Shared Paged Binding Array: A Universal Datastructure for Parallel Logic Programming

Gopal Gupta
Enrico Pontelli

Lab for Logic and Databases
Dept of Computer Science
New Mexico State University
Las Cruces NM USA
gupta,epontell@nmsu.edu

Vítor Santos Costa

Advanced Computing Research Center
Dept of Computer Science
University of Bristol
Bristol BS8 1TR UK
costa@compsci.bristol.ac.uk

Abstract

Two major problems that arise in parallel logic programming systems are: (i) redundant computation during and-parallel execution of dependent goals, and, (ii) efficient representation of multiple environments at runtime. Both these problems are caused by non-determinism present in logic programs—responsible for much of the power of logic programming. This paper is mainly concerned with solving the second problem, namely, the efficient representation of multiple environments at runtime in parallel logic programming systems. We present a datastructure called the *Shared Paged Binding Array* that arguably is most suited for implementing any arbitrary parallel logic programming system (i.e., a system that exploits any arbitrary combination of dependent-and parallelism, independent and-parallelism and or-parallelism). This datastructure can also be used for and-or parallel execution of Committed Choice Languages with Deep Guards as well as for realizing the implementations of more advanced models of parallel execution such as the Extended Andorra Model and the Andorra Kernel Language. Details of the Shared Paged Binding Array are presented, its merits over other techniques are shown, and, finally, implementations that combine different flavors of and- and or-parallelism using Shared Paged Binding Arrays are briefly sketched.

1 Introduction

Logic Programming is a declarative programming paradigm that is becoming increasingly popular. One of the distinguishing features of logic programming languages is that they allow considerable freedom in the way programs are executed. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution. Indeed, three main types of parallelism have been identified and successfully exploited in logic programs:

(i). *Or-parallelism:* (abbreviated OP) arises when more than a single rule define some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solution(s) to the top-level query.

(ii). *Independent And-parallelism:* (abbreviated IAP) arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another, that is, their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables [22].

(iii). *Dependent And-parallelism:* (abbreviated DAP) arises when mutually dependent goals that share a common variable are executed in parallel to co-operatively produce a binding for this variable. Dependent and-parallelism is readily found in applications that involve producer-consumer and reactive interactions [27].

Parallel Execution of Logic Programs has been an active area of research since a decade [24, 5, 7, 26]. However, parallel logic programming systems that can *efficiently exploit all* forms of parallelism present in Logic Programs—or-parallelism, independent and-parallelism, and dependent and-parallelism—still elude us. The main source of difficulty has been the non-determinism—responsible for much of the power of logic programming—present in logic programs. Non-determinism affects parallel execution in two ways: (i) Unrestricted parallel execution of non-determinate goals that have data-dependencies between them leads to redundant computations being performed; (ii) Due to non-determinism, variables in goals may be imparted multiple bindings during or-parallel execution, since each matching clause may bind the variable differently; efficient maintenance and representation of these multiple environments is a non-trivial problem which is compounded further in the presence of independent and dependent and-parallelism. We believe that problem (i) demands appropriate control mechanisms in logic programming languages, while problem (ii) can be solved by designing smart datastructures which allow for efficient representation and maintenance of multiple environments. This paper is concerned with solving problem (ii). We present a datastructure called *Shared Paged Binding Array*, and show how it can be (universally) used for implementing any *arbitrary* parallel logic programming system, i.e., a system that exploits any arbitrary combination of OP, (determinate or non-determinate) DAP and (determinate or non-determinate) IAP. In particular, we will show how the Shared Paged Binding Array datastructure can be used for supporting efficient implementation of systems combining:

1. OP and IAP (with goal recomputation)

2. OP and IAP (with goal reuse)

3. OP and determinate DAP

4. OP, determinate DAP, and IAP

5. OP, non-determinate DAP and IAP

6. Committed Choice Languages with deep guards

It can also be used for implementing parallel Constraint Logic Programming Systems.

The rest of the paper is organized as follows: In Section 2 we further elaborate on problems related to control in dependent and-parallel execution. In Section 3 we discuss the problem of representing multiple environments that arise due to or-parallelism in parallel logic programming systems. We briefly discuss criteria, based on very pragmatic considerations, that any solution to this problem should satisfy. We mention the various earlier attempts and describe how the lack of a general and efficient environment representation technique has hindered progress in parallel logic programming. In Section 4 we describe the Binding Arrays technique that has been used for implementing purely or-parallel systems and show why it is ineffective when and-parallelism is present as well. In Section 5 we present an extension of Binding Arrays called Shared Paged Binding Arrays that satisfies all the criteria developed in earlier sections. We next show in Section 6 how different systems that combine different forms of parallelism (listed above) can be realized using the Shared Paged Binding Arrays. Finally, we show some other versatile properties of the Shared Paged Binding Arrays and present our conclusions.

2 Speculative Computation in Non-deterministic Dependent Goals

Unrestricted and-parallel execution of dependent goals with non-determinism can lead to large amount of speculative computation. Consider the goal `p(X), q(X)`, in which the non-deterministic goals `p` and `q` have a data-dependency due to shared variable `X`[1]. Suppose they are defined by the following clauses:

```
p(X) :- g1, X = 1, s1.
p(X) :- g2, X = 2, s2
p(X) :- g3, X = 3, s3.

q(3) :- h1.
q(4) :- h2.
q(5) :- h3.
```

where $\mathtt{g}_i$ and $\mathtt{h}_i$ involve a fair amount of computation. Clearly, there is only one solution for `X`, namely, 3. In a sequential execution, only $\mathtt{h}_1$ will ever get executed, and that too only once when `X` in `q(X)` is bound to 3 by `p(X)`. However, in unrestricted and-parallel execution of `p(X)` and `q(X)`, `p` will produce three bindings for `X` (`X = 1`, `X = 2`, and `X = 3`). So will `q` (`X = 3`, `X = 4`, and `X = 5`) and in the process it will also execute the goals $\mathtt{h}_1$, $\mathtt{h}_2$, and $\mathtt{h}_3$. When the bindings produced by `p` and `q` are compared, only `X = 3` will produce an answer, the rest will be thrown away. Thus compared to sequential execution, dependent and-parallel execution performed redundant execution of $\mathtt{h}_2$ and $\mathtt{h}_3$.

However, if we are too cautious, and execute `q` only after a binding for `X` has been produced by `p` then we may lose parallelism, because for a different set of instantiations all bindings for `X` in `q` may be consistent with those in `p`. Therefore, there is the problem of striking a balance between the amount of speculative computing and the amount of parallelism exploited since the speculativeness or usefulness of a computation depends on the nature of instantiation of variables in the goal (note that the problem of determining variable instantiations in advance is undecidable).

In general, we believe that to extract maximum parallelism and obtain the best possible execution, programs will need suitable (but simple!) explicit control constructs (such as the soft and hard sequencing

[1] Following [28] we term `X` the dependent variable.

operators [20], wait guards [21], etc.), as well as implicit control constructs such as suspension on variables [33, 21, 20, 9, 6, 29, 32]. In this paper we do not pursue this problem further since it is not our main concern.

3 Environment Representation in presence of Non-determinism

Given a non-deterministic goal, its or-parallel execution will entail executing the bodies of more than one matching clauses in parallel. If an unbound variable exists in the goal, then each one of the matching clauses may impart a binding to this unbound variable distinct from others. In other words, during or-parallel execution multiple environments may exist at run-time, each corresponding to a branch in the SLD tree (search tree) of the goal. Efficient representation and maintenance of these multiple environments is a difficult problem [36, 18].

3.1 Problems in Environment Representation

In principle, or-parallelism should be easy to implement since various branches of the search tree are independent of each other, thus requiring little communication between processors. However, in practice, implementation of or-parallelism is difficult because of the sharing of nodes in the search tree.

Consider two nodes in two different branches of search tree. All nodes above (and including) the least common ancestor node of these two nodes are (logically) shared between the two branches. If a binding for a variable, created in one of the common ancestor nodes, is generated above (or at) the least common ancestor node then this binding is the same for both branches and hence should be shared (such a binding is known as an unconditional binding and such a variable as an unconditional variable). However, if a binding to a variable created in a shared ancestor node is generated by a node below the least common ancestor node then that binding should be visible *only to the branch to which the binding node belongs* (such a binding is known as a conditional binding and such a variable as a conditional variable). Bindings of such variables should not be shared. It is this peculiar nature of or-parallel execution, that part of the environment is to be (logically) shared between different branches while part of it is to be kept (logically) distinct, that makes efficient representation of multiple environments so difficult.

Consider the goal `p(X)` and the clauses for `p` defined in section 2. The three different clauses (each giving rise to a different branch in the search tree) each impart different bindings to the conditional variable `X`. Unlike in sequential execution a binding for `X` can't be recorded in the activation record (environment frame) created for the goal since there are many such bindings. Therefore, special arrangements have to be made. If `X` is accessed in the goal s_i then the environment representation scheme has to make sure that the appropriate binding is accessed for `X`. Thus, the main problem in implementing or-parallelism is the efficient representation of multiple environments that co-exist simultaneously in the search tree corresponding to a program. Note that the main problem in management of multiple environments is that of efficiently representing and accessing the conditional bindings; the unconditional bindings can be treated as in normal sequential execution of logic programs (i.e., they can be stored in-place in the activation frames).

3.2 Criteria for Environment Representation in Or-parallel Execution

Essentially, the problem of multiple environment management has to be solved by devising a mechanism where each branch has some private area where it stores conditional bindings applicable to itself. There are many ways of doing this [36, 18]. For example:

- storing the conditional binding created by a branch in an array or a hash table private to that branch, from where the binding is accessed whenever it is needed.
- keeping a separate copy of the environment for each branch of the tree, so that every time branching occurs at a node the environment of the old branch is copied to each new branch.
- recording all the conditional bindings in a global datastructure and attaching a unique identifier with each binding which identifies the branch a binding belongs to.

Each approach has its associated cost. This cost is non-constant time and is incurred either at the time of variable access, or at the time of node creation, or at the time a processor begins execution of a new branch. In [18] three criteria were derived for an ideal or-parallel system, namely:

1. The cost of *environment creation* (also termed task creation) should be constant-time;

2. The cost of *variable access and binding* should be constant-time; and

3. The cost of *task switching*[2] should be constant-time.

[2]That is, the cost associated with updating the state of a processor when it switches from one node of the tree to another.

It has been shown that it is impossible to satisfy these three criteria simultaneously [18]. At least one of the three criteria has to be sacrificed. In other words, the non-constant time costs in managing multiple or-parallel environments cannot be avoided. However, one can argue that systems that choose to have non-constant time task-switching are better than those that choose to have non-constant time task-creation or variable access for the following reasons:

a. The number of tasks created and the number of variable-accesses depends on the program, whereas the number of task-switches done during execution depends on the work-scheduler. Clearly, the system implementor has no control over the number of tasks created or variables accessed. However, when, and how often, task-switching is done can be controlled by the system implementor (for example, by controlling task granularity) by building a smart scheduler.

b. Experience gained from the implementation of or-parallel systems such as Muse [2] and Aurora [10] has shown that the quality of the work scheduler has greatest influence on the performance of an or-parallel system. Scheduling involves selecting the most promising node in the search tree from where to pick work (taking into account *speculativeness* of nodes, task granularity considerations, etc.). The operation of picking the most promising node requires searching how much work is available at different nodes, and in general will not be a constant time operation. (this is irrespective of how the information about the work available at each node is kept, i.e., whether in the node itself or in a global datastructure separate from the search-tree). Thus, a scheduler that picks most promising work at any given time will incur non-constant time cost. Thus, an or-parallel system that has non-constant time task-creation or non-constant variable access will incur non-constant time cost during task-switching as well if it desires to have smart scheduling (note that task-switching involves invoking the scheduler). Note that in an or-parallel system with non-constant time task-switching, the cost of scheduling will be amortized in the task-switching overhead (for example, in Aurora, the updating of the binding array, the cause of non-constant time task switching, is done while looking for most promising work [31, 30, 3]).

The observations above are also borne by experimental results of Szeredi [31] for the Aurora system [10], a system with non-constant time task-switching. He found that the actual cost of updating the binding array is not a very significant part of the total cost incurred in task-switching, and that the most significant portion of the task-switching overhead was incurred in looking for node with most promising work. This overhead will be present in *any* or-parallel system that includes a good scheduler.

In the presence of and-parallelism the problem of environment representation is compounded further, because not only we have to worry about which bindings are shared and which are distinct but we also have to worry about synchronizing access to variables when two and-parallel goals attempt to bind the same variable, and making sure that the environment of one and-parallel goal is 'visible' to the other.

Based on the above, we conclude that the environment representation technique used for realizing or-parallel execution (i) Should have constant-time variable access and task-creation, and non-constant time task-switching (Binding Arrays [37, 38] and Stack Copying [2] are two such environment representation approaches[3]) and (ii) should be able to work even in the presence of IAP or DAP. In this paper we present such an environment representation scheme called the Shared Paged Binding Arrays.

3.3 Earlier Solutions to the Environment Representation Problem

Due to the difficult nature of solving the environment representation problem initial efforts for parallelizing logic programming languages either focussed on designing abstract models of parallel execution, ignoring the implementation issues, or did not take full advantage of the sequential implementation technology that has been developed for logic programming (such as the WAM [34]), or banished non-determinism altogether. An example of the last approach are the committed-choice languages [6, 32, 27, 26]. These languages did not banish non-determinism altogether, but retained it partially in the form of *deep guards*. However, when it came to parallel implementation, a further restriction was placed on the deep-guards— they could only be tried sequentially, or they could only be deterministic. Eventually, the deep-guards gave way to *flat* guards. The point we want to make is that lack of efficient environment representation techniques has been a severe hurdle in realizing fully general parallel logic programming systems in the past.

In the past few years, many researchers have proposed schemes for environment representation [8, 39, 2, 37, 38] (See [18] for a comprehensive comparative survey of these schemes) in or-parallel execution of logic programs. However, these scheme are:

[3]The discussion above also explains why Aurora, based on Binding Arrays, and Muse, based on Stack Copying, are the two most successful or-parallel systems. Performance of systems that use techniques with constant-time task switching, such as ROPM [25], ParaAKL [23], and, to an extent, PEPSys [39], is not up to the mark because of precisely the reasons just given.

a. either too inefficient because they do not take advantage of sequential implementation technology: such schemes are based on the *process model* of execution. In the process based approaches, prominent examples of which are Conery's And-Or Process Model [8] and the Reduce-Or Process Model [25], a process is created for every goal encountered during execution (in contrast, in multisequential approaches, multiple WAM like threads are created that are executed in parallel to produce answers to a query). These processes communicate bindings and control information to each other to finally produce a solution to the top-level query. Process based approaches have also been used for implementing committed choice languages [27].

To be competitive, the performance of a parallel system on a single processor should be comparable to sequential logic programming implementations (ideally there should not be any slow down compared to a sequential system, but most parallel systems will have some overheads). To achieve this goal, parallel systems should take full advantage of the well-developed sequential compilation technology [35]. Experience has shown that process based system do not achieve this. Moreover, presence of backtracking makes the process model too complex for non-deterministic parallel logic programming (although the approach is still good for committed choice languages where there is no backtracking involved). Finally, the process based approach exploits parallelism at a level that is too fine grained, resulting in high parallel overhead and unpromising absolute performances.

b. or, they choose to have non-constant time task creation or non-constant time variable access. Methods of this class were proved to be inferior to methods that have non-constant time task switching in the previous section.

c. or, they are targeted towards exploiting *only* or-parallelism and break down when and-parallelism is present.

With recent advances in design of abstract models for parallel execution of logic programs (such as the Andorra Model [33, 21, 20, 9] and others [28]) which exploit all three forms of parallelism, the need for an environment representation scheme that can be used for implementing these models is being felt more than ever. We believe that the *Shared Paged Binding Arrays* that we propose in this paper will fulfill these needs. The Shared Paged Binding Array owes much to the recent advances made in representing environment in purely or-parallel logic programming systems [8, 38, 2, 19, 11, 14, 4]. As we will shortly demonstrate, the Shared Paged Binding Array (SPBA for brevity) can be used for environment representation in any arbitrary parallel logic programming system (that is, any arbitrary combination of or-, independent and-, and dependent and-parallelism), as well as for implementation of Committed Choice Languages with Deep Guards.

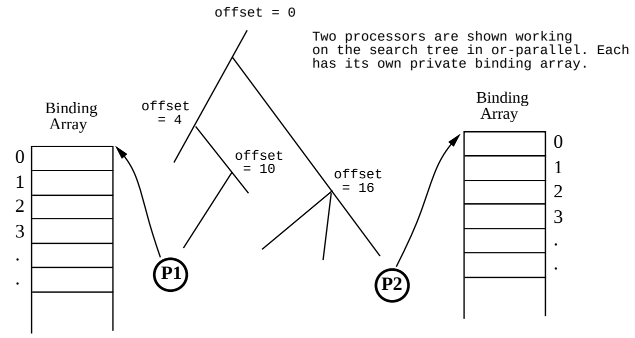


Figure 1: Binding Arrays

4 Simple Binding Arrays

Or-parallel execution can be viewed as parallel unfolding of a *search tree*[4]. Each branch node of the search tree is a *choicepoint*, and represents a set of alternative matching clauses for some goal. Each branch of the search tree represents the set of single-matching clause reductions between two choicepoints. Processing agents can therefore be seen as placed in a node of the search tree, which they unfold by expanding one of the outgoing branches, and possibly by creating new nodes.

As mentioned earlier, the principal problem when implementing or-parallelism is that of managing multiple bindings for variables. The *Binding Arrays* method [38, 37] has proven very successful in managing conditional bindings efficiently in purely or-parallel systems [10], and is outlined below.

Binding arrays are simple one dimensional arrays. One binding array is allocated for each processor of the multiprocessor system. Moreover, a counter is associated with each node in the search tree of the program. The value of the counter is initialised to 0 at the root node. Whenever branching takes place in the (or-parallel) search tree the parent node's counter is copied into the children nodes. Whenever a variable that might receive more than one binding is created by a processor, it is tagged as conditional, and the value of the counter, say i, is recorded in that variable. This recorded value is called the *variable's offset*. The counter for that branch is then atomically incremented.

[4]The search tree is also called the or-parallel tree. An example or-tree is shown in Figure 1.

The ith location in the binding array of the processor is allocated for holding the binding for the variable with offset i. Note that one step of indirection is therefore introduced in dereferencing a variable—when a conditional variable is to be dereferenced, its offset is obtained and used to index into the binding array to access the location where the binding is to be recorded/found (fig. 1). But like in sequential logic programming implementations, access to a variable's value remains a constant-time operation—the principal advantage of the BA method over other methods (see the dereferencing algorithm given below).

```
/*unbound variables are bound to themselves*/
deref(V)
term *V {
    if V→tag == VAR
            if notV→value == V
                  deref(V→value)
            else  V
    else    if V→tag == NON-VAR
                  V
            else {   /*conditional var bound to offset i*/
                  val = BA[i];  /*BA is the binding array.*/
                  if val→value == val
                  V
                  else deref(val) }}
```

However, with this advantage comes the requirement that the BA of a processor should reflect the correct environment of the node at which the processor is currently stationed. Therefore, when a processor switches from one node of the search tree to another, it has to update its BA so that it reflects the environment of its target node. For this purpose, conditional bindings are recorded in a *trail stack* along with the addresses of the corresponding variables. Processors update their BAs from the trail stack as they move up and down the or-parallel tree while searching for nodes with unexplored alternatives.

5 Environment Representation in And-Or Parallel Systems

In this section we present an extension of the simple BAs called the Shared Paged Binding Arrays, but before we do that: (i) we introduce models for abstractly representing and-or parallel execution; (ii) resolve the conflicting requirements imposed by and- and or-parallelism on processor organization; and, (iii) show why simple BAs fail to work in the presence of and-parallelism.

5.1 Abstract Representation of And-Or Parallel Execution

Before we delve into the implementation of and-or parallel logic programming systems, we first need to have an abstract representation for and-or parallel execution (like the or-parallel tree for or-parallel execution).

In and-or parallel systems there are two choices with respect to executing a conjunction of two non-deterministic (dependent or independent) goals `p & q`[5]: For every solution produced by `p` compute `q` in its entirety (also known as and-or parallelism with goal recomputation); or, compute solutions for `p` and `q` and then (incrementally) construct their join (also known as and-or parallelism with goal reuse). We have proposed the Extended And-Or Tree [19] for abstractly representing and-or parallel execution with goal reuse and the Composition-tree for representing and-or parallel computation with goal recomputation [15]. Examples of these are shown in figure 2.(i) and 2.(ii). These two abstract representations are very similar to the or-tree representation except that when and-parallel goals are encountered a *parallel conjunct node* (rectangular box in figure 2.(i)-(ii)) is created. The execution tree for each and-parallel goal is then rooted at this parallel conjunct node (Figure 2). The execution tree for the continuation of the and-parallel goals is appended to the parallel-conjunct node in the extended and-or tree (i.e., when goal reuse is adopted), and is rooted below the composition-subtree of one of the and-parallel goals in the case of composition-tree (i.e., when goal recomputation is adopted; in figure 2.(ii) it is put below the subtree of rightmost and-subgoal) [6]. These two representations (that use goal reuse and goal recomputation respectively) are sufficient for abstractly representing any and-or parallel execution where the and-parallelism may be either IAP or DAP.

5.2 And-Or Parallelism and Teams of Processors

In an and-or parallel logic programming system the following problem arises: For efficient execution, two processors that are executing in and-parallel should see all the bindings created by each other so that wasteful computation is not performed by one due to lack of knowledge of values of variables created by the other (as discussed in the example in section 2). This is in contrast 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 and-parallelism seem anti-thetical to each other. The solution that have been proposed (mostly for independent and- and or-parallel systems) range from

[5] Throughout this paper `p & q` indicates that the goals `p` and `q` may be executed in and-parallel.

[6] Note that one can have goal reuse or goal recomputation within the same computation—the execution tree can be abstractly represented as a combination of the extended and-or tree and the composition tree

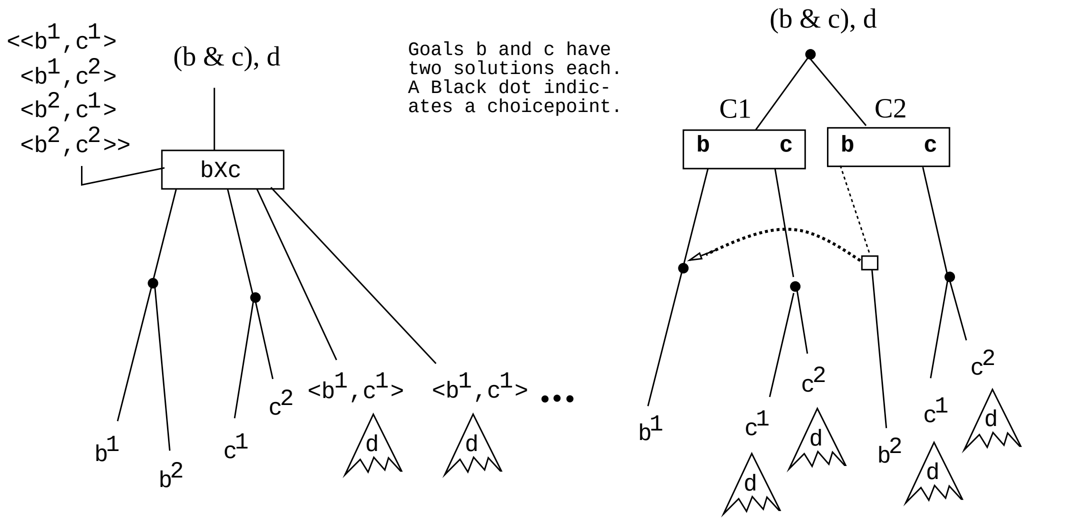


Fig (i): Extended And-Or Tree

Fig (ii): Composition Tree

The Extended And-Or Tree shown above is used for representing and-or parallel computation with goal reuse. That is, when a parallel conjunct is encountered, the rectangular node is created. Solutions are found for goals in the parallel conjunct, which are then cross-produced. The continuation of the parallel conjunct is executed for each element tuple in the cross-product set constructed.

The Composition tree shown above is used for representing and-or parallel computation with goal recomputation: for each solution for the goal to the left in the parallel conjunct, the goals to the right are re-executed. A parallel conjunct (the rectangular box in (ii)) represents a set of solutions. In the tree above a maximum of four such rectangular boxes could have been created, each corresponding to one solution for (b&c). The number of boxes created depends on the processing resources available. In the tree above, solutions of goal c will be found via backtracking assuming that not enough resources are present. The dashed arrow above indic--ates that C1 and C2 share branches up to the root of the tree

Figure 2: Abstract Representation of And-Or Parallel Execution

updating the environment at the time independent and-parallel computations are combined [25, 19] to having a complex dereferencing scheme [39]. All of these operations have their cost.

We contend that this cost can be eliminated by organising the processors of a multiprocessor system into *teams*, such that 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.

The concept of teams of processors has been successfully used in the Andorra-I system [9], which extends an or-parallel system to accommodate (determinate) dependent and-parallelism. It has also been used in independent and- and or-parallel systems [14, 11] We firmly believe that the organization of processors into teams is absolutely essential for design-simplicity and execution-efficiency of and-or parallel systems.

5.3 Simple Binding Arrays and And-Or Parallelism

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[7].

In the presence of 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 the goal: `a, (b & c), d`. The part of its tree corresponding to the parallel conjunction is shown in figure 3.(i) (the figure also shows 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. 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 [19] 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 5.2 about having teams of processors where all processors in a team would share a common binding array.

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 tree shown in figure 3.(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 parallel conjunct 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 3.(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 4.(i)), while overestimating them can lead to allocating space in the BA which will never be used (figure 4.(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.

5.4 Shared 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

[7] For instance, in Aurora [10] more than 1Mb of space gets allocated for each BA.

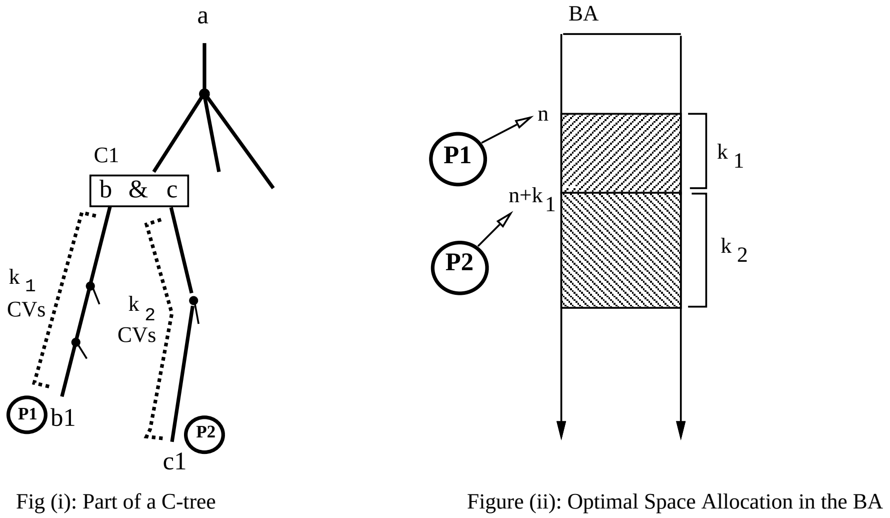
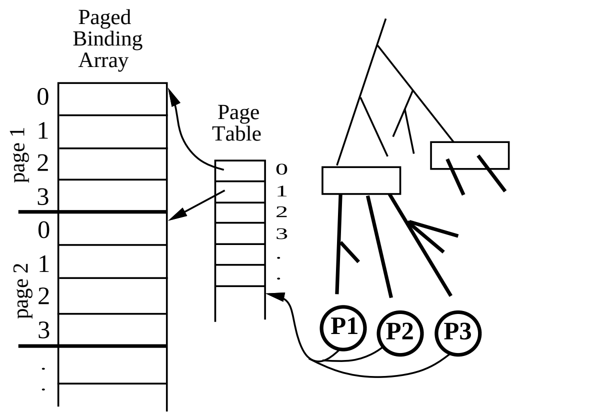


Figure 3: BAs and And-Parallelism

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)[8]. 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) and the auxiliary array as the Page Table (PT)[9]. Since a PBA is shared between all member processors of a team, we call it *Shared Paged Binding Array* or SPBA.

Every time execution of an and-parallel goal is started by a processor, or the current page in the SPBA 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 (Figure 5). 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 SPBA, 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, its 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 a new page-marker node is pushed.

A list of free pages in the SPBA 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-



An and-parallel computation (delimited by a rectangular box) is performed by a team of three processors which share a common paged binding array and page table. The branches that are part of and-parallel computation are shown in dark in the figure.

Figure 5: Paged Binding Arrays

[8] Thanks to David H. D. Warren for pointing out this similarity.

[9] A binding array has also been used in the ElipSys system of ECRC [1], 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.

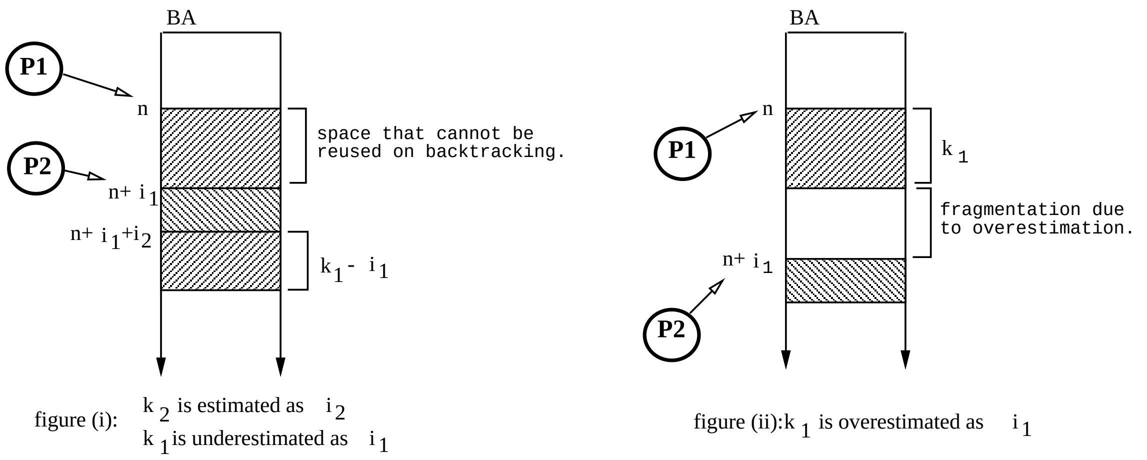


Figure 4: Problems with Simple BA

list. The free-list is kept ordered so that pages higher up in the SPBA occur before those that are lower down. This way it is always guaranteed that space at the top of the SPBA would be used first, resulting in optimum space usage of space in the SPBA. 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) }
   V = <i, oc>;
   oc = oc+1}


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

Note that the SPBA would allow backtracking without any space in it being used only once or never being used (the problems mentioned in section 5.3). However, some internal fragmentation can occur in an individual page, because when a set of conjunctive goals to be executed in and-parallel 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 and-parallelism is preferred to be large, we expect such wastage as a percentage of total space used to be marginal.

Returning to the tree shown in Figure 3.(i), when the parallel conjunct node is reached, processor P1 will request a new page in the SPBA and push a marker node in the trail-stack. The conditional variables created in branch `b-b1` would be allocated space in this new page. In the meantime P2 would request another page from the SPBA 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 SPBA 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 SPBA, 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 SPBA 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 and-parallel goal, then task-switching is more or less like in purely or-parallel system (such as Aurora), modulo allocation/deallocation of pages in the SPBA. If, however, the untried alternative that is selected is in the scope of an and-parallel goal `g`, then the team updates its SPBA with all the conditional bindings created in the branches corresponding to goals which are to the left of `g`.

Note that the technique of paged binding array is a generalisation of environment representation technique of AO-WAM [19], hence some of the optimisations [17] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the SPBA model. Also note that the SPBA is a generalization of the environment representation scheme used in the Andorra-I system [9] that exploits or-parallelism and deterministic dependent and-parallelism. 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.

6 And-Or Parallel Systems and SPBAs

We now describe how different systems that combine or-parallelism with different flavours of (dependent and independent) and-parallelism can be realized using the SPBA. For each kind of system only a rough outline is sketched—details are omitted due to lack of space. The SPBAs can also be used for implementing Parallel CLP Systems [12].

6.1 Combined IAP and OP Systems

Implementation of systems that combine Or-parallelism and Independent and-parallelism using the shared paged binding arrays is quite simple. It is customary to express independent and-parallelism through *Conditional Graph Expressions* [22], which determine if a conjunction of goals can be executed in independent and-parallel or not through simple run-time checks . Note that, as mentioned, and-parallelism is exploited by processors within teams, while or-parallelism is exploited between teams. Since the goals are independent, no special action needs to be taken after the execution of independent and-parallel goals is over.

6.1.1 Combined IAP and OP Systems with Goal Recomputation

In this case, a composition tree is constructed. There are three distinct cases that should be considered: (i) when a parallel conjunct is encountered by a team; (ii) when an alternative from within a goal in a CGE is picked by another team; and, (iii) an untried alternative in continuation of the parallel conjunct is picked. Regarding case (i), when the parallel conjunct is encountered a parallel conjunct node is created. For example, in the tree shown in fig. 2.(ii) a parallel conjunct node is created by the processor P1. 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` by P1 would be allocated space in this new page. In the meantime processor P2, teammate of P1, will pick up goal `c` for execution and 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 the size of the page. 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`. The continuation of the independent and-parallel conjunct will be begun by a team when a solution has been found for all goals in the parallel conjunct. Note that as soon as a solution is found for all goals in the parallel conjunct, execution of its continuation can begin since all the bindings needed are already present in the SPBA.

In case (ii), when a team picks an untried alterative from a choice point in an and-parallel goal, the team first task switches to that node, updating its SPBA and the Page Table along the way. The solution to the alternative picked is found, and then the subgoals to the right of this subgoal in the parallel conjunct are re-executed (goal recomputation). Execution from then on is similar to what is described in the previous paragraph.

In case (iii), again the team will have to task switch to the node in the continuation that has available work. In the course of task-switch, the parallel conjunct will be encountered and the SPBA and Page Table updated.

The implementation of combined or- and independent and-parallel systems (with or without recomputation of goals) is quite simplified with Shared Paged Binding Arrays. An implementation of an or- and independent and-parallel system with recomputation of goals based on SPBA has the nice property that in the presence of only or-parallelism the system will behave exactly like a purely or-parallel system based on simple binding arrays, such as Aurora [10] (modulo an extra level of indirection in accessing conditional bindings). In the presence of only and-parallelism it will behave exactly as a purely independent and-parallel system such as &-Prolog [22] (modulo the indirection in accessing conditional variables).

6.1.2 Combined IAP and OP Systems with Goal Reuse

In this case an extended and-or tree will be constructed. Here again, three cases can be distinguished. (i) when a parallel conjunct is encountered by a team; (ii) when an alternative from within a goal in a CGE is picked by another team; and, (iii) an untried alternative in continuation of the parallel conjunct is picked. Regarding Case (i), a parallel conjunct node will be created by a team when an independent and-parallel conjunct is encountered. Processors in the team pick up goals in the parallel conjunct for independent and-parallel execution. Before commencing the execution of the and-parallel goals, they each request a page for allocating space for conditional variables. When the execution of goals is finished, the team will continue with the execution of the continuation. No extra work is required since all the requisite pages are already in the SPBA. For example, consider figure 2.(i). A team begins execution by creating a parallel conjunct. Processors P1, P2, that are members of the team work on goals `b` and `c` respectively. When solutions $\mathtt{b}_1$ and $\mathtt{c}_1$ are found, the cross-product tuple $\langle \mathtt{b}_1, \mathtt{c}_1 \rangle$ is constructed and execution of the continuation of the conjunct for this tuple is begun.

Regarding Case (ii), the team will first have to task switch to the node containing the alternative, updating the SPBA and the Page Table in the process with conditional bindings created earlier. The team will begin execution of the alternative; other member processors will execute other alternatives in other and-parallel goals. For instance, in the example in Fig. 2.(i) another team may generate $\mathtt{b}_2$ and $\mathtt{c}_2$ while the team that created the conjunct is still generating $\mathtt{b}_1$ and $\mathtt{c}_1$. Once $\mathtt{b}_2$ and $\mathtt{c}_2$ are generated, the other team can generate all cross-product tuples in which $\mathtt{b}_2$ and $\mathtt{c}_2$ take part, and then it can continue execution of the continuation of the and-parallel conjunct for one of these generated tuples, exactly as the first team.

Case (iii) arises when a team simply picks up new tuples, whose components have been generated by other teams. In this case the team has to first taskswitch to the parallel conjunct node, followed by installing all conditional bindings created while finding the solutions that constitute the tuple.

Note that often the conditional bindings of different goals in the parallel conjunct can be themselves installed in parallel. Implementation of the conjunct node itself should also be much simplified, as the node now simply has only to maintain a list of tasks which have unexplored alternatives, so that a team can move there and exploit or-parallelism. The list of cross-product tuples generated is kept in the parallel conjunct node so that that a team can pick untried tuples. We believe that the SPBA should make it practical to implement and-or parallel systems that recompute goals, that share goals, or that can follow both strategies, depending on which is best for a particular procedure.

6.2 Combined DAP and OP Systems

We now describe how the Shared Paged Binding Array can be used for implementing systems with or-parallelism and dependent and-parallelism. Let us consider the goal `p(X) & q(X)`, where `p` and `q` are non-deterministic (i.e., have more than one matching clauses) and suppose `p` and `q` are executed in parallel. Most systems exploiting dependent and-parallelism will distinguish between goals `p` and `q`, designating `p` as the *producer* of the bindings of `X`, and `q` its *consumer*. Only producers can bind an unbound dependent variable. Consumer goals will suspend if they attempt to do so.

Like in independent and-or parallel systems, or-parallelism will be exploited by teams while and-parallelism will be exploited by processors within teams. Execution will begin by a team T1 reaching the goal in question. A processor in the team will begin executing `p` while another processor will pick up `q`. Within `p` and `q` they will pick one of the alternatives (since `p` and `q` are non-deterministic). The variable `X` would have been allocated space in the shared paged binding array of the team. Since the Paged Binding Array is shared, a processor accessing `X` (for binding it, or reading its value), will have to obtain exclusive access to it.

As a result of sharing the Binding Array, the moment a binding for `X` is produced by one of the processors and recorded in the SPBA it is immediately visible to the others in the same team. Thus, other team-members can tell the binding status of each dependent variable with ease.

Other teams may come along and execute other alternatives of `p` and `q` while T1 is busy producing the first solution. Here we consider a number of cases:

6.2.1 Combined Determinate DAP and OP Systems

An example of such a system is the Andorra-I system based on the Basic Andorra Model. Note that the Andorra-I system uses Binding Arrays and the notion of teams of processors in its implementation. In Andorra-I, processors in a team share a binding array, each processor in the team allocates a chunk of locations in the common Binding Array in which it allocates conditional variables. Note that since the SPBA is a generalization of the chunk scheme of Andorra-I, the SPBA can be trivially used instead for implementing Andorra-I where size of a chunk equals size of a page. (Note that offsets assigned to CVs in Andorra-I are single values, while if we use SPBA instead, CVs will have to be assigned pairs. The implementation of Andorra-I doesn't need pairs because dependent and-goals are always deterministic, while the SPBA is designed for non-deterministic goals as well.)

6.2.2 Combined DAP and OP Systems with Goal Recomputation

We consider here execution models such as Prometheus [28, 29] which suspend a consumer goal if it attempts to bind the dependent variable, and the consumer goal is executed for every solution of the producer goal. Execution of models such as Prometheus can be abstractly represented quite easily using the composition tree.

Execution takes place by a team first reaching the goal `p(X) & q(X)`. Two processors in the team pick the first alternatives from `p` and `q` respectively. The alternative for `q` will be executed so far as the processor executing `q` doesn't attempt to bind `X`. If it attempts to bind `X` it will suspend and will resume only after the alternative for `p` has produced a binding for `X`. Since the team has a common SPBA, checking whether `X` is unbound or bound is quite easy.

While a team is computing the first solution for `p(X) & q(X)`, other teams may pick alternatives from `p`. In Prometheus they will recompute `q` in its entirety, rather than make use of the computation done by the first team.

A team may also pick alternatives from `q`, but before doing so the team must update its SPBA from the trail in the corresponding solution for `p`. However, all dependent and-parallel goals to the right of `q` (if any) will be completely recomputed.

Prometheus [28] and similar dependent and- and or-parallel systems can be viewed as examples of dependent and-parallel system that extend independent and-parallel systems with recomputation. The SPBA allows easy implementation of the main extension, namely, the mechanism where consumer goals that attempt to bind an unbound dependent variable suspend.

6.2.3 Combined DAP and OP Systems with Goal Reuse

There are other execution models that exploit dependent and-parallelism and or-parallelism and that do not recompute goals (e.g., the Extended Andorra Model (EAM) [33, 20] and the Andorra Kernel Language [21]). In the EAM given the goal `p(X) & q(X)`, `q` is not recomputed for every solution for `p`, rather both `p(X)` and `q(X)` are executed independently. However, an alternative of `q` (the consumer goal) will suspend if it attempts to bind `X` before `p`. The execution of suspended consumer goals is restarted by making a copy of computation state of goal `q` for each binding produced by the producer `p`. This operation, known as the non-determinate promotion, is akin to taking a join of the solutions of the producer and the consumer goals (where the solutions of the consumer have not been found completely; the remaining computation in the consumer goal can be regarded as part of the continuation of the parallel conjunct so that the model becomes somewhat similar to the combined IAP and OP model with goal reuse). However, the consumer `q` is allowed to bind `X` unconditionally if the binding is *deterministic*.

Execution of such models can be abstractly represented using the extended and-or tree. Given such an execution model, execution using SPBA will proceed as follows: A team will pick up the and-parallel goals for execution. A processor in the team will pick the first alternative of `p` and another processor will pick the first alternative of `q`. Conditional bindings made by the two processors in the team will be recorded in the SPBA. If `q` produces a binding for `X` and `X` is unbound, it will suspend unless it is determinate (code for checking if a goal is determinate can be generated at compile-time, say as in [9]). If the processor executing `q` produces a binding that is inconsistent with the binding produced by its teammate processor executing `p`, then the one executing `q` will backtrack. In the process of backtracking it will only untrail bindings from the paged binding array that were produced as a result of execution of `q`. The stack frames, trail, etc., produced by `q` will remain intact (since they may be used by another team that has produced a consistent binding for `X` from `p`). The backtracking processor then either installs bindings from an alternative of `q` that has been executed by another processor and that has a consistent binding for its `X`, or it will try to execute another alternative for `q`. Other teams will behave in a manner similar to the first, their different processors executing `p` and `q` to obtain consistent bindings. Note that deterministic bindings will not be put in the Binding Array, rather they can be put in-place in the environment frames.

We have discussed so far the main ideas of these models. Implementation will be more complex when we consider mechanisms such as the *eager execution of producer goals* in the EAM, or local execution of guards in AKL. In both cases goals that work in and-parallel are not allowed to access everyone else's bindings, thus synchronisation between elements of a team is more complex. In this case, the SPBA serves as a very useful temporary datastructure where processors can store synchonisation information.

6.3 Combined IAP, OP and DAP Systems

As may be obvious by now, there are numerous similarities between systems implementing independent and- and or-parallelism and those implementing dependent and-parallelism and or-parallelism. Also, note that independent and-parallelism is a special case of dependent and-parallelism, therefore the implementation of a system using SPBA that exploits independent and-, dependent and- and or-parallelism does not need any extra machinery beyond what has already been described in the previous sections. We omit the description due to lack of space.

6.4 Committed Choice Languages with Deep Guards

Implementation of Committed choice languages with deep guards (such as Concurrent Prolog) in which or-parallel execution of such guards is permitted can also be implemented quite easily with SPBAs. Execution of a CCL with deep guards requires a model that is very similar to the ones described in sections 6.2.2 and 6.2.3. The only difference is when a commit operator is reached, in which case all choicepoints up to the corresponding parallel conjunct must be removed. Removal of choicepoints is accompanied by cleaning up of parts of the SPBA that correspond to the pruned alternatives.

Execution will commence as follows: a team picks up the top level goal for execution. A parallel conjunct is created and member processors attempt to solve the guards in the respective goals. A new page is used by each processor in the SPBA for executing the goal it selects. An attempt by a processor to bind a variable in the goal while executing the guard will result in suspension. The processor will record this suspension and will attempt to do some other useful work. When the variable on which execution is suspended gets bound, a member processor will resume the execution of the suspended goal.

Meanwhile another team can come along and pick an alternative from one of the deep guards. One of the member processor can start executing the untried alternative after updating the SPBA, while the other processors can pick untried alternatives from deep guards of sibling conjunctive goals. A failure in any of the processors in the team will simply cause that processor to backtrack and try another untried alternative. A success, however, will prompt that processor to prune all other alternatives in that deep guard. Once the alternatives are pruned, the body of the clause will be executed output bindings will be performed.

7 Versatility of the Shared Paged Binding Arrays

The Shared Paged Binding Array is intended for parallel logic programming systems targeted towards Shared Memory Multiprocessors. So far we had one SPBA for each team. We can go one step further and have one SPBA that is shared by *all processors of all the teams*. However, each team has its own individual page table, which indicates the pages that the team is currently using. The pages are allocated in the single Paged Binding Array. There are many advantages to having a single system-wide Paged Binding Array: (i) Space can be utilized much better. (ii) Processors in a team can park their page table at some place in the execution tree, break from the team, go and help other teams, and later reform the team and pick up execution from the point where the page table was parked (the memory space that will be blocked due to parking will be exactly equal to the number of pages in use by that page table); parking of binding arrays is very expensive when there are multiple binding arrays. (iii) The use of Binding Arrays in the style of ElipSys [1] is also possible: duplication of a binding array just involves duplicating the page table; pages get physically copied during a write (copy-on-write), thus ensuring that read-only pages are shared.

We can also distinguish between local conditional variables (those that are allocated space in the local stack in the WAM) and global conditional variables (those conditional variables that are allocated space in the heap in the WAM). To do this we will need a separate trail for global conditional variables and local conditional variables. We will also need a separate page tables for the two kind of conditional variables in each team. However, we can use the same SPBA. The advantage of making this distinction is that during task-switching, in certain cases we only need install bindings from the global trail [17]. As a result, task-switching overhead, the principal source of overhead in the SPBA scheme, is reduced.

The page size in the SPBA can be made equal to the page size of the cache of the multiprocessor system. We expect the page size of pages in the SPBA to be of the order of 64 or 128 words. This is also the range of page sizes of caches. As a result of this matching of sizes of pages, we hope that SPBAs will deliver better cache performance (in terms of successful cache hits).

The page size of the SPBA can also be used for controlling granularity of computation. The idea is that we can estimate the granularity for an and-task through the number of pages the tasks needs. If we are creating many tasks which only fill at most one or two pages before creating new and-tasks, we can assume thet these tasks are fine grained, and easily prevent the system from creating extra tasks. As an example, consider the Fibonacci program:

```
fib(0, 1).
fib(1, 1).
fib(X, Y) :- X > 1,
      ( X1 is X-1, fib(X1, Y1) &
        X2 is X-2, fib(X2, Y2) ),
      Y is Y1+Y2.
```

When executing this program in an SPBA based system, we would create new pages for each recursive call to `fib`, each page containing the variables `X1, Y1, X2`, and `Y2`. To implement granularity control, the system will notice it is creating many mostly empty pages, and thus stop generating new and-tasks for `fib` until it sees pages are being better used. Notice that we have two ways to control granularity. First, through the total number of pages we can check whether there are sufficient and-tasks and whether we need to create more. Second, through the number of pages per and-task we can notice whether we are only creating fine-grained tasks. The same information can be used to support a top-level scheduler, which dynamically reconfigures teams structures depending on the availability of parallelism.

A Criticism one could advance against SPBA based techniques is that they will not work well on Non-Uniform Memory Access architectures such as the BBN Butterfly. However, this can easily be rectified using *generalized stack copying* [13]. The idea is that whenever locality of reference due to our "share everything" approach causes performance degradation on a subset of processors, we can improve locality by forking over a choicepoint and making a copy of the entire computation for that set of processors in the style of MUSE. Thus the copied part and the original copy will execute in or-parallel with each other. Note that during copying all the structures, including SPBAs, will be copied (the only thing shared is the choicepoint on which forking was done). Since we also exploit and-parallelism, the copying done is in the style of ACE [16] rather than MUSE and is not explained here due to lack of space. The generalized copying technique is based on the observation that "stack copying" based and "binding array" based environment representation techniques, each of which have their own advantages, are two extremes of a spectrum of choices (share everything *vs* share very little) one can make for representing environment, and ideally one should combine the two to get benefits of both.

8 Conclusions

In this paper we presented a versatile data-structure called the Shared Paged Binding Array (SPBA) for representing multiple environments in parallel logic programming systems. The SPBA can be used for implementing any arbitrary parallel logic programming system, that is, for implementing a system that exploits any arbitrary combination of independent and-parallelism, or-parallelism, and dependent and-parallelism.

The technique is quite versatile in that if a system, that exploits all forms of parallelism and that has been implemented using SPBA, is used for executing a program that has only or-parallelism, then its performance will be comparable to a system that exploits only or-parallelism (such as Aurora [10]). Likewise, for executing programs that have only independent and-parallelism or only dependent and-parallelism. In the presence of only one processor, the performance of this parallel system will be comparable to a sequential logic programming system because the SPBA can take full advantage of the sequential WAM technology.

9 Acknowledgements

Thanks are due to Rong Yang, David Warren, Kish Shen of the University of Bristol and Manuel Hermenegildo of University of Madrid for many stimulating discussions. This research was initially supported by U.K. Science and Engineering Research Council grant GR/F 27420. Continued support is provided by NSF Grant CCR 92-11732 and HRD 93-53271, Grant AE-1680, AI-1929 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, and by NATO Grant CRG 921318. Vítor Santos Costa would like to thank the Gulbenkian Foundation and the European Commission for supporting his stay at Bristol.

References

[1] A. Véron, J. Xu, et al. Virtual memory support for parallel logic programming systems. In *Proceedings of PARLE'91*, Lecture Notes in Computer Science, pages 365–396. Springer Verlag, June 1991.

[2] K.A.M. Ali and R. Karlsson. The muse or-parallel prolog model and its performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, 1990.

[3] T. Beaumont, S. Muthu Raman, and et al. Flexible Scheduling or Or-parallelism in Aurora: The Bristol Scheduler. In *Parallel Architectures and Languages Europe '92, LNCS 506*, pages 403–420. Springer Verlag, 1991.

[4] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. Anlwam: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, Il 60439, 1986.

[5] A. Ciepielewski and S. Haridi. Control of Activities in the Or-Parallel Token Machine. In *International Symposium on Logic Programming*, pages 49–58, Silver Spring, Md, February 1984. Atlantic City, Ieee Computer Society.

[6] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *Journal of the Acm*, 8:1–49, January 1986.

[7] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[8] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *International Symposium on Logic Programming*, pages 457–467. San Francisco, Ieee Computer Society, August 1987.

[9] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP*, 1990.

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

[11] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92*, pages 617–632. Springer-Verlag, June 1992.

[12] G. Gupta and V. Santos Costa. Parallel Implementation of CLP Systems. Internal report, NMSU, 1992.

[13] G. Gupta, V. Santos Costa, and M. Hermenegildo. Gerneralized Stack Copying. In *Proc. Workshop on Parallel Execution JICSLP'92*, Nov 1992.

[14] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569*. Springer-Verlag, 1991.

[15] G. Gupta, M. Hermenegildo, and V. Santos Costa. And-or parallel prolog: A recomputation based approach. *New Generation Computing*, 11(3,4):297–322, 1993.

[16] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. Tech. report, NMSU, Nov 1993.

[17] G. Gupta and B. Jayaraman. Optimizing And-Or Parallel Implementations. In *1990 N. American Conf. on Logic Prog.*, pages 737–736. MIT Press, 1990.

[18] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[19] G. Gupta and B. Jayaraman. And-or parallelism on shared memory multiprocessors. *Journal of Logic Programming*, 17(1):59–89, 1993.

[20] G. Gupta and D.H.D. Warren. An Interpreter for the Extended Andorra Model. Internal report, University of Brsitol, 1991.

[21] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In *Proc. 7th Int'l Conf. on Logic Prog.* MIT Press, 1990.

[22] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

[23] R. Moolenaar and B. Demoen. A Parallel implementation for AKL. In *PLILP'93*, pages 246–261. LNCS 714, 1993.

[24] G. H. Pollard. *Parallel Execution of Horn Clause Programs.* PhD thesis, Imperial College, London, 1981. Dept. of Computing.

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

[26] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[27] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

[28] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-parallel Scheme. In *Proc. Joint Int'l Conf. and Symp. on Logic Prog.* MIT Press, 1992.

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

[30] R. Sindaha. The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture. In *Proc. of ILPS'93*, 1993.

[31] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[32] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, March 1986.

[33] D. H. D. Warren. The Extended Andorra Model with Implicit Control. ICLP'90 Parallel Logic Programming workshop.

[34] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[35] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[36] D. H. D. Warren. Or-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT'87*. Springer Verlag, LNCS 250, 1987.

[37] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Proc. IEEE Int'l Symp. on Logic Prog.*, pages 92–102, 1987.

[38] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *IEEE Int'l Symp. on Logic Prog.*, pages 198–203, 1984.

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