A Systematic Approach to Exploiting Implicit Parallelism in Full Prolog

Gopal Gupta

Department of Computer Science
New Mexico State University
Las Cruces, NM, USA
gupta@nmsu.edu

Vítor Santos Costa

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

Abstract

Prolog is a practical declarative programming language based on Horn Logic. In this paper we argue that implicit parallelism can be extracted from *full* Prolog. By full Prolog we mean pure Prolog with extra-logical features such as side-effects, database predicates and cuts. Prolog programs have three main forms of implicit parallelism present in them: or-parallelism, independent and-parallelism and dependent and-parallelism. Starting from an or-parallel version of full Prolog, we show how independent and-parallel and dependent and-parallel versions can be systematically derived. We present details of how the environment is represented using *Paged Binding Arrays* to handle multiple bindings of variables in presence of or-, independent and-, and dependent and-parallelism. We also show how extra-logical predicates can be supported in this combined parallel implementation of Prolog.

Our approach can be summarized as follows: We begin with an or-parallel implementation of Prolog that uses *Binding Arrays* for environment representation; we next show how independent and-parallelism can be added to such an implementation by annotating Prolog programs at compile time with *conditional graph expressions*, and generalizing the binding arrays used for environment representation to *paged* binding arrays. Next we show how extra-logical predicates can be supported in this combined independent and-or parallel system to support full Prolog. We then show how (determinate) dependent and-parallelism can be added to exploit parallelism further, using Warren's *Andorra* Principle, and how the resulting system subsumes Prolog as well as flat concurrent logic languages such as *Flat Guarded Horn Clauses (FGHC)*. The details of environment representation and implementation in the combined independent and-or-dependent and parallel system are completely specified. The resulting system is a powerful system that implicitly supports parallel execution of full Prolog, as well as reactive computations. We also elaborate how our techniques for environment representation etc. can be extended to support non-determinate dependent and-parallelism (the Extended Andorra Model)

1. Introduction

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

Or-parallelism: arises when more than one rule defines some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently executing a nondeterministic procedure; it speeds up the search for a solution to the top-level query by exploring alternative solutions in parallel. Or-parallelism has been efficiently implemented in the Aurora [LWH90] and Muse [AK90] systems.

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

Dependent And-parallelism 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 interactions.

So far researchers have been focusing their attention on devising systems that exploit only a single form of parallelism [AK90, LWH90, HG90, S87], or combining or-parallelism with independent and-parallelism [GS92, GH91, RK89, WR87] or dependent and-parallelism [SWY91c]. However, recently proposals have been made for exploiting all three sources of parallelism in a single framework, such as the Extended Andorra Principle [W90, HJ90], which allows for exploitation of all three forms of parallelism in an integrated framework. However, it is still not very clear how a system exploiting all three sources of parallelism can be implemented—the principle problem being the management of multiple environments that arise at runtime. This paper is an attempt in solving the problem for systems that exploit all sources of parallelism. We present a data structure called the *Paged Binding Array* for representing multiple or-parallel environments in the presence of independent and- and dependent and-parallelism. The Paged Binding Array (or PBA for brevity) is an extension of the Binding Arrays that have been devised for purely or-parallel execution [W84, W87]. Our approach is incremental, in that we first show how a purely or-parallel system can be implemented using PBAs, we then show independent and-parallelism can be accommodated, followed by how determinate dependent and-parallelism can be accommodated. Finally we show briefly how non-determinate dependent and-parallelism can be accommodated. The final proposal is in effect an implementation scheme for the Extended Andorra Model.

The rest of this paper is organized as follows: Section 2 reviews the binding arrays technique for purely or-parallel systems. Section 3 discusses the problem of adding independent and-parallelism, and a solution to the problem using paged binding arrays. Section 4 discusses the addition of determinate dependent and-parallelism to or-parallel and independent and-parallel system. Finally section 5 presents ideas about implementing the Extended Andorra Model. We assume some familiarity with sequential implementation of Logic Programming Systems [W83], and with parallelism in logic programs.

2. Or-parallelism and Binding Arrays

The principle problem in implementing or-parallelism is that of managing multiple bindings for variables. A variable in a goal may get bindings from each one of the matching clauses during or-parallel execution. The multiple conditional bindings of such a variable should be readily accessible in the respective environments for efficient realization of or-parallelism. A comprehensive survey of problems, solutions and trade-offs for implementing or-parallelism can be found in [GJ90]. The Binding Arrays method [W84, W87] has proven very successful in managing conditional bindings efficiently, and is outlined below.

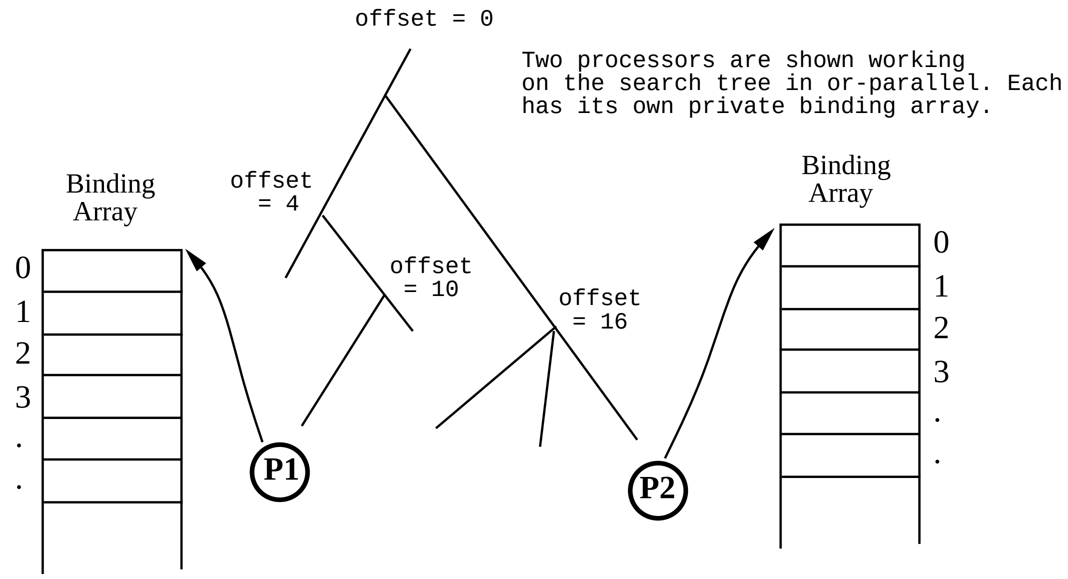

Fig. 1: Binding Arrays

The method consists of allocating a binding array, a simple one dimensional array, for each processor of the multiprocessor system. With each node in the search tree of the program a counter is associated. The value of the counter is 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 bindings 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. The ith location in the binding array of the processor is allocated for holding the binding. 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, it still remains a constant-time operation—the principle advantage of the BA method over other methods (see the dereferencing algorithm given below). 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 also need to be recorded in the trail stack along with thier addresses. Processors update their BAs from the trail stack as they move up and down the or-parallel tree while switching nodes.

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

The Aurora system [LWH90], a highly successful or-parallel system, has been implemented using Binding Arrays.

3. Adding Independent And-Parallelism

Exploitation of independent and-parallelism requires the identification of *independent* goals, i.e., goals that do not influence each other's execution at runtime. The most successful technique for accomplishing this has been that annotating of a logic program with *Conditional Graph Expressions.* Program clauses are compiled into expressions (called Conditional Graph Expressions) of the form

$$(condition \Rightarrow goal_1 \ \& \ goal_2 \ \& \ \ldots \ \& \ goal_n)$$

meaning that, if *condition* is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can be either $ground(v_1, \ldots, v_n)$, which checks whether all of the variables $v_1, \ldots, v_n$ are bound to ground terms, or it can be $independent(v_1, \ldots, v_n)$, which checks whether the set of variables reachable from each of $v_1 \ldots v_n$ are disjoint from one another. Checking for *ground* and *independence* involve very simple run-time tests, details of which are presented in [D84]. Conditional Graph Expressions can be automatically generated from a given logic program using compile-time analysis tools [MH89].

In the presence of or-parallelism the multiple solutions of independent goals in the CGE can be found in or-parallel. This suggests further optimization of logic program execution—one can compute all possible solutions of independent goals and then *join* them to produce all possible solutions for the CGE. This is in contrast to what is done in sequential (Prolog) execution—an independent goal to the right will be re-computed in its entirety for every solution of an independent goal to the left. However, we have argued that although adopting the former approach leads to fewer inferences being performed, it requires complex machinery to implement it and, further, supporting Prolog as a user language becomes trickier [GH92, GS92, GH91]. We therefore advocate an approach where even in the presence of or-parallelism independent goals to the right are recomputed, just as in sequential Prolog.

Taking the approach of re-computing independent goals to the right in a CGE, we have developed the notion of a *composition-tree* to represent or-parallelism and independent and-parallelism [GH92] in a single framework.

Consider the CGE `(true => a & b)`, the classical and-or tree for which is shown in figure 2.(i). With recomputation, 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, each composition node corresponds to a different solution for the parallel conjunction, i.e., a different "continuation". The extended tree, called the *Composition-tree* (C-tree for brevity), for the above query might appear as shown in figure 2.(ii)—for each alternative of the and-parallel goal `a`, goal `b` is entirely recomputed (in fact, the tree could contain up to 9 c-nodes, one for each combination of solutions of `a` and `b`). To represent the fact that a parallel conjunction can have multiple solutions we add a branch point (choice point) before the different composition nodes. (Note that c-nodes and branch points serve purposes very similar to the Parcall frames and markers of the RAP-WAM [H86].) The C-tree can represent or- and independent and-parallelism quite naturally—execution of goals in a c-node gives rise to independent and-parallelism while parallel execution of untried alternatives gives rise to or-parallelism. Note that there is a direct mapping (or transformation) from the (or-parallel) search tree to the C-tree. This transformation is such that the independent and-parallel parts (corresponding to the parallel goals in the CGE) can be readily identified and computed in and-parallel.

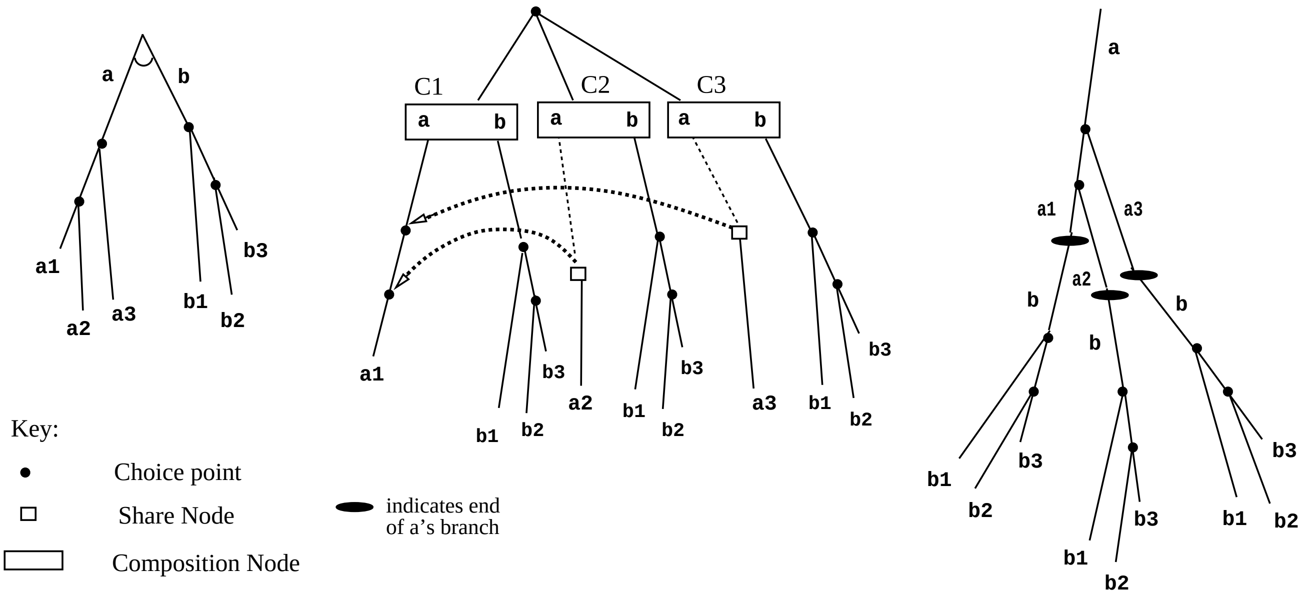

fig 1: And-Or Tree fig 2: Composition Tree fig. 3: Or-Parallel Tree

Fig. 2: Composition Tree

While exploiting and- and or-parallelism in the same framework, we are faced with the seemingly anti-thetical requirement that a group of processors working in and-parallel *should* have total access to each other's environment, while a group of processors working in or-parallel *should not* have access to each other's environment. To get around this problem we introduce the notion of *teams of processors*—all the available processors are organized into *teams*†. All processor in a team share a single environment, while different teams have different environments. Or-parallelism is exploited at the level of teams while independent and-parallelism is exploited among processors within a team. The notion of teams of processors meshes very well with the notion of C-trees. Different teams work on different c-nodes, where member processors of each team execute the independent goals in the CGE corresponding to that c-node in and-parallel. The number of c-nodes created in the C-tree is governed by the amount of resources (number of teams) available. If enough teams are not available other solutions are found by backtracking, similar to RAPWAM [H86] (thus, if there were only one team only one c-node will be created).

Having shown how the or-parallel tree can be extended to represent independent and-parallelism, we now show how the binding arrays scheme can be extended to work for the C-tree.

3.1. Paged Binding Arrays

In the presence of and-parallelism different segments of a branch of the or-parallel tree may be executed in parallel. Consequently, assignment of offsets to conditional variables such that they form an increasing sequence along a particular or-branch becomes difficult (see [GS92] for detailed discussion), and results in large amount of space going unused in the binding array.

To solve this problem and allow for and-parallelism in the presence of or-parallelism 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 conditional variables 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 conditional variables that have been allocated space in the same logical segment (i.e. conditional variables 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 (PT) (figure 3). Note that since all processors in a team need to see the same environment, the auxiliary and the binding array are made common to all the processors in a team, i.e. there is a single binding array and page table per team (See [GS92] for more details).

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 conditional variables 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 conditional variables within a

† Teams of processors were first employed in the Andorra-I system [SWY91a, SWY91b, SWY91c].

‡ Thanks are due to David H. D. Warren for pointing out this similarity.

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 conditional variable 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.

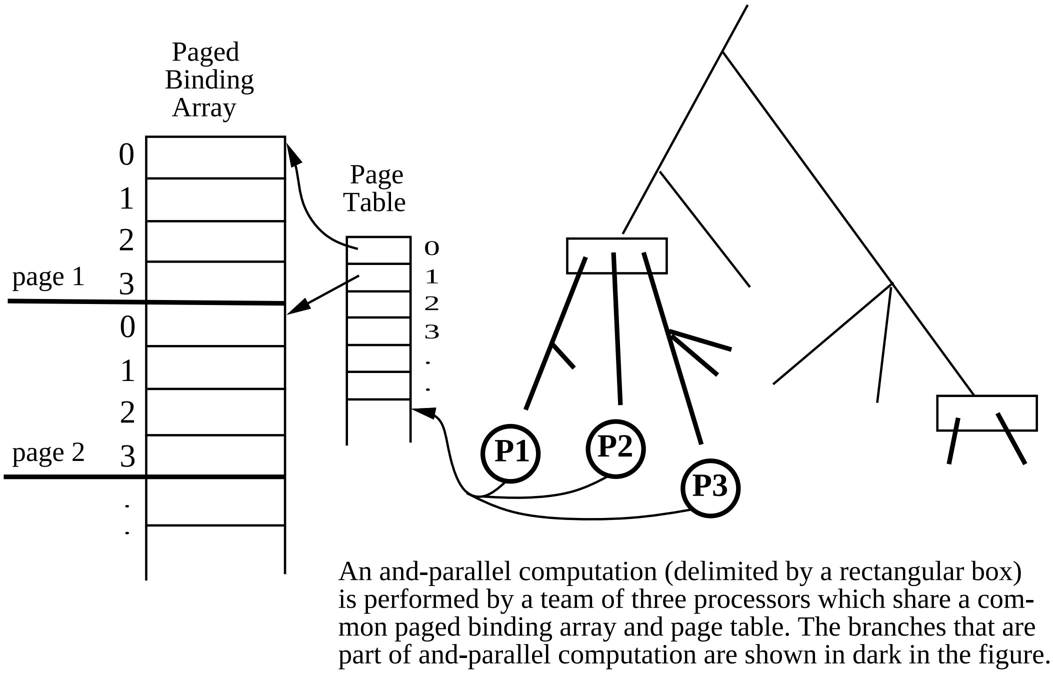

Fig. 3: Paged Binding Arrays

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 usage of space in the PBA.

Note that empty unused slots can be created in the page table (PT). 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. 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.

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>}


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 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 pure or-parallel case, a conditional binding 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.

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. Also, seniority of conditional variables, which needs to be known for efficient compilation so that "older" variables do not 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.

One can argue with a reasonable degree of certainty that an and-or parallel system based on PBA will be efficient. This is because in the presence of only or-parallelism, our system's behaviour will be identical to Aurora (except for an extra level of indirection), while in the presence of only independent and-parallelism our system's behaviour will be identical to &-Prolog (except for extra levels of indirection while accessing conditional variables).

3.2. Supporting full Prolog

One of the principle goal of our approach is to support *full* Prolog as the user language of our parallel logic programming system. By supporting full Prolog we mean supporting the extralogical (e.g. `assert` and `retract`), metalogical (e.g. `cut` and `var`), and side-effect predicates (e.g. `read` and `write`) of Prolog in a manner that the user-observed behaviour during parallel execution of a program is identical to its behaviour during sequential execution (in such a case we say that the parallel system preserves Prolog's *sequential semantics*). The execution of these predicates is sensitive to the order in which computation is done, therefore, preserving sequential semantics is not an easy problem. To achieve our goal we have devised techniques for preserving Prolog's sequential semantics in systems combining or- and independent and-parallelism.

To ensure that sequential semantics of Prolog is preserved during parallel execution in a system based on C-trees, the following rules have to be observed [GS91] when an execution-order sensitive predicate is reached in a goal inside a CGE. Consider an extralogical predicate e encountered in one of the alternatives of the goal $\mathtt{g}_i$ in the CGE $(\mathtt{true} \Rightarrow \mathtt{g}_1 \ \& \ \ldots \& \ \mathtt{g}_n)$ that is encountered during execution of a program. During combined independent and- and or-parallel execution, the extra-logical predicate e can be executed if and only if: (i) it is in the leftmost branch of the subtree rooted at $\mathtt{g}_i$, and; (ii) All subgoals to the left of $\mathtt{g}_i$ in the CGE, i.e. $\mathtt{g}_1$ through $\mathtt{g}_{i-1}$ have finished execution, and; (iii) The CGE containing $\mathtt{g}_i$ recursively satisfies conditions (i) and (ii) with respect to the subtree and outer CGEs that may contain it. (In the case of a cut, the conditions above can be somewhat relaxed and the cut can be executed earlier [GS91]. This is because a cut does not affect the solutions of a query, rather it affects the number of solutions reported and the amount of space searched. See [GS91] for more details.)

4. Adding Dependent And-Parallelism

Exploiting dependent and-parallelism in presence of non-determinism (specifically or-parallelism) has been a challenging problem since the inception of parallel logic programming research. Recently, however, techniques have been proposed to exploit dependent and-parallelism and or-parallelism. together. We first consider the exploitation of *determinate* dependent and-parallelism with other forms of parallelisms. By determinate dependent and-parallelism we mean that only those goals are executed in dependent and-parallel that have *atmost* one matching clause. Such a combination of or-parallelism and dependent and-parallelism was first proposed by David Warren and the framework termed *Basic Andorra Model* [SWY91c]. An advantage of the Basic Andorra Model is that in addition to exploiting parallelism, it also leads to the reduction of program's search space because it employs a goal selection rule different from that used by Prolog. It gives priority to determinate goals over non-determinate goals. As a result there is a clear separation of or-parallelism and dependent and-parallelism (which also make this model simpler to implement, and indeed it has been implemented in the Andorra-I system [SWY91c]).

The control of the Basic Andorra Model is depicted in figure 4. Execution in Basic Andorra Model consists of two phases: The Dependent And-parallel Phase (DAP) and the Or-parallel Phase (ORP). Execution starts in DAP, in which the goal-list is examined and all determinate goals executed (in parallel). Execution of these determinate goals may give rise to other determinate goals, which will be subsequently executed. Once there are no determinate goals left to execute, DAP will be exited and ORP entered. In the ORP the leftmost goal in the goal-list will be picked up and a choice point created for it. The first alternative of this choice point will then be selected (the other alternative will be made available to other processors for or-parallel processing), and the DAP phase entered to process determinate goals along this alternative in parallel.

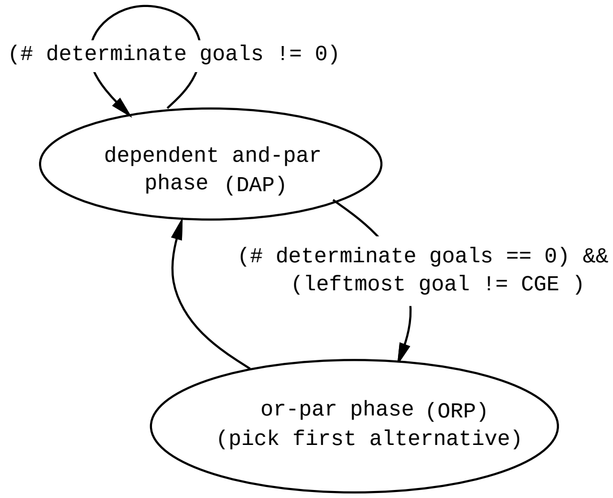

Fig. 4: Control in Basic Andorra

Independent and-parallelism can be incorporated into the Basic Andorra Model to exploit all three forms of parallelism in a single framework (although dependent and-parallelism will only be exploited for determinate goals). We call such an integrated framework IDIOM (Integrated Dependent and-, Independent and-, and Or-parallel Model). Thus, an extra state will be introduced in the control diagram of Fig. 4, namely, for independent and-parallel execution (fig. 5).

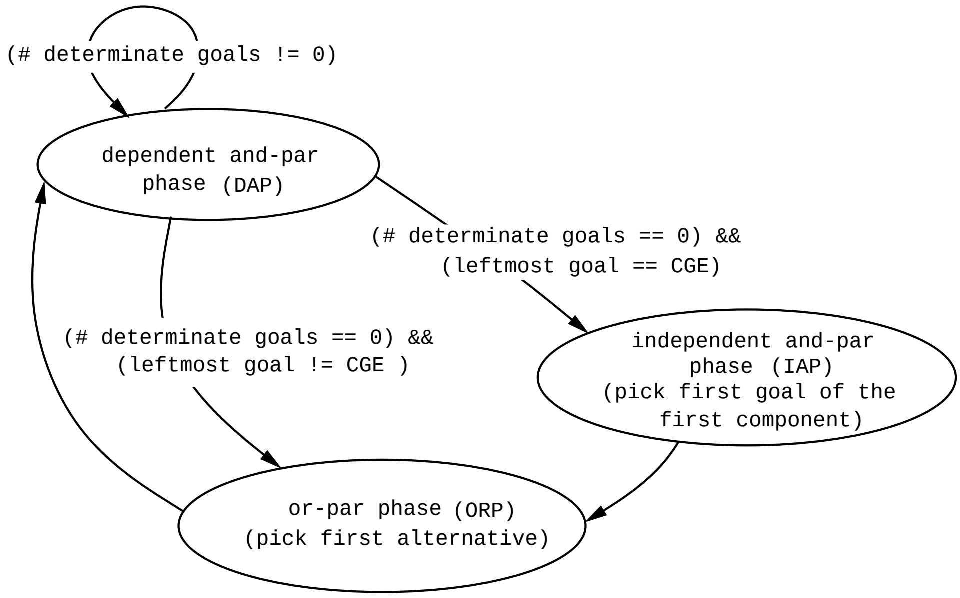

Fig. 5: Control in IDIOM

Thus, execution in IDIOM consists of 3 phases: the Dependent And-parallel Phase (DAP), the Independent And-parallel (IAP) phase, and the Or-parallel (ORP) phase. In the DAP phase all goals that can be determinately reduced are evaluated in parallel (including those inside the CGEs) until none are left. The leftmost goal is then examined to see if it is (i) a simple goal; or (ii) a CGE. In case (i) the ORP phase is entered while in case (ii) the IAP phase is entered. In the ORP phase the first alternative to the goal is selected (other alternatives are made available for or-parallel execution), head unification is performed, and the DAP phase is entered again. In the IAP phase, firstly the condition in the CGE is evaluated. If true, the components of the CGE are made available for independent and-parallel processing. In practice, the leftmost component is selected immediately and the other components are made available for parallel execution. The ORP phase will then be entered to process the leftmost goal of the selected component in or-parallel. If the condition in the CGE evaluates to false, the ORP phase is entered immediately to process the leftmost goal in the CGE in or-parallel. Execution of the continuation of the CGE then continues in the DAP phase.

The eager evaluation of determinate goals in the presence of CGEs can result in some unusual interactions. This is because the compile time analysis for generating CGEs assumes left-to-right Prolog execution order while the Andorra Principle employs a different selection rule (that of executing determinate goals eagerly). These interactions can be put into two categories:

(i) (eager) evaluation of determinate goals affecting the evaluation of conditions in CGEs.

(ii) (parallel) evaluation of goals in CGEs affecting the (eager) evaluation of determinate goals.

These problems have been studied by us and solutions that do not compromise with independent and- and dependent and-parallelism have been proposed. They have been omitted from here for simplicity, details can be found in [GSY91].

Note that IDIOM subsumes both Prolog and Flat Committed Choice Languages such as FGHC. This is because the Basic Andorra Model, to which IDIOM adds independent and-parallelism, possesses that property.

4.1 Paged Binding Array for IDIOM

The Paged Binding Array can easily be adapted for accommodating dependent and-parallelism in the presence of or-parallelism and independent and-parallelism. The notion of teams of processors that was used for executing independent goals in and-parallel can also be employed for executing determinate dependent goals in and-parallel. Idle processors in the team pick determinate goals from ready-to-run-goals queue for dependent and-parallel processing. Processors working in dependent and-parallel allocate a separate page in the PBA for conditional variables that are created (the variable creation and dereferencing algorithm remains unchanged). Note that because of the paged nature of our binding array, we do not have to resort to the "chunk scheme" used by Andorra-I implementation to allow for shared use of BA. In fact, the paged binding array can be viewed as a generalization of the chunk scheme that allows for or-parallelism within independent as well as dependent and-parallel goals (in IDIOM, though, only independent and-parallel goals are allowed to have or-parallelism within them).

We envisage that each processor in a team has a queue for recording ready to execute determinate goals. How these queues are examined for selecting dependent and-parallel work depends on the *mode* of the processors. A processor in a team can be in a *master* mode or a *slave* mode. Every processor in *slave* mode has a unique master associated with it. A processor in the slave mode will record ready-to-run determinate goals in its master's queue. It will also pick up dependent and-parallel goals from its master's queue. When a team is not executing a CGE, there is only a single team-wide master in whose queue determinate goals generated are recorded, and picked for dependent and-parallel execution. When a CGE is encountered, the (slave) processors that pick up independent and-parallel goals temporarily change their mode to master. The other slave processors that do not have an independent goal to execute distribute themselves among the new master processors and help them in executing determinate goals in dependent and-parallel. Once the execution of a CGE is over, the temporary master processors restore their mode to slave. The effect is to create sub-teams within teams to solve determinate goals in dependent and-parallel that arise within independent and-parallel execution of goals in the CGE.

4.2. Supporting full Prolog in IDIOM

To support extralogical predicates in the presence of (determinate) dependent and-parallelism we will take the approach similar to that taken by Andorra-I system [SYW91a]. Since to exploit dependent and-parallelism determinate predicates are executed eagerly, to ensure that sequential semantics of Prolog is preserved during parallel execution, we have to make sure that a goal whose eager execution can adversely affect the evaluation of an extralogical predicate is executed only after that extralogical predicate has been executed. For example consider the clauses:

```
p(X):- write(X).

p(X):- ...
```

and the query `?- p(X), X = f(a,b)`. Executing `X = f(a,b)` eagerly (since it is determinate) will cause the `write` predicate in the first clause of `p` to print `X` as `f(a,b)`, while in sequential execution `X` will be printed as an unbound variable. In the Andorra-I preprocessor, such predicates as `p` are recognized by compile-time analysis and declared "sensitive." Determinate goals which can affect sensitive predicates are then not executed eagerly at runtime.

5. Extended Andorra Model

Having addressed the question of adding determinate dependent and-parallelism, we next consider adding non-determinate dependent and-parallelism. Eager or-parallel execution of dependent goals can lead to the problem of doing speculative computation which may turn out to be wasteful (in extreme cases, this may lead to non-termination). For instance, consider the goal `p(X), q(X)` where both `p` and `q` have multiple matching clauses. In normal sequential execution `p` will act as a producer for bindings of `X`, which will be consumed by `q`. However, if dependent and-parallelism is exploited both `p` and `q` may attempt to bind `X`, and in that process `q` may search a larger space compared to if it had known the binding for `X`. Therefore, some means are needed to control the eager or-parallel execution of dependent goals.

A model that addresses all these problems has been proposed by Warren, called the Extended Andorra Model (EAM). It allows for or-parallel execution of dependent/independent goals, but only so far as their execution does not bind a variable external to the goal (in other words goals execute *locally*). However, if a goal is deemed as a producer goal for an external variable, it is allowed to proceed when it generates a binding for that variable†. Note that or-parallel execution of non-determinate dependent goals is carried out together with eager reduction of determinate goals in dependent and-parallel. Thus, the EAM augments the Basic Andorra Model by allowing even non-determinate dependent goals to be executed in advance. In the example just mentioned, non-determinate goals `p` and `q` will be executed in parallel together with other determinate goals, but since `q` is considered a consumer for the binding of `X` it will be suspended the moment it accesses `X`'s binding. On the other hand, `p`, the producer of the binding, will continue execution even though the variable `X` is external to the goal (variable `X` is external to `p` and `q` because it is defined in the environment that *contains* `p` and `q` and not in the environment within `p`) or `q`.

At some point, the computations done by `p` and `q` have to be put together. The operation to accomplish this is called `global forking` in EAM. Essentially, for each binding produced for `X` by `p` a copy of `q`'s computation is made and the binding of `X` made visible in it (this is akin to taking a join of the or- alternatives in `p` and `q`. See fig. 6). Only those conjoined goals will continue executing which agree on the bindings for `X` produced by `p` and `q` ‡. The rest will fail.

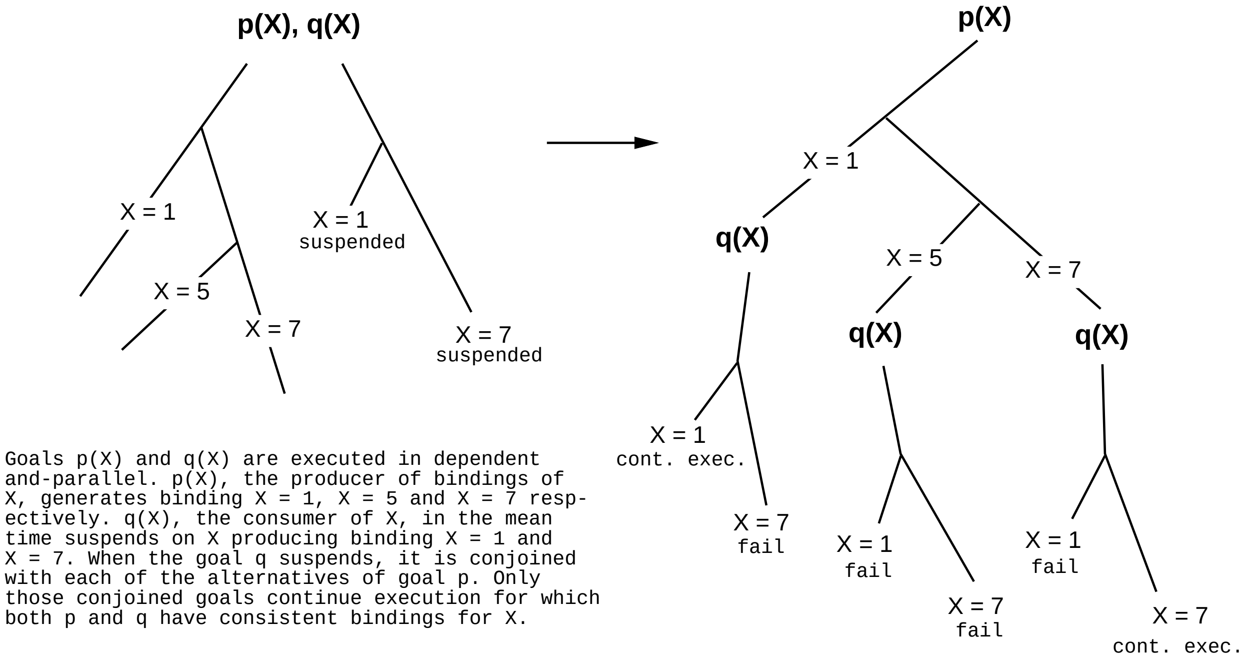

Fig. 6: Global Forking in the Extended Andorra Model

The Extended Andorra Model not only exploits more parallelism than any other model, it also leads to a greater reduction in search space compared to the Basic Andorra Model. The Extended Andorra Model subsumes both Prolog and the Committed Choice Languages with non-flat guards. Note that the Extended Andorra Model does not make a distinction between independent and-parallelism and dependent and-parallelism. This is because independent and-parallelism is a special case of dependent and-parallelism, and since the model incorporates full dependent and-parallelism,

† In fact, in the presence of or-parallelism within dependent and-parallel goals, an additional condition for suspension of goals is needed. At present, it is the sequential operator which requires that all goals to the left be solved before goals to the right can begin execution. More details can be found in [GW92]

‡ The above is a very coarse description of the EAM, a more detailed description can be found in [W90, GW92].

it doesn't need to take any special care of independent and-parallelism (unlike IDIOM).

5.1 Towards an Implementation of the EAM

We believe that Paged Binding Array can be easily adapted for implementing the Extended Andorra Model. It is easy to capture local execution of goals in Paged Binding Arrays—the local environment of a goal can be identified through the pages in the Binding Array that are allocated by that goal during execution. A binding to an external variable can be identified by checking the page number in the pair assigned to the external variable. If it's smaller than the smallest page number being used by the local environment of the goal, the binding is external.

One can envisage having a *single* paged binding array for *all* teams from which pages are allocated. However, each team has its own page table which reflects the environment of that team. During a task switch, the team need not install-deinstall variable bindings from the trail to update its environment, as most traditional Binding Arrays based systems (such as Aurora) will do. Rather, it will just bring in pointers to those pages in its Page Table †. Likewise, in the Extended Andorra Model the operation of Global Forking just involves making copies of pages of the consumer goal's environment and installing the pointers to these pages in the page table.

Note that our design for implementation of the Extended Andorra Model is far from complete. However, we strongly believe that Paged Binding Arrays have a very important role to play in attaining this goal. Efforts are under way for achieving it.

6. Conclusions

In this paper we demonstrated how one can incrementally add the various kinds of parallelism found in logic programs to obtain an efficient implementation. Starting with an or-parallel implementation based on binding arrays, we showed how one can extend it to support independent and-parallelism. We then showed how the existing system can be augmented by *determinate* dependent and-parallelism. Addition of determinate dependent and-parallelism not only leads to more parallelism being exploited but also results in fewer inferences being performed at runtime. Finally, we discussed Warren's Extended Andorra Model which is a general framework for exploiting all forms of parallelism from logic programs and showed the suitability of the Paged Binding Array for implementing it.

Work is in progress for implementing the IDIOM execution model and designing an implementation of the Extended Andorra Model based on the Paged Binding Arrays.

7. Acknowledgements

Thanks are due to Manuel Hermenegildo, Rong Yang, and David H. D. Warren for their work which serves as the foundation for this paper. They have also contributed directly and indirectly in many other ways. Vìtor Santos Costa will like to thank the Gulbenkien Foundation, the University da Oporto, Portugal, and the European Commission for supporting his work and stay at Bristol. This project was partially supported by U.K. SERC grant GR/F 27420. Continued funding is provided by NSF grant CCR 9211732.

† This technique can also be used to reduce the the task switching time in a purely or-parallel system and an independent and-or parallel system based on the Paged Binding Array.

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.

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

[GH92] G. Gupta, "Recomputation based Implementation of And-Or Parallel Prolog," To appear in *Proc. of International Conference on Fifth Generation Computer Systems (FGCS '92)*, Tokyo, Japan, June, 1992.

[GH91] G. Gupta, M. Hermenegildo, "ACE: And/Or-parallel Copying-based Execution of Logic Programs," *Lecture Notes in Computer Science 569*, Springer Verlag.

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

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

[GJ90a] G. Gupta, B. Jayaraman, "Optimising And-Or Parallel Implementations," In *Proc. N. American Conf. 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.

[GS92] G. Gupta, V. Santos Costa, "And-Or Parallel Execution of Prolog based on the Paged Binding Array," To appear in *Parallel Architecture and Languages Europe (PARLE)*, June, 1992, Paris.

[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 International Logic Programming Symposium*, San Diego, CA, MIT Press, Oct. 1991.

[GW92] G. Gupta, D.H.D. Warren, "An Interpreter for the Extended Andorra Model *(Preliminary Report)*," Technical Report, Department of Computer Science, New Mexico State University. Also University of Bristol TR.

[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. Hermenegildo, K.J. Greene, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of International Conference on Logic Programming '90*, Jerusalem, MIT Press, pp. 253-268.

[HJ90] S. Haridi and S. Janson, "Kernel Andorra Prolog and its Computation Model," In *Proceedings of ICLP*, MIT Press, June, 1990.

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

[RK89] B. Ramkumar and L. V. Kale, "Compiled Execution of the REDUCE-OR Process Model," In *Proceedings fo the North American Conference on Logic Programming '89*, MIT Press, pp. 313-331.

[S87] E. Shapiro, "Concurrent Prolog : Collected Papers," MIT Press, 1987.

[SWY91a] V. Santos Costa, D. H. D. Warren, R. Yang, "The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model," In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991, pp. 443-456.

[SWY91b] V. Santos Costa, D. H. D. Warren, R. Yang, "The Andorra-I Engine: A parallel implementation of the Basic Andorra model," In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, pp. 825-839.

[SWY91c] 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.

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

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

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues", *1987 IEEE International Symposium in Logic Programming*, San Francisco.

[W90] D. H. D. Warren, "The Extended Andorra Model with Implicit Control," In *Proceedings ICLP '90 Workshop on Parallel Logic Programming*, June, 1990, Eilat, Isreal.

[WR87] H. Westphal, P. Robert, J. Chassin and J. Syre, "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," In *1987 IEEE Int'l Symp. in Logic Prog.*, San Francisco, pp. 436-448.