Combined And-Or Parallel Execution of Logic Programs

TR88-012

March 1988

(Revised December 1988)

Gopal Gupta

Bharat Jayaraman

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Combined And-Or Parallel Execution of Logic Programs†

Gopal Gupta
Bharat Jayaraman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract

This paper presents an extended and-or tree and an extended WAM (Warren Abstract Machine) model for efficiently supporting both and-parallelism and or-parallelism on shared-memory multiprocessors. Our approach is based on the binding-arrays method for or-parallelism and the RAP (Restricted And-Parallelism) method for and-parallelism. Our combined and-or model avoids backtracking, because of or-parallelism, and avoids redundant computations when goals exhibit both and- and or-parallelism, by representing the cross-product of the solutions from the and-or-parallel goals rather than re-computing it. Thus the extended and-or tree has two new nodes: a 'sequential' node (for RAP's sequential goals), and a 'cross-product' node (for the cross-product of solutions from and-or-parallel goals). The other main features of our approach are: (i) each processor's binding-array is accompanied by a base-array, for constant access-time to variables in the presence of and-parallelism; (ii) coarse-grain parallelism is supported by our processor scheduling policy, to minimize the cost of binding-array updates during task-switching; (iii) essentially, two new classes of WAM instructions are introduced: the 'check' instructions of RAP-WAM, and 'allocate' instructions for the different types of nodes. (iv) Several optimizations are proposed to minimize the cost of task switching. This extended WAM is currently being implemented on a Balance Sequent 8000.

† This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

1. Introduction

A number of approaches have been proposed for the parallel execution of logic programming languages, but the bulk of current research emphasizing practical implementations has dealt with either or-parallelism [HCH87, LWH88] or and-parallelism [D84, H86, LK88]. The obvious reason for combining and-parallelism and or-parallelism in a single framework is that any implementation that caters to either alone is suboptimal compared with one that caters to both. But there are other benefits: First, a combined model does not have to support backtracking unlike a pure and-parallel model; it suffices to determine subgoal independence, and initiate forward execution. Second, 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, given the usual definition of append, the pair of goals `? append(X, Y, [1,...,m]), append(P, Q, [1,...,n])`, leads to a `O(m*n)` computational cost under a pure or-parallel or pure and-parallel implementation (and also a sequential implementation), because all `n + 1` solutions for `P` and `Q` are re-computed for each of the `m + 1` solutions for `X` and `Y`. Instead of re-computing the solutions, if we represent the cross product of their solutions, the computational cost would only be of the `O(m + n)`. Note that such goals frequently arise in database searching.

In contrast to recently emerging approaches to combined and-or parallelism which emphasize execution on distributed systems [BSY88, C87, WR87, K87], we describe in this paper the combined and-or parallel execution for shared-memory multiprocessors, because we feel the latter best support the dynamic data creation of logic languages. We believe that ultimately a combination of the two approaches would be used in systems of the future. At a strategic level, we adopt a *processor-oriented* view of the computation [H86], which contrasts with most earlier models which adopt a process-oriented view. That is, there is one process per processor in our model, and different processors cooperate to build the and-or tree. This approach is well-suited to shared-memory multiprocessors; it avoids the excessive cost of process creation and also facilitates sharing of data across processors. Since shared-memory multiprocessors can be viewed as "multi-sequential" systems, it is also possible to adapt existing compilation techniques for sequential execution of logic programs, i.e. Warren Abstract Machine technology [W83], to such systems.

Our proposed approach is based upon our recently developed method for and-or parallelism [GJ88], which combines the binding-arrays method for or-parallelism [W84, LWH88] and the Restricted And-Parallelism (RAP) method [D84, H86]. We select the

binding-arrays method because it provides constant-time performance for the two most frequently occurring operations: variable access and task-creation. While task-switching is not constant-time in this method, one can minimize its cost by keeping the granularity of parallelism high [LWH88, HCH87]—an assumption that is compatible with current shared-memory multiprocessors. We also propose a number of new techniques to reduce the cost of task switching. We select the RAP method because it supports divide-and-conquer parallelism, which frequently occurs in algorithm design. Because the RAP method does not support stream-and-parallelism, our combined model also does not.

The rest of this paper is organized as follows: section 2 describes the major issues in or-parallelism and and-parallelism and states desirable criteria for their implementation; section 3 gives an overview of the combined and-or model, including the extended and-or tree, the variable-access method and the parallel execution strategy; section 4 describes the data areas and instruction set of the extended WAM, and the code for a simple example; section 5 discusses how task switching can be made more efficient; and section 6 presents conclusions and comparisons with related work.

We assume the reader has some familiarity with and-or parallel execution and the Warren Abstract Machine.

2. Parallelism in Logic Languages

We will now develop criteria which or- and and-parallel models should satisfy, and then mention some of the execution models for or-parallelism and and-parallelism. These criteria would be used in assessing our own combined and-or model to be described later.

2.1 Or Parallelism

Or-parallelism manifests itself whenever there is a non-deterministic search for solutions. In logic programs, or-parallelism arises when multiple clause heads unify with a goal. The subgoals arising from these multiple matches can be executed in parallel. There are three significant issues to be addressed in any or-parallel implementation:

1. An or-parallel implementation must be able to represent multiple bindings for variables which are unbound at the time of the match. D.H.D. Warren refers to such variables as *conditional variables* [W87a].

2. In addition to representing multiple bindings, an efficient implementation must ensure that the *access time* to such conditional variables and the *task-creation time* needed

is not prohibitive; ideally, they should be a constant independent of the size of the goal tree and independent of the size of the arguments of a goal.

3. A further requirement on or-parallel implementations arises from the finite nature of the underlying parallel machine. Because it is very likely that the number of or-parallel tasks will exceed the number of available processors—a valid assumption for commercially available shared-memory multiprocessors —the *task switching time* should also not be prohibitive, and ideally a constant independent of the goal tree.

We can thus sum up the criteria for an ideal or-parallel implementation as follows: constant access time to all variables; constant task-creation time; and constant task-switch time. 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 and environment trimming.

No method in the literature achieves all the criteria mentioned earlier; however, we think that the binding-arrays method comes closest, since it provides constant time performance for the two most frequently occurring operations: access to variables and task creation.

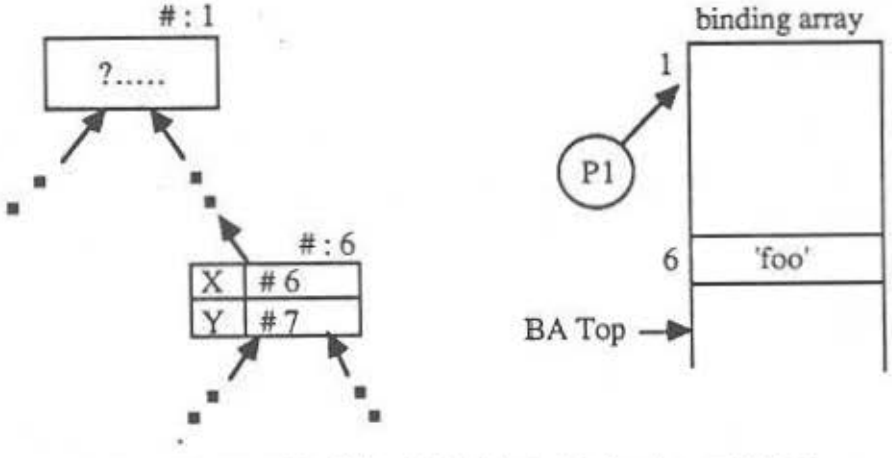

Fig 1 : Access to Conditional Variables in Binding Arrays Method

The binding-arrays method works by assigning to each conditional variable a unique offset, which is used to index into a *binding array*, local to the processor, to obtain the binding of these variables. Figure 1 shows how the variable `X` bound to the offset 6 is dereferenced by processor P1 to the atomic constant `foo`. The # indicates the offset counter value along the branch. The binding array method has the disadvantage that processor context-switch time is large since binding arrays need to be updated upon context switch.

This overhead can be alleviated 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, a depth-first search would perform comparably to a sequential implementation, and it supports standard sequential optimizations.

2.2 And Parallelism

And-parallelism in logic programs arises because multiple subgoals in a goal can be executed in parallel. Because executing dependent subgoals in parallel may result in wasteful computation we would like to execute only independent subgoals in parallel. Four different approaches have been proposed to handle dependencies: (1) by requiring explicit *annotations* from the programmer indicating which are "input" variables and which are "output" variables [CG86, S83]; (2) by monitoring the status for variables (bound or unbound), and dynamically re-structuring tasks [C87]; (3) by global compile time analysis (requiring no runtime checks) assuming worst cases for subgoal dependencies [CDD85]; and (4) by monitoring at runtime the status for terms (ground or nonground) and using a static task-structure, conditioned upon the status of terms, to obtain *restricted and-parallelism* [D84]. Approach (1) differs from (2), (3) and (4) in that the programmer has to explicitly specify the dependencies, using annotations. We do not further consider this approach here, because we are interested in automatic detection of and-parallelism. We consider (4) as a nice compromise between (2), where considerable run-time analysis is needed, and (3), where extensive compile time analysis is done.

Three important 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.

We use RAP to exploit independent and-parallelism, though our model would work for other methods such as [LK88, CDD85]. In the RAP method program clauses are compiled into Conditional Graph Expressions (CGEs) of the form

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

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 $ground(v_1, \ldots, v_n)$, 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 mutually exclusive of one another. Checking for *ground* and *independence* involve very simple runtime tests, details of which are presented in [D84]. 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". This model has been extended by Hermenegildo and Nasr, and has been efficiently implemented using WAM-like instructions.

2.3. Combined And-Or Parallelism

The criteria for combined and-or parallel implementation are 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). Also as mentioned in section 1, the combined model does not have to support any backtracking, unlike a pure and-parallel model, because of the presence of or-parallelism. The realization of and-parallelism is simplified in this respect; it suffices to detect subgoal independence and initiate their forward execution. Also recomputation should be avoided whenever both and- and or-parallelism arise in solving a goal.

Finally, we should expect an and-or parallel implementation to produce solutions at least as fast as (if not much faster than) a sequential implementation. This implies that preference should be given to and-parallel tasks over or-parallel tasks if there are more tasks than available processors. 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, and also favor and-parallelism over or-parallelism if there are limited processors.

3. Overview of the Combined And-Or Model

Our approach is to combine the binding-arrays model for or-parallelism and the RAP model for and-parallelism. Thus programs are compiled into CGEs before execution. We begin by describing extensions to the basic and-or tree. Figure 2 shows an extended and-or tree for a simple example. There are four kinds of nodes in the extended tree. In addition to *and* nodes and *or* nodes, we also have *cross-product* nodes, to hold the cross-product of solutions from and-or parallel goals, and *sequential* nodes, which correspond to sequential

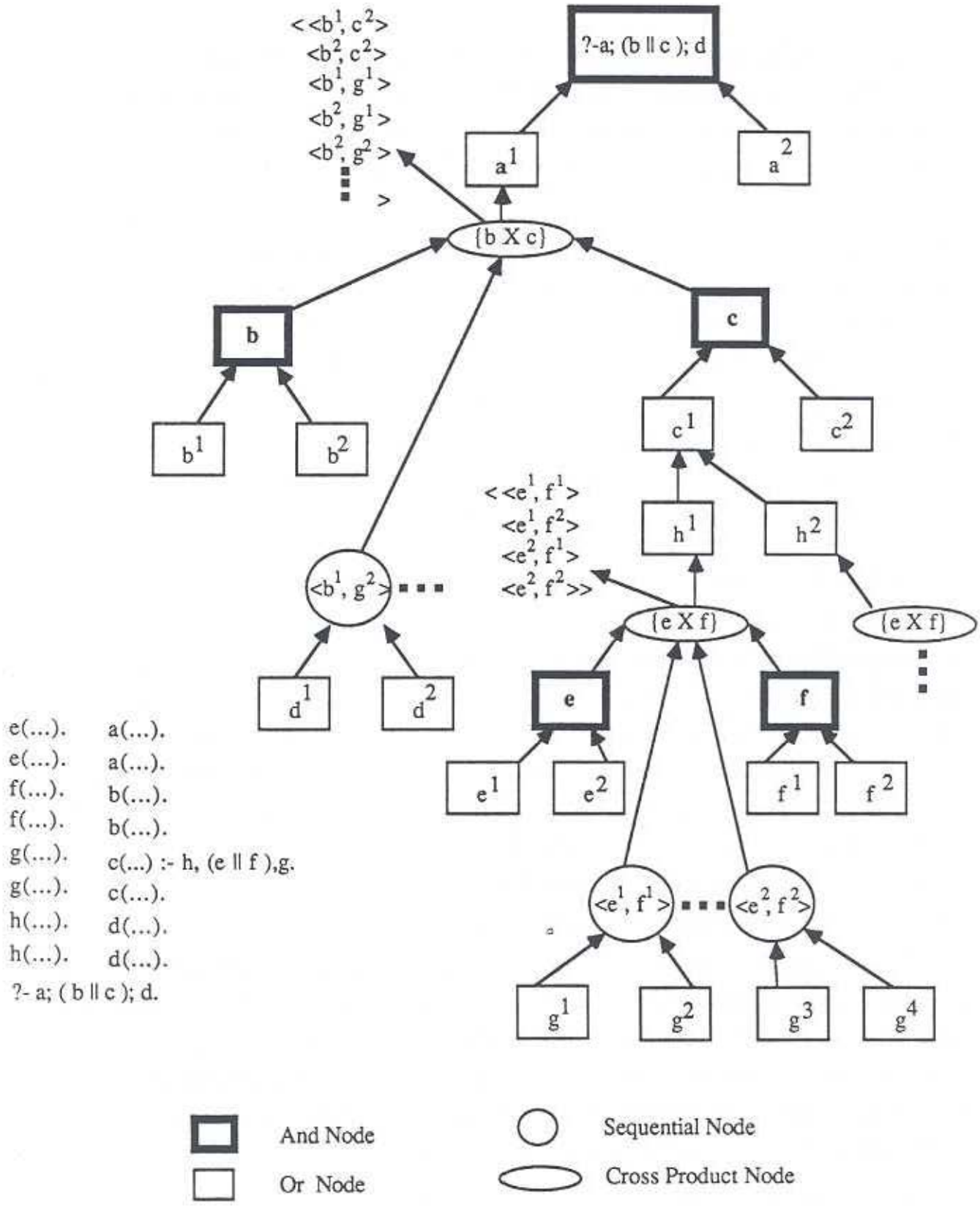

Figure 2. An Example of an And-Or Tree

goals in the RAP model. All cross-product nodes are parents of and-nodes and sequential nodes, and correspond to join-nodes in the Pepsys model [WR87]. There is one sequential node for each tuple in the cross-product set. Nodes have space for their subgoals (also called goal-list), and or-nodes have space for the bindings of the variables occuring in the sub-goals.

Note that branches rooted at a cross-product node can be grown in and-parallel; such branches do not have data-dependencies between them. Branches rooted at nodes other than cross-product nodes can be grown in or-parallel; for such branches we have to ensure that the environment of each or-parallel branch is correctly maintained. This is a general scheme for exploiting and-or parallelism in logic programs. Not described in the extended and-or tree is (a) how variables are accessed in constant-time, (b) how a collection of processors in a shared-memory multiprocessor cooperate to grow the tree, and (c) how task-switching is done. We address these issues in the next two subsections.

3.1. Variable Access

When dealing with both and- 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 ';' stands for sequencing and 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, 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. 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 [W84, W87] 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 and at run time a processor can traverse them in any order. We introduce another level of indirection in the binding array to get around this problem.

In addition to the binding array, each processor also maintains another array called the *base array*. As each and-node is created, it is assigned a unique integer id. When a processor encounters an and-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 id of the and-node.

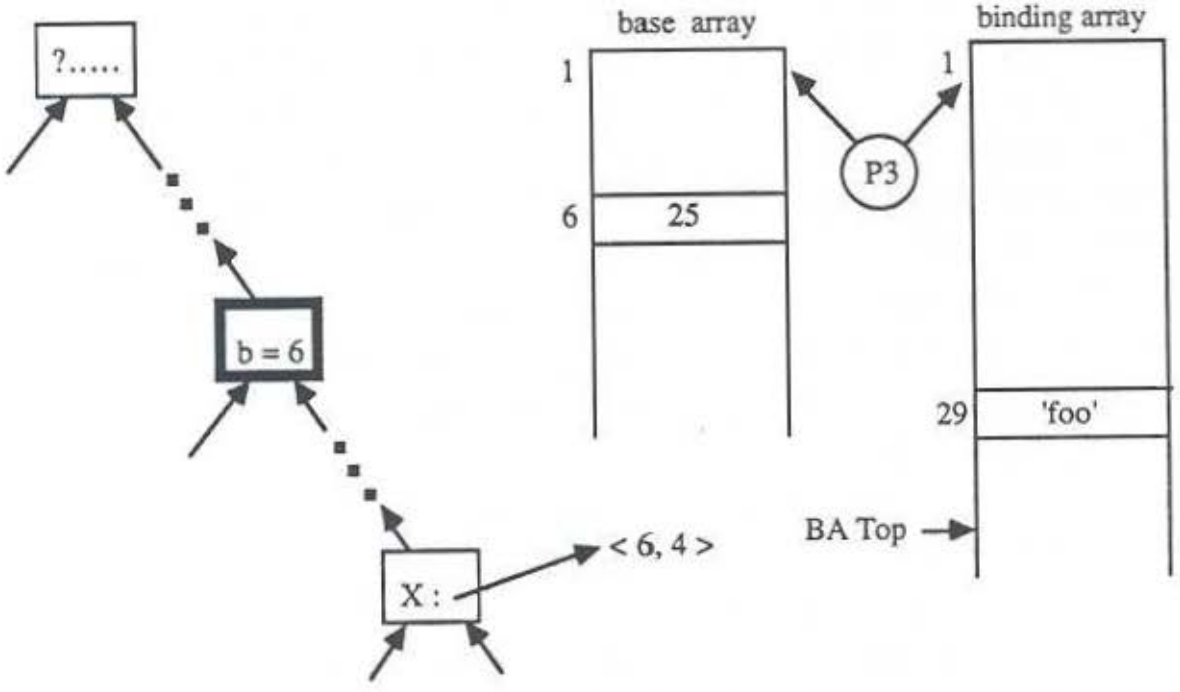

Fig 3 : Access to Conditional Variables in Extended Binding Array Method

The offset-counter is reset to zero when an and-node is created. Subsequent conditional variables are bound to the pair $\langle i, v \rangle$, where v is the value of the counter. The counter is incremented after each conditional variable is bound to this pair. The dereferencing algorithm is described below.

```
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 value→val == val
                V
        else deref(val) }}
```

Note that access to variables is constant-time, though 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. Figure 3 shows how a variable bound to the pair $(6, 4)$ is dereferenced to the atomic value `foo`

3.2. Parallel Execution

Because we are targeting our implementation at shared-memory multiprocessors, we can assume the extended and-or execution tree lies in a memory space accessible to all processors. In our proposed scheme, processors traverse the branches of the tree, executing sub-goals 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 it. 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. Below we provide the algorithm that a processor uses for selecting work in the extended and-or tree. In the algorithm, two basic operations *load* and *unload* are assumed. In the load operation, a processor, given a cross-product tuple, updates its binding array with the conditional variables that are found in the and-branches of the solutions corresponding to the tuple; the base array is simultaneously updated. We therefore say the processor loads its binding array with the tuple. During the unload operation, conditional variables occuring in the and-branch of the solutions in the tuple are purged from the binding array; the base array is also purged.

```
task_switch()
            /*A is the current node.*/
   Case A of
      and_node:          if A is not equal to root
                            if there are more untried siblings of A
                                pick A's untried sibling and-node for execution.
                            else
                                get a tuple containing the solution found.
                                load the binding array with the tuple.
                                execute sequential goal after the CGE.
      or_node:           if or-node has more alternatives
                            execute an alternative clause
                         else
                            if work available at a node below /* explained later */
                               move to the node where work is available.
                               update binding array (and base array, if needed).
                               carry out the work.
                            else
                               move to the node further up.
                               update binding array.
```

```
                    task_switch().
sequential_node:    unload the tuple in the sequential node.
                    If while unloading, a node with work is found in the and-branch
                       execute that work.
                    else
                       move up to the parent cross-product node
                       if an untried tuple is found
                          load binding array with the tuple
                          execute the sequential goal for that tuple.
                       else
                          move one node up
                          task_switch().
```

Note that the case for a cross-product need not be treated explicitly since a processor reaching a sequential node also has access to its parent cross-product node. In order to determine when work is available below an or-node, we assume that each processor maintains a pool of work it produces, which it will eventually carry out if unaided by no other processor. While choosing available work preference is given to untried and-nodes over untried or-nodes and cross-product tuples, so that solutions are produced at least as fast as (if not much faster than) a sequential implementation.

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 move further up.

4. An Extended WAM for the And-Or Model

Figure 4 summarizes the state of an AO-WAM processor — all processors have a similar state. As a processor executes the extended WAM instructions (described in section 3.2) 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 [W83].

4.1. AO-WAM Machine State

A. 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

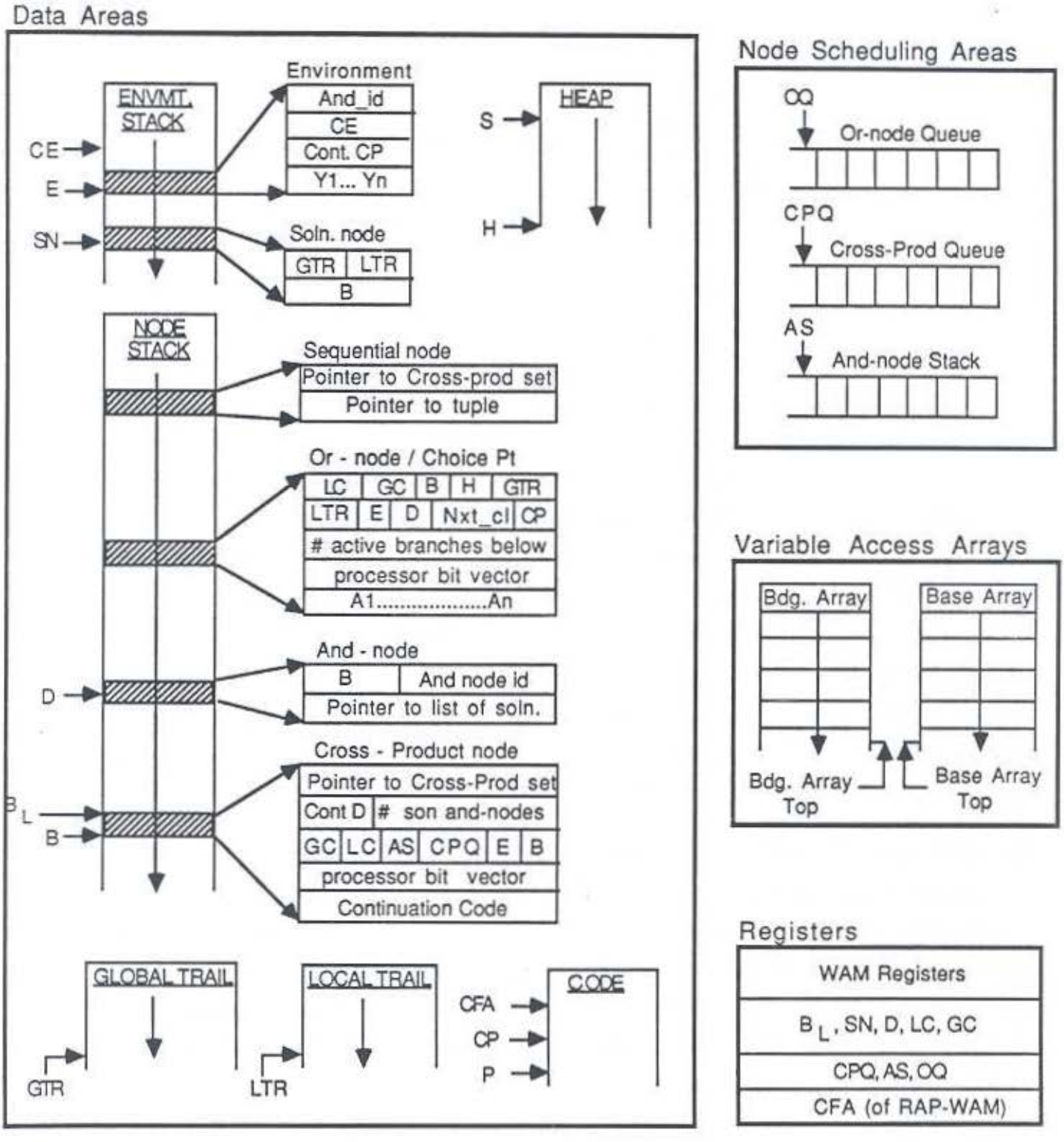

Figure 4 AO-WAM Processor State

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*: The advantage of this separation is two-fold: 1) During space allocation, it is easy for the processors to access the top-most node in their local stack (through B_L). 2) It simplifies 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 4).

(iv) *Introduction of solution nodes*: A solution node is pushed on the environment stack when the end of an and-branch is reached. It serves two purposes: 1) Its address is used as the *name* for the corresponding solution in cross-product tuple. 2) It makes sure that an and-parallel solution does not get deleted from the stack until the entire cross-product has been tried.

B. 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 cross-product 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.

Or-nodes and Cross-product nodes have a *processor bit-vector* (similar to [HCH87]). 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 id 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.

C. Variable Access Arrays: These have been in explained in section 2.1.

D. Registers: In addition to the regular WAM registers we have the following extra registers. (i) B_L, which points to the top of the node stack. (ii) 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 cross-product nodes so that it can be restored when sequential nodes are pushed. (iii) SN, the solution node register, which points to the top most solution node. Space below SN cannot be reclaimed until the and-branch has been fully solved. (iv) LC and GC, which are the offset counters for the local and global conditional variables, respectively. (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.

4.2. AO-WAM Instruction Set.

The AO-WAM supports most of the instructions supported by WAM. However, some of the instructions have been modified. Some new instructions have also been added.

The modified instructions are `call P, n, k`, `put_variable Yn, Ai, k` and `allocate`. These instructions are similar to those in Aurora [LWH88]. The third argument in `call` is needed to implement an optimization similar to environment trimming on the local binding array. The third argument in `put_variable` is used to allocate offsets to local conditional variables at compile time. The `allocate` instruction has to take care that it allocates space above the top-most solution node so that and-branches that are still needed are not destroyed.

The new instructions introduced in the AO-WAM consist of the *check* instructions (`check_me_else, check_ground` and `check_independent`) of RAP-WAM [H86] for compiling CGEs, and instructions for allocating space for various nodes: `alloc_cross_prod 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` (`check_independent`) instruction checks if the variables in their arguments are grounded (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. Two interesting instructions are:

`put_and_variable Yn, Ai, j`: This instruction is the same as the `put_variable Yn, Ai, j` instruction except that the variable `Yn` is globalized and a reference to the global value is saved in `Ai`. This instruction globalizes unbound variables in the and-parallel subgoals, so that during update (loading and unloading tuples) of the binding arrays,

the processor has to only look at the trail for global variables.

`push_and_call Code/n`: similar to the `push_call` instruction in RAP-WAM. Push an entry into the and-goal stack, i.e., push the instruction address of the sub-goal, 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.

4.3. Example

In this section we give the compiler generated AO-WAM code for the a simple clause. The code is annotated to explain the effect of the instructions. The source program is

```
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,Z) || c(X,Y)), 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 argument register to execute a.
    put_value Y, A2
    call a/2, 3, 1            Call a
    check_me_else SEQ_CODE    store the address SEQ_CODE in CFA
    check_ground X            If X is not ground jump to SEQ_CODE
    check_ground Y            If Y is not ground jump to SEQ_CODE
    alloc_cross_prod ADDR     Allocate a cross product node.  ADDR
                              is the address from where execution
                              continues when a tuple is picked up.
    put_value X, A1           load argument registers for b.
    put_value Y, A2
    push_and_call b1/2        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,A3,0   globalize Z for split trail optimization
    call c1/3,3,1             the value of the 2nd & 3rd arguments doesn't
                              matter since B will be more recent than E.
HWC :                         Return here when an and-branch finishes.
    alloc_solution ADDR       Push a solution node, store the solution found &
```

```
                                check to see if more and-goal still unsolved.  If
                                yes, load regs.  & execute one, else load BA with a
                                tuple containing current solution, load E reg.  from
                                the parent cross-product node and branch to ADDR
ADDR :
      alloc_sequential          Push the sequential node and update
                                the BA to execute sequential code d.
      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, 1
      put_value X, A1
      put_value Y, A2
      put_variable Z, A3
      call c/3, 3, 1
CALL_d:
      put_value X, A1
      put_value Y, A2
      put_value Z, A3
      execute d/3
b1:   alloc_and HWC             alloc.  an and node (and-parallel execution).
                                set the continuation code to HWC
b/2 : ... b's code ....
c1:   alloc_and HWC             alloc.  an and node (and-parallel execution).
c/3 : ... c's code ....
d/3 : ... d's code ....
```

5. Efficient Task Switching

Task switching is an overhead in the AO-WAM which is inherited from the binding arrays method. We have tried to minimize it by choosing a suitable scheduling strategy, as discussed in section 2.1, so that processors switch less often. We now discuss further ways to minimize the work involved in task switching; all of them aid in reducing the amount of work involved in updating binding arrays.

5.1. 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, while loading the binding array we need only consider the conditional variables in G and ignore those of its descendent. 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 since 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 binding array loading. 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 instruction `put_and_variable`. Note that the binding array has to be loaded from both global and local trails during a task switch.

5.2. 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, when a processor while backtracking 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 [W87]. 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.

5.3. Cross-product Enumeration

When a processor is backtracking and possibly unloading a cross-product tuple, it is very likely that after getting to the cross-product 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.

5.4. Grounded 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 cross-product 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 tremendously improve the performance of the system, since the ground condition is very frequently found in CGEs.

6. Conclusions and Related Work

The combined and-or model presented in this paper preserves the 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, there is no need to restart and-parallel goals, as required in [H86], and the computation of and-parallel subgoals are shared across different solution paths, resulting in better time and space performance. Standard optimizations, such as last-call and environment trimming, still apply, though the conditions under which they can be applied would slightly change due to the sharing of nodes. Furthermore, if there is only one processor available, the execution would be as fast as a sequential implementation, with the added advantages of limited intelligent backtracking, no redundant computations and no restarting of and-parallel goals. Even the main source of overhead of the binding arrays approach, i.e., task switching, is minimized in our model.

Note that sharing frames across different or-paths would always save time compared with re-computing goals. If there was no sharing, all and-branches would be replicated and recomputed. During this computation all variables encountered would be accessed *at least once* during unification. However, if branches are shared, we save time by not pushing frames on the stacks, and also during loading only the conditional variables are accessed, which are far fewer in number. If we take into account our optimizations for task switching, the number of such conditional variables would be even less.

Another noteworthy point is that the amount of traversing the processors would perform in context switching would not be more than in the case of purely or-parallel binding array method (for example, [LWH88]). In fact, since the solutions to and-parallel sub-goals are shared, the amount the traversing would be much less. If we take into account the optimizations discussed in section 4, the amount of traversing involved in a task switch would be reduced further significantly. Thus, in our model, and-parallelism has the effect of improving the performance obtained from or-parallelism.

A number of other research projects have aimed at realizing both and- and or-parallelism in a single implementation. Of these [WR87, K87] come closest to ours; most others are intended for specialized architectures. The PEPSys model from ECRC [WR87] is a practical model for and-or parallelism. Its limitations, however, are that variable access, based on time-stamping, is not a constant-time operation. Since the join appears not to be incrementally computed, processor cycles could be wasted in synchronizing. The PEPSys model, however, is not tied down to a fixed architecture and can be implemented on shared or non-shared memory multiprocessors. Kale's reduce-or model [K87] uses *Data Join Graphs* for exploiting parallelism. Although the technique also exploits dependent and-parallelism it requires potentially time-consuming *back-unification* to ensure the consistency of bindings. Also Or-processes require full copies of arguments, which might be a significant overhead.

We have begun implementing the extended WAM model described in this paper on a Balance Sequent 8000. We expect experimental results on our implementation by April 1989.

References

[BSY88] 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.

[CG86] K. Clark and S. Gregory, "Parlog: Parallel Programming in Logic", In *A.C.M. TOPLAS*, Vol. 8, No. 1, Jan. 1986.

[C87] J. S. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors", *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 457-467

[CDD85] 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.

[D84] D. DeGroot, "Restricted AND-parallelism", *Int'l Conf. on Fifth Generation Computer Systems*, Nov., 1984.

[GJ88] G. Gupta and B. Jayaraman, "Combined And-Or Parallel Execution of Logic Programming Languages", Technical Report TR-88-012, Dept. of Computer Science, UNC Chapel Hill, Mar '88, 23 pages.

[HCH87] B. Hausman, A. Ciepielewski, and S. Haridi, "Or-Parallel Prolog made efficient on shared memory multiprocessors", in *1987 IEEE International Symposium in Logic Programming*, San Francisco, pp. 69-79.

[H86] 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.

[K87] L. V. Kale, "The REDUCE-OR model for Parallel Evaluation", In *4th International Conference on Logic Programming*, Melbourne, 1987, pp. 616-632.

[LK88] 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.

[LWH88] E. Lusk, D.H.D. Warren, S. Haridi et. al. "The Aurora Or-Prolog System", Internal Report, Gigalips Project, 19 pages.

[S83] E. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", ICOT Tech. Report TR-003, ICOT, Tokyo, Feb., 1983.

[W83] D. H. D. Warren, "An Abstract Instruction Set for Prolog", Tech. Note 309, SRI International, 1983, 28 pages.

[W87] 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.

[W87a] D. H. D. Warren, "Or-Parallel Execution Models of Prolog", TAPSOFT '87, Springer Verlag, LNCS 250.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies", In *The 1984 International Symposium on Logic Programming*, Atlantic City, pp. 198-202.

[WR87] 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.