Paged Binding Array:
Environment Representation for And-Or Parallel Prolog

Gopal Gupta

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.
`gupta@compsci.bristol.ac.uk`

Abstract

We show that to exploit both Independent And- and Or-parallelism from Prolog programs independent goals should be recomputed (*vs* their solutions reused). We present an abstract model, called the Composition-Tree, for representing and-or parallelism in Prolog Programs. The Composition-tree closely mirrors sequential Prolog execution (by recomputing independent goal rather than re-using them). We also present an extension of the Binding Array scheme for And-Or parallel execution of *full* Prolog based on the Composition-tree. A complete abstract machine for and-or parallel Prolog is presented along with scheduling and memory management issues. We demonstrate why our system is better than the earlier proposals for exploiting and-or parallelism.

Keywords: Or-parallelism, Independent And-parallelism, Binding Arrays, RAP-WAM, Paging.

1. Introduction

One of the main features of logic programming languages is that they allow implicit parallel execution of programs. There are three main forms of parallelism that are present in logic programs: Or-parallelism, Independent And-parallelism and Dependent and-parallelism. In this paper we restrict ourselves to Or-parallelism and Independent and-parallelism. There have been numerous proposals to exploit or-parallelism in logic programs [AK90, HCH87, LWH90, W84, W87, etc.]† and quite a few to exploit independent and-parallelism [H86, LK88, etc.] Models have also been proposed to exploit both or-parallelism and independent and-parallelism in a single framework [GJ89, RK89, BK88]. It is this aspect of combining independent and- and or-parallelism that this paper addresses.

Most models that have been proposed (or implemented) so far for combining or-parallelism and independent and-parallelism have either considered only pure logic programs (pure Prolog) [RK89, GJ89], or have modified the language to separate parts of the program that contain extra-logical predicates (such as cuts and side-effects) from those that contain purely logical predicates, allowing parallel execution only in parts containing purely logical predicates [RS87, BK88]. In the former case practical Prolog programs cannot be executed since most practical Prolog programs have extra-logical features. In the latter case programmers have to divide the program into sequential and parallel parts themselves, as a result of which parallelism is not exploited completely implicitly since some programmer intervention is required. The latter approach also rules out the possibility of taking "dusty decks" of existing Prolog programs and running them in parallel. In

† See [GJ90] for a systematic analysis of the various models

addition, some parallelism may also be lost since parts of the program that contain side-effects may also be the parts that contain parallelism. It has been shown that or-parallelism and independent and-parallelism can be exploited in full Prolog completely implicitly (for example, the Aurora and Muse System [HCC88, LWH90, AK91], and the &-Prolog system [HG90, MH89, CC89]). We contend that the same can be done for systems that combine independent and- and or-parallelism. We present techniques for incorporating side effects in and-or parallel systems to execute full Prolog in a companion paper [GS91]. In this paper we present a concrete model which facilitates the inclusion of side effects. We discuss issues such as environment representation, scheduling, and memory management for this model. Our approach is two-pronged, first we present an abstract model of and-or parallelism for logic programs which mirrors sequential Prolog execution more closely (essentially by recomputing independent goal rather than re-using them); second, we introduce the concept of teams of processors to exploit and-or parallelism and present an efficient environment representation scheme based on Binding Arrays [W84,W87] to support it.

Our environment representation technique is based on Binding Arrays and is an improvement of the technique used in the AO-WAM [GJ89]. We use Conditional Graph Expressions (CGEs) [HN86, H86] to express independent and-parallelism†. Hence our model can be viewed as a combination of the &-Prolog system and the Aurora system—in presence of only independent and-parallelism our model behaves *exactly* like &-Prolog while in the presence of or-parallelism it behaves *exactly* like Aurora.

The rest of the paper is organised as follows: Section 2 presents arguments for favouring recomputation of independent and-parallel goal over their reuse; section 3 presents an abstract model called the Composition-tree for representing and-or parallel execution of Prolog with recomputation; Section 4 presents reasons for favouring binding arrays method over other environment representation techniques (notably stack copying [GH91]). Section 4 also presents an extension of the Binding arrays method for and-or parallel execution of Prolog programs based on the composition tree. Section 5 describes an abstract machine for compiled execution of and-or parallel Prolog based on the Composition-tree and extended Binding Arrays; while section 6 presents our conclusions. We assume the reader is familiar with Binding Arrays [W84, W87], the Aurora System [LWH90], Conditional Graph Expressions (CGEs) [HN86], and RAP-WAM [H86].

2. Recomputation *vs* Reuse

In the presence of both and- and or-parallelism in logic programs, it is possible to avoid recomputing certain goals (termed as solution sharing [GJ89]). For example, consider two independent goals `a(X), b(Y)`, each of which has multiple solutions. The most efficient way to execute this goal would be to execute `a` and `b` in their entirety and combine their solutions (possibly incrementally) through a join [BK88, GJ89, RK89]. However, to solve the above goal in this way one needs to be sure that the set of solutions for `a` and `b` are *static* (i.e., if either goal is executed multiple times, then each invocation produces an identical set of solutions). Unfortunately, this can hold true only if clauses for `a` and `b` are pure logic programs. If side-effects are present (as is usually the case with Prolog programs), then set of solutions for these goals may not be static. For example, consider the case where, within `b`, the value of a variable is read from the standard input and

† Note that CGEs can be generated at compile time [MH89a] using abstract interpretation and thus the programmer is not burdened with inserting them.

then some action taken which depends on the value read. The solutions for `b` may be different for every invocation of `b` (where each invocation corresponds to a different solution of `a`). Hence solution sharing would yield wrong results in such a case. The simple solution of sequentialising such and-parallel computations results in loss of too much and-parallelism, because if `a(X), b(Y)` falls in the scope of some other goal, which is being executed in and-parallel, then that goal has to be sequentialised too, and we have to carry on this sequentialisation process right up to the top level query. If, however, the goals are recomputed then this sequentialisation can be avoided, and parallelism exploited even in the presence of cuts and side-effects [GS91]. Hence, there is a strong argument for recomputing non-deterministic and-parallel goals, especially, if they are not pure, and even more so if we want to support Prolog as the user language†.

Additionally, recent simulations done on and-or parallelism [SH91] show that typical Prolog programs perform very little recomputation, thus providing further evidence that the amount of computation saved by a system which avoids recomputation may be quite small in practice‡. Presumably this behaviour is due to the fact that Prolog programmers, aware of the selection and computation rules of Prolog, order literals in ways which result in efficient search which minimises the recomputation of goals. Note that the use of full or partial recomputation can never produce any slowdown with respect to Prolog since Prolog itself uses full recomputation.

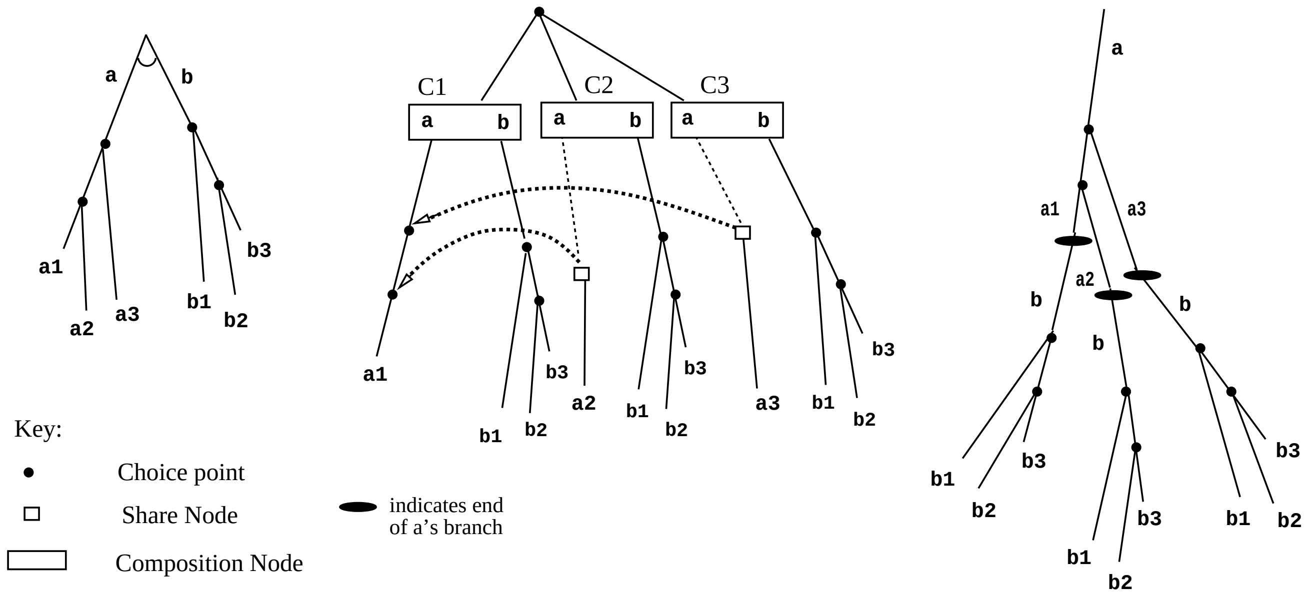

fig 1: And-Or Tree fig 2: Composition Tree fig. 3: Or-Parallel Tree

3. And-Or Composition Tree: And-Or parallelism with Recomputation

The most common way to express and- and or-parallelism in logic programs is through traditional and-or trees, which consist of or-nodes and and-nodes. Or-nodes represent multiple clause heads matching a goal while and-nodes represent multiple subgoals in the body of a clause being executed in and-parallel. Since in our model and-parallelism is permitted only via CGEs, we restrict and-nodes to represent only CGEs. Thus, given a clause `q :- (true => a & b)`,

† There is a third possibility as well: to recompute those independent and-parallel goals that have side-effects and share those that don't.

‡ The idea of recomputation is also referred to as "or-under-and" in [SK91].

where a and b have 3 or-parallel solutions each, and the query ?- q, then the resulting and-or tree would appear as shown in figure 1.

The problem with traditional and-or trees is that bindings made by different alternatives of a are not visible to different alternatives of b, and vice versa, and hence the correct environment has to be created before the continuation goal of the CGE can be executed. Creation of the proper environments requires a global operation, for example, *Binding Array loading* in AO-WAM [GJ89], the complex dereferencing scheme of PEPSys [BK88], or the "global forking" operation of Extended Andorra Model [W90]. To eliminate this problem in our model, we extend the traditional and-or tree so that the various or-parallel environments that simultaneously exist are always separate.

The extension essentially uses the idea of recomputing independent goals of a CGE. Thus, for every alternative of a, the goal b is computed in its entirety. Each separate combination of a and b is represented by, what we term as a *composition* node (c-node for brevity)†. Thus the extended tree for the above query would appear as shown in figure 2—for each alternative of and-parallel goal a, goal b is computed in its entirety. This extended tree, which we call the *Composition-tree* (C-tree for brevity), is somewhat similar to the Reduce-Or tree of Kalé [K85]. Note that each composition node in the tree corresponds to a different solution for the CGE. To represent the fact that a CGE can have multiple solutions we add a branch point (choice point) before the different composition nodes. The C-tree can represent or- and independent and-parallelism quite naturally—execution of goals within a single c-node gives rise to independent and-parallelism while execution of different c-nodes, and of untried alternatives which are not below any c-node, gives rise to or-parallelism. The C-tree of a logic program is dynamic in nature, and its shape may depend on the number of processors and how they are scheduled, as will be explained shortly.

Notice the topological similarity of the C-tree with the purely or-parallel tree shown in figure 3 for the program above. Essentially, branches that are shared in the purely or-parallel tree are also shared in the C-tree. This sharing is represented by means of a *share-node*, which has a pointer to the shared branch and a pointer to the composition node where that branch is needed (figure 2). Due to sharing the subtrees of some independent and-parallel goals maybe spread out across different composition nodes. Thus, the subtree of goal a is spread out over c-nodes C1, C2 and C3 in the C-tree of figure 2.

3.1 Independent And-Parallelism and Teams of Processors

When a purely or-parallel model is extended to exploit independent and-parallelism then the following problem arises: at the end of independent and-parallel computation, all participating processors should see all the bindings created by each other. However, this is completely opposite to what is needed for or-parallelism where processors working in or-parallel should not see the (conditional) bindings created by each other. Thus, the requirements of or-parallelism and independent and-parallelism seem anti-thetical to each other. The solution that have been proposed range from updating the environment at the time independent and-parallel computations are combined [RK89, GJ89] to having a complex dereferencing scheme [BK88]. All of these operations have their cost.

† The composition-node is very similar to the Parcall node of RAP-WAM [H86].

We contend that this cost can be eliminated by organising the processors into teams. Independent and-parallelism is exploited between processors within a team while or-parallelism is exploited between teams. Thus a processor within a team would behave like a processor in a purely and-parallel system while all the processors in a given team would collectively behave like a processor in a purely or-parallel system. This entails that all processors within each team share the data structures that are used to maintain the separate or-parallel environments. For example, if binding arrays are being used to represent multiple or-parallel environments, then only one binding array should exist per team, so that the whole environment is visible to each member processor of the team. Note that in the worst case there will be only one processor per team. Also note that in a team setup a processor is free to migrate to another team as long as it is not the only one left in the team. Results of simulation of combined independent and- and or-parallelism [SK91, SH91] have shown that most CGEs have 2 or 3 independent goals, hence a team size of 2 or 3 is adequate for most programs.

The concept of teams of processors has been successfully used in the Andorra-I system [SWY91], which extends an or-parallel system to accommodate dependent and-parallelism.

3.2. C-tree and Independent And-Parallelism

The concept of organising processors into teams also meshes very well with C-trees. A team can work on a c-node in the C-tree—each of its member processors working on one of the independent and-parallel goal in that c-node. We illustrate this by means of an example. Consider the query corresponding to the and-or tree of figure 1. Suppose we have 6 processors P1, P2, ..., P6, grouped into 3 teams of 2 processors each. Let us suppose P1 and P2 are in team 1, P3 and P4 in team 2, and P5 and P6 in team 3. We illustrate how the C-tree shown in figure 2 would be created.

Execution commences by processor P1 of team 1 picking up the query `q` and executing it. Execution continues like normal sequential execution until the CGE is encountered, at which point a choice point node is created to keep track of the information about the different solutions that the CGE would generate. A c-node is then created (node C1 in figure 2). The CGE consists of two and-parallel goals `a` and `b`, of which `a` is picked up by processor P1, while `b` is made available for and-parallel execution. The goal `b` is subsequently picked up by processor P2, teammate of processor P1. Processor P1 and P2 execute the CGE in and-parallel producing solutions `a1` and `b1` respectively. In the process they leave choice points behind. Since we allow or-parallelism below and-parallel goals, these untried alternatives can be processed in or-parallel by other teams. Thus the second team, consisting of P3 and P4 picks up the untried alternative corresponding to `a2`, and the third team, consisting of P5 and P6, picks up the untried alternative corresponding to `a3`. Both these teams create a new c-node, and restart the execution of and-parallel goal `b` (the goal to the right of goal `a`): the first processor in each team (P3 and P5, respectively) executes the alternative for `a`, while the second processor in each team (P4 and P6, respectively) executes the restarted goal `b`. Thus, there are 3 copies of `b` executing, one for each alternative of `a`. Note that the nodes in the subtree of `a`, between c-node C1 and the choice points from where untried alternatives were picked, are shared between different teams.

Since there are only three teams, the untried alternatives of `b` have to be executed by backtracking. In the C-tree, backtracking always takes place from the right to mimic Prolog's behaviour—goals to the right are completely explored before a processor can backtrack inside a goal to the left. Thus, if we had only one team with 2 processors, then only one composition node would be created, and all solutions would be found via backtracking, exactly as in &-Prolog [H86, HG90]. On the other hand if we had 5 teams of 2 processors each, then the C-tree could appear as shown in fig 4. In figure 4, the 2 extra teams steal the untried alternatives of goal `b` in c-node C3, This results in 2 new c-nodes being created, C4 and C5 and the subtree of goal `b` in c-node C3 being spread across c-nodes C3, C4 and C5. The topologically equivalent purely or-parallel tree of this C-tree is still the one shown in figure 3. The most important point to note is that new c-nodes get created only if there are resources to execute that c-node in parallel. Thus, the number of c-nodes in a C-tree can vary depending on the availability of processors.

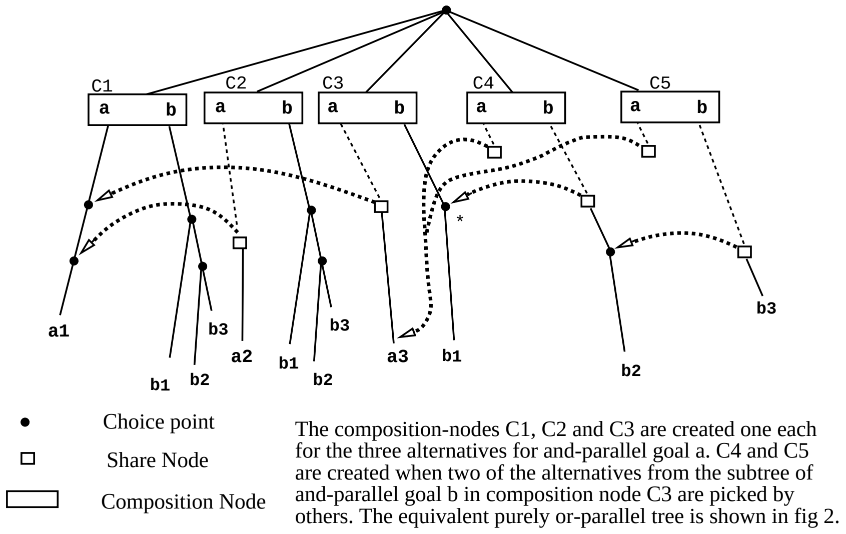

Figure 4: C-tree for 5 Teams

4. Environment Representation for And-Or parallelism

So far we have described and-or parallel execution with recomputation at an abstract level. We have not addressed the problem of environment representation in the C-tree. In this section we discuss how to extend the Binding Arrays (BA) method [W84,W87] to solve this problem. These extensions enable a team of processors to share a single BA without wasting too much space.

4.1 Sharing *vs* Non-Sharing

In an earlier paper [GJ90] we argued that environment representation schemes that have constant-time task creation and constant-time access to variables, but non-constant time task-switching, are superior to those methods which have non-constant time task creation or non-constant time variable-access. The reason being that the number of task-creation operations and the number of variable-access operations are dependent on the program, while the number of task-switches can be controlled by the implementor by carefully designing the work-scheduler.

The schemes that have constant-time task creation and variable-access can be further subdivided into those that physically share the execution tree, such as Binding Arrays Scheme [W84, W87, LWH90] and Versions Vectors [HCH87] scheme, and those that do not, such as MUSE [AK90] and Delphi [CA88]. Both these kinds of schemes have their advantages. The advantage of non-sharing schemes such as Muse and Delphi are that less synchronisation is needed in general since each processor has its own copy of the tree and thus there is less parallel overhead [AK90]. This also means that they can be implemented on non-shared memory machines more efficiently. However, operations that may require synchronisation and voluntary suspension such as side effects, cuts and speculative scheduling are more overhead prone to implement. When an or-parallel system reaches a side effect which is in a non-leftmost or-branch, it has two choices: (i) it can suspend the current branch and switch to some other node where there is work available, the suspended branch would be woken up when it becomes leftmost; or (ii) it can busy-wait at the current branch until it becomes left most. In case (i) an or-parallel systems that does not share the execution tree, such as Muse, will have to save its current execution stack in a scratch memory-area since switching to a new node means that the current stack would be overwritten due to copying of the branches corresponding to the new node. This can lead to large memory overheads, because, for example, during speculative scheduling a processor may voluntarily suspend as much as 10 to 100s of times for large sized programs [SI91]. Thus, the only alternative left for a non-sharing or-parallel system is to busy wait until the branch containing the side-effect becomes leftmost. However, this would lead to wasted processor cycles and hence will reduce speed-ups obtained.

The essential conclusion is that for some applications (those that require processors to synchronise often due to presence of a large number of side-effects and cuts) environment representation schemes which share the or-tree are better, and for some other applications (those that require processor to synchronise less often) schemes which maintain an independent or-tree per processor are better. With this observation in mind we have extended both types of environment representation schemes to accommodate independent and-parallelism with recomputation of goals. In a separate paper we report how Muse can be extended to incorporate independent and-parallelism [GH91]. In this paper we show how binding arrays can be extended for independent and-parallelism.

4.2 The Fragmentation Problem in Binding Arrays

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 (CVs)† that arise in that branch. The 2 main properties of the BA method for or-parallelism are the following:

(i) The offset of a conditional variable is fixed for its entire life.

(ii) The offsets of two consecutive conditional variables in an or-branch are also consecutive.

The implication of these two properties is that conditional variables get allocated space consecutively in the binding array of a given processor, resulting in optimum space usage in the BA. This is important because large number of conditional variables can exist at runtime‡.

In the presence of independent and-parallel goals, each of which have multiple solutions, maintaining contiguity in the BA can be a problem, especially if processors are allowed (via backtracking or or-parallelism) to search for these multiple solutions. Consider a goal with a CGE: `a, (true => b & c), d`. A part of its C-tree is shown in figure 5(i) (the figure also shows

† Conditional variables are variables that receive different bindings in different environments [GJ90].

‡ For instance, in Aurora [LWH90] about 1Mb of space is allocated for each BA.

the number of conditional variables that are created in different parts of the tree). If `b` and `c` are executed in independent and-parallel by two different processors P1 and P2, then assuming that both have private binding arrays of their own, all the conditional variables created in branch `b-b1` would be allocated space in BA of P1 and those created in branch of `c-c1` would be allocated space in BA of P2. Likewise conditional bindings created in `b` would be recorded in BA of P1 and those in `c` would be recorded in BA of P2. Before P1 or P2 can continue with `d` after finding solutions `b1` and `c1`, their binding arrays will have to be merged somehow. In AO-WAM [GJ89] the approach taken was that one of P1 or P2 would execute `d` after updating its Binding Array with conditional bindings made in the other branch (known as the BA loading operation). The problem with BA loading operation is that it acts as a sequential bottleneck which can delay the execution of `d`, and reduce speedups. To get rid of the BA loading overhead we can have a common binding array for P1 and P2, so that once P1 and P2 finish execution of `b` and `c`, one of them immediately begins execution of `d` since all conditional bindings needed would already be there in the common BA. This is consistent with our discussion in section 3.1 about having teams of processors where all processors in a team would share a common binding array.

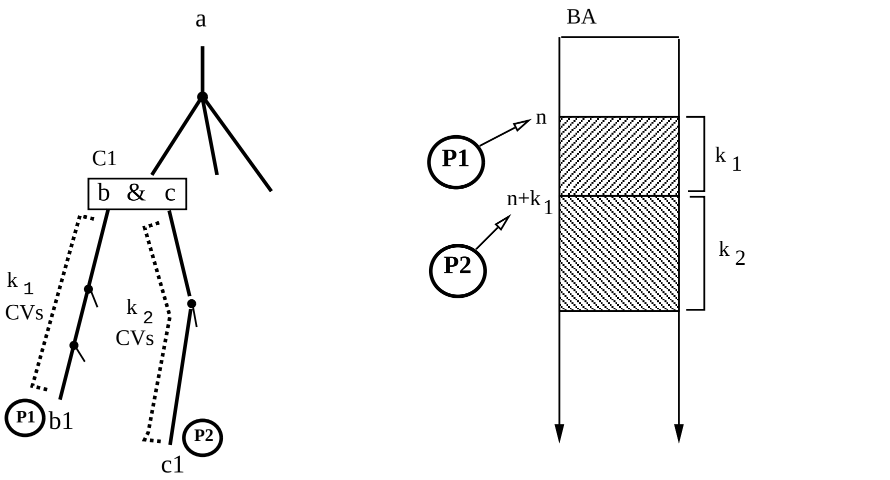

Fig (i): Part of a C-tree

Figure (ii): Optimal Space Allocation in the BA

Figure 5: Binding Arrays and Independent And-Parallelism

However, if processors in a team share a binding array, then backtracking can cause inefficient usage of space, because it can create large unused holes in the BA. Referring again to the C-tree shown in figure 5(i), when processors P1 and P2 start executing goals `b` and `c`, then to be able to efficiently utilise space in their common binding array they should be able to exactly tell the amount of space that should be allocated for conditional variables in branches `b-b1` and `c-c1`. Let us suppose we know the exact number of conditional variables that arise in branches `b-b1` and `c-c1` (say k_1 and k_2 respectively). If n is the value of the offset counter at the composition node C1, then P1 would set its offset counter to n and allocate space between locations n and $n + k_1 - 1$ in the BA for conditional variables in branch `b-b1`. P2 would set it to $n + k_1$ and allocate space between locations $n + k_1$ and $n + k_1 + k_2 - 1$ in the BA for conditional variables in branch `c-c1` (figure 5(ii)). However, if P1 fails and tries an untried alternative in one of the choice-points created along branch `b-b1`, and if this new alternative leads to creation of more number of conditional variables than k_1, then there is going to be a problem since P1 will have to get a new value of the offset-counter which does not interfere with that of P2 (i.e. it does not lead to overwriting of BA locations between $n + k_1$ and $n + k_1 + k_2 - 1$. Maintaining the offset counter such that it does not interfere with BA locations being used by P2 can be very complex. Also once P1 has completely searched or backtracked over goal `b` it will be difficult to reuse the free space in the BA created by backtracking of P1 since if P2 has not completely backtracked over `c` yet, then the space after location $n + k_1$ would still be in use, trapping the space freed by P1. Even if we manage to reuse the space freed, it will lead to BA offsets of consecutive CVs in a branch of the tree becoming non-monotonic. This has the determental effect that the offsets, traditionally used as indicators of "seniority" of CVs [LWH90], cannot be used any longer for determining their "age".

To simplify the book-keeping of the offset-counter, one can determine the maximum number, m, of conditional variables created among all branches of the goal `b` and then set P1's counter to n and P2's counter to $n + m$. In this case space would be definitely wasted, since all the m allocated locations would be used by only one of the branches of `b`. The others branches of `b` may have far fewer conditional variables than m.

Unfortunately, knowing the exact number of conditional variables that are going to arise in a branch (k_1, k_2, m, etc.) is impossible since their number is only determined at runtime; the best one can do is to estimate them. Underestimating their number can lead to lot of small holes in the BA which will be used only once (figure 6(i)), while overestimating them can lead to allocating space in the BA which will never be used (figure 6(ii)). Thus, in either case space is wasted and a very large binding array may be needed to exploit independent and-parallelism with or-parallelism.

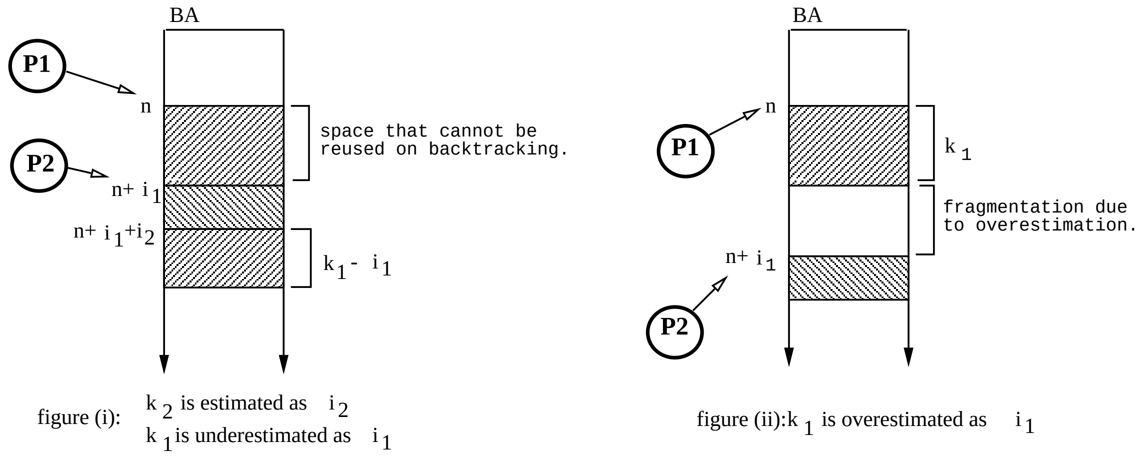

Figure 6: Problems with BA in presence of And-parallelism

4.3. Paged Binding Array

To solve the above problem we divide the binding array into *fixed sized segments*. Each conditional variable is bound to a pair consisting of a segment number and an offset within the segment. An auxiliary array keeps track of the mapping between the segment number and its starting location in the binding array. Dereferencing CVs now involves double indirection: given a conditional variable bound to $\langle i, o \rangle$, the starting address of its segment in the BA is first found from location i of the auxiliary array, and then the value at offset o from that address is accessed. A set of CVs that have been allocated space in the same logical segment (i.e. CVs which have common i) can reside in any physical page in the BA, as long as the starting address of that physical page is recorded in the ith slot in the auxiliary array. Note the similarity of this scheme to memory management using paging in Operating Systems, hence the name Paged Binding Array (PBA)†. Thus a segment is identical to a page and the auxiliary array is essentially the same as a page table. The auxiliary and the binding array are common to all the processors in a team. From now on we will refer to the BA as Paged Binding Array (PBA), the auxiliary array as the Page Table (PT), and our model of and-or parallel execution as the PBA model‡.

Every time execution of an and-parallel goal in a CGE is started by a processor, or the current page in the PBA being used by that processor for allocating CVs becomes full, a *page-marker node* containing a unique integer id i is pushed onto the trail-stack. The unique integer id is obtained from a shared counter (called `pt_counter`); there is one such counter per team. A new page is requested from the PBA, and the starting address of the new page is recorded in the ith location of the Page Table. i is referred to as the page number of the new page. Each processor in a team maintains an offset-counter, which is used to assign offsets to CVs within a page. When a new page is obtained by a processor, the offset-counter is reset. Conditional variables are bound to the pair $<i, o>$, where i is the page number, and o is the value of the offset-counter, which indicates the offset at which the value of the CV would be recorded in the page. Every time a conditional variable is bound to such a pair, the offset counter o is incremented. If the value of o becomes greater than K, the fixed page size, a new page is requested and new page-marker node is pushed.

A list of free pages in the PBA is maintained separately (as a linked list). When a new page is requested, the page at the head of the list is returned. When a page is freed by a processor, it is inserted in the free-list. The free-list is kept ordered so that pages higher up in the PBA occur before those that are lower down. This way it is always guaranteed that space at the top of the PBA would be used first, resulting in optimum space usage of space in the PBA. The algorithms for dereferencing and creation of a conditional variable are given below:

```
/*K is max. no. of slots in a page*/
create(V)
term *V {
      if (oc > K) ∨ (new and-parallel goal is begun) {
                oc = 0;
                p = head(free_page_list);
                i = pt_counter;     /*exclusive access to pt_counter is obtained.*/
                pt_counter++;
                PT[i] = p;       /*PT is the Page Table.*/
                push_page_marker(i) }
      oc = oc+1;
      V = <i, oc>}
```

† Thanks to David H. D. Warren for pointing out this similarity.

‡ A paged binding array has also been used in the ElipSys system of ECRC [VX91], but for entirely different reasons. In ElipSys, when a choice point is reached the BA is replicated for each new branch. To reduce the overhead of replication, the BA is paged. Pages of the BA are copied in the children branches on demand, by using "copy-on-write" strategy. In ElipSys, unlike our model, paging is not necessitated by independent and-parallelism.

```
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 vars bound to <i, o>*/
              val = PBA[o + PT[i]];        /*PBA is the paged binding array.*/
              if val→value == val
                       V
              else deref(val) }}
```

Note that empty slots which will never be used again can be created in the page table. The main point, however, is that the amount of space rendered unusable in the PT would be negligible compared to the space wasted in the BA if it was not paged. Note that the PBA would allow backtracking without any space in it being used only once or never being used (the problems mentioned in section 4.2). However, some internal fragmentation can occur in an individual page, because when a CGE is encountered, conditional variables in each of its and-parallel goal are allocated space in a new page, and so if part of the current page is unused, it will be wasted (it may, however, be used when backtracking takes place and another alternative is tried). Given that the granularity of goals in CGEs is preferred to be large, we expect such wastage as a percentage of total space used to be marginal.

Returning to the C-tree shown in fig. 5(i), when the c-node is reached, processor P1 will request a new page in the PBA and push a marker node in the trail-stack. The CVs created in branch `b-b1` would be allocated space in this new page. In the meantime P2 would request another page from the PBA where it will allocate CVs created in branch `c-c1`. P1 and P2 may request more pages, if the number of CVs created in respective branches is greater than K. If the branch `b-b1` fails, then as P1 backtracks, it frees the pages in the PBA that it allocated during execution. These pages are entered in the free-list of pages, and can now be used by other processors in the team, or by P1 itself when it tries another alternative of `b`.

Note that with the PBA, when a team picks an untried alternative from a choice point, it has to task-switch from the node where it is currently stationed to that choice point. In the process it has to install/deinstall the conditional bindings created along that path, so that the correct environment at that choice point is reflected (like in Aurora, a conditional binding made to a variable is trailed in the trail-stack, along with address of that variable). While installing conditional bindings during task-switching, if a team encounters a page-marker node in the trail-stack whose id is j, it requests a new page in the PBA from the free list, records the starting location of this page in jth location of the page table, and continues. Likewise, if it encounters a page-marker node in the trail-stack during backtracking whose id is k, it frees the page being pointed to by the kth location of the page table.

If the untried alternative that is selected is not in the scope of any CGE, then task-switching is more or less like in purely or-parallel system (such as Aurora), modulo allocation/deallocation of pages in the PBA. If, however, the untried alternative that is selected is in the and-parallel goal `g` of a CGE, then the team updates its PBA with all the conditional bindings created in the branches corresponding to goals which are to the left of `g`. Conditional bindings created in `g` above the choice point are also installed. Goals to the right of g are restarted and made available to other member processors in the team for and-parallel execution. Notice that if a C-tree is folded into an or-parallel tree according to the relationship shown in figure 2 and 3, then the behaviour of (and the number of conditional bindings installed/deinstalled during) task switching would closely follow that of a purely or-parallel system such as Aurora, if one follows the same scheduling order. Note that the technique of paged binding array is a generalisation of environment representation technique of AO-WAM [GJ89], hence some of the optimisations [GJ90a] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the PBA model. Lastly, seniority of conditional variables, which needs to be known so that "older" variables never point to "younger ones", can be easily determined with the help of the $<i, o>$ pair. Older variables will have a smaller value of i; and if i is the same, then a smaller value of o.

5. An Abstract Machine for the PBA Model

It is possible to design an abstract machine similar to WAM [W83] for compiled and-or parallel execution of Prolog programs. Prolog programs are compiled into an abstract instruction set, which are then executed to obtain fast execution. It turns out that the abstract instruction set for the PBA model is identical to the RAP-WAM [H86], except for the following differences:

- The dereferencing algorithm is different, since paged binding array is used for accessing conditional bindings.
- Multiple composition nodes (which are functionally similar to the parcall frame of the RAP-WAM) have to be supported.
- A choice point has to be locked when it is accessed by a team (for picking untried alternatives, for instance) since other teams may also try to access that choice point at the same time.
- An extra choice point has to be pushed on the stack to which the various c-nodes corresponding to a CGE can point to.
- Marker nodes have to be pushed on the stack whenever the value of the offset-counter becomes equal to page-size, K.
- Share-nodes have to be pushed when an untried alternative is picked by another processor from a choice point nested in a CGE.
- A new data structure for recording pointers to choice points with untried alternatives is also added to the data-areas of the RAP-WAM. It is used by a team for picking work from another team. There is one such data-structure per team.
- A new marker node called the end-marker is also introduced, which is added to the stack whenever the end of an and-parallel goal in a CGE is reached. It marks the tip of the and-branch of a CGE. It records the trail top register and is used by teams during task-switching. A pointer to the end-marker of a goal in a CGE is also recorded in the slot corresponding to that and-parallel goal in the c-node .

Like the RAP-WAM, each processor has its own local stack, heap, and trail stack, which is accessible to all processors in its team as well as to processors of other teams. Thus, the abstract machine for the PBA model is targeted towards shared memory multiprocessor machines.

We next outline the forward and backward execution algorithms of this abstract machine, along with memory management and work scheduling issues.

5.1. Forward Execution

Forward execution is identical to the RAP-WAM, except that when a choice-point is created, a pointer to it is placed in the choice point queue (techniques that do not involve maintaining a queue of work, such as those used in Aurora [S89, BM91, S91] can also be used). This pointer will be removed from the queue when all the untried alternatives of the choice point have been tried. When a CGE is encountered, first a choice point, and then a c-node is pushed on the stack. The fields in a c-node are very similar to those in the `parcall` frame of RAP-WAM [H86]. For each goal in the CGE, the c-node has an entry to record the processor-id of the processor that is working on it; a field to record the completion status (set by the corresponding processor when it finishes the goal, marking whether the goal has other alternatives or not); a `ready-field`, to help select goals to restart on backtracking from outside the CGE; and a field to record a pointer to the end-marker for that goal. The c-node also has a `put_instruction_pointer` field whose content and usage is identical to that in the RAP-WAM. The processor that creates the composition node places all but one of the goals in the and-parallel goal stack, in the manner of RAP-WAM. The one which is not placed is processed locally. Goals from the goal-stack are picked up by member processors of the same team. The goal-stack of each processor is visible only to processors in its team and not to processors in other teams. The choice point queue of each team is however visible to all other teams.

When a team is idle, it scans choice point queues of other processors. If a team finds a choice point with untried alternative, then if that choice point is not in the scope of any c-node, execution takes place in a way very similar to that in Aurora [LWH90]. That is, the team task-switches to the node with the untried alternative, updating its PBA in the process by removing conditional bindings that are no longer applicable and installing conditional bindings that are needed. However, if the choice point is in the scope of a c-node, the sequence of action is different. Let us suppose `(true =>` $\mathtt{g}_1$ `&` $\mathtt{g}_2$ `...` `&` $\mathtt{g}_n$`)` is the CGE associated with the c-node, and $\mathtt{g}_i$ the goal that contains the choice point with the untried alternative. The following actions are taken: The team task-switches from its current site to the choice point where work is available; in the process its paged binding array and page table are updated. The composition node is copied into the stack of one of the processors of the stealing team. For each goal $\mathtt{g}_1 \ldots \mathtt{g}_{i-1}$, a share node is inserted. Each share node contains a pointer to the copied composition node as well as to the end-marker node of the corresponding goal. A share node is also pushed for g_i, which contains a pointer to the copied composition node and to the choice point from where the untried alternative was picked. The ready field is adjusted so that on branching to the instruction pointed to by put-instruction pointer [H86], the execution of the goals to the right of g_i is restarted. These goals get placed in the goal stack, and are picked up for and-parallel execution by other team members (via `push_call, check_ready`, etc. instructions of the RAP-WAM [H86]). Execution in this team continues like normal RAP-WAM execution.

For the sake of simplicity we impose the restriction that if the untried alternative lies in the scope of c-node, then it is allowed to be picked up by another team for or-parallel processing only if all the sibling and-parallel goals to the left of the goal containing this untried alternative have produced at least one solution. This also means that if the c-node containing the goal is nested inside another c-node, then all and-parallel goals of the outer c-node, which are to the left of the inner c-node, must also have produced at least one solution. Thus, given the CGE `(true =>` g_1 `&` g_2 ... `&` g_n`)` an untried alternative can be picked from choice point in g_i only if goals g_1 ... g_{i-1} have found at least one solution each. This restriction also helps ensure that we do not give work to other teams which is of very small granularity or is speculative.

5.2. Backward Execution

When failure occurs backward execution phase is entered. If the failing processor is in a branch which is not in the scope of any composition-node, the sequence of actions performed is similar to normal sequential backtracking (except that if a node is still in use by another team, it is not reclaimed). The processor moves up the tree looking for choice points with untried alternatives. Like the RAP-WAM [H86] backtracking on a CGE can take place in two modes: (a) the "inside" mode, if the failure occurs in a branch in the subtree of an and-parallel goal of the CGE; (b) the "outside mode" when failure occurs in a goal to the right of the CGE. If the failing processor is in the scope of a composition node, and the mode is "inside" [H86], it backtracks on its and-branch. If it can find a choice point in this and-branch which has an untried alternative, it selects it and starts its execution. Its team mates, who are possibly working on sibling and-branches, keep working on them without any interruption. If such a choice point is not found, and a share-node is reached then it follows the pointer to the choice-point in that share node and keeps backtracking, looking for a choice point with untried alternative. If it can find one, it updates the share node over which it just backtracked to point to this new choice-point and starts executing the alternative (note that a share-node that points to the end-marker of another goal cannot be reached during backtracking in the "inside" mode). Like in the previous case its teammates are not disturbed and they continue executing their respective goals. If, however, its composition node is reached during backtracking, then this means that the composition-node is not going to yield any solution, so the processor sends a kill message to all its team-mate processors which are working on sibling branches. The failed team is then free to look for untried alternatives in any available choice-point.

If the mode during backtracking is "outside" then a redo message is sent to the member processor which executed the rightmost goal in the CGE. If goals to the right cannot produce any new solutions, backtracking proceeds to goals to the left (just as in the RAP-WAM). When a choice point with untried alternatives is found in a non-rightmost goal then, all the goals to its right have to be restarted. This is done by changing the `ready/not-ready` field of the corresponding goals in the composition node to `ready`, setting the program counter to the contents of the `put_instruction_pointer` field in the composition node, and resuming the forward execution phase.

The abstract machine for the PBA model is very similar to the RAP-WAM. In fact, not even a single new instruction needs to be added to the RAP-WAM; hence, Prolog programs can still be compiled to the RAP-WAM instruction set. The only necessary changes are in the dereferencing, forward execution, and the backward execution algorithms of the RAP-WAM.

5.3. Memory Management

Memory management in the PBA model is very similar to that of RAP-WAM and Aurora.

A node can be reclaimed when no processor is using it. One can use bit-vectors for recording which processor is using which nodes in the tree, following Aurora's Manchester scheduler [S89] (strategies used by other schedulers of Aurora [BM91, S91] can also be adapted). There is one bit-vector per node. When team number i is using a node, then the ith bit of that node's bit vector is set to 1 (this is done either while creating the node or while going through that node to pick work below). When that node is not longer in use, the ith bit is reset to 0 (this is done when the node is passed through during backtracking). Note that there is a bit for each team, rather than for each processor. The last team exiting verifies that all bits are reset to 0, and can mark the node as a *ghost node* to be reclaimed later.

5.4. Work Scheduling

And-parallel work is shared within a team, thus a processor can steal and-parallel work only from members of its own team. Or-parallel work is shared at the level of teams, thus only an idle team can steal an untried alternative from a choice point. An idle processor first looks for and-parallel work in its own team. If no and-parallel work is found, it can decide to migrate to another team where there is work, provided it is not the last remaining processor in that team. If no such team exists it can start a new team of its own, perhaps with idle processors of other teams, and the new team can steal or-parallel work. One has to carefully balance the number of teams and the number of processors in each team, to fully exploit all the and- and or-parallelism available in a given Prolog program. We plan to use some of the 'flexible scheduling' techniques that have been developed for the Andorra-I system [D91] for this purpose.

6. Conclusions and Comparison with Other Work

In this paper we presented a model for combining or-parallelism and independent and-parallelism which can support full Prolog. The model uses an extension of the binding array called *paged* binding array for environment representation. The model is a direct combination of the &-Prolog [HG90] system and the Aurora system [LWH90], both of which have been very successful in exploiting and-parallelism and or-parallelism respectively. If an input program has only or-parallelism, then the PBA model will behave *exactly* like Aurora (except that another level of indirection would be introduced in binding array references). If a program has only independent and-parallelism the PBA model will behave *exactly* like &-Prolog (except that conditional bindings would be allocated in the binding array). Our model can also support the extra-logical features of Prolog (such as cuts and side-effects) transparently [GS91], which other and-or models [BK88, GJ89, RK89] cannot. In this regard the PBA model is superior to these other models. Control in the PBA model is quite simple, due to recomputation of independent goals. Memory management is also much simpler. We firmly believe that the PBA model can be implemented very efficiently, and indeed an implementation of the PBA model is to begin shortly. The implementation techniques of the PBA model can be used for even those models that have dependent and-parallelism, such as Prometheus [SK91], and IDIOM (with recomputation) [GSY91]. It can also be extended to implement the Extended Andorra Model [W90].

Acknowledgements

Thanks to Vítor Santos Costa for his numerous comments on this paper, and to Raéd Sindaha and Tony Beaumont for answering many questions about Aurora and its schedulers. Thanks are also due to David H.D. Warren for his support. This research was supported by U.K. Science and Engineering Research Council grant GR/F 27420.

References

[AK90] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, *to appear*

[AK91] K. Ali, R. Karlsson, "Full Prolog and Scheduling Or-parallelism in Muse," To appear in *International Journal of Parallel Programming*, 1991.

[BK88] Uri Baron, et. al., "The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results," In *Proceedings of FGCS '88*, Tokyo, pp. 841-850.

[BM91] 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.

[CA88] W. F. Clocksin and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors," In *New Generation Computing, 5(1988),* 361-376.

[CC89] S-E. Chang and Y.P. Chiang, "Restrict And-Parallelism Model with Side Effects," Proceedings of North American Conference on Logic Programming, 1989, MIT Press, pp. 350-368.

[D91] I. Dutra, "Flexible Scheduling in the Andorra-I System," In *Proc. ICLP'91 Workshop on Parallel Logic Prog.*, Springer Verlag, *to appear.*

[GH91] G. Gupta and M. Hermenegildo, "ACE: And/Or-parallel Copying-based Execution of Logic Programs," Technical Report TR-91-25, Department of Computer Science, University of Bristol, Oct. 1991.

[GJ89] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 332-349.

[GJ90] G. Gupta and B. Jayaraman, "On Criteria for Or-Parallel Execution Models of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 604-623.

[GJ90a] 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.

[GS91] G. Gupta, V. Santos-Costa, "Cut and Side Effects in And-Or Parallel Prolog," Technical Report TR-91-26, Department of Computer Science, University of Bristol, Oct. 1991.

[GSY91] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, "IDIOM: A Model for Integrating Dependent-and, Independent-and and Or-parallelism," In *Proceedings of ILPS*, MIT Press, Oct. 1991.

[H86] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd International Conference on Logic Programming*, London, 1986.

[HG90] 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.

[HN86] M. V. Hermenegildo and R. I. Nasr, "Efficient Implementation of backtracking in AND-parallelism", *3rd International Conference on Logic Programming*, London, 1986.

[HCH87] B. Hausman, et. al., "Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, CA.

[HCC88] B. Hausman, A. Ciepielewski, and A. Calderwood, "Cut and Side-Effects in Or-Parallel Prolog," In *International Conference on Fifth Generation Computer Systems,* Tokyo, Nov. 88, pp. 831-840.

[K85] L. V. Kale, "Parallel Architectures for Problem Solving", Ph.D. Thesis, Dept. of Computer Science, SUNY-Stony Brook, 1985.

[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.

[LWH90] 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.

[MH89] K. Muthukumar and M. Hermenegildo, "Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted And-Parallelism," In *Proc. of ICLP*, 1989.

[MH89a] K. Muthukumar, M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation," In *Proc. of NACLP '89*, MIT Press.

[RS87] M. Ratcliffe, J-C Syre, "A Parallel Logic Programming Language for PEPSys" In *Proceedings of IJCAI '87*, Milan, pp. 48-55.

[RK89] B. Ramkumar and L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of NACLP '89*, MIT Press, pp. 313-331.

[S91] R. Sindaha, "The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture," Technical Report, Department of Computer Science, University of Bristol, forthcoming.

[S89] P. Szeredi, "Performance Analysis of the Aurora Or-parallel Prolog System," In *Proc. of NACLP*, MIT Press, 1989, pp. 713-732.

[SH91] K. Shen and M. Hermenegildo, "A Simulation Study of Or- Independent And-Parallelism," In *Proc. of ILPS* MIT Press, 1991.

[SI91] R. Sindaha, Personal Communication, Sep. 1991.

[SK91] K. Shen, "Studies of And-Or Parallelism in Prolog," Ph.D. thesis, Cambridge University, 1991, forthcoming.

[SWY91] 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.

[VX91] A. Véron, J. Xu, et. al., "Virtual Memory Support for Parallel Logic Programming Systems," In *PARLE'91*, Springer Verlag, LNCS 506, 1991.

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

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

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco.

[W90] D.H.D. Warren, "Extended Andorra Model with Implicit Control" Talk given at Workshop on Parallel Logic Programming, 7th ICLP, Eilat.