such a case p will not prune as many or-branches as quickly as possible, and will risk wasting processor cycles because other processors will be expanding these or-branches which will eventually get pruned.

The Aurora or-parallel system treats cuts in a manner different from side-effects. As mentioned earlier, part of the operations associated with a cut can be performed as soon as it is encountered rather than when it becomes leftmost. This is based on the observation that a cut only affects the shape of the or-tree. The Aurora system, therefore, immediately performs many of the actions associated with a cut [17] when one is encountered. Essentially, all the pruning that can be done *safely* is performed immediately. This is accomplished as follows: a *cut boundary flag* ($CutB$) and a cut counter ($CutC$) is associated with each or-node (choice point). A cut boundary flag indicates if the corresponding procedure calls a cut; cut counter

[9] Note that pruning in purely and-parallel models is not as involved as in purely or-parallel models. Discussion of pruning in and-parallel models can be found in [26, 28].

indicate how many cuts the node has in its continuation (thus, the higher the cut counter, the more pending cuts to perform). Each processor also maintains a global cut counter that keeps track of how many cuts are pending in the or-branch the processor is currently executing. A processor decrements its global cut counter every time it executes a cut. Also, when a processor encounters a clause containing n cuts, it increments its cut counter by n. A cut counters of a node is obtained from the current value of the cut counter of the processor that created it. Once the cut counter of a node is initialized it is never changed.

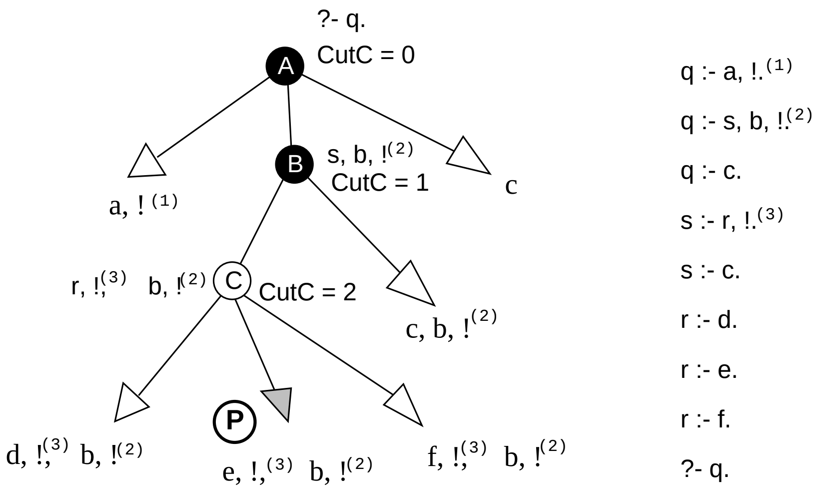
When a processor encounters a cut it should ideally cut all branches to the right up to the point where the cut was introduced (its cut boundary node). However, if this cut is not in the leftmost branch, then incorrect pruning may result since this cut may itself be pruned by a younger cut to the left. Thus, a cut should prune only those branches which will get pruned even if the cut in question was itself pruned by another cut to the left. The cut counter is used to recognize such branches. When a processor executes a cut, it checks if any of the nodes *between* the current node (the node where the cut is being executed) and the cut boundary node of this cut:

- have alternatives to the left and a cut counter value *exceeding* that of the processor's;
  or
- have alternatives to the left and are marked with the cut boundary flag.

If either of the above condition holds, the processor must postpone completion of the "higher" part of the cut. By completion of higher part of the cut we mean pruning of branches above the lowest such node where one of the above conditions holds true. When the processor's cut counter value is less than that of node's and it is not in the leftmost branch with respect to the corresponding cut boundary node, it means the there may still exist a younger cut to the left. An older cut is always in the danger of getting pruned itself, if it is not in the leftmost branch with respect to its cut boundary node.

Figure 2 shows an example of pruning in a search tree decorated with cut counters and cut boundary flags. The fastest processor, p, is shown shaded. Processor p tries first to execute $!^{(3)}$, that is, to prune alternatives for nodes B and C. Node B is $!^{(3)}$'s cut boundary node. Although not leftmost for C, p's $CutC$ has the same value as the C's $CutC$, and C's cut boundary flag is false, therefore p can prune alternatives in C to the right. Moving up, $!^{(3)}$'s cut boundary node, namely B, is reached, therefore p can also prune branches to the right in B.

The state after pruning B and C is shown in figure 3. Note that after executing $!^{(3)}$, p has a single cut to execute, and its cut counter is decreased to 1. Imagine now that again p executes `b` very quickly and tries to execute $!^{(2)}$ next. Processor p is still not leftmost at C, and because its cut counter is smaller than B's $CutC$, p cannot try to prune any higher. This corresponds to the branch to the left executing $!^{(3)}$, and pruning p's branch, but then failing in `b`, hence preventing execution of $!^{(2)}$. Only if `d` fails meanwhile, making p's branch leftmost, will p be able to prune at node A.



The figure shows the or-parallel execution for a query q. Black nodes are nodes with cut boundary flag set. For each node the goal list and the value of the cut counter is shown. Arrows represent active processors. The processor corresponding to the shaded arrow executes !(3) and !(2) before any other cut gets executed.

Figure 11. Executing a Cut

6.2. Pruning in And-Or Parallel Models with Goal Recomputation

Having described how a cut is implemented in a purely or-parallel system, we now explain its implementation in an and-or parallel system based on C-trees (note that implementation of cuts in purely and-parallel systems is fairly straightforward and, therefore, we do not discuss it here). The techniques are an extension of those developed for purely or-parallel systems (described above), such as Aurora, by Hausman et al [17]. Once again we make use of the topological similarity between the purely or-parallel tree and the C-tree of a query, which allows us to directly import all the techniques and algorithms developed for a purely or-parallel model into our and-or parallel model. Thus, to support cuts in the C-trees, we maintain a cut counter and a cut boundary flag in each choicepoint node and a cut counter for each processor. All the actions that are needed for executing the cut in the purely or-parallel tree are mapped directly into the C-tree. Essentially, all nodes in the C-tree corresponding to the nodes that would be in the scope of the cut in the equivalent purely or-parallel tree would be cut (see Figure 4). The algorithm that accomplishes this is given below. The only complication arises when a execution of a cut causes pruning in an and-parallel goal in a CGE and while moving up to prune further, the pruning team encounters a c-node. Pruning in this case should proceed into the and-parallel goal to the left.

```
cut(n, cut_level) {
  while (n != cut_level) {
     { n→alternatives_available = false;
       if (exists n→right_sibling)
         prune_trees(n→right_sibling)
       p = n→parent;
       if (p is a c-node) {
```

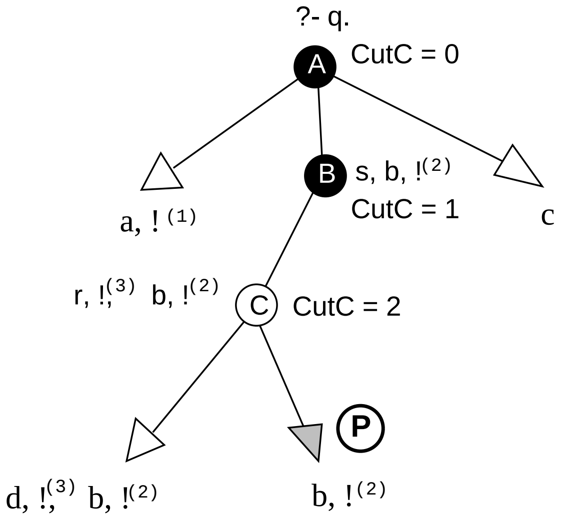


Figure 12. Executing a Cut (cont'd)

```
      g_i = and-parallel goal of n;
      prune_c_node(p, g_i);
      n = c-node→parent;}
    else if ( (exists n→left_sibling) &&
      ( (p→CutB== TRUE) || (p→CutC > CutC) ) )
     WAIT;
    else n = p;
    } }
```

```
prune_c_node(up_to, g_i) {
  if (g_i is not leftmost in the c-node)
        if (g_{i-1} has finished)
          {n = bottom-most choice point in g_{i-1};
          cut(n, up_to); }
        else
          WAIT;}
```

```
prune_trees(n) {
      prune_tree(n);
      if (exists n→right_sibling)
          prune_trees(n→right_sibling); }
```

The procedure `cut(n, cut_level)` is invoked when a cut is encountered during execution; `n` is the node in which the cut is encountered and `cut_level` is the level in the tree up to which pruning has to be done. The algorithm treats or-nodes as discussed previously, and guarantees the correct pruning of alternatives below c-nodes. Note that `WAIT` can be implemented as a busy wait, i.e., the team waits for the `if` condition to become true, or as a suspension, i.e., the team suspends

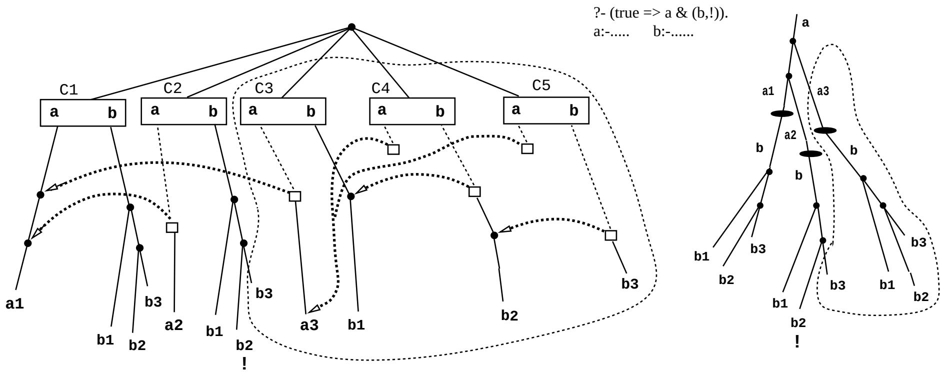


FIG. (i): Composition Tree

FIG. (ii): Equivalent Or-parallel Tree

Consider the cut after b2 corresponding to solution a2 for a. The scope of this cut is shown by the dotted closed area, both for the C-tree as well as for its equivalent or-parallel tree. Nodes can be pruned only upto the point where the "locally leftmost" check succeeds. Thus, only node b3 can be pruned. The others will be pruned only after c-node C1 and branch b1 in c-node C2 have finished. The information that some pruning may be needed later, is recorded in the subroot node of b2 in subtree of goal b of c-node C2.

Figure 13. Implementing Cut in the PBA Model

execution to find work elsewhere and this cut is retried later (possibly by another team). Also, note that the invocation of `prune_tree(n)` in `prune_trees` interrupts all the teams active in a subtree rooted at `n`, the subtree is removed, and the teams look for new work outside the pruned part of the tree.

Note that, as mentioned, a cut in one and-parallel goal may also need to prune subtrees of sibling and-goals to the left. This can happen if:

(i). The continuation of a CGE contains a cut, for example, in `(true => a & b & c), d, !, ...` In such a case when the cut is executed, since we are already in the continuation of the CGE we are assured that at least the leftmost branches of subtrees of goals `a`, `b`, and `c` have completed. Thus, when pruning reaches `c`'s subtree, it is performed in the normal manner. Once pruning of `c`'s subtree is done, it proceeds into the subtree of and-goal to the left, namely `b`, and so on.

(ii). The CGE contains a cut, for example, in the CGE `(true => a & b & (c, !) & d)`, execution of cut after goal `c` should also prune branches in goals `a` and `b`. In this case, either the c-node is complete and we can prune as long as we are leftmost, or and-tasks are still executing. Some optimizations are then possible. First, pruning in `b` may be performed as soon as `b` reaches a solution regardless of `a`. Second, if the subgoal `b` does not contain any side-effects, pruning of `a` may be performed as soon as `a` completes, that is without having to wait for `b` (this is because either `b` succeeds and hence pruning should be done, or `b` deterministically fails, the entire computation fails, and hence doing pruning will not affect the final result).

7. CONCLUSIONS

In this paper we argued that supporting full Prolog is crucially important in an and-or parallel logic programming system. We showed that, in general, dynamic approaches to supporting side-effects and cuts require no user intervention and exploit more parallelism compared to static approaches that partially sequentialize logic programs or require the programmer to divide a program into sequential and parallel parts.

We also argued that it is easier to support full Prolog in models that use goal recomputation rather than those that reuse solutions. We presented an abstraction called Composition Tree that permits and-or parallel execution with goal recomputation. We presented complete and efficient techniques for supporting extra-logical predicates in an and-or parallel model based on the Composition-tree abstraction on a shared memory multiprocessor. Our techniques rely on proven and efficient techniques used for purely or-parallel and purely and-parallel systems, and hence we believe they will be quite efficient. The underlying principles of our approach to supporting side-effects and cuts in and-or parallel systems are quite general and can be applied to other models as well, even to those that have dependent and-parallelism, such as Prometheus [26] and IDIOM (with recomputation) [11].

Acknowledgements

We are grateful to David H.D. Warren for his support, and to Raéd Sindaha and Tony Beaumont for answering many questions about Aurora and its schedulers. Thanks also to Manuel Hermenegildo for numerous discussions that have greatly contributed to this research. Special thanks are due to Enrico Pontelli for a thorough reading of the paper and for his suggestions for improvements. We also would like to acknowledge the anonymous referees; their comments improved the paper considerably. This research is supported by NSF Grants 92-11732 and 93-53271, Grant AE-1680 and AI-1929 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, by NATO Collaborative Research Grant CRG921318, and by an Esprit fellowship from the European Commission to Vítor Santos Costa while he was at the Department of Computer Science of the University of Bristol.

REFERENCES

1. K.A.M. Ali. A Method for Implementing Cut in Parallel Execution of Prolog. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 449–456. Computer Society Press of the IEEE, 1987.
2. K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 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. S. Auwatanamongkol, A. Ciepielewski, and P. Biswas. Cuts and Side-Effects in a Data-Driven Implementation of Prolog. Technical report, Southern Methodist University, 1990.
5. Uri Baron et al. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proceedings of International Conference on Fifth Generation Computer Systems '88*, pages 841–850, 1988.
6. T. Beaumont, S. Muthu Raman, and et al. Flexible Scheduling or Or-parallelism in Aurora: The Bristol Scheduler. In *Parallel Architectures and Languages Europe '92, LNCS 506*, pages 403–420. Springer Verlag, 1991.

7. S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368, 1989.
8. 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 *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.
9. V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.
10. D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.
11. G. Gupta, V. Santos Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism. In *International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.
12. G. Gupta, M. Hermenegildo, and V. Santos Costa. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.
13. G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.
14. G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.
15. Gopal Gupta and Vítor Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.
16. B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.
17. Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.
18. 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.
19. M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.
20. L. V. Kalé, D. A. Padua, and D. C. Sehr. OR-Parallel execution of Prolog with side effects. *The Journal of Supercomputing*, 2(2):209–223, 1988.
21. E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3):243–273, Nov. 1990.
22. 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.
23. 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.
24. B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

25. M. Ratcliffe and J. C. Syre. A Parallel Logic Programming Language for PEPSys. In *Proceedings of IJCAI'87*, pages 48–55. Milan, 1987.
26. K. Shen. *Studies of And/Or Parallel Prolog*. PhD thesis, University of Cambridge, 1992.
27. K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. Technical Report FIM-60.1/AI/90, Facultad de Informatica, U. P. Madrid, 28660 Boadilla del Monte, Madrid, Spain, October 1990.
28. K. Shen and M. Hermenegildo. Divided We Stand: Parallel Distributed Stack Memory Management. In *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994. E. Tick and G.Succi (eds).
29. R. Sindaha. The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture. In *Proc. of 4th IEEE Symposium on Parallel and Distributed Processing 1992*, pages 296–303, 1992.
30. D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.