And-Or Parallelism on Shared-Memory Multiprocessors†

Gopal Gupta

Department of Computer Science
New Mexico State University
E-Mail: gupta@nmsu.edu
Tel: 505-646-6236

Bharat Jayaraman

Department of Computer Science
SUNY at Buffalo, NY 14260, U.S.A.
E-Mail: bharat@cs.buffalo.edu
Tel: 716-645-3194

Abstract

This paper presents an extended and-or tree and an extended WAM (Warren Abstract Machine) for efficiently supporting both and-parallel and or-parallel execution of logic programs on shared-memory multiprocessors. Our approach for exploiting both and- and or-parallelism is based on the binding-arrays method for or-parallelism and the RAP (Restricted And-Parallelism) method for and-parallelism, two successful methods for implementing or-parallelism and and-parallelism respectively. Our combined and-or model avoids redundant computations when goals exhibit both and- and or-parallelism, by representing the crossproduct of the solutions from the and-or parallel goals rather than re-computing them. We extend the classical and-or tree with two new nodes: a 'sequential' node (for RAP's sequential goals), and a 'crossproduct' node (for the crossproduct of solutions from and-or-parallel goals). The paper also presents an extension of the WAM, called AO-WAM, which is used to compile logic programs for and-or parallel execution based on the extended and-or tree. The AO-WAM incorporates a number of novel features: (i) inclusion of a base array with each processor's binding-array for constant-time access to variables in the presence of and-parallelism; (ii) inclusion of new stack frames and instructions to express *solution sharing*, (iii) novel optimizations which minimize the cost of binding-array updates in the presence of and-parallelism.

---

† This is an expanded version of the paper, "Compiled And-Or Parallel Execution on Shared Memory Multiprocessors," In *Proc. N. American Conference on Logic Programming*, Cleveland OH, 1989.

1. Introduction

There are three main forms of parallelism in logic programming languages:

1. *or-parallelism*,
2. *independent and-parallelism*, and
3. *dependent and-parallelism.*

Or-parallelism arises when more than one rule defines some relation and a procedure call unifies with more than one rule 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. *Independent and-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another, i.e., their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. And-parallelism is thus a way of speeding up a divide-and-conquer algorithms by executing the subproblems in parallel. *Dependent and-parallelism* arises when two or more goals in the body of a procedure have a common variable and are executed in parallel. Dependent and-parallelism can be exploited in two ways: (i) the two goals can be executed independently until one of them accesses/binds the common variable; (ii) once the common variable is accessed by one of the goals, it is bound to a structure, or *stream* (the goal generating this binding is called the *producer*), and the structure is read as an input argument of the other goal (called the *consumer*). Case (i) is very similar to independent and-parallelism. Case (ii) is sometimes also referred to as *stream-parallelism* and is useful for speeding up producer-consumer interactions found in system programs, by allowing the consumer goal to compute with one element of the stream while the producer goal is computing the next element. Stream-parallelism forms the basis for Committed Choice Languages (e.g., Parlog [5], GHC [34], and Concurrent Prolog [30]).

In this paper we have chosen to focus on or-parallelism and independent and-parallelism for two reasons: (i) we believe that or-parallelism and independent and-parallelism tend to occur more commonly at the applications level; and (ii) dependent and-parallelism together with or-parallelism is somewhat more difficult to realize in an implementation.‡

We note at the outset that the target machines of interest to us in this paper are shared-memory rather than distributed-memory multiprocessors. The reason for this fo-

‡ Recently, however, attempts have been made to exploit or-parallelism together with dependent and-parallelism [15, 17, 32, 39].

cus is that, given the dynamic nature of memory access in logic programs, we believe that the performance during parallel execution would be better on shared-memory multiprocessors rather than on distributed-memory multiprocessors, where access to remote memories is expensive. Because it is likely that there will be large units of essentially sequential execution during parallel execution, it is very important that we be able to take advantage of implementation techniques for sequential execution, i.e., the WAM (Warren Abstract Machine) [38]. In this connection, it may be noted that we can readily adapt the WAM for shared-memory machines. Thus, our approach contrasts with approaches such as the Reduce-Or Process Model (ROPM) [22], Limited–OR/Restricted-And Parallel Model (LORAP) [2] and others which are primarily designed for non-shared memory multiprocessors.

While there are many opportunities for or-parallelism and and-parallelism in logic programs, realizing them in an actual implementation poses significant challenges. Therefore, before describing our proposed approach, we present in the next section the major problems to be addressed and objectives that should be met by any and-or parallel execution system. We first discuss the problems in exploiting pure or-parallelism and pure and-parallelism separately, and then discuss the problems in combined and-or parallelism. Briefly, we propose three criteria for or-parallel implementations: constant-time access to variables, constant-time task-creation, and constant-time task-switch; and three criteria for and-parallel implementations: avoid wasteful over-computation (e.g. back-unification [35]), avoid complex run-time dependency analysis, and support intelligent backtracking. The criteria for a combined and-or parallel implementation are essentially the union of the criteria for pure or-parallel and pure and-parallel implementations. In addition, it is desirable to avoid over-computation when both and-parallelism and or-parallelism arise within a set of goals.

In view of the above criteria, our system for realizing and-or parallelism combines or-parallelism with restricted and-parallelism (RAP) [9, 19]. This system was developed (and can be understood) in four stages.

1. We first developed an abstract model for representing and-or parallel computations, called the extended and-or tree model. The extended and-or tree has, besides *and* nodes and *or* nodes, two new nodes: the *sequential* node, to support RAP's sequential goals, and the *crossproduct* node, to help avoid overcomputation when and-parallel goals also exhibit or-parallelism.

2. The extended and-or tree model is a high-level model in that it does not address how multiple or-parallel environments are efficiently maintained, and how parallel

execution and scheduling is done. While the model allows these issues to be independently addressed, in our work we consider the binding-arrays method [36, 37] for environment representation, and a scheduling method for coarse-grain parallelism. We chose the binding-arrays method since it provides constant-time access to the two most frequently occurring operations: variable-access and task-creation. The emphasis on coarse-grain parallelism follows from our interest in shared-memory multiprocessors. The binding arrays (BA) method for purely or-parallel systems cannot be directly used for supporting multiple environments. In this paper we present an extension of binding arrays that accommodates independent and-parallelism in the presence of or-parallelism.

3. We then showed how to map the extended and-or tree on to a collection of stacks that lie in the address space of a shared-memory multiprocessor. We developed an extended instruction set for parallel execution, called And-Or WAM (AO-WAM), into which source programs would be compiled. AO-WAM has all the instructions of WAM plus some extra instructions for compiling and-parallelism, as well as for allocating space for the new nodes introduced in the extended and-or tree, and for incorporating optimizations.

4. To minimize the overheads of task-switching inherited from the binding-arrays approach, we developed a number of optimizations for AO-WAM by exploiting and-parallelism. Essentially, we reduce the overhead by reducing the number of conditionally bound variables† whose bindings need to be installed during task-switching. These techniques can be incorporated in a compiler, and can lead to substantial savings in execution time.

The remainder of this paper is organized as follows: section 2 presents objectives for and-or parallel implementations; section 3 describes the extended and-or tree; section 4 describes our environment representation and strategy for parallel execution, and compares them with other schemes; section 5 describes the data areas and instruction set of AO-WAM, including code for a simple example; section 6 describes how the overheads in AO-WAM due to the cost of task-switching can be reduced; and section 7 presents conclusions.

We assume the reader has some familiarity with the Warren Abstract Machine (WAM), binding arrays, and Restricted And-Parallelism (RAP).

† Conditionally bound variables, or conditional variables, are those variables that can potentially receive multiple bindings during execution.

2. Objectives for And-Or Parallel Implementations

2.1 Objectives for Or-parallelism

The following three criteria are of central importance in any or-parallel implementation:

1. the cost of *environment creation* should be constant-time;
2. the cost of *variable access and binding* should be constant-time; and
3. the cost of *task switching* should be constant-time.

The term "constant-time" here means that the time for these operations is independent of the number of nodes in the or-parallel search tree, as well as the number of goals and the size of terms that appear in goals. These criteria are derived from a consideration of the three important operations in any logic programming system (sequential or parallel): allocation of space for variables (environment creation), unification (variable access and binding), and resumption after success or failure (task switching). While it would be ideal if an or-parallel execution model could satisfy all three criteria, we have shown [12] that this ideal cannot be achieved by any or-parallel model using a finite number of processors and constant-time addressable memory. Other desirable characteristics of an ideal or-parallel implementation are that it should execute as efficiently as a sequential implementation in case only one processor is available. Also, it should be amenable to optimizations that apply to sequential implementations, such as last-call optimization.

The fact that at least one of the criteria must be sacrificed partly explains why so many or-parallel execution models have been proposed in the literature. Based on which criteria one chooses to satisfy one has a different execution model for or-parallelism. This also suggests a natural scheme for classifying various execution models for or-parallelism, which we have given in [12].

In the and-or implementation to be described later, the environment representation used is based on the *binding arrays* method [36, 37] because it provides constant-time access to the two more frequently occurring operations: variable access and task creation. The Binding Array method works by constructing an or-parallel tree of nodes, where each node contains the local variables of some clause. Each conditional variable is assigned a unique offset in a *binding array*, which stores the values of conditional variables. Each processor maintains its own binding array; the binding arrays of two processors working on two different or-parallel paths would have the same entries only at those indices that correspond to variables in the common ancestor nodes of the two paths. A processor creates a new or-parallel task by assigning unique offsets in its binding array for the unbound variables in the calling node and extending its binding array accordingly (the task of assigning offsets, in practice, is relegated to unification). The main overhead of this method is that binding arrays need to be updated upon context-switch (conditional bindings are also recorded in the trail, in addition to the binding array, for the purpose of updating during task-switching); however, this overhead can be minimized by not switching too often or not switching to too distant nodes in the tree. Other properties of this method are that, if there is only one processor available, the performance from a depth-first search strategy would compare well with a sequential implementation, and it supports standard sequential optimizations.

2.2. Objectives for And-parallelism

As noted earlier, exploiting and-parallelism requires the identification of *independent* subgoals, because executing dependent subgoals in parallel often results in wasteful computation. Three different approaches have been proposed to detect dependencies: (1) by run-time monitoring of the status of variables (bound or unbound) and dynamically restructuring tasks to obtain optimal and-parallelism [6]; (2) by global compile-time analysis assuming worst cases for subgoal dependencies [8]; and (3) by monitoring at run-time the status of terms (ground or nonground) and using a static task-structure, conditioned upon the status of terms, to obtain restricted and-parallelism [9]. The third approach is a nice compromise between (1), where a run-time overheads are high, and (2), where exploited parallelism is low. While a naive approach would traverse arguments of subgoals to determine if they are ground or not, clearly a desirable solution is one that avoids such a time-consuming run-time analysis. Thus, the time taken for detecting subgoal independence should be independent of the size of their respective arguments.

Unlike or-parallel implementations, a pure and-parallel implementation must be able to backtrack upon failure [20]. To understand the problem, consider the subgoals shown below, where ',' is used between sequential subgoals (because of data dependencies) and '‖' for parallel subgoals (no data dependencies).

a, b, (c ‖ d ‖ e), g, h

Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails: If subgoal a or b fails, sequential backtracking occurs, as usual. Because c, d, and e are mutually independent, if either one of them fails, backtracking must proceed to b—but see further below. If g fails, backtracking must proceed to the right-most choice point within the parallel subgoals c ‖ d ‖ e, and recompute all goals to the right of this choice point. If e were the rightmost choice point and e should subsequently fail, backtracking would proceed to d, and, if necessary, to c. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., "from the right". If initiated from within, backtracking proceeds outside all these goals, i.e., "to the left". This latter behavior is a form of "intelligent" backtracking [20].

To sum up, the following criteria should be satisfied by an ideal and-parallel implementation: avoid wasteful over-computation, avoid complex run-time dependency analysis, and support intelligent backtracking. As with or-parallel implementations, it is desirable that an and-parallel implementation perform comparably with a sequential implementation in the single-processor case and support standard sequential optimizations.

Among the well-known methods for and-parallel execution of logic programs are Conery's abstract model [7], improvements of this model by Kumar [24], and the Restricted And-Parallel (RAP) model of DeGroot [9] and its refinement by Hermenegildo and Nasr [20]. Of these, the refined RAP method comes closest to realizing the criteria mentioned earlier. In this method program clauses are compiled into Conditional Graph Expressions (CGEs) of the form

$$(condition \Rightarrow goal_1 \parallel goal_2 \parallel, \ldots, \parallel goal_n),$$

meaning that, if *condition* is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can be either the constant *true*, or it can be $ground(v_1, \ldots, v_n)$, which checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms, or it can be $independent(v_1, \ldots, v_n)$, which checks whether the set of variables reachable from each of $v_1 \ldots v_n$ are disjoint from one another. Checking for *ground* and *independence* involve very simple run-time tests, details of which are presented in [9]. The method is conservative in that it may type a term as nonground even when it is ground—another reason why the method is regarded as "restricted". Techniques to perform these checks at compile time may be found in [26, 27]. This model has been efficiently implemented on shared-memory multiprocessors using a variant of WAM [21].

2.3 Objectives for And-Or Parallel Implementations

Since an and-or parallel implementation must exploit both and-parallelism as well as or-parallelism, it is reasonable to adopt the union of the criteria for pure or-parallel and pure and-parallel implementations: constant variable-access, task-creation and task-switch times (pure or-parallel case); and avoidance of wasteful computation and efficient determination of subgoal independence (pure and-parallel case). However, when there is potential for both and- and or-parallelism in a single program, exploiting either form of parallelism alone can lead to unnecessary over-computation. For example, consider the following program for finding "cousins at the same generation" taken from [33]:

```
sg(X, X) :- person(X).
sg(X, Y) :- parent(X, Xp), parent(Y, Yp), sg(Xp, Yp).
```

In executing a query such as `?- sg(fred, john)` under a (typical) purely or-parallel or a purely and-parallel or a sequential implementation, the goal `parent(john, Yp)` will be re-executed for every solution to `parent(fred, Xp)`†. This is clearly redundant since the two parent goals are independent of each other. It would be better to compute their solutions separately, take a crossproduct of these solutions, and then try the goal `sg(Xp, Yp)` for each of the combinations. In general, for two independent goals G_1 and G_2 with m and n solutions respectively, the cost of the computation can be brought down from $m * n$ to $m + n$ by computing the solutions separately and combining them through a crossproduct (assuming the cost of computing the crossproduct is negligible). However, for independent goals with very small granularity, the gain from solution sharing may be overshadowed by the cost of computing the cross-product etc., therefore, such goals should either be executed serially, or they should be recomputed instead of being shared [14]. Independent goals that contain side-effects and extra-logical predicates should also be treated similarly [14, 16]. This is because the number of times, and the order in which, these side-effects will be executed in the solution sharing approach will be different from that in sequential execution or parallel execution with recomputation, altering the meaning of the logic program. While solution sharing may not be applicable in all cases, it does reduce the number of logical inferences performed at run-time, and therefore is also used in "optimal" computational models of logic programming, such as the Extended Andorra Model [18, 39].

To sum up, the criteria for a combined and-or parallel implementation are essentially the union of the criteria for pure or-parallel and pure and-parallel implementations. In addition, it is desirable to avoid over-computation when both and-parallelism and or-parallelism arise within a set of goals.

3. The Extended And-Or Tree Model

We begin by describing extensions to the basic and-or tree. Figure 1 shows an extended and-or tree for a simple program. There are four kinds of nodes in this tree: In addition to *and* nodes and *or* nodes corresponding to and-parallel and or-parallel goals respectively, we also have *crossproduct* nodes, to hold the crossproduct of solutions from and-or parallel goals, and *sequential* nodes, which correspond to sequential goals in the RAP model. The crossproduct node and the sequential node act as delimiters between which and-parallel

† A purely and-parallel system can avoid re-execution of independent goals but most existing ones do not.

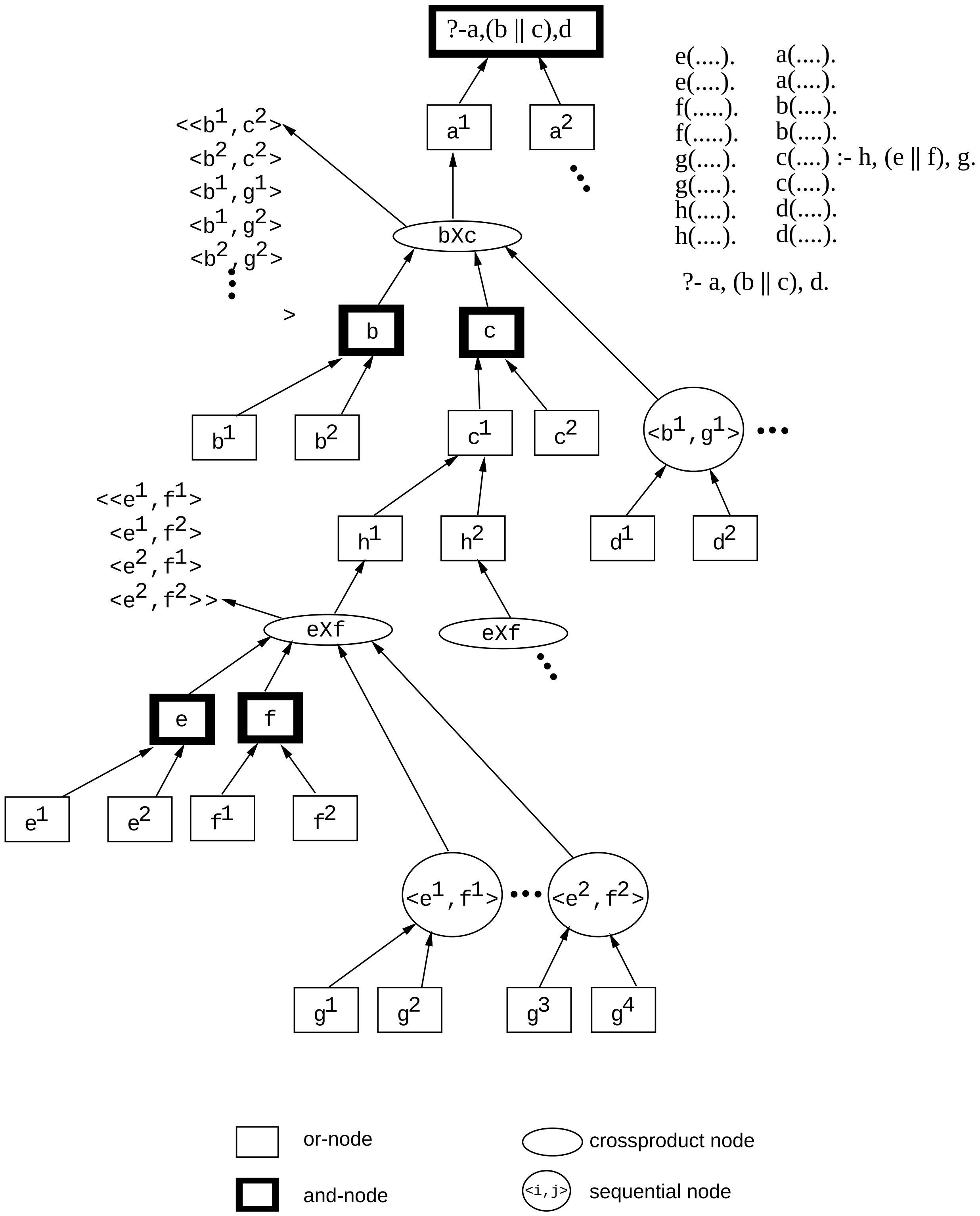

Fig 1 : An Extended And-Or Tree

execution takes place. Thus, the sequential node marks the end of and-parallel execution and resumption of and-sequential (but or-parallel) execution. Nodes have space for their subgoals (also called goal-list), and or-nodes have space for the bindings of the variables occurring in the subgoals.

Cross-product nodes are introduced to facilitate sharing of solutions‡ of and-parallel goals and serve to avoid the over-computation alluded to in the previous section. They are analogous to join-nodes in the PEPSys model [23, 40], and are parents of and-nodes and sequential nodes. Each element in the crossproduct set is a *tuple* which corresponds to one specific solution for the and-parallel goals. The crossproduct set `e X f` shown in the example tree gets created as follows: Once the and-nodes labeled `e` and `f` have been created for and-parallel goals with corresponding names, the solutions for their goals are found (by exploring the subtrees rooted below `e` and `f`, respectively). Next, the set of solutions for `e` and `f` are cross-produced and the resulting set stored in the crossproduct node, the parent node of `e` and `f`; for example, the crossproduct set $\{\langle e^1, f^1\rangle, \langle e^2, f^1\rangle, \langle e^1, f^2\rangle, \langle e^2, f^2\rangle\}$ corresponding to the crossproduct node labeled `e X f`. The components of tuple are labels (addresses) of terminal nodes of the and-branches. An and-branch is a branch rooted at an and-node and consists of those nodes lying along a path from this and-node to a leaf node representing *one* solution for the associated and-parallel subgoal (e.g., nodes labeled `b` and b^1 form an and-branch rooted at `b`). Note that we represent the crossproduct set symbolically, i.e., using label of terminal nodes (e.g. the memory address of the terminal node could serve as its label). However, one can also perform an explicit join of bindings produced for variables in the and-parallel subgoals, as done for example in ROPM [22]. The crossproduct set can either be computed incrementally as individual solutions for `e` and `f` are produced or it can be computed in one single operation after the solutions to `e` and `f` have been found—clearly the former approach is better since it avoids potentially nonterminating computations.

The structure of the extended and-or tree, except perhaps for the sequential nodes, should be evident now from the description in the previous paragraphs and figure 1. We therefore clarify the role of sequential nodes. (Note that, in the figure, (`b` $\|$ `c`) indicates that `b` and `c` can be executed in and-parallel.) For example, suppose that the and-nodes `e` and `f` have produced solutions e^1 and f^1 respectively. A tuple $\langle e^1, f^1\rangle$ would then be inserted in the crossproduct set associated with node labeled `e X f`, as described earlier. A sequential node labeled $\langle e^1, f^1\rangle$ would then be created corresponding to this tuple to solve goal `g`. Similarly, the creation of other tuples $\langle e^1, f^2\rangle$, $\langle e^2, f^1\rangle$, etc., result in the creation

‡ Solution-sharing has also been proposed in other models for and-or parallelism such as ROPM [22] and PEPSys [40].

of the corresponding sequential nodes rooted under `e X f`. Likewise, when goals g^1, g^2, etc., are solved, tuples $\langle b^1, g^1 \rangle$, $\langle b^1, g^2 \rangle$, etc., are inserted in the crossproduct set `b X c`, and similarly sequential nodes labeled $\langle b^1, g^1 \rangle$, etc., are created for the execution of goal `d`. Thus, a sequential-node is created for every possible continuation of the CGE; and-parallel computations in the extended and-or tree are "bracketed" between a crossproduct node and sequential node.

Note that initiating the execution of a sequential goal corresponding to a sequential node such as $\langle e^1, f^1 \rangle$ depends only on the completion of and-branches e^1 and f^1 and not on other and-branches of `e` and `f`. Hence execution of a sequential goal (which corresponds to the goal in the continuation of the CGE) may be initiated as soon as its corresponding sequential node has been created, even though the preceding and-parallel goals have not been completely solved.

An or-node, a sequential-node, or an and-node fails if all its children nodes fail, in which case it is deleted from the tree. A crossproduct node fails if any one of its children and-nodes fail, in which case the entire subtree rooted at the crossproduct is deleted from the tree. A limited form of intelligent backtracking is obtained in this way—if an and-parallel subgoal fails, the computation does not unnecessarily backtrack over sibling and-parallel goals.

Different forms of parallelism naturally arise in the extended and-or tree: (i) execution of two or-nodes in parallel gives rise to or-parallelism; (ii) execution of two and-nodes in parallel gives rise to and-parallelism; iii) execution of two sequential nodes with same parent crossproduct node gives rise to or-parallelism (also called *consumer instance* parallelism in [28]). Also note that the crossproduct set can be incrementally generated in parallel†.

4. Execution in the Extended And-Or Tree

The extended and-or tree is a general model for exploiting and-or parallelism in logic programs. Not described in this model are

(a) how multiple or-parallel environments are efficiently maintained;

(b) how and-parallel computations are expressed; and

(c) how parallel execution and scheduling is performed.

We believe that any existing scheme described in the literature can be used (perhaps with slight modification) for solving problems (a) and (b), although we advocate the binding arrays for the former and CGEs for the latter. The binding arrays scheme requires some

† An efficient, general method for incremental parallel generation of crossproduct set is given in [10].

extensions in order to work in the presence of and-parallelism‡. In the next few subsections we describe these extensions along with how various operations are performed in our realization of the extended and-or tree.

4.1. Variable Access

In presence of both and-parallelism and or-parallelism, the binding-arrays method for the pure or-parallel case must be extended to achieve constant-time access to variables. To see the problem, consider the goals (p, (q1 $\|$ q2), r), where q1 and q2 also exhibit or-parallelism. Suppose further that goal p has been completed. In order to execute goals q1 and q2 in and-parallel (exploiting or-parallelism within them at the same time), it is necessary to maintain separate binding arrays for them. As a result, the binding-array offsets for any conditional variables that come into existence within these two goals will overlap. (Recall that conditional variables are variables which are unbound at the time of branching.) Thus, when goal r is attempted, we are faced with problem of merging the binding arrays for q1 and q2 into one composite binding-array or maintaining fragmented binding arrays.

To solve the above problem, first recall that in the binding-array method [36, 37] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables. However, offsets to the conditional variables in the and-parallel branches cannot be uniquely assigned, since there is no implicit ordering among them; at run-time a processor can traverse them in any order. We introduce one more level of indirection in the binding array to solve this problem.

In addition to the binding array, each processor also maintains another array called the *base array*. As each or-node is created, it is assigned a unique integer id. When a processor encounters an or-node, it stores the offset of the next free location of the binding array in the i-th location of its base array, where i is the identifier of the or-node. The offset-counter is reset to zero when an or-node is created. Subsequent conditional variables are bound to the pair $\langle i, v \rangle$, where v is value of the counter. The counter is incremented after each conditional variable is bound to this pair. The dereferencing algorithm is given below in Fig 2. Note that, in this algorithm, if a variable has the tag value VAR, then this indicates that it is either bound to another variable or to itself. If a variable is bound to itself, it means that it is unbound, following the convention used by most Prolog systems. The tag NON-VAR indicates that the variable is bound to an atomic value or a structure.

‡ In [13] we show how some other or-parallel environment representation techniques can be adapted for implementing the extended and-or tree.

```
deref(V)         /*unbound variables are bound to themselves*/
term *V {
    if V→tag == VAR
            if not V→value == V
                    deref(V→value)
            else V
    else if V→tag == NON-VAR
            V
        else {              /*conditional var bound to <i, v>*/
            val = BA[v + base[i]];        /*BA is the binding array.*/
            if val→value == val
                    V
            else deref(val) }}
```

Fig 2: Dereferencing Algorithm

In the above scheme, access to variables is still constant-time, although the constant is somewhat larger compared to the binding-arrays method for pure or-parallelism. Also note that now the base array is also to be updated on a task-switch.

This variable binding scheme is very general and not dependent on any scheduling strategy. A processor may switch from any node to any other node, and provided it makes all the appropriate changes to its binding array and base array, it can still access all the variables in its environment correctly. However, assigning a unique identifier to every or-node may be a large overhead, since it is also incurred for or-nodes which are not part of and-branches. By using a modified technique which requires a particular scheduling strategy described below we can avoid this work for the or-nodes. In this technique we incur the labeling overhead only in the presence of and-parallelism; hence, pure or-parallelism is exploited without the extra overhead of assigning ids to or-nodes.

In the improved technique we label the and-nodes rather than or-nodes with unique integer ids†. This would also require that rather than resetting the offset counter when an or-node is created, it must be reset when an and-node is created. Thus, we incur an overhead only on creation of and-nodes, i.e., only in the presence of and-parallelism. If the identifier of an and-node is j and this and-node is encountered by a processor, the offset of the next free location in the binding array of that processor is stored in the jth location of its base array. All conditional variables arising in an and-branch corresponding to the and-node are bound to the pair $\langle j, v\rangle$, where v is the value of the offset-counter. The

† The ids need to be unique across an or-branch, but for simplicity we assume that they are unique across the whole and-or tree.

dereferencing algorithm remains the same. The node scheduling strategy should be such that when conditional variables are *loaded* in the binding array, they occupy contiguous locations (the importance of conditional variables occupying contiguous locations in the BA is further explained in Section 5.1.3.3). It turns out that a simple scheduling strategy suffices, namely, one in which a processor does not work on another and-goal, unless it has found a solution to the current and-goal (i.e. once an and-goal or an alternative inside an and-goal is picked, the processor will not arbitrarily switch tasks unless it has produced *a* solution or failure occurs). Other processors, of course, may work in parallel on other and-goals. Also during *unloading* of the binding array, the processor should traverse the and-branches in the opposite order of traversal during *loading* (the loading and unloading operations are explained in the next subsection). This ensures contiguity in the binding array when a processor selects an unfinished choice point in the course of unloading an and-branch from its binding array. In addition, it ensures that conditional variables occur in the binding array in order of their seniority, like in the binding-arrays scheme for pure or-parallelism.

4.2. Parallel Execution and Scheduling

We first note that we can assume the extended and-or tree lies in a memory space accessible to all processors, because we are targeting our implementation at shared-memory multiprocessors. At the start of execution it simply consists of the root node. In our scheme, processors traverse the branches of the tree, executing subgoals in the nodes, and growing and contracting the tree in the process. Since the number of branches in the and-or tree would be much larger than the number of processors, each processor ends up executing more than one branch of the tree. This is accomplished through backtracking on success/failure. The movement of a processor from its current site to the place where work is available is called *task switching*. A processor that has created a node is eventually responsible for solving the entire tree rooted at the node. However, other idle processors may eagerly help, by taking up any available work from this subtree. A processor does not become idle until the entire sub-tree rooted at the node it undertook to solve is explored. This ensures coarse granularity of parallelism, and results in less task switching.

We now describe the algorithm that a processor uses for selecting work in the extended and-or tree. The algorithm, which is invoked when a processor runs out of work, makes use of two basic operations: *load* and *unload*. In the load operation, a processor, given a crossproduct tuple, updates its binding array with the conditional bindings that are found from the trails of the and-parallel branches of the solutions corresponding to the tuple; the base array is simultaneously updated. We therefore say the processor *loads the tuple* *into its binding array.* For example, referring to figure 1 once again, if a processor has its binding array (and base array) stationed at the crossproduct node `e X f`, then before it can continue execution below the sequential node labeled $\langle \mathtt{e}^1, \mathtt{f}^1 \rangle$ it must load all the conditional bindings along the and-branches $\mathtt{e}$-$\mathtt{e}^1$ and $\mathtt{f}$-$\mathtt{f}^1$. If, however, the processor's arrays were stationed at node $\mathtt{e}^1$ and it wanted to continue with the tuple $\langle \mathtt{e}^1, \mathtt{f}^1 \rangle$ then it just needs to load conditional bindings made along $\mathtt{f}$-$\mathtt{f}^1$. The unload operation is the opposite of the load operation, i.e., during the unload operation, conditional bindings occurring in the and-branch of the solutions in the tuple are purged from the binding array; the base array is also purged. (See also Sections 5.1.3.1 and 5.3.1.2.)

The load and unload operations are the costs we incur because we are not re-computing the and-parallel goals, an activity that is generally more prone to overhead. In our scheme, all solutions to an and-parallel goal are computed only *once*. One can show that the loading and unloading operations due to solution sharing entail fewer operations than recomputing the and-parallel goals. If there was no sharing, all and-branches would be recomputed, and during this recomputation *all* variables encountered would be accessed *at least once* during unification. However, if they are shared instead, no frames are pushed on the stacks and hence these stack operations are avoided. Also, only conditional variables (rather than all variables) are accessed during loading; furthermore, they are accessed only once.

In order to realize parallelism from the extended and-or tree, every processor works on a branch until it exhausts the goal-list. If it is working on an and-branch it grows that branch until a solution for the and-subgoal is found. If at least one solution has been produced for each of the other sibling and-parallel goals, the processor inserts all the tuples that can be formed from this solution into the crossproduct set. It then selects the appropriate tuple from the crossproduct set, loads its binding array, and continues execution. If a solution has not been found for some of the sibling and-goals, the processor commences execution of one of these and-goals, rather than searching for *all* solutions of the current and-subgoal. This ensures that solutions to top-level query are produced as quickly as possible. If it is working on an or-branch, it likewise grows this branch until a solution is found for the top-level query. After the processor has found a solution, it traverses the tree upwards, and if it finds any untried alternatives it takes them up (updating appropriately the pool to which this untried work belongs). However, if there are no untried alternatives, it examines the work pool of the processors which are working in the subtree below it. We assume that each processor maintains a pool of work it produces, which it will eventually carry out if aided by no other processor. It assists these processors by taking work from them so that all solutions in the subtree are computed quickly. Notice that the behavior of the processor depends on whether it is executing an or-branch or an and-branch. The algorithm for work selection is described in figure 3 below.

```
task_switch()
  case A of        /*A is the current node*/
    OR_NODE :
            if or_node has untried alternatives
                execute the alternative clause
            else if or-node is not part of and-branch
                  L1: steal work from processors below
                          /*return here only if no work found*/
                     if no processor below
                         untrail and move one node up ; task_switch();
                     else goto L1
                else  /*or-node is part of and-branch*/
                   L2: if no solution produced in subtree below
                         if no processors below
                            untrail and move one node up ; task_switch();
                         else
                            steal work from processors below
                               /*return here only if no work found*/
                            goto L2
                      else
                         untrail and move one node up ; task_switch();
    AND_NODE :
            if no solution found for this and-node and no processor working below it
                send kill messages to all processors working below the
                                  parent crossproduct node.
                untrail and move one node up ; task_switch();
            else
                move one node up  to the crossproduct node ; task_switch();
    CROSS_PROD_NODE :
            if there is work available (untried and-node or tuple) at this node
                   execute it
            else if there is an untried choice point in subtree below
                   execute it.
                else if there are processors working in subtree below
                     L3: steal work from these processor  and execute it.
                               /*return here only if no work found*/
                        if there are processors working below
                            goto L3;
                     untrail and move one node up ; task_switch();
    SEQ_NODE :
            unload tuple in sequential node from binding array.
            If while unloading an and-branch, an or-node with
              untried alternative is found, execute that alternative.
            else
               move one node up to the parent crossproduct node ;
               task_switch();
```

Fig 3: Algorithm for Selecting Work

Note that the binding array is updated not only during loading and unloading of tuples, but also when a processor moves up from one node to another. Also note that while moving up, if the processor happens to be the creator of that node, it has to wait for other processors working in the sub-tree below to finish before it can reclaim this node from memory or move further up. We can avoid this idling by letting the processors work in other sections of the and-or tree, without reclaiming the node they created even though there are other processors working in the subtree below. This will lead to creation of *ghost* nodes which will have to be reclaimed later. Such a strategy has been used in Aurora [25].

4.3. Comparison with Other Schemes

We now compare briefly our extended and-or tree model with the ROPM [22] and PEPSys [40]—the two other prominent models proposed for and-or parallel execution. Basically, the three systems take a different approach to or-parallelism, each incurring a different kind of cost: non-constant-time variable access (PEPSyS), non-constant-time task creation (ROPM), and non-constant-time task switch (our approach). The ROPM model is based on the concept of Data Join Graphs (DJGs). The DJG is used for the dual purpose of representing dependencies between subgoals in a clause as well as for recording the execution state at run-time. Although DJGs are more general than CGEs, they seem more prone to overheads, e.g., intricate operations have to be performed to remove redundancies during join evaluation [29]. In the PEPSyS system, the join algorithm does not allow more than two and-parallel goals to be joined together, making and-parallel execution of more than two goals slightly inefficient. Some of the complexity of this system is due to the designers' goal to incorporate backtracking along with and- and or-parallelism.

Our model is derived from the intuition that, for shared-memory multiprocessors, schemes that have non-constant time task-creation or non-constant-time variable-access are less efficient than those with non-constant time task-switching. This is because the number of variable accesses and task creations is dependent on the logic program being executed, while the number of task switches is dependent on the scheduler. While the scheduler can be carefully tuned by the implementor to minimize the number of task-switches, this kind of tuning is virtually impossible to perform in minimizing the number of variable accesses or task creations. In systems with non-constant time task-creation (such as ROPM), it is possible to reduce the number of expensive task-creation operations by adopting some technique for granularity control, e.g., goals whose terms are "smaller" than a certain threshold are not executed in parallel, thereby avoiding the overhead for creating parallel subtasks within this goal. However, adopting such a technique may lead to loss of parallelism, because grains are fixed at compile-time. On the contrary, by adopting the approach where task-switching is non constant-time operation, the same effect is achieved dynamically, e.g., in the Aurora system, a distinction is made between shared and private section. The shared section grows as more processors become available.

A noteworthy point about our proposed system is that the exploitation of and-parallelism and or-parallelism does not appreciably degrade the performance of programs that contain only pure or-parallelism or pure and-parallelism: If we exploit only or-parallelism, we believe that our system could be as efficient as the Aurora system [25, 31], a purely or-parallel system based on binding arrays; likewise, if we exploit only and-parallelism, we believe that our system could be as efficient as the RAP-WAM system [21], a purely and-parallel system based on CGEs. In both cases, the indirection in accessing conditional variables in our system (due to base arrays) would marginally degrade performance. This overhead is, we believe, a small price to pay for obtaining one system for both and-parallelism and or-parallelism, with the added benefits of solution sharing.

5. AO-WAM: A WAM Extension for And-Or Parallel Execution

Figure 4 summarizes the state of an AO-WAM processor—all processors have a similar storage model. As a processor executes the extended WAM instructions (described in section 5.2) it pushes nodes along a branch in the extended and-or tree on to its stacks. Because idle processors may eagerly help other processors, it can happen that nodes along a branch are distributed across the stacks of different processors. In the remaining description we explain the processor-state, concentrating on features not present in the standard WAM model [38].

5.1. AO-WAM Machine State

5.1.1. Data Areas:

(i) *Correspondence of nodes in extended and-or tree to frames in stacks*: All nodes in the extended and-or tree map directly to the stack frames. However, a single choice point is created for a set of sibling or-nodes of the extended and-or tree, and one environment record is created for each or-node. In subsequent sections we shall refer to choice-points as or-nodes, by abuse of terminology. Given two nodes n1 and n2, where n1 is above n2 in the stack, it is true that n1 is a descendent of n2 in the and-or tree or they are in independent and-branches. As a corollary of this, space from the node and environment stacks is always reclaimed from the top.

(ii) *Separation of local stack into environment and node stacks*: There are two advantages of this separation: (1) During space allocation, it is easy for the processors to access the top-most node in the stack (through register $\mathtt{B}_L$, described later). (2) It simplifies

Figure 4: AO-WAM Data Areas and Registers

the task of updating the binding array. It also enables incorporation of other scheduling strategies and thus makes the architecture amenable to modifications.

(iii) *Separation of trail into local and global trails*: This is done to reduce the amount of work during task switching (explained in section 6).

(iv) *Introduction of solution nodes*: A solution node is pushed on the node stack when the end of an and-branch is reached. Thus, it corresponds to the tip of an and-branch. It serves two purposes: (1) Its memory address is used as the symbolic *name* for the corresponding solution in crossproduct tuple. (2) It helps ensure that an and-parallel solution does not get deleted from the stack until the entire crossproduct has been tried.

5.1.2. Node Scheduling Areas: The node-scheduling areas are used to identify available work, and are organized as follow: *(i) Or-node Queue*: The untried or-nodes are organized as a queue for scheduling because we believe that those closer to the root would contain bigger subtrees, maximizing the granularity of work. *(ii) Cross-product Queue*: The untried crossproduct tuples are organized as a queue for the same reason. *(iii) And-node Stack*: Untried and-nodes are organized as a stack because later and-subgoals must be solved before earlier and-subgoals, which we think helps in avoiding speculative work and finding the first solution faster.

Or-nodes and crossproduct nodes have a *processor bit-vector* (similar to [4]). This vector tells which processors are working in the subtree rooted at that node. A processor sets the bit at position *pid*, where *pid* is the processor identifier of the processor, when it passes through the node while moving to the site where work is available. It resets this bit when it returns while traveling up the tree. The bit vector scheme is suitable for only a small number of processor (maximum of 32 in our implementation). Other scheduling schemes which allow more processors can also be adapted to our model, e.g. those of [1, 3] which are designed for or-parallel systems.

5.1.3. Variable Access Arrays: These were introduced in section 4.1; here, we describe in more detail the loading and unloading of conditional bindings in the binding and base arrays. We also discuss how space in the binding array is managed.

5.1.3.1. Untrailing Variables: Unloading

As a processor moves up from the node where it is currently stationed to the parent of that node, it updates its global environment so that correct bindings are accessed during variable dereferencing. The untrailing operation consists of marking as unbound all the conditional variables in the section of the trail corresponding to the intervening local environments (or-nodes) (Fig 5(ii) and Fig 5(iii)). This involves marking as unbound the binding array slots corresponding to these conditional variables. We mentioned earlier that the conditional bindings created by an environment are also recorded in the trail stack. Since the trail stack contains the conditional bindings of the environment frames residing in the local stack, we must identify the section of the trail that contains the conditional binding created by these intervening environments. This section of the trail-stack frame can be identified by two pointers, one pointing to the beginning (bottom) and one to the end (top). These pointers can be stored in the nodes itself, as part of the machine state. We can avoid storing the pointer to the bottom of the trail-section, since the bottom of one trail-section will be the top of the trail-section of the node preceding the current node in the stack (Fig 4). We need not store a pointer to the top of trail-section in the and-node or a sequential node since there are no intervening local-environments between it and its parent node. Thus, it is only stored for the choice points and the cross-product nodes. The top of trail-section at any given time is easily obtained, since it would be the same as the top of trail stack at the time the node is pushed onto the local-stack.

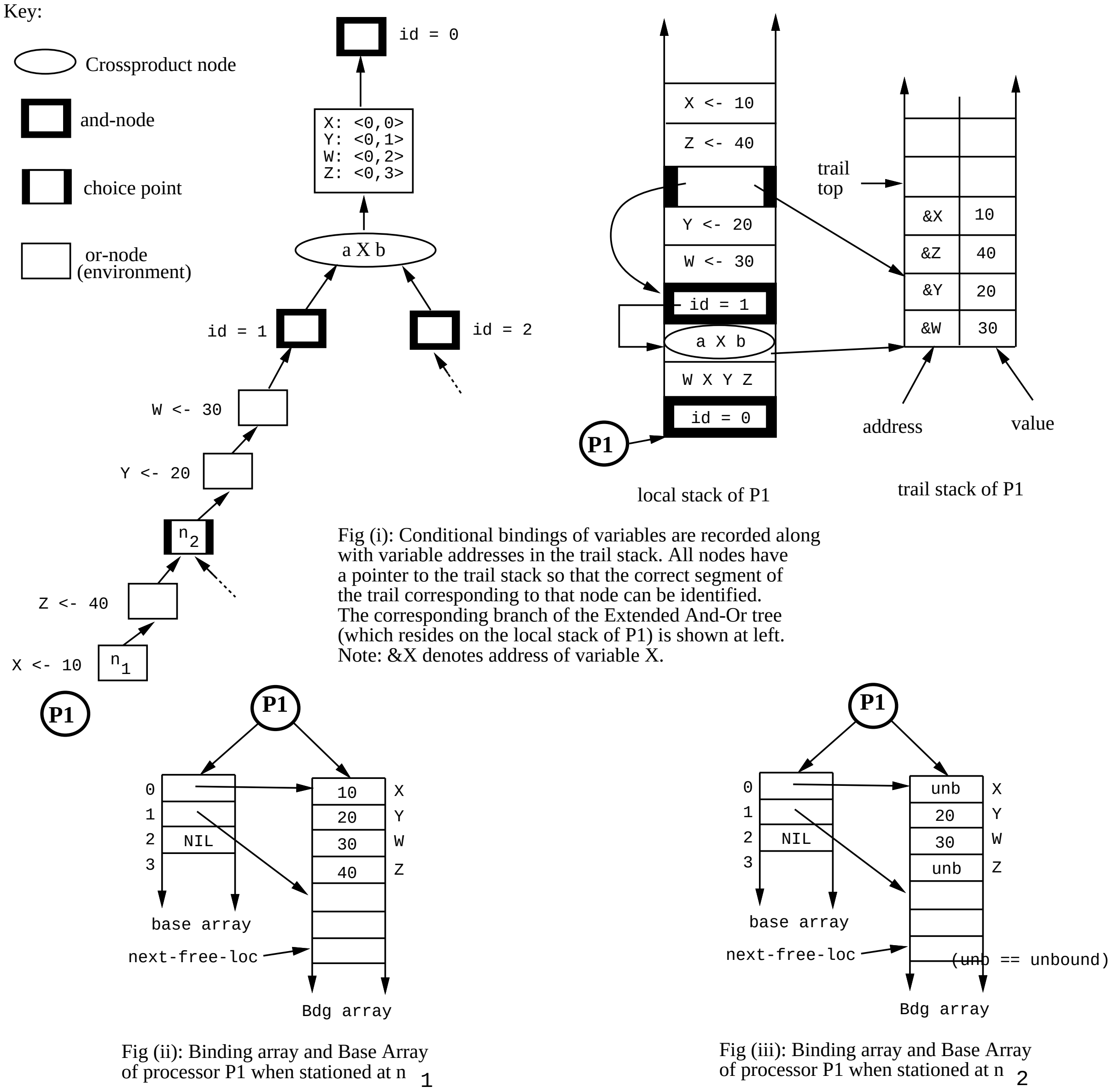

Fig (i): Conditional bindings of variables are recorded along with variable addresses in the trail stack. All nodes have a pointer to the trail stack so that the correct segment of the trail corresponding to that node can be identified. The corresponding branch of the Extended And-Or tree (which resides on the local stack of P1) is shown at left. Note: &X denotes address of variable X.

Fig (ii): Binding array and Base Array of processor P1 when stationed at n_1

Fig (iii): Binding array and Base Array of processor P1 when stationed at n_2

Fig 5 : Untrailing and Unloading

Note that though we call this operation untrailing, only the binding array gets modified; the trail is left untouched, in contrast with the untrailing operation in sequential systems where part of the trail stack is also reclaimed. This is because the trail is also used for loading the binding array of a processor that may later pick work from that region of the tree. A section of trail is reclaimed only when its associated frame is reclaimed from the local stack (this reclamation will be performed only when we are sure that the subtree rooted at the node corresponding to this frame has been completely explored).

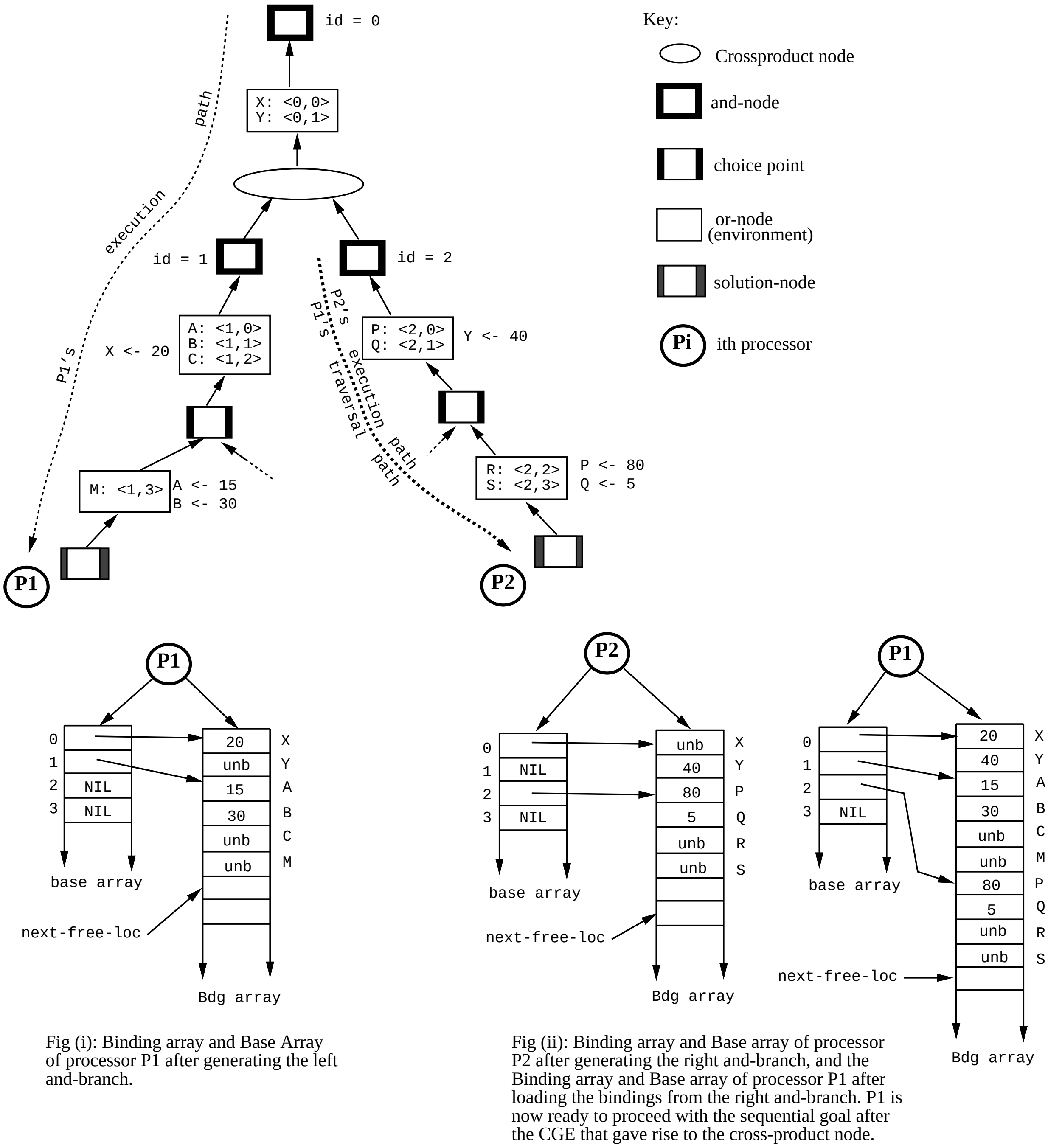

Fig (i): Binding array and Base Array of processor P1 after generating the left and-branch.

Fig (ii): Binding array and Base array of processor P2 after generating the right and-branch, and the Binding array and Base array of processor P1 after loading the bindings from the right and-branch. P1 is now ready to proceed with the sequential goal after the CGE that gave rise to the cross-product node.

Fig 6 : Binding Installation during Loading

5.1.3.2. Binding Installation: Loading

During forward execution when a processor finds a solution for an and-parallel goal in a CGE, and wants to continue with the execution of the continuation of the CGE, then it has to install all conditional bindings created during execution of other goals in the CGE in its binding array. This operation, termed loading in section 4.1, is further illustrated in figure 6. Thus, the operation of binding installation is very similar in nature to untrailing, except that instead of marking variables as unbound in the binding array their correct binding is put instead. It has the effect of merging of binding arrays alluded to earlier.

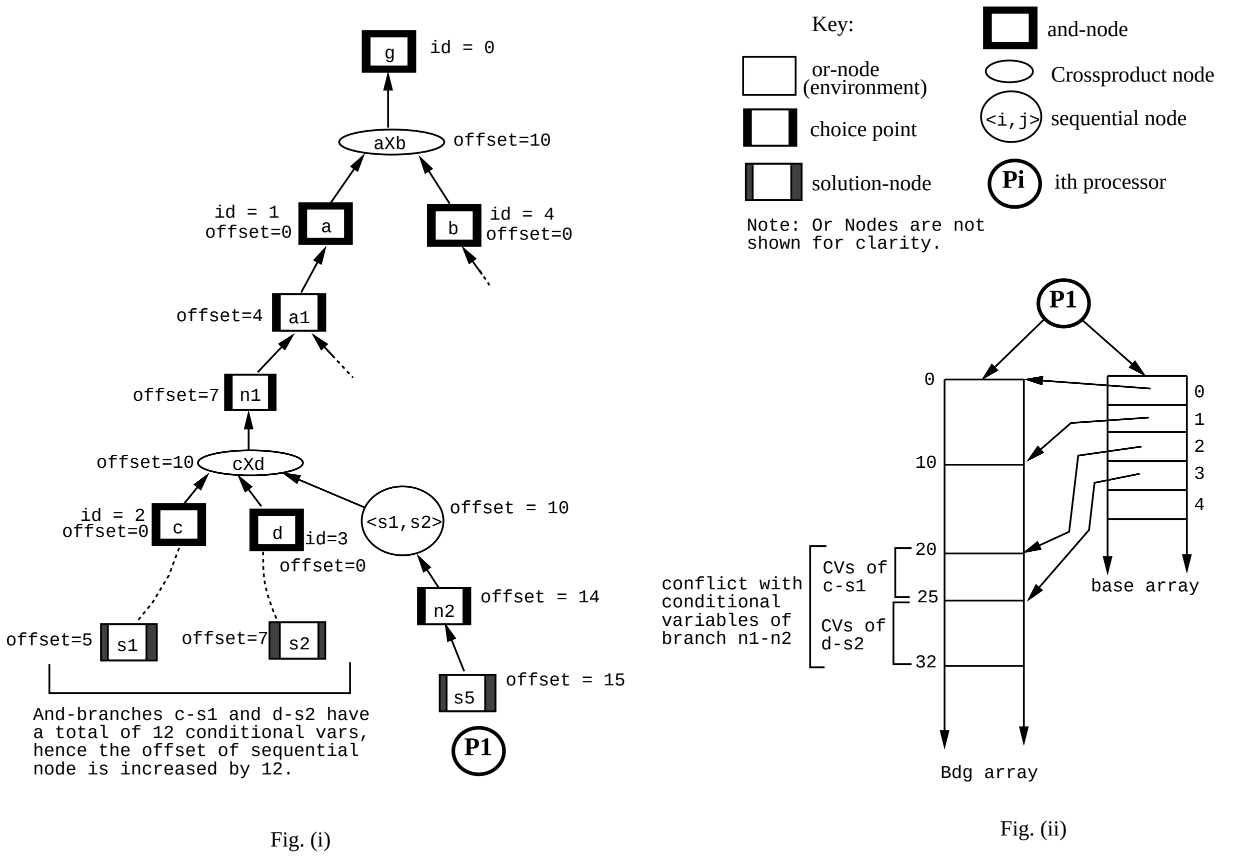

Fig 7 : Problems with Binding Arrays

Loading and Unloading are also performed during task-switching. When a processor task-switches from node n_1 to node n_2, it moves up from n_2 to the common ancestor node (say c) of n_1 and n_2, unloading conditional bindings from the binding array along the way. It then installs conditional bindings from the binding lists of nodes lying between the node c and n_2. To install the bindings, the processor has to traverse the path from c to node n_2. During a task-switch when a sequential node is encountered, then conditional bindings made along and-branches corresponding to tuple associated with this sequential have to be unloaded or loaded, depending upon if the sequential node lies between n_1 and c or between c and n_2.

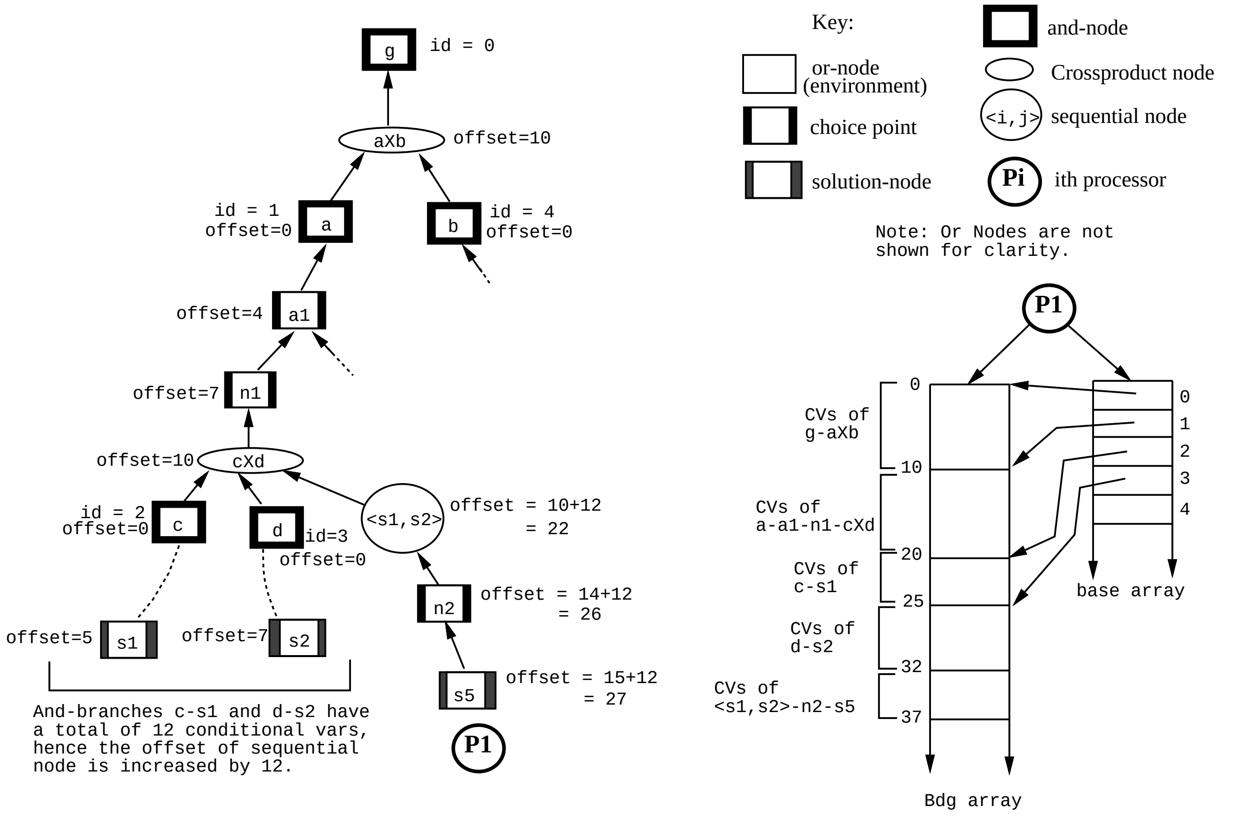

Fig 8 : Ensuring Contiguity of a Binding Array

5.1.3.3. Contiguity in Binding Arrays

Every conditional variable in the Extended And-Or tree has an associated and-node whose and-id is used for dereferencing its value. Consider the tree shown in Fig 7(i). The nodes labeled n_1 and n_2 have a common ancestor and-node (labeled a). All the conditional variables along the branch a-a_1-n_1-n_2 ... would use a_1's and-id as their and-id (the first element in the pair to which conditional variables are bound). If the offsets (the second element in the pair) of two conditional variables corresponding to nodes n_1 and n_2 respectively along the and-branch differ by k, then their corresponding slots in the binding array should also differ by k. In other words, all conditional variables created in a given and-branch should be allocated space contiguously. If this is not the case, then the dereferencing algorithm would not work correctly for these variables. We call this condition the *contiguity condition.* It is hard to ensure this condition because n_1 and n_2 may have intervening (nested) and-branches where the offset counter is reset. Since these nested and-branches would always be traversed while going from n_1 to n_2 (or vice versa), the conditional variables of these and-branches would occupy slots in the binding array somewhere in between those of n_1 and n_2. Since the counter is reset on arriving at an and-branch, the constraint mentioned earlier would be violated, leading to incorrect dereferencing of variables (Fig. 7(ii)).

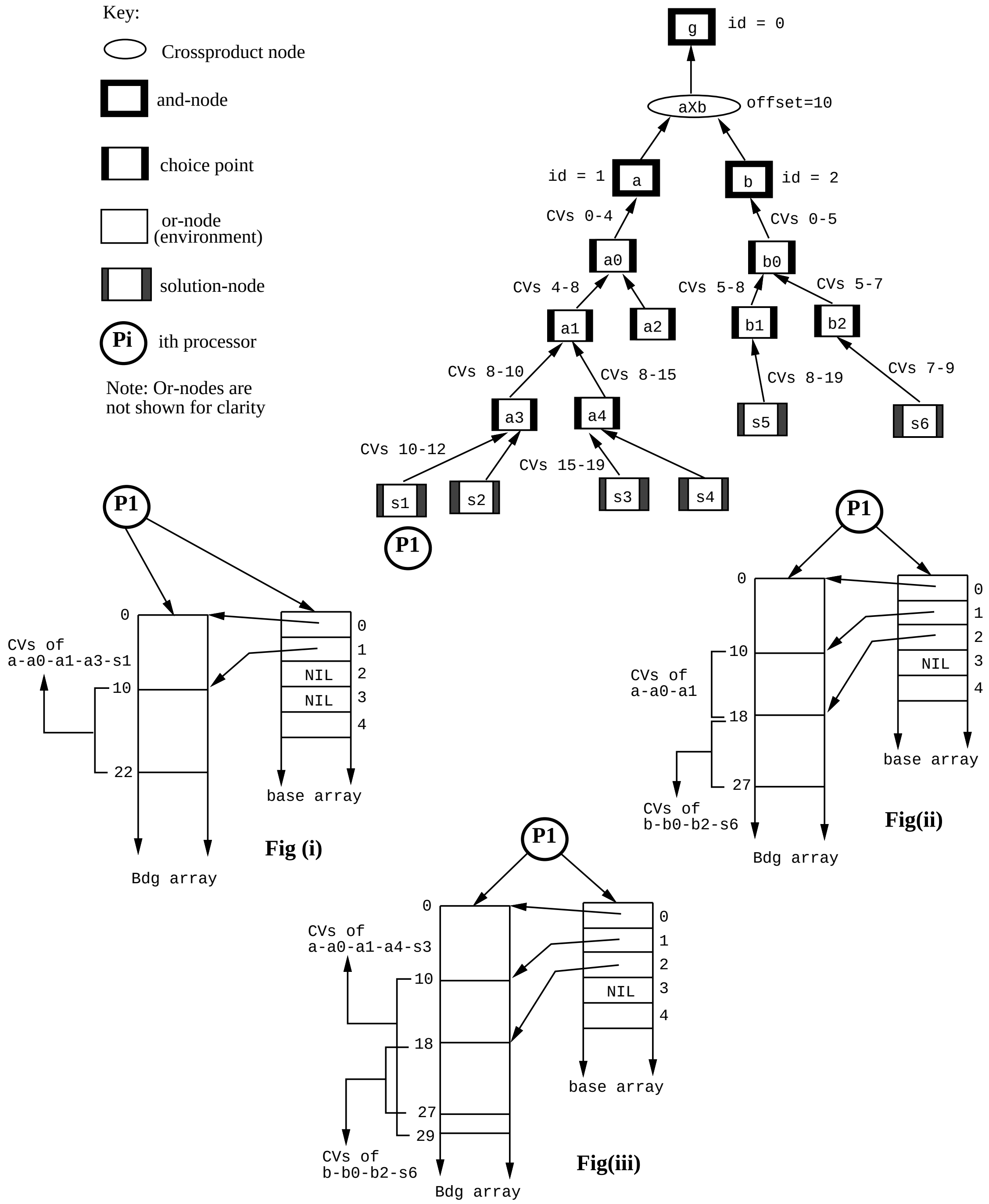

Fig 9 : Backtracking and Contiguity of BAs

To circumvent the foregoing problem, we propose the following solution. When a CGE is encountered in an and-branch, the current value of the offset counter is recorded in the cross-product node. When a cross-product tuple corresponding to this CGE is generated, the offset counter is restored to the value recorded in the cross-product node. The number of conditional variables found along each and-branch comprising the tuple are then summed and added to the offset counter. The number of conditional variables in an and-branch is easily determined since it is recorded in its solution node (by recording the value of its local offset counter during the time of its creation). The updated value of the offset counter is then stored in the sequential node and used for assigning offsets to the subsequent conditional variables (Fig 8). As a result, the contiguity constrain is not violated.

The contiguity constraint requires that processors move up the tree strictly along the path they followed during forward execution; otherwise, variables would be incorrectly dereferenced. To illustrate why this is essential, consider the example tree shown in Fig 9. Suppose a processor creates the branch $\mathtt{a}$-$\mathtt{a}_0$-$\mathtt{a}_1$-$\mathtt{a}_3$-$\mathtt{s}_1$. Its base array and binding array would now appear as shown in Fig. 9(i). Let us assume that the processor moves up to $\mathtt{a}_1$ from $\mathtt{s}_1$ and then decides to execute an untried alternative of $\mathtt{b}_1$. When it moves up to $\mathtt{a}_1$ it would remove the conditional variables with offsets `8` through `12` from the binding array since they are no longer in its environment. Suppose it produces the branch $\mathtt{b}_0$-$\mathtt{b}_2$-$\mathtt{s}_6$ with conditional variable offsets as shown. Now the binding array and base array would be as shown in Figure 9(ii). At this point, if P1 decides to return to $\mathtt{a}_1$ and pursue its other alternative, then binding of conditional variables along the branch below `b` would be overwritten and therefore lost. If we store conditional variables with offsets `8` through `15` and `15` through `19` (i.e. conditional variables along the path $\mathtt{a}_1$-$\mathtt{a}_4$-$\mathtt{s}_3$) after the conditional variables of `b` in the binding array, then conditional variables below $\mathtt{a}_1$ cannot be dereferenced correctly, because the contiguity property is destroyed for the branch rooted at a. Thus it is important that, while moving up the tree, nodes with available work be tried in the order they were created, i.e., P1 should pursue the untried alternative of $\mathtt{a}_1$ before trying the untried alternative of $\mathtt{b}_1$. Note that this order is indeed the one that would be dictated by the work-scheduling strategy of sections 4.1 and 4.2. As a result, the binding array behaves like a stack. Node reclamation is now simplified, since the associated conditional variables that are no longer needed can be reclaimed from the binding array by simply moving the next-free-location pointer.

5.1.4. Registers: In addition to the regular WAM registers we have the following extra registers. (i) $\mathtt{B}_L$, which points to the top of the local node stack. (ii) $\mathtt{L}_e$, which points to the top of the local environment stack. (iii) `D`, which points to the current and-node, i.e., the and-node in whose scope the current environment falls. The current value of `D` is saved in the `Cont D` field in crossproduct nodes so that it can be restored when sequential nodes are pushed. (iv) `OC`, which is the offset counter for the conditional variables. (v) `CFA`, in which the address of the code sequence to be executed, if the CGE fails, is loaded. (vi) `CPQ`, `AS` and `OQ`, which hold pointers to the heads of the work queues/stacks. The `CPQ`, `AS` and `OQ` pointers are stored in nodes to restore the respective work queues/stacks on failure.

5.2. AO-WAM Instruction Set

The AO-WAM supports all of the instructions supported by WAM. The new instructions introduced in the AO-WAM consist of the *check* instructions (`check_me_else`, `check_ground` and `check_independent`) of RAP-WAM [19] for compiling CGEs, and instructions for allocating space for various nodes: `alloc_cross_prod n, Addr`, `alloc_and Addr`, `alloc_sequential` and `alloc_solution Addr`. The `check_me_else` instruction loads a register with the address (called Check Fail Address or CFA) where the execution is to branch if the CGE condition evaluates to false. The `check_ground` (respectively `check_independent`) instruction checks if the variables in their arguments are grounded (respectively independent). The instructions for allocating space are used to allocate space for the various nodes. The example in the next section illustrates their meaning and use. The first argument, `n`, in `alloc_cross_prod n, Addr` is used to trim the environment of the calling predicate, in a similar fashion as the `allocate` instruction of WAM. Three interesting new instructions are:

`put_and_variable Yn, Ai`: is same as the `put_variable Yn, Ai` instruction except that the variable `Yn` is globalized and a reference to the global value is saved in `Ai`. `Yn` is also initialized with the same reference. This instruction is used to globalize unbound variables in the and-parallel subgoals, so that during update (loading and unloading of tuples) of the binding arrays, the processor has to only look at the trail for global (or heap) variables.

`put_and_value Yn, Ai`: is same as the `put_value Yn, Ai` instruction except that if the variable `Yn` dereferences to an unbound local variable then it is globalized and a reference to it is saved in `Ai`. `Yn` is also initialized with the same reference. This instruction is used to globalize variables in the and-parallel subgoals, which are bound to unbound variables, so that during update (loading and unloading of tuples) of the binding arrays, the processor has to only look at the trail for global variables.

`push_and_call Code/n`: similar to `push_call` instruction in RAP-WAM. Push an entry into the and-goal stack, i.e., push the instruction address of the subgoal, argument registers `A1` through `An` (loaded through the regular `put` instructions) and the current environment register. Exclusive access to the stack is obtained while pushing the entry.

In addition to the operations associated with specific instructions, each processor performs certain other actions to handle exceptions such as failure and messages from other processors. These non-instruction related actions are sketched below:

failure: If failure occurs the task_switch routine in section 4.2 is invoked. As a processor moves up the tree looking for work, it reclaims space from nodes which it created. If it reaches an and-node, for which no solution has been found, and there are no processors working below in the branches of that and-node, then the crossproduct corresponding to that and-node has failed. The processor sends a *kill* message to all the processors working below the parent crossproduct node to signal this failure of the crossproduct. It restores its registers, node stack, environment stack, and heap up to the parent node of the crossproduct node. It also purges its binding array and base array from the trail.

kill: This message is received by a processor when a crossproduct fails. The address of the crossproduct node that has failed is also received. The processor restores its registers and stacks up to the parent-node of the crossproduct node whose address is received. The binding array and base array are also purged. Next the processor calls the task_switch routine to look for work.

A prototype implementation of the AO-WAM is operational, and preliminary results from the small test cases are very encouraging.

5.3. Example

In this section we give the compiler generated AO-WAM code for a simple clause. The code, which has been verified on our sequential implementation, is annotated to explain the effect of the instructions.

`f(X, Y) :- a(X, Y), b(X,Y), c(X, Y, Z), d(X,Y,Z).`

Suppose the graph expression generated is the following:

`f(X, Y) :- a(X, Y), (ground(X,Y) ⇒ b(X,Y) || c(X,Y,Z)), d(X,Y,Z).`

where `a` is expected to ground `X` and `Y` so that `b` and `c` can be executed in parallel. The code is as follows

```
f/3:                            Entry point for procedure f
      allocate                  Push environment for f.
      get_variable X, A1
      get_variable Y, A2        unify arguments of f.
```

```
      put_value X, A1        load arg.  registers to execute a.
      put_value Y, A2
      call a/2, 3            Call a
      check_me_else SEQ_CODE store the address SEQ_CODE in CFA
      check_ground X         If X not ground jump to SEQ_CODE
      check_ground Y         If Y not ground jump to SEQ_CODE
      alloc_cross_prod 3, ADDR  Allocate a crossproduct node.
                                ADDR is the address from where
                                execution continues when a
                                tuple is picked up.
      put_and_value X, A1    load argument registers for b.
      put_and_value Y, A2
      push_and_call b1       push the and-call entry in the
                             and-goal stack.
      put_value X, A1        load argument registers for c.
      put_value Y, A2        Pick up c for execution.
      put_and_variable Z,A3  globalize Z for split trail optim.
      call c1,3              start c's execution.
HWC :                        Return here after a solution to
                             the and-subgoal found.
      alloc_solution ADDR    Push a solution node, store the
                             solution found, and check to see
                             if more unsolved and-goal present.
                             If yes, load registers & execute
                             one, else pick a tuple containing
                             the current solution, load E
                             register from parent cross-
                             product node and branch to ADDR.
ADDR :
      alloc_sequential       Push the sequential node, update
                             BA to execute sequential goal d.
      deallocate             Dealloc the env.  frame for f.
      execute CALL_d         execute d.
SEQ_CODE:                    branch here if CGE can't be
                             executed in parallel.
      put_value X, A1
      put_value Y, A2
      call b/2, 3
      put_value X, A1
      put_value Y, A2
      put_variable Z, A3
      call c/3, 3
CALL_d:
      put_value X, A1
      put_value Y, A2
      put_value Z, A3
      deallocate             Dealloc the env.  frame for f.
      execute d/3
a/2 : ... a's code ....
```

```
b1:   alloc_and HWC           allocate an and node for and-
                              parallel execution and set the
                              continuation code pointer to HWC.
b/2 : ... b's code ....

c1:   alloc_and HWC                        "

c/3 : ... c's code ....

d/3 : ... d's code ....
```

In the code above, when the processor reaches the end of an and-branch it executes an `alloc_solution` instruction. This instruction is responsible for checking if there are more untried and-nodes, and if there are none, picking a tuple so that the execution can continue with the next sequential goal. The binding array is loaded with the conditional bindings made along the and-branches corresponding to the tuple-elements in `alloc_sequential` instruction. When the processor reaches the end of an or-branch (recognized by the condition that the continuation pointer register points to the end of the top-level query) it reports the solution and then calls the task_switch routine of section 4.2. If failure occurs, while executing an and-branch or an or-branch, then too the task_switch routine is called.

6. Optimizing Binding Array Update

The major overhead in our scheme is incurred updating the binding arrays. There are two situations where we need to update binding arrays: (i) during task-switching, and, (ii) during the loading/unloading operation. The update overhead in task switching is inherited from the binding-arrays method, while the update overhead in loading/unloading results from the need to have flexibility in processor movement during task scheduling. We try to minimize the overhead due to task-switching by choosing a suitable scheduling strategy (keeping task granularity large so that processors switch tasks less often) but that has no effect on the loading/unloading overhead. In this section we discuss some optimizations which reduce the overhead in both task-switching and loading/unloading operations. These optimizations reduce the number of binding array updates to be performed during the operation. A more detailed description can be found in [13].

Splitting the Trail

Note that after an and-parallel subgoal G has been solved, subsequent goals are only interested in the bindings produced for G's unbound variables. Thus, once an and-parallel goal has been solved, then while loading the binding array we need only consider the conditional variables in G and ignore those of its descendents. This can be safely done because, even if the conditional variables in G get bound to conditional variables of its descendents, the conditional variables of the descendent nodes would not be accessed when G's variables are dereferenced because in WAM younger variables point to older ones. However, bindings of G's conditional variables might reside in the trail section of descendent frames. If we globalize the conditional variables in the and-parallel subgoal and split the trail into a global trail and a local trail, we need only consider the global trail during loading of a tuple in the binding array. Although we would still be loading some unneeded variables, we would save the work of loading all local conditional variables in the descendent subgoals. This justifies the inclusion of the instructions `put_and_variable` and `put_and_value`.

Promoting Variables

There are two instances where conditional variables can be *promoted* to unconditional variables, resulting in less task switch time: first, when a processor takes the last alternative from an or-node and is the last one using that or-node; and second, while moving up the tree when a processor passes an or-node which has just one active path below it. The first is similar to the WAM `trust` operation and to the *contraction* operation in the SRI model [36]. In both cases conditional variables up to the previous or-node can be made unconditional. When a variable is promoted, it also needs to be removed from the binding array.

Cross-product Enumeration

When a processor is moving up the tree and possibly unloading a crossproduct tuple, it is very likely that after getting to the crossproduct node it will pick up another tuple to continue execution. The branches corresponding to the new tuple would be loaded before execution is begun. However, the new tuple might have some elements common with the old tuple just unloaded, which we would have to load again. Thus, an obvious improvement would be to save the loading/unloading steps for the common elements in the tuple. This improvement has two advantages – not only less work is done, the contention for the node-stacks and trail is also reduced.

"Ground" CGEs

Frequently, the CGEs are of the form:

`(ground(X,Y) ⇒ b(X,Y) || c(X,Y) || d(X,Y,Z)).`

In such CGEs, X and Y would be *ground* if the condition succeeds; hence there is no need for processors to load their binding arrays from branches of `b` and `c` when they pick up a tuple from the crossproduct set of `b`, `c` and `d`. However, they do need to load their binding arrays from `d`'s branch, since `d` has a potential conditional variable, `Z`, as its argument. We believe that this optimization would improve the performance of the system, since the ground condition is frequently found in CGEs.

Trimming Binding Arrays

Because conditional variables are either local or global, if we have a separate binding array for local and global conditional variables, the concept of environment trimming can be extended to the local binding array. Separation of binding arrays also necessitates two base arrays, one each for the two binding arrays. Also offsets to local conditional variables can be determined at compile time. Thus the `call` and `put_variable` instructions are modified by adding an extra argument, similar to [25]. The extra argument in `call` is used for trimming the local binding array and that in `put_variable` for assigning the offset to the local conditional variable. Since local binding array gets trimmed, there are fewer conditional bindings to unload on a task-switch.

7. Conclusion

In this paper we have presented (i) a general model for exploiting and- and or-parallelism in a single framework, (ii) an extension of the binding-arrays method for environment representation in the presence of and-parallelism, (iii) a parallel execution strategy for coarse-grain parallelism, (iv) an extension of the WAM instruction set in terms of which combined and-or parallelism is initiated, and (v) optimizations to reduce the cost of task-switching incurred by the binding-arrays approach. The resulting system, called AO-WAM, differs from the WAM in that the new instructions support compilation of CGEs, sharing of and-parallel solutions, and efficient task-switching. Standard optimizations, such as last-call optimization and environment trimming, still apply, although the conditions under which they can be applied would slightly change due to solution sharing. A prototype implementation of AO-WAM is operational, and has shown encouraging results. In particular, the prototype has shown that the parallel overhead in the AO-WAM is only a small constant factor of the total sequential execution time [11].

Our combined and-or model preserves the performance characteristics of the binding-arrays method for pure or-parallelism and the RAP method for pure and-parallelism, namely, constant-time variable access, constant-time task creation, efficient dependency checking of subgoals, and restricted intelligent backtracking. Additionally, the computation of and-parallel subgoals are shared across different solution paths when these subgoals also exhibit or-parallelism, thus yielding better time and space performance. Even the main sources of overhead of the binding-arrays approach, i.e., task switching, as well as the loading/unloading overhead arising from sharing computation, are reduced in our model because of our optimizations.

Acknowledgments

Thanks are due to anonymous referees whose comments led to many improvements in the paper. This research was supported by grant DCR-8603609 from the National Science Foundation. Gopal Gupta was supported by SERC U.K. grant GR/F 27420, NSF grant CCR 9211732, and by grants from Sandia Labs (AE-1680), Oak Ridge Associated Universities (CK 004448), and NATO (CRG 921318) during the preparation and revision of this paper.

References

[1] T. Beaumont, S. Muthu Raman, et al. "Flexible Scheduling or Or-parallelism in Aurora: The Bristol Scheduler," In *Proceedings of PARLE '91*, Springer Verlag, LNCS 506, pp. 403-420.

[2] P. Biswas, S-C Su, D.Y.Y. Yun, "A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP) in Logic Programs," in *Fifth International Logic Programming Conference,* Seattle, WA., pp. 1160-1179.

[3] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek and R. Stevens, "Scheduling Or-Parallelism : An Argonne Perspective", In *Proceedings of Joint International Conference and Symposium on Logic Programming*, Seattle, Washington, 1988.

[4] A. Ciepielewski, S. Haridi, B. Hausman "Or-Parallel Prolog made efficient on shared memory multiprocessors", In *Journal of Logic Programming*, Vol 7, pp. 125-149.

[5] K. Clark, S. Gregory, "Parlog: Parallel Programming in Logic", In *A.C.M. TOPLAS*, Vol. 8, No. 1, Jan. 1986.

[6] J. S. Conery and D. F. Kibler. "And Parallelism in Logic Programs", In *Proceedings of the International Joint Conference in AI*, 1983.

[7] J. Conery, "Parallel Interpretation of Logic Programs," Kluwer Academic Press, 1987.

[8] J-H. Chang, A. M. Despain, and D. DeGroot, "And-Parallelism of Logic Programs based on Static Data Dependency Analysis", In *Digest of Papers of COMPCON Spring 1985*, pp. 218-225, 1985.

[9] D. DeGroot, "Restricted AND-parallelism", *Int'l Conf. on Fifth Generation Computer Systems*, Nov., 1984.

[10] G. Gupta, "A Timestamp-based Technique for Parallel Evaluation of Crossproduct Sets," In *Information Processing Letters* 44 (1992) 273-280.

[11] G. Gupta, "Parallel Execution of Logic Programs on Shared Memory Multiprocessors," Ph.D. Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, Dec. 1991.

[12] G. Gupta and B. Jayaraman, "On Criteria for Or-Parallel Execution Models of of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 604-623.

[13] G. Gupta and B. Jayaraman, "Optimizing And-Or Parallel Implementations," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 737-756.

[14] G. Gupta, V. Santos Costa, "And-Or Parallelism in Full Prolog with Paged Binding Arrays," In *Proc. 1992 Parallel Architectures and Languages Europe,* Lecture Notes in Computer Science, Springer-Verlag.

[15] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, "IDIOM: A Model for Integrating Dependent-and, Independent-and and Or-parallelism," In *Proceedings of International Logic Programming Symposium*, MIT Press, Oct. 1991.

[16] G. Gupta, M. Hermenegildo, "Recomputation based And-Or Parallel Execution of Prolog," In *Proceedings of International Conference on Fifth Generation Computer Systems (FGCS '92)*

[17] G. Gupta, D.H.D. Warren, "An Interpreter for the Extended Andorra Model," Technical Report, Department of Computer Science, University of Bristol, Forthcoming.

[18] S. Haridi and S. Janson, "Kernel Andorra Prolog and its Computation Model," In *Proceedings of ICLP*, MIT Press, June, 1990.

[19] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs", *3rd International Conference on Logic Programming*, London, 1986, pp. 25-39.

[20] M. V. Hermenegildo and R. I. Nasr, "Efficient Implementation of backtracking in AND-parallelism", *3rd International Conference on Logic Programming*, London, 1986.

[21] M. V. Hermenegildo, K.J. Green, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of the 7th International Conference on Logic Programming*, 1990, pp. 253-268.

[22] L. V. Kalé, "The REDUCE-OR model for Parallel Evaluation", In *4th International Conference on Logic Programming*, Melbourne, 1987, pp. 616-632.

[23] J. C. de Kergommeaux and P. Robert, "An Abstract Machine To Implement Or-And Parallel Prolog Efficiently," In *Journal of Logic Programming*, Vol 8, pp. 249-264, 1990.

[24] Y-J. Lin and V. Kumar, "AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor : A Summary of Results", in *Fifth International Logic Programming Conference,* Seattle, WA.

[25] E. Lusk, D.H.D. Warren, S. Haridi et. al. "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990 pp. 243-273.

[26] K. Muthukumar, M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation," In *Proc. of NACLP '89*, MIT Press.

[27] K. Muthukumar, M. V. Hermenegildo, "The DCG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism," In *Proceedings of the 7th International Conference on Logic Programming*, pp. 221-236.

[28] B. Ramkumar and L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of NACLP '89*, MIT Press.

[29] B. Ramkumar and L. V. Kalé, "Joining And Parallel Solutions in And/Or Parallel Systems," In *Proc. of NACLP '90*, MIT Press, pp. 624-641.

[30] E. Y. Shapiro (ed), "Concurrent Prolog: Collected Papers," MIT Press, 1987.

[31] P. Szeredi, "Performance Analysis of the Aurora Or-Parallel Prolog System," In *Proceedings fo the North American Conference on Logic Programming '89*, MIT Press, pp. 713-734.

[32] V. Santos Costa, D. H. D. Warren, R. Yang, "Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism," In *Proceedings of Principles & Practice of Parallel Programming*, Apr. '91, pp. 83-93.

[33] J. D. Ullman, Principles of Database and Knowledge-base Systems, Computer Science Press, 1989. Volume II.

[34] K. Ueda, "Guarded Horn Clauses," Ph.D. thesis, University of Tokyo, 1986.

[35] D. S. Wise, "Prolog Multiprocessors", Prentice-Hall, 1986.

[36] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues", *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 92-102.

[37] D. S. Warren, “Efficient Prolog Memory Management for Flexible Control Strategies”, In *The 1984 International Symposium on Logic Programming,* Atlantic City, pp. 198-202.

[38] D. H. D. Warren, “An Abstract Instruction Set for Prolog”, Tech. Note 309, SRI International, 1983, 28 pages.

[39] D.H.D. Warren, “Extended Andorra Model with Implicit Control” Talk given at Workshop on Parallel Logic Programming, 7th International Conference in Logic Programming, Eilat, Isreal, July 1990.

[40] H. Westphal, P. Robert, J. Chassin and J. Syre, “The PEPSys Model: Combining Backtracking, AND- and OR-parallelism”, In *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 436-448.