# Extended Dynamic Dependent And-parallelism in ACE*

Enrico Pontelli and Gopal Gupta
Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Box 30001, Dept. CS
Las Cruces, NM 88003
{epontell,gupta}@cs.nmsu.edu

**Abstract**

We present an execution model for Dynamic Dependent And-parallelism as well as an extension of the scheme in which *deterministic* consumer goals are allowed to bind their *dependent variables* ahead of time. The extended scheme leads to: (i) improved efficiency due to pruning of program search space; and, (ii) exploitation of more parallelism due to increased overlapping of dependent executions. In dynamic dependent and-parallel execution, given a parallel conjunction with a shared variable `X`, the leftmost goal is normally designated as the producer of the binding for `X`, all others goals are designated as consumers. If the producer goal finishes execution without binding `X`, then the leftmost consumer goal becomes the next producer and so on. In the extended scheme a deterministic consumer goal is also allowed to bind the dependent variable. Our extension leads to incorporation of coroutining in a dynamic dependent and-parallel system. The extended dynamic dependent and-parallel scheme can be regarded as a weak form of the *Extended Andorra Model*. The implementation of

*This paper is an extended version of the paper "Extended Dynamic Dependent And-Parallelism in ACE" which appeared in the proceedings of ACM PASCO'97.

the extended scheme has been realized, based on the *Filtered Binding Model* for implementing dependent and-parallelism, and has shown excellent results.
**Keywords:** And Parallelism, Andorra Model, Optimizations

# 1 Introduction

Logic programming is a popular computer programming paradigm that has been used in a wide variety of symbolic applications, ranging from Artificial Intelligence, Genetic Sequencing, Database programming, Expert Systems, Natural Language Processing, Constraint based Optimization, to general purpose programming and problem solving. The most popular logic programming language is *Prolog*.

Given a Prolog program, we would like to execute it as fast as possible. One way of executing a program faster is to execute it in parallel. The declarative nature of logic programs allows the compiler/runtime system to automatically extract parallelism, without any effort from the programmer. Note that this approach considerably differs from the approach in which parallelism is explicitly programmed by the user (as in most of the traditional programming languages). Two types of (implicit) parallelism have been identified and successfully exploited in logic programs:

1. *Or-parallelism:* 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 has been efficiently implemented in various systems, such as *Aurora* [27] and *Muse* [1].

2. *And-parallelism:* arises when more than one goal is present in the query or in the body of a procedure, and the resolution of the different (sub)goals is attempted in parallel. Two classes of and-parallel systems are typically identified:

   (a) *Independent And-Parallelism (IAP):* arises when run-time bindings for the variables in two or more goals are such to make the goals *independent* of one another, i.e., the bindings produced by each subgoal will not affect the computation of the other subgoals (e.g., they do not have any unbound variables in common [21]);

such independent goals can be run in parallel. &-Prolog [21] and ACE [37] are two systems that exploit IAP.

(b) *Dependent And-parallelism (DAP):* arises when dependent goals, i.e. goals that "compete" in the creation of the binding for common variables (also known as *dependent* or *shared* variables) are executed in parallel. Dependent and-parallelism is readily found, for example, in applications that involve producer-consumer interactions. Systems like DASWAM [45] and ACE [35] implement DAP.

In this paper we are mostly interested in dependent and-parallelism. Given a goal `p(X), q(X)` dependent and-parallelism can be exploited in varying degrees [35]. The two "extreme" schemes are:

1. *Curative Scheme:* The two subgoals can be executed independently. Each subgoal is operating on local environments containing duplicate instances of variables shared across parallel subgoals. Upon termination of a collection of parallel subgoals, a curative action is required to guarantee that all subgoals have produced consistent bindings—and eventually guarantee that the bindings respect the order of solutions of the corresponding sequential semantics. This curative activity is commonly indicated as *back unification.* This unrestricted form of execution of dependent goals is typically not recommended because it can lead to redundant execution [35].

2. *Preventive Scheme:* parallel dependent subgoals are executed concurrently only as far as they do not attempt to bind shared resources. Only subgoals that will preserve the correct sequential semantics will be allowed to proceed in binding shared variables (*producer goals*), while the remaining goals (*consumer goals*) will be suspended until the correct producers have accomplished their actions.

The first case is very similar to independent and-parallelism. Various models have been developed (see [35] for a brief survey) which are based on intermediate situations between the two extremes represented by the curative and the preventing scheme—e.g., *stream and-parallelism* where consumers are released as soon as a partial binding is created by the producer.

Determining a-priori (i.e., *statically*) which goal is going to bind a variable first is an *undecidable* problem [35]. In the description of the preventive scheme above, we assigned the roles of producer and consumer to goals. If we follow Prolog semantics, then the leftmost goal binding the dependent variable should be chosen as the producer while all others to the right will be chosen as the consumers.

Due to the undecidability of the problem, any a-priori determination of producers and consumers is necessarily approximate. It is possible that the role of the producer may have to be reassigned during execution, as the current producer goal may finish execution without binding the dependent variable—i.e., the initial approximation of producer/consumers was incorrect. In such a case, the leftmost consumer goal becomes the new producer. This way of realizing DAP is called *Dynamic Dependent And-parallelism (DDAP)* [44], and was originally proposed in detail by Shen [44, 45].

In dynamic DAP, bindings can be communicated only from the left to the right, i.e., bindings are done in Prolog order. A consumer goal that attempts to bind a dependent variable, suspends. Dynamic DAP thus allows only for 1-way communication, i.e., goals to the left can communicate bindings to goals to the right. Recently, we proposed and implemented an efficient scheme for dynamic DAP called the Filtered Binding Model [35] (earlier implementations also exist such as DASWAM [44]). In this paper we overview dependent and-parallelism and its implementation using the Filtered Binding Model, and report on an extension of dynamic dependent and-parallelism based on the filtered binding model in which, under certain conditions, a designated consumer goal is allowed to become a producer. The condition is very simple: a consumer goal is allowed to become the producer of a dependent variable if it binds that variable deterministically, otherwise it has to suspend. Binding deterministically means that there are no choice-points between the point in the consumer goal where the binding is being attempted and the beginning of the consumer goal. Allowing consumer goals to bind dependent variables leads us towards the Andorra Principle [50], and adds the ability to prune search space. It also allows incorporation of coroutining into our system, allowing for various flavors of two–way communication. In fact, this scheme of allowing consumers to make deterministic bindings is a *weak form* of the Extended Andorra Model (EAM) [51] rather than the Basic Andorra Model. Thus, our implementation can be regarded as a step towards the realization of the Extended Andorra Model. In the rest of this paper, we refer to this extension of DDAP as *Extended Dynamic Dependent And-Parallelism* or *EDDAP*. EDDAP can be seen as an elegant way of synthesizing dynamic DAP and the Andorra Principle. This combination of dynamic DAP and the Andorra Principle is novel, and to the best of our knowledge has never been proposed or implemented before.

Our implementation of EDDAP is based on the Filtered Binding Model. The ease with which the Filtered Binding Model allowed this extension to be implemented, shows its sophistication. In fact, in this paper, we argue that any arbitrary strategy for choosing consumers/producers can be easily implemented with the help of the Filtered Binding Model. For example, committed choice languages can be easily and efficiently implemented with the help of the Filtered Binding Model.

The rest of the paper is organized as follows, Section 2 describes dependent and-parallelism, dynamic dependent and-parallelism, and our extension of it. Section 3 introduces the Andorra and Extended Andorra Models, and shows that our extended scheme is a weak form of the Extended Andorra Model. Section 4 describes the Filtered Binding Model and its use in the ACE system to realize dynamic DAP. Section 6 describes how reactive computations can be realized in EDDAP, while Section 7 describes performance results of implementation of coroutining in the ACE system.

# 2 Dependent And-parallelism

## 2.1 Preliminary Notions

An and-parallel execution can be represented using a tree structure—see Figure 1. During program execution, we distinguish between *sequential* and *parallel* conjunctions. Parallel conjunctions will (potentially) lead to parallel execution of the goals which appear in the conjunction; sequential conjunctions will lead to the sequential execution of the goals in the conjunction. The detection of which conjunctions should be parallel and which should be sequential, in order to achieve the best parallel performance, can be either realized via manual code annotations or through compile-time analysis. In this work we will use the notation "&" to indicate parallel conjunctions, and the symbol "," to indicate sequential conjunctions. In the tree representations, the larger boxes represent parallel calls, i.e., conjunctions which can be executed in parallel; each parallel call in the tree has a child node for each goal present in the conjunction. Each parallel call may optionally indicate the dependencies existing between goals—e.g., in Figure 1 we have indicated the dependence created by the shared variable `X` between the first and the third goal. Sequential executions are represented as successive segments along the same branch. Circles are used to represent choice points created by the computations.

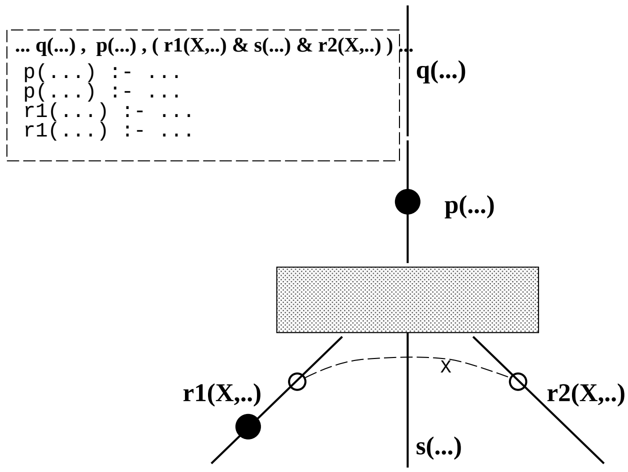


Figure 1: Computation Tree with DAP

A sequential execution will build the tree depth-first from left to right. A DAP execution will extend the tree in parallel along different branches. Sequential conjunctions simply extends one branch, while parallel conjunctions generates new branching points.

In the rest of this paper we will use the following notation:

- given a node $n$ in the tree, $\mathit{Type}(n)$ is either $\mathit{ChoicePoint}$ or $\mathit{ParCall}$, depending on whether $n$ is a choice point in the tree or a parallel call
- given a node $n$, $\mathit{Parent}(n)$ indicates the immediate predecessor of $n$ in the tree
- given a node $n$ such that $\mathit{Parent}(n) = m$ and $\mathit{Type}(m) = \mathit{Parcall}$, then $\mathit{Index}(n)$ is the position of $n$ in the left-right ordering of the children of $m$—i.e., $\mathit{Index}(n) = 1$ for the leftmost child of $m$, and so on
- given a node $n$ such that $\mathit{Type}(n) = \mathit{ParCall}$, then $\mathit{Vars}(n)$ is the set of dependent variables which have been introduced in the parallel call corresponding to node $n$, and $\mathit{Child}(n, i)$ is the i-th children of $n$
- given a node $n$ such that $\mathit{Type}(n) = \mathit{ChoicePoint}$, then $\mathit{Alts}(n)$ indicates

how many unexplored alternatives are still associated to the choice point corresponding to node $n$

- given a node $n$, $Bottom(n)$ represents the node $m$ which is the rightmost leaf in the subtree which has $n$ as root.

## 2.2 Introduction to DAP

DAP is exploited when two subgoals that have data dependencies between them are executed in parallel. Dependent subgoals have at least one variable in common, and the binding produced by one of them for such a variable influences the structure of the computation of the other subgoals [35, 45]. The "classical" example of Dependent And-Parallelism (DAP) execution is represented by a conjunctive goal of the form: $p(X)$ , $q(X)$ in which the two subgoals "compete" (in parallel) to construct the binding for the common (or *shared*, or *dependent*) variable $X$.

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-dependence due to shared variable `X`[1]. Suppose they are defined by the following clauses:

```
p(X) :- g1, X = 1, s1.       q(3) :- h1.
p(X) :- g2, X = 2, s2.       q(4) :- h2.
p(X) :- g3, X = 3, s3.       q(5) :- h3.
```

where $g_i$ and $h_i$ involve a fair amount of computation. Clearly, there is only one solution for `X`, namely, 3. In a sequential execution, only $h_1$ will ever get executed (while solving `q`), 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 $h_1$, $h_2$, and $h_3$. 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 $h_2$ and $h_3$.

[1] Following [45] we term `X` the dependent variable.

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).

It follows from the discussion above that dependent and-parallel execution has to be carried out under certain constraints, so that redundant computations are not performed. Observe that, in general, speculative computation (which may become redundant later) cannot be completely avoided—a failure in a subgoal will make the execution of any subgoal on its right speculative. While we may not be able to completely avoid speculative computation, we can aim for the following goal: for a given goal and program, *no slow-down* is produced with respect to sequential execution. That is, parallel execution takes an amount of time bounded by the time taken for sequential execution. This implies that the speculative computation should always be performed in parallel with a non-speculative computation. To achieve this the constraint that is usually laid down is that only one of the subgoals (the producer subgoal) binds the dependent variable, the others (the consumer subgoals) only read its value. Thus, the dependent variable becomes a one way communication channel (or a *stream*). Hence, dependent and-parallelism is also known as *stream and-parallelism.* This approach guarantees (modulo overheads) that minimum redundant computation is performed, since the producer goal will be executed both during parallel as well as sequential execution. The (speculative) consumer goals are not allowed to compute ahead of the producer, and are executed in parallel with (or after) the producer, thus they cannot perform any redundant execution that will violate the *no slow down* principle.

## 2.3 Dynamic DAP

In determining the producer goal we can either let the leftmost goal containing the shared variable be the producer, following Prolog semantics, or use some other strategy for choosing a producer that will not lead to redundant computations [43]. Considerable amount of work has been done in designing models and building systems where the current active leftmost goal containing a dependent variable is designated as its producer (see [35] for a short overview). All other goals are consumers. This form of DAP has been termed Dynamic DAP in the literature [45], as the runtime system has to dynamically keep track of the leftmost active subgoal for each dependent variable at all time. The leftmost active subgoal for a dependent variable may have to be updated dynamically because the current leftmost active goal may terminate without producing a binding for the dependent variable (since the problem of deciding the actual producer for a variable is in general undecidable). The Dynamic DAP strategy guarantees that the bindings will be made in Prolog order, thus, the search space during parallel execution cannot be larger than for sequential execution.

Maintaining Prolog semantics during parallel execution also means supporting non-deterministic computations, i.e., computations that can potentially produce multiple solutions. In many approaches DAP has been restricted to only those cases where $p$ and $q$ are deterministic [43, 2]. This is largely due to the complexity of dealing with distributed backtracking. Nevertheless, it has been shown [45] that imposing this kind of restriction on DAP execution may severely limit the amount of parallelism exploited. Our objective is to exploit DAP even in non-deterministic goals. Techniques for fast backtracking in independent and dependent and-parallel systems have been studied by various researchers [22, 46, 36].

## 2.4 Extended Dynamic DAP

We can also employ the Andorra principle in designating producer and consumer status. The Andorra principle states that during execution of a logic program, whenever possible, deterministic steps should be done in preference to non-deterministic ones. Adoption of the Andorra Principle, in general, may lead to a drastic reduction in the search space. We can adopt the Andorra Principle in Dynamic DAP execution as follows: a non-leftmost active subgoal for a dependent variable `X`, that will normally be designated as a consumer goal, is permitted to bind `X` *if it does so deterministically.* Binding deterministically means that no further bindings shall be produced for the dependent variable by this goal. Thus, if a designated consumer goal attempts to bind the unbound dependent variable deterministically, then instead of causing it to suspend, the binding is immediately performed. Thus, the consumer goal becomes the producer goal. The normally designated producer goal (that has not produced a binding for `X` yet) now is turned into a consumer of this binding. The producer goal may have been highly non-deterministic, but after the consumption of this deterministic binding, it might become deterministic. This may prune the search space of the producer goal dramatically, resulting in fewer number of inferences performed, and a much improved performance.

For example, consider the query

```
?- p(X), q(X)
```

with respect to the following simple program:

```
p(Y)  :-  ... , Y = 1, ...
p(Y)  :-  ... , Y = 2, ...
p(Y)  :-  ... , Y = 3, ...
q(Z)  :-  ⟨ deterministic comp.⟩ , Z = 2, ...
```

If `q` binds `X` deterministically with the value `2` before `p` gets a chance to bind it (either to `1, 2` or `3`), then the deterministic computation in `q` will be executed only once. In normal computation, the deterministic computation will be executed thrice, two times out of which it will fail. Thus, the search space is reduced as the deterministic computation in `q` is executed only once.

The EDDAP scheme can also lead to more parallelism being exploited, since as soon as the deterministic consumer binds the dependent variable, other suspended consumer goals can be woken up. The fact that a deterministic consumer binds the dependent variables means that the normally designated (the leftmost) producer goal has not produced any bindings yet (otherwise the deterministic consumer would have found the variable bound). Thus, all consumer goals are still suspended. These suspended goals can be started sooner. In dynamic DAP these consumer goals would have waited until the current leftmost producer goal produces a binding.

A dynamic dependent and-parallel system that allows deterministic consumer goals to bind, can be regarded as a simplified implementation of the Extended Andorra Model. We present argument to this effect in the next section, along with a brief description of the Basic Andorra Model and Extended Andorra Model.

# 3 Basic and Extended Andorra Models

The Andorra Principle states that deterministic computations should be performed as early as possible. The simplest form of the Andorra Principle has been realized in the Basic Andorra Model [42]. In the Basic Andorra Model, goals in the current resolvent can be executed ahead of their turn ("turn" in the sense of Prolog's depth first search) in parallel *if they are determinate*, i.e., if at most one clause matches the goal. These determinate goals can be dependent on each other. If no determinate goals can be found for execution, a branch point is created for the leftmost goal in the goal list (non-determinate phase) and parallel execution of determinate goals along each alternative of the branch point continues. Dependent and-parallelism is obtained by having determinate goals execute in parallel. Thus, in parallel execution performed in accordance with the Basic Andorra Model, deterministic goals are never executed in parallel with non-deterministic goals.

The Extended Andorra Model [51] is an extension of the Basic Andorra Model. There are many manifestations of the Extended Andorra Model, but the essential ideas are summarized next. The Extended Andorra Model goes a step further and removes the constraint that goals become determinate before they can execute ahead of their turn. However, goals which do start computing ahead of their turn must compute only as far as the (multiple) bindings they produce for the uninstantiated variables in their arguments are consistent with those produced by the "outside environment." If such goals attempt to bind a variable in the outside environment, they suspend. Once a state is reached where execution cannot proceed, then each suspended goal which is a producer of bindings for one (or more) of its argument variables will "publish" these bindings to the outside environment. For each binding published a copy of the consumer goal is made and its execution continued. This operation of "publication" and creation of copies of the consumer is known as a *non-determinate promotion* step. The producer of bindings of a variable is typically the goal where that variable occurs first. However, if a goal produces only a single binding (i.e., it is determinate) then it doesn't need to suspend, it can publish its binding immediately, thus automatically becoming the producer for that goal, irrespective of whether it contains the left most occurrence of that variable or not (as in the Basic Andorra Model). This operation is termed a *determinate promotion* step. An alternative way of looking at the EAM is to view it as an extension of the Basic Andorra model where non-determinate goals are allowed to execute locally so far as they do not influence the computation going on outside of them. This amounts to including in the Basic Andorra Model the ability to execute independent parts of subgoals in parallel.

We next illustrate the EAM with the following very simple program:

```
p(X, Y) :- X = 2, m(Y).
p(X, Y) :- X = 3, n(Y).
q(X, Y) :- X = 3, t(Y).
q(X, Y) :- X = 3, s(Y).
r(Y) :- Y = 5.
?- p(X, Y), q(X, Y), r(Y).
```

When the top-level goal begins execution, all three goals will be started concurrently. Note that variables `X` and `Y` in the top-level query are considered to be in the environment "outside" of goals `p`, `q`, and `r` (this is depicted by existential quantification of `X` and `Y` in figure 2). Any attempt to bind non-deterministically these variables from inside these goals will lead to the suspension of these goals. Thus, as soon as these three goals begin execution, they immediately suspend since they try to constrain either `X` or `Y`. Of these, `r` is allowed to proceed and constrain `Y` to value 5, because it binds `Y` determinately (determinate promotion step). Since `p` will be reckoned the producer goal for the binding of `X`, it will continue as well and publish its binding. The goal `q` will, however, suspend since it is neither determinate nor the producer of bindings of either `X` or `Y`. To resolve the suspension of `q` and make it active again, the *non-determinate* promotion step will have to be performed. The non-determinate promotion step will match all alternatives of `p` with those for `q`, resulting in only two combination remaining active (the rest having failed because of non-matching bindings of `X`). These steps are shown in figure 2 (note that a few intermediate steps have been omitted between Step 2 and Step 3 for the sake of simplicity).

The above is a very coarse description of the Extended Andorra Model, a full description of the model is beyond the scope of this paper. More details can be found elsewhere [51, 16, 15]. The EAM is a very general model, more powerful than the Basic Model, since it can narrow down the search even further by local searching. Implementing the EAM is an extremely complicated task—the only attempt made, the AKL system [25], suffers the drawback of being based on a rather different language and semantics w.r.t. Prolog. Different parallel implementations of AKL have been recently completed with good performance results [29, 28, 33].

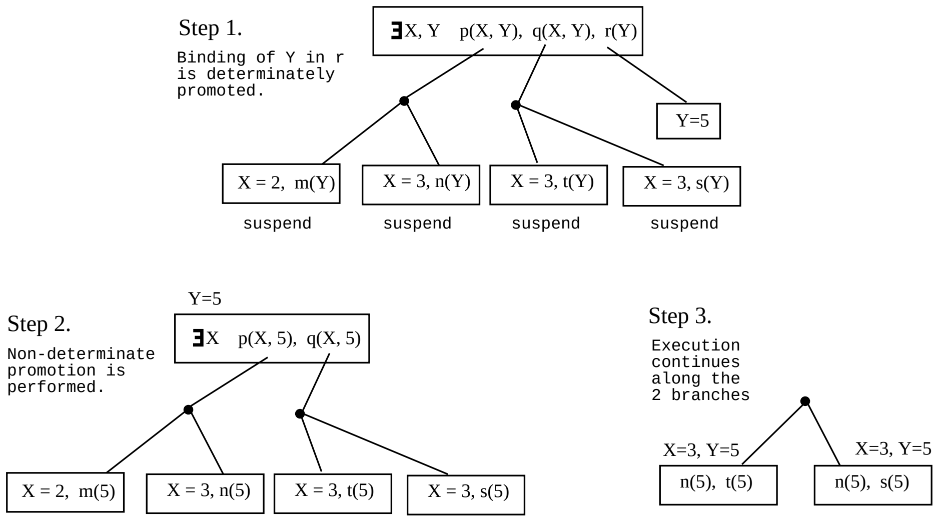


Figure 2: Execution in EAM

## 3.1 EDDAP: A Step Towards EAM

Dynamic DAP extended with allowing deterministic consumers to become producers (*Extended Dynamic Dependent And-Parallelism (EDDAP)*) is arguably an implementation of a weak form of the EAM. Allowing a consumer goal to bind the dependent variable, if it is doing so deterministically, clearly, is equivalent to the determinate promotion step. If both the producer and the consumer are non-deterministic, the dynamic DAP scheme finds all solutions by backtracking over them. Thus, dynamic DAP will recompute the non-deterministic consumer goal for each binding generated by the producer, while the EAM will reuse that part of the consumer goal that is independent of the dependent variable. Thus, the extended dynamic DAP model, while not exactly like EAM, can be regarded as an approximation to it. We would argue that the determinate promotion is the central idea in the EAM. Non-determinate promotion is just a way of starting a computation that is stalled due to all goals being non-deterministic. Thus, whether goal reuse is used or goal recomputation is used for combining the alternatives of the producer goal with the consumer goal is not a matter of great importance, as long as the stalled computation is started somehow. It is our opinion [13] that goal reuse is quite inefficient to implement (especially in the presence of or-parallelism) compared to goal recomputation. Nevertheless, extending the ACE system so that it becomes a true implementation of the EAM is a topic of our current investigation. Implementation mechanisms for flavors of the EAM have been also proposed in the AKL project [28, 33] and in the work by Costa and Lopes [26].

The Extended Dynamic DAP has different termination properties than the Basic Andorra Model. This is because in the Basic Andorra Model the goals are reordered, i.e., determinate goals are placed before non-determinate ones. This reordering can completely alter the search space explored. In fact, a terminating Prolog program may become non-terminating under Basic Andorra, as in the case of the pathological example below: given the query

```
?- p(X), q(X,Y).
```

with respect to the program

```
p(a).
p(b).
q(c, Y)  :-  q(c, Y).
```

On the other hand, the Extended Dynamic DAP does not reorder goals, rather it re-orders the *sequence in which bindings are made*, thus if a program terminates during sequential execution, it will also terminate in Extended Dynamic DAP. Indeed, if only one processor is available, then execution is *exactly* like Prolog. In fact, EDDAP execution has better termination properties than that of sequential Prolog execution—when execution makes use of more than one processor[2]. A program that is non-terminating under sequential execution may terminate under extended dynamic DAP, and this is achieved without the overhead of goals reordering.

Reordering of subgoals in the current goal-list, as done in the Basic Andorra Model, adds considerable overhead to the implementation, as any reordering done should be backtrackable. It is our belief that this is a major source of overhead in the Andorra-I system, a system based on the Basic Andorra Model [42]. Also, deterministic goals are not allowed to execute in parallel with non-deterministic goals, thus potentially limiting the exploitation of parallelism. The advantage that runtime reordering of subgoals has is that, even if execution is done by a single processor, search space is still pruned. In contrast, in the EDDAP scheme, subgoals are not reordered, and deterministic goals can run in parallel with non-deterministic goals. Thus, if only one processor is available, then execution is exactly like Prolog and no pruning of the search space takes place. The Andorra-I system may un-

[2]Or when a preemptive scheduling mechanism is used.

cover more determinacy due to the re-ordering. It is our belief, however, that this re-ordering can be accomplished quite effectively using compile-time techniques, like those developed by Ramakrishnan et al [41, 7]. Using their techniques at compile-time, determinate goals are moved from deep down in the search tree to the top. Executing the resulting program in EDDAP will, we hope, result in determinate goals getting executed sooner, as they have been moved to the left. EDDAP, in conjunction with such compile time techniques, thus, is a better alternative to the Basic Andorra Model, as it incurs less overhead, exploits more parallelism, has better termination properties, and uncovers similar amount of determinacy.

In the next section, we describe the Filtered Binding Scheme for implementing Dynamic DAP and its extension for implementing Extended Dynamic DAP. The generality of the Filtered Binding Scheme is also demonstrated.

# 4 Implementing Dynamic DAP

Any implementation scheme for dynamic DAP should provide means for keeping track of the producer and consumer subgoals for a given dependent variable. We assume that during compilation a program is statically annotated to identify the promising sources of parallelism (in the same style as DASWAM [45]—details on how to generate annotations are omitted due to lack of space [38]). For each dependent variable identified in the program, we need to maintain the producer and the consumer goals. If a producer goal finishes without binding the dependent variable, then the producer status should be dynamically passed to the leftmost consumer goal.

Our implementation of dynamic DAP is based on the concept of *instance* of a shared variable. Every parallel subgoal that has access to the same shared variable owns an instance of such variable. The instance is the "view" that the subgoal has of the shared variable. The actual status of producer/consumer can thus be associated with the variable instance instead of with the subgoal itself.

This concept can also be understood through an analogy to or-parallelism—based on the "dual" nature of or- and and-parallelism [34]. Implementation of or-parallelism has to deal with dependencies between parallel threads because different or-parallel threads may produce distinct (conditional) bindings for the same variable. The main difference is that, while in or-parallelism the alternative threads are disjunct (since we are actually trying to prove

$$\exists X(B_1(X) \vee B_2(X))$$

where $B_1$ and $B_2$ are the bodies of two alternative clauses), and consequently the bindings produced can be kept separate—since

$$\exists X(B_1(X) \vee B_2(X)) \equiv \exists X(B_1(X)) \vee \exists X(B_2(X))$$

while in the case of DAP the bindings always need to be kept consistent—since

$$\exists X(B_1(X) \wedge B_2(X)) \not\equiv \exists X(B_1(X)) \wedge \exists X(B_2(X))$$

Nevertheless, we do not need to go to the extreme of keeping a unique location for these bindings, as done in most of the models [44, 48, 8], since this requires complex synchronization mechanisms, locks, etc. An interesting alternative is exactly the one suggested by the or-parallel scheme, by trying to split the existential quantification in two separate quantifications:

$$\exists X(B_1(X) \wedge B_2(X)) \equiv \exists X_1 X_2(B_1(X_1) \wedge B_2(X_2) \wedge X_1 = X_2)$$

where the equality $X_1 = X_2$ is used to maintain consistency between the bindings produced by $B_1(\cdot)$ and $B_2(\cdot)$.

This gives rise to various degrees of possible implementations of DAP, that successively differ from each other based upon the moment at which the equality "=" is enforced. The simplest model is the one in which the parallel computations (e.g., $B_1(X_1)$ and $B_2(X_2)$) are carried on independently and '=' is applied only after they have both produced a binding for their variables. Speculative work can appear (and, in extreme cases, there can be infinite speculative work—i.e., non-terminating computations). On the other hand, the most complex model is the one in which the = is performed before starting the parallel execution, which leads to the various DAP models implemented (like DASWAM). We are interested in finding a solution between these two extremes. That is, we are interested in solutions in which the actual instances $(X_1, X_2)$ are kept separate as long as the different subgoals have different "views" of the shared variable, while performing (in a more or less explicit fashion) the unification as soon as it is needed (i.e., whenever the unification of instances requires propagation of some value from one instance to the other).

## 4.1 Filtered Binding Model for DAP

As mentioned in the previous discussion, given a parallel conjunction, each subgoal that can (directly or indirectly) access the corresponding dependent variable should maintain an independent path to it in order to access it. The idea behind the Filtered Binding model is to directly encode in the access path itself the information (the *filter* or *view*) that allows a subgoal to discriminate between producer and consumer accesses. This is in contrast to schemes used in other implementations, such as DASWAM, where the producer and consumer information for subgoals is implicit, and considerable processing may be required to infer the status of a given subgoal [45].

Figure 3 presents an intuitive schema of the Filtered Binding Model. Each subgoal has a local path to access the shared object (in this case a heap location allocated to hold the value of the shared variable) and the path contains a filter. In the figure, the filter is linked to information stored in the subgoal descriptor—this common information will be used to verify when the subgoal is a viable producer (i.e., it is the leftmost active subgoal in the parallel call).

Every access to a shared variable by a subgoal will go through the filter corresponding to that subgoal, which will allow it to determine the "type" (producer or consumer) of the access.

By properly organizing the unification process, as long as there is guarantee that no aliasing between shared variables occurs (unless they are both producer accesses), it can be proved that at any time a variable access will require traversal of at most one filter—which means determining in constant-time if an access is a producer access or a consumer access. The intuition is that the only case which requires maintaining more than one filter per path is when distinct shared variables are aliased and at most one of them is in consumer mode (i.e., the filter is thus required to prevent the consumer access to bind the variable). This scheme is an implementation of the optimal algorithm described in [40]. The more general problem of handling proper semantics in presence of bindings between dependent variables has been proved to be not solvable in constant-time [40]. The setup of a parallel call and the creation of the filters can also be done in constant-time (cost is always bounded by the number of dependent variables detected by the compiler in that parallel call). We are also working under the assumption that the compiler marks goals for DAP execution conservatively, i.e., during execution if a shared variable `X` is bound to a structure containing an unbound

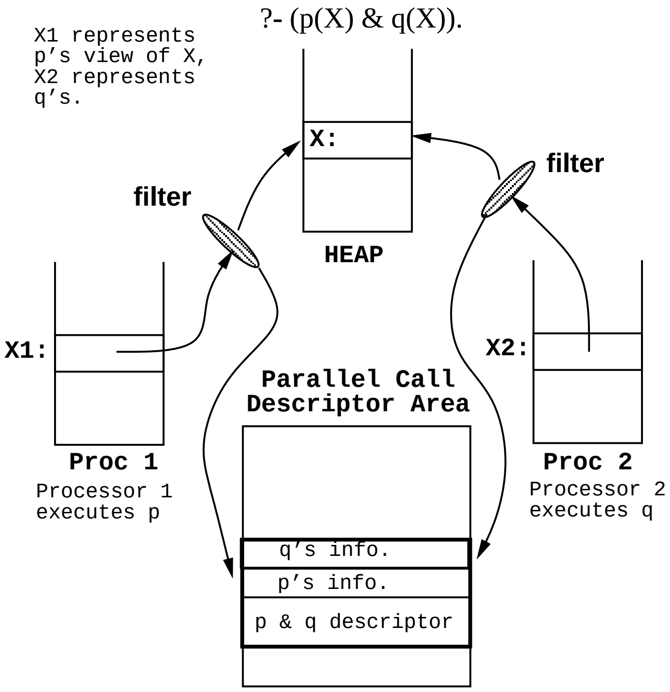


Figure 3: The Filtered Binding Model

variable `Y` before the parallel conjunction corresponding to `X` is reached then both `X` and `Y` are marked as shared. Otherwise, for correctness, the structure `X` is bound to will have to be traversed to find all unbound variables occurring in it and mark them as shared. ACE includes a lazy approach to this worst-case situation. Nevertheless, all the programs we have parallelized (even benchmarks not reported in this work) could be completely handled a-priori, without run-time structural traversal. It is also important to point out that any scheme based on a variable level annotation, such as DDAS [44] and GHC [47], will encounter similar problems; the Filtered Binding Model suggests more efficient schemes to handle this sort of situations.

The fact that producer and consumer detection is realized at the level of the individual variables allows to maintain different producers for different variables within the same CGE. The compiler identifies the initial producer for each shared variable, as the leftmost goal accessing the shared variable and potentially modifying it (this information is detected using freeness information at compile-time). During the execution of the CGE the producer role moves to different subgoals upon termination of each producer goal.

To our knowledge, the Filtered Binding Model is the first model to satisfy this ideal theoretical properties—all existing models fail to maintain the constant time behaviour [40]. More details on the Filtered Model and on its properties can be found in [35, 40]. The interested reader is referred to [35, 38] for additional details. An additional step is required when a subgoal terminates: if it is a producer goal, then on termination it should transfer the producer status to the next active subgoal in the parallel call by changing its filter. This is also a constant-time operation as the next goal to the right can be found by looking at the descriptor of the parallel call. In ACE a more intelligent transfer of control is realized by allowing the compiler to encode the dependence graph in the code for tranfering producer role. The scheme resembles a finer grain implementation of the methodology used for independent and-parallelism in [**?**].

The same model can be extended (by adding different status information per subgoal) to achieve a larger amount of parallelism, by allowing multiple producers per parallel call. This generalization is still under investigation.

Thus, the Filtered Binding model is a model that exploits restricted DAP and *performs all operations in constant-time*. The restriction is that unbound shared variables are not allowed to be bound to each other (unless the goal doing the aliasing is a producer for both—a very unlikely situation, as also discussed in [45]). The constant time nature of the model gets disrupted if binding between unbound shared variables (one in producer access and one in consumer access) are allowed. A more detailed discussion of these pathological cases is beyond the scope of this work and can be found elsewhere [35]. The Filtered Binding model can be implemented in various ways. The current implementation represents each filter as a word in the subgoal descriptor, and paths as a pair of words, one pointing to the actual variable and one pointing to the filter. Local paths related to shared variables introduced in the same parallel call share the same filter. Consumer accesses suspend in presence of unbound variables. Variable suspensions have been implemented using the traditional *suspension lists* [6].

The current implementation supports both busy-waiting and goal suspension (e.g., release of suspended computation). During consumer access, if an unbound variable is reached, the processor busy-waits for a small fixed amount of time, and if no bindings have been produced in that time it releases the current subgoal (to a queue of suspended computations) and reenters scheduling—this operation is called *relinquishing suspension*. Though quite naive, this approach has produced good results. For small granularity computations (e.g., *Primes*) the short busy waiting allows the processor to receive the binding without paying the overhead of going through a sequence of suspension/awakening. For computations with a large number of goals and a considerable delay between starting of the consumer execution and completion of the corresponding producer (e.g., *Nrev*), the short busy waiting does not seriously affect the efficiency of exploitation of parallelism.

The current implementation of DAP using the Filtered Binding Model has been shown to be competitive, and in most cases superior, with the existing implementations of DAP [35].

The Filtered Binding Model is a relatively intuitive model. Its nature shares some similarities with the idea of *attributed variables* [23, 24]; Hermenegildo et al. also proposed the use of attributed variables to support some forms of dependent and-parallelism [20]. Comparisons between the Filtered Binding Model and the various existing schemes used to support dependent and-parallelism can be found in [35].

## 4.2 Implementing Extended DDAP

The EDDAP scheme has been implemented in the current version of the ACE system. The implementation of the dynamic DAP mechanisms, based on the previously described Filtered Binding Model, has already been described elsewhere [35]—and briefly presented in an earlier section. In this section we will focus exclusively on the implementation of our extension to DDAP.

### 4.2.1 Determinacy Detection

First and foremost, to support EDDAP we need a mechanism to detect if a consumer goal is deterministic. This mechanism will be invoked when a consumer goal attempts to bind an unbound dependent variable. Detection of determinacy at this stage will lead to overriding the suspension associated to the consumer status and allow immediate execution of the binding.

Three approaches are possible to detect determinacy:

1. run-time detection: automatically performed during program execution, specifically whenever a consumer access to an unbound shared variable is performed;

2. compile-time annotations: the ACE compiler [38], derived from the CIAO compiler [19, 18], is capable of detecting determinacy of certain classes of goals and of producing suitable code annotations to communicate this information to the run-time engine;

3. manual annotations: the same annotations generated by the compiler can be manually introduced by the programmer.

In the current system we are mostly relying on run-time detection—our experience shows that the run-time cost for determinacy detection in most cases is negligible.

The run-time determinacy test is performed by verifying that no choice points have been generated between the current point of execution—i.e., the point of execution where a consumer is attempting a binding of a dependent variable—and the parallel call where this dependent variable was introduced. This is illustrated in figure 5—the topmost parallel call introduces a new dependent variable, `X`, and a consumer goal attempts a binding of `X` later in the computation (under a nested parallel call). The dotted line denotes the part of the computation that has to be explored for determinacy purposes. We term this region, which needs to be checked for absence of choice-points, the *scope* of the binding. Note that if the point where a binding is being attempted is nested inside dependent and-parallel calls then the scope will include goals to the left as well in each of the parallel conjunction—as depicted in Figure 5.

Note that the consumer goal needs to be deterministic only up to the point where the binding is attempted, it may be non-deterministic once the current point in execution is crossed. If the consumer goal turns out to be deterministic up to the point of binding, then this binding will be immediately performed.

This is an approximation of the actual scope. In fact, eventual choice points appearing in the scope may not affect the binding that the consumer is creating (i.e., all the alternatives are producing the same binding for the variable). Similarly, choice points present in the scope may be "pruned" by successive operation, either via cut operations or by bindings generated by other consumers (see later). Nevertheless, maintaining more precise information would require static knowledge about the bindings created by each alternative of a choice point and/or knowledge about execution of pruning operations.

```
BOOLEAN DeterminacyCheck(Variable X, Node m)
begin
NODE n;

    n = Parent(m);
    if (Type(n) = ParCall and
            X ∈ Vars(n)) then
        return true;
    else
    if (Type(n) = ChoicePoint) then
        if (Alts(n) > 0) then
            return false;
        else return DeterminacyCheck(X,n);
    else
        begin
            BOOLEAN temp;
            INTEGER ind,i;
            Node ch;

            ind := Index(m);
            temp := true;
            for i:= ind-1 downto 1 do
                ch := Bottom(Child(n,i));
                temp := temp and
                    DeterminacyCheck(X,ch);
            endfor;
            if (temp = false) then
                return false;
            else return DeterminacyCheck(X,n);
        end;
end
```

Figure 4: Simple Determinacy Check

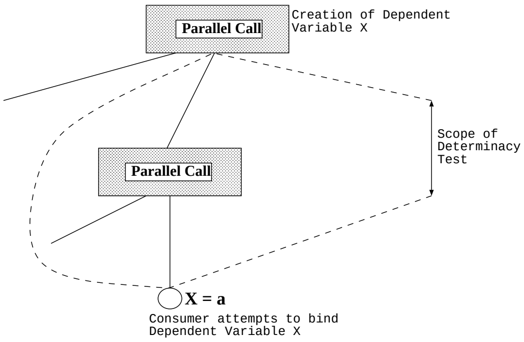


Figure 5: Determinacy Test

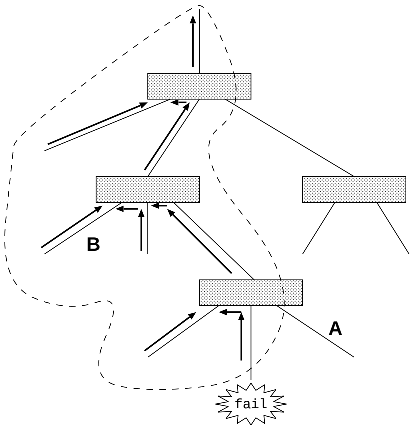


Figure 6: Backtracking in And-parallel Systems

An intuitive view of the determinacy check is shown in Figure 4. The initial call to `DeterminacyCheck` will receive as arguments the variable the consumer is attempting to bind and a node indicating the current position of the consumer in the computation tree. The algorithm visits the tree, starting from the current position of the consumer and stops only when either a choice point (with alternatives) is found or when it reaches the parallel call where the dependent variable was initially introduced. Every parallel call encountered during this traversal is checked to verify that no choice points are present in the branches to the left of the current one.

The current implementation of the determinacy check in ACE is slightly more efficient, and takes advantage of simple flags that can be stored in the parallel call to indicate the deterministic/non-deterministic nature of the subgoal. In particular, the determinacy check proceeds by:

1. determining the parallel call $\tau$ representing the highest point in the determinacy scope. This information is readily available through the filter stored with each dependent variable. Such filter, implicitly, points to the parallel call descriptor. It must be noted that the same information will also be immediately available in the DASWAM model for DAP.

2. verifying the absence of choice-points. This is realized by keeping a flag associated to each subgoal of a parallel call. This flag is set as soon as a

choice point is created during the execution of such subgoal. The flag is easily updated when the computation backtracks over the initial choice point in the subgoal. Propagation of the non-determinacy information to outermost parallel calls is performed upon subgoal termination (see also point 4. below).

3. a scan of the parallel calls between the current execution point and $\tau$ is performed, and the various flags verified. As soon as a choice point is detected (i.e., one of the flags is set) the search is terminated (with unsuccessful result). If $\tau$ is reached without finding any flag set, then the computation is declared determinate.

4. observe that the determinacy should hold also on all the branches on the left of the current one up to the parallel call $\tau$ (excluded). This also means that the process may require temporary suspensions whenever some computations on the left are still active and determinate.

Step 4. may imply occasional delays in the propagation of the non-determinacy information. Additional delays may originate from the imprecision of the flags associated to the subgoals. In fact, a goal marked as non-deterministic may indeed have become deterministic due to the effect of successive ahead-of-time bindings performed by other consumers acting on some other branches of the computation tree. Nevertheless, in all practical applications we have not observed any severe consequence of these side effects, and this last scheme has largely outperformed the more naive search of Figure 4.
The main advantage of the Filtered Binding Model over most of the other models proposed is that it allows a very fast implementation of the above mentioned check[3].

Additionally, the ACE analyzer [38] is capable of discovering the determinacy of certain subgoals by using the sharing and freeness information [38]. Our compiler annotates this information in the program. During execution, the filter for such a consumer goal is set to a permanently deterministic state, so that any binding created within this consumer goal will be immediately recorded in the dependent variable; the consumer goal will not have to suspend at all. Furthermore, knowledge about sharing between variables [31]

[3]To our knowledge, DASWAM [44] is the only other model that will allow a similar operation with similar performance. Nevertheless, even DASWAM does not meet the theoretical ideal constant-time behaviour.

allows very often to avoid suspending during the determinacy test (due to the presence of incomplete executions on the left).

#### 4.2.2 Backtracking

A very important issue in the implementation of the EDDAP model is backtracking. We show that EDDAP, in presence of a failure, does not require any special mechanism in addition to what is already needed to backtrack in normal DAP executions. The design of the backtracking mechanisms in EDDAP has been realized with the goal of producing the same backtracking semantics as in standard DAP which, in turn, is designed to explore alternatives in the same order as in a sequential Prolog execution. This essentially means that backtracking will explore choice points in the same order as in a sequential execution.

Backtracking in a IAP and DAP system has been analyzed in detail in the literature [22, 36, 45]. Backtracking denotes the series of steps that are performed following the failure of a test (e.g., an arithmetic test) or a unification failure. Since an and-parallel system explores only one alternative per choice point at a time, backward execution involves backtracking and searching for new alternatives in previous choice points. In the presence of parallel conjunctions, sequential backtracking must be modified in order to deal with computations which are spread across processors. As long as backtracking occurs over the sequential part of the computation plain Prolog-like backtracking can be used. However, the situation is more involved when backtracking involves a CGE, because more than one goal may be executing in parallel, one or more of which may encounter failure and backtrack at the same time. Thus, unlike in a sequential system, there is no unique backtracking point. Furthermore, the choice-point which offers a cure to a failure may lie within a memory segment belonging to a processor different from the one that encountered the failure, creating dangerous situations where different agents may be operating on the same stack segments. The communication among the processors involved in such backtracking process is performed using "signals", which can be viewed as specialized messages carrying the information needed to perform backtracking correctly.

One such backtracking semantics has been presented in [22] and represents the most natural generalization of standard Prolog backtracking (also similar to what described by Conery [5]). Consider the subgoals shown below, where, as usual, "," is used between sequential subgoals (because of data-dependencies) and “&” for parallel subgoals:

`a, b, (c & d & e), g, h`

Assuming that all subgoals can unify with more than one rule, there are several possible cases depending upon which subgoal fails: If subgoal `a` or `b` fails, sequential backtracking occurs, as usual. Since `c`, `d`, and `e` are mutually independent, if either one of them fails, backtracking must proceed to `b`, because `c`, `d` and `e` do not affect each other's search space—but see further below. If `g` fails, backtracking must proceed to the right-most choice point within the parallel subgoals `c & d & e`, and re-compute all goals to the right of this choice point. If `e` were the rightmost choice point and `e` should subsequently fail, backtracking would proceed to `d`, and, if necessary, to `c`. Thus, backtracking within a set of and-parallel subgoals occurs only if initiated by a failure from outside these goals, i.e., “from the right” (also known as *outside backtracking*—since it is initiated when the parallel call has already reached outside status). When backtracking is initiated from outside, once a choice point is found in a subgoal $g$, an untried alternative is picked from it and then all the subgoals to the right of $g$ in the parallel conjunction are restarted.

The backtracking activity has to “move” in the correct direction. This means that the different “segments” of the computation need to be properly *linked*, in order to allow backtracking to flow from one segment to the *logically* preceding one—i.e., the one immediately above or the one immediately to the left. Thus, the logical structure that the backtracking algorithm has to see is the one depicted in Figure 6.

Backtracking is from right to left in the parallel call similar to backtracking in sequential execution. The only difference is that a goal to be backtracked over may have been executed by a remote processor if another processor stole the goal. ACE [36] allows each processor to freely move in the segments created by other processor, thus avoiding the need of message passing and complex synchronizations between processors—the amount of synchronization required to guarantee safety when multiple processors backtrack on the same goal is significantly simpler than other approaches proposed [22]. If a new solution is found during backtracking, the goals to the right of this goal in the parallel conjunction have to be re-executed. If outside backtracking fails to produce additional answers, the goal before the CGE will be backtracked over as in normal sequential execution.

In a traditional DAP system, as long as bindings for the dependent variables are produced exactly in the same order as in a sequential execution, backtracking follows the same pattern. When a failure occurs, backtracking is activated in order to explore the computation performed so far in search of unexplored alternatives, which may "cure" the failure. With respect to the computation tree developed by a DAP execution, as illustrated in Figure 6, the choice point explored during backtracking lies in the part of the tree which is above and to the left of the point where the failure as occurred. This is the case since the part of the computation tree above and to the left represents the computation which "logically" precedes the point of failure with respect to the sequential semantics of Prolog. The only added complexity in DAP arises due to the need to kill the computations in the branches to the right of the point of failure which may have consumed bindings produced by the failed computation. In Figure 6 the computation **A** consumed a binding produced by the computation **B**; if backtracking moves over **B**, then also **A** has to be removed.

The situation may appear more complex in presence of EDDAP. The arbitrary intermixing of bindings produced by different subgoals may lead to a very different search space compared to DDAP execution. In particular, a binding made by a consumer ahead of time may prune alternatives from choice points of the producer. Thus, if the producer fails, it will not be able to see certain choice points/alternatives that were available during a sequential execution. We show that these "missed" alternatives would lead to failure eventually even during sequential execution. Let us analyze the possible cases:

- Suppose the determinate consumer produces a "correct" binding, i.e., the same binding that would have been produced during sequential execution; in this case early binding of the variable by the consumer will not produce complete failure of the computation on its left. The only effect it may have is to prune alternatives in the producer that are incompatible, thus, reducing the search space of the producer. If the producer fails, then the pruned alternatives would not have helped in any way, since they produce different values for the dependent variables (which do not correspond with the value that produces a success).

- Suppose the consumer produces a binding ahead of time for a dependent variable $Y$. Suppose this binding is inconsistent with the binding produced for $Y$ by the rest of the computation. If the consumer is completely determinate (i.e., its determinacy is independent of the bindings created within the parallel call), then it will attempt to produce such binding for $Y$ no matter when it is executed. Thus the parallel call does not have any solution (since the consumer does not have any alternative value for $Y$). This means that any pruning of alternatives performed due to early binding by the consumer will not affect the final outcome of the parallel call. This is illustrated in figure 7(i).

Suppose now the consumer becomes determinate because of some bindings previously performed by other subgoals in the parallel call. Let us consider the simple case in which binding a variable $X$ made the consumer deterministic, as shown in figure 7(ii).

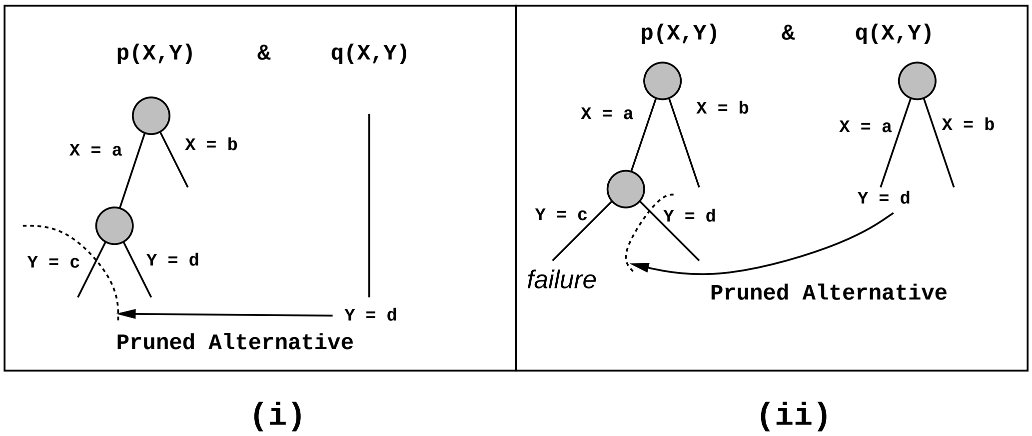


Figure 7: Possible situations during backtracking

The consumer `q` becomes deterministic due to the binding for `X` produced by `p`. In turn, `q` produces a binding for `Y` which prunes some alternatives in the producer's computation. Upon failure the producer (`p`) will not be able to see the pruned alternatives. If the branch selected by the consumer is the correct one (i.e. it leads to a successful solution in the sequential execution), then the producer (`p`) must be able to find a cure to its failure in some other alternative (the one labeled `Y = d`).

If the branch used by the consumer is incorrect, then this means that the binding which made such branch determinate (in the example the binding for `X`) is incorrect. Since such binding was produced before the consumer accessed it (i.e., the consumer is passive w.r.t. such binding), then we are guaranteed that the consumer has not pruned any alternative value for such binding. In the example, the producer will backtrack immediately to a new value for `X`, which will cause the consumer to take a different branch.

The same considerations apply for more complex combinations of producers and consumers of different variables.

The observations above lead to two important conclusions:

1. backtracking in presence of EDDAP does not require extra mechanisms than those required for the implementation of backtracking in presence of dynamic dependent and-parallelism;

2. the execution will not lead to any violations of the *no-slowdown* requirement, i.e., ignoring the cost of overheads, the parallel computation will not take more time than the sequential one.

The early deterministic binding of a dependent variable by a consumer goal allows to remove a number of choice points. Whenever the consumer produces a binding ahead of time, all the alternatives/choice-points that are pruned in the search space of the producer are actually irrelevant to the computation at hand. The search space may become smaller than that explored in the sequential execution.

As consequence of the above considerations, the implementation of backtracking in presence of EDDAP is relatively straightforward, given an existing implementation of non-deterministic DAP, as in the ACE system. The only minor change that is required is to guarantee the proper management of the non-deterministic flags which are associated to the subgoal descriptors in the parallel calls, as described in the previous section. This is realized by storing a reference to the flag in the trail upon creation of the choice-point—the entry in the trail is going to be untrailed only when backtracking requests the last alternative of the choice point.

Detection of the computations affected by a certain binding (in order to remove it whenever the binding is undone) can be made more precise and efficient through the use of static analysis, by identifying at compile-time the "dependencies" that exist between the different dependent variables.

# 5 Side-effects

While executing two goals in dependent and-parallel, one of our primary concerns is to preserve Prolog semantics. Ideally, we would like that (i) the search space explored during parallel execution is identical (or smaller) to that during sequential execution; and, (ii) the sequence in which order-sensitive predicates (meta-logical, side-effect, and extra-logical predicates) are executed is the same as in sequential execution. During parallel execution, (i) is not possible, as whenever and-parallel execution is done there is some amount of speculativeness involved because the goal to the left in a parallel conjunction might fail causing the work invested in other goals to the right to go waste. However, the blow-up of search space should not be excessive (as is the case with unrestricted execution of two dependent goals). On the other hand, if parallel execution can help us in narrowing down the search space, compared to sequential execution (as done by the Andorra Principle for instance), then that is fine. The narrowing of search space however inevitably leads to condition (ii) above being violated, because if the search space that is pruned happens to contain extra-logical predicates, then these will not be executed during parallel execution.

The approach we are taking is to enforce Prolog semantics during the parallel execution, even in presence of side-effects. Currently the system is relying on code annotations to detect the possible "impurity" of subgoals, i.e., the presence of side-effects in their computations. The non-determinism checks used by the consumer to support EDDAP have been extended to include additional checks on the purity of the computations that may get pruned—thus avoiding ahead-of-time bindings if they may endanger impure computations.

Techniques to achieve this behaviour have been studied by various researchers [12, 30, 42, 17, 45, 3] and ACE incorporates them with little changes. Considerable work is still needed to optimized the execution of side-effects during parallel computations in order to avoid excessive overhead and/or loss of parallelism.

We would also like to point out that both the annotations for purity as well as the direction of the filters on the dependent variables can be explicitly programmed by the user, thus allowing the user to modify the semantics of the computations, e.g., allowing pruning of impure computations as well.

# 6 Reactive Computations in EDDAP

The Extended Dynamic DAP model can also be used for programming reactive computations. Both unidirectional (1-way) and bidirectional (2-way) communication can be obtained. However, bidirectional communication is limited the following cases:

- the consumer goals, i.e., goals lying in the branches "on the right", are deterministic, so that EDDAP is triggered;
- the programmer explicitly annotates the dependent variables to specify the direction of the bindings, thus explicitly forcing deviations from Prolog semantics.

In the rest of this section we will focus on the first of these two cases.

Recall that dynamic DAP itself allows for one way communication between subgoals. Subgoals to the left can communicate bindings to subgoals to the right. This can be illustrated with a simple example, taken from the Strand book [9], in which process `p` generates *ferraris*, while process `q` consumes (rides) them.

```
produce(0, []).
produce(N, [ferrari | Ms])  :-
                               N > 0,
                               N1 is N - 1,
                               produce(N1, Ms).
consume([]).
consume([ferrari | Ms])     :-
                               go_ride_ferrari,
                               consume(Ms).
```

When the goal

```
?-produce(10, X), consume(X).
```

is executed, as soon as `X` is partially bound to `[ferrari | Ms]` by the `produce` goal, the `consume` goal begins execution. Thus, while `produce` is busy generating more elements of stream `X`, `consume` can start processing the elements already generated, thereby giving rise to dependent and-parallelism. The `consume` goal will suspend if it attempts to access the binding of `Ms` and `Ms` is unbound.

Because programs with 1-way communication can be expressed in dynamic DAP, they can obviously be expressed in extended dynamic DAP.

However, 2-way communication in which a goal to the left communicates to goals to the right and vice versa is not possible in dynamic DAP, or Prolog. In fact, 2-way communication goes against our philosophy of maintaining Prolog semantics. However, if we decided not to care about Prolog semantics, it is possible to simulate 2 way communication (coroutining) in EDDAP, i.e., bindings can be communicated not only from left to right but also from right to left, given that the consumer goal on the right is deterministic. Consider an extension of the example above, where the consumer process should pay the price of the ferrari (by instantiating a variable to a specified constant) before riding it. The producer process reads the instantiated variable, and processes it further (spends it).

```
produce(0, []).
produce(N, [ferrari(Envelope)|Ms]) :-
             N > 0,
             N1 is N - 1,
             spend(Envelope),
             produce(N1, Ms).
consume([]).
consume([ferrari(Envelope) | Ms]) :-
             Envelope = big_bucks,
             go_ride_ferrari,
             consume(Ms).
spend(big_bucks).
spend(small_bucks).
```

This example is very similar to the previous example, except that the query

```
?- produce(10, X), consume(X).
```

now each element in the stream `X` is a structure that contains an unbound variable called `Envelope`. Assuming sufficient resources are available, the `Envelope` variable is bound by the `consume` goal, this value is then processed by `produce` in the goal `spend`. In traditional Prolog as well as dynamic DAP, `spend` will create a choice-point since the argument of `spend` gets bound only in the goal to the right of the `produce` goal in the top-level query. Extended dynamic DAP will, however, execute `produce` and `consume` concurrently, feeding the binding of `Envelope` from `consume` goal into `produce` goal in the backward direction, thus permitting a bidirectional communication of bindings. The example above can be easily coded in EDDAP by using separate unidirectional streams, with opposite communication directions. Observe that the switching from one subgoal to the other will *actually* take place, since workers associated to a suspended computation can task switch to any other active subgoal. This will not lead to a complete fairness but will allow to tackle most of the desired situations (such as the one in the previous example). Extended Dynamic DAP can thus constrain the search space of both the producer and the consumer goals. In the case of lack of determinacy on the consumer side, e.g., the `consume` subgoal is not deterministic, then coroutining will turn into sequential execution; the `produce` goal will "guess" the value of `Envelope` and eventually fail and backtrack in case of discordance with the `consume` goal. As mentioned at the beginning, user annotations are available in the ACE system to override Prolog semantics and explicitly program 2-way communication and reactive behavior; the relevant classes of annotations are:

- *dependency graphs:* ACE allows to describe the dependency graph relative to the dependent variables present in a parallel call, thus allowing the user to create arbitrary ordering between the goals w.r.t. bindings dependent variables

- *guards:* ACE allows flat guards and non-deterministic selection of guarded clauses [43]

Extensions of the execution mechanisms to better handle 2-way communication are still under investigation and will rely on existing technology [32, 43, 4].

## 6.1 Further Improvements to EDDAP

EDDAP scheme can be improved further. The availability of compile-time (or user supplied) information can considerably cut the overhead present in the scheme. For example, freeness and sharing information can be combined to develop dependence graphs for the shared variables. These dependence graphs can then be analyzed at compile-time to reassign producer/consumer roles to better fit the EDDAP model. For example, it would be preferable to allow a given subgoal to be a producer for a variable if that will allow many other subgoals to become deterministic. A dependence graph which emphasizes the fact that a certain variable can be bound determinately whenever a certain set of variables have been bound, can be employed to simplify the code (the filter for such variable can be omitted).

Constraint Solving over Finite Domains [49] can also be incorporated to further prune the search space [10]. This is an area of current investigation.

## 6.2 Generality of the Filtered Model

The Filtered Binding Model plays a fundamental role in the implementation of the EDDAP scheme. The Filtered Binding Model efficiently implements a very general and flexible scheme to manage dependent variables. In particular, the concept of "filter" can be easily modified to suit different needs. Virtually all the schemes that have been developed in the literature for implementing DAP in Prolog are tightly connected to the Prolog execution semantics. In particular, the management of the dependent variables is closely related to the enforcement of the left-to-right order of bindings typical to Prolog.

The Filtered Binding Model can not only be efficiently used for implementing this left to right order of bindings, it can be used for implementing any arbitrary order with very minor changes. The reason for this flexibility of the filtered binding scheme is that it encodes the producer and consumer status of different subgoals in a very efficient way. The producer and consumer status can be found out by checking the corresponding filter in constant time. In other systems such as DASWAM, this information is implicit in the execution tree and is quite expensive to compute. Thus, our system can be easily modified to obtain an implementation of a simple committed choice language (Parlog, GHC, etc.). The only extra mechanism that is absent from our implementation and that is needed to implement committed choice languages is the *commit* operator. Work is under way to include an implementation of the commit operator in our system, so that committed-choice programs can also be executed. Although the Filtered Binding Model is not expected to be competitive in terms of sequential efficiency with the specialized implementation techniques used by modern committed-choice languages, e.g., [11], the ability to support committed-choice semantics with minimal effort is extremely valuable. Furthermore, we expect the parallel efficiency of ACE with the Filtered Binding Model to easily compete with the efficiency of parallel implementations of committed-choice languages.

In the previous section we have considered the issue of 1-way vs. 2-way communications. The EDDAP implements a limited amount of 2-way communications between parallel subgoals—the most that can be done without affecting Prolog semantics. On the other hand we may want to implement a different kind of semantics. For example, we may be interested in a more complete 2-way communication scheme like that supplied by committed-choice languages [43]. The filtered binding model, as it is, supplies the complete support needed for managing this type of communication (producers/consumers are known a-priori and the filters are fixed). With the addition of a simple order-insensitive version of certain meta-logical predicates (e.g., `var`) we can easily achieve an efficient implementation of languages like *Parlog* and *Concurrent Prolog*.

Our conjecture is that the generality and flexibility of the Filtered Binding Model allows it to be used to implement virtually any execution strategies on the computation tree.

# 7 Performance Results

As mentioned earlier, the Filtered Binding model for implementing Dynamic DAP has been recently incorporated in the ACE system [14], an and/or parallel implementation of Prolog developed at New Mexico State University. This implementation was then further extended to incorporate Extended Dynamic DAP. The current version of the ACE system runs on both Sequent Symmetry, Sun Sparc and Pentium-based multiprocessors. The performance figures presented in this paper are from the Sequent Symmetry implementation—the largest shared memory machine available in our laboratory. Speedups and overheads that are relatively similar to those presented here have been achieved on Sun Sparc and Pentium-based Multiprocessors. Results on these architectures are not reported here because of the small size of the machines available to us. All the figures presented have been obtained as average over 10 runs on a dedicated machine—all the execution times observed were highly stable and presented negligible variations from the average.

We first present some performance figures for Dynamic DAP, followed by figures for Extended Dynamic DAP.

## 7.1 Dependent And-Parallelism

The current implementation is still prototypical: it has been realized to test the feasibility of the implementation model. No optimizations are present in

| **Goals** | **ACE agents** | | | | |
|---|---|---|---|---|---|
| *Query* | 1 | 2 | 4 | 6 | 8 |
| *Genetic* | 17779 | 9419 (1.89) | 4810 (3.70) | 3269 (5.44) | 2540 (7.0) |
| *Dynam* | 58939 | 29469 (2.0) | 14730 (4.00) | 9989 (5.9) | 7429 (7.93) |
| *Peephole* | 2949 | 1620 (1.82) | 1299 (2.27) | 870 (3.39) | 809 (3.65) |
| *TSP* | 23059 | 11519 (2.0) | 5770 (4.0) | 3850 (5.99) | 3009 (7.66) |
| *Pqsort* | 43579 | 21830 (2.0) | 12500 (3.5) | 8939 (4.88) | 6939 (6.28) |
| *ClS* | 11750 | 5889 (2.00) | 2959 (3.97) | 2019 (5.82) | 1549 (7.59) |
| *Nrev* | 8120 | 4580 (1.77) | 2659 (3.05) | 2020 (4.02) | 1674 (4.85) |

Table 1: Execution times in msec (Speedups are shown in parenthesis)

the current implementation (and many optimizations [39] can be applied). Implementation of garbage collection and various other optimizations is under way. Table 1 presents the execution times and speedups obtained on various benchmarks for the Sequent (more results have been presented elsewhere [35]. The speedups presented are obtained as ratio between execution time on a single processor and execution time on $n$ processors. Execution time on a single processor for the benchmarks considered has been observed to be from 4% to 20% slower than the execution time on the SICStus Prolog engine used as a base for the ACE implementation (SICStus Prolog 0.7). Considering that the implementation is largely unoptimized, these overhead figures are very encouraging.

Table 2 presents, for comparison purposes, the execution times achieved for the *Genetic* and *Dynam* benchmarks on Sun Sparc Multiprocessors. As it can be observed, the speedups are comparable to those observed on Sequent machines. Speedups on Pentium multiprocessors have been observed to be occasionally better than those on Sun Sparc machines, possibly due to the larger cache size and the good locality of execution.

| **Goals** | **ACE agents** | | |
|---|---|---|---|
| *Query* | 1 | 2 | 4 |
| *Genetic* | 3066 | 1630 (1.88) | 833 (3.68) |
| *Dynam* | 10162 | 5106 (1.99) | 2553 (3.98) |

Table 2: Execution times in msec on Sparc

The benchmarks have been mostly chosen from the benchmark suite of the

Aquarius Prolog system. In many benchmarks the amount of DAP present is limited. The speed-ups for such benchmarks are also reported in table 1. Note that some benchmarks are programs that are quite big in size. The results obtained are quite satisfactory. Good speedups have been achieved on many of the benchmarks used (several benchmarks have produced a speedup $\geq 7$ on 8 processors), which proves that DAP can be fruitfully exploited and is a non-trivial source of parallelism. Analogous results have been achieved on Sparc Multiprocessors.

Furthermore, the results have been particularly encouraging in terms of overhead. Given that our implementation is only a preliminary prototype, the parallel overhead observed ranged from a minimum of 4% (on the *TSP* problem) to a maximum of 20% (on *Pqsort*). Higher overheads arise due to more frequent occurrence of suspensions. This suggests that the mechanisms for suspending a computation and releasing the processor needs some further tuning. Nevertheless, *Nrev* is a very good indicator of our system's efficiency, that it can produce good speed-ups with low overhead even when the granularity of tasks is small.

## 7.2 Extended Dynamic DAP

Our implementation of dynamic DAP demonstrates that DAP can be fruitfully exploited and that it is present in diverse applications. Nevertheless, there are certain kinds of applications that cannot take any advantage of DAP. The key for a successful dependent and-parallel execution is the presence of an adequate amount of overlapping between producer and consumers [38]. Programs which are based on certain programming styles, such as *generate and test*, typically cannot be effectively parallelized—since the consumer and producer have virtually no overlapping. The use of the EDDAP allows to overcome part of this problem. Avoidance of suspensions forces the consumer to overlap with the producer. Table 3 shows performance results for a set of benchmark that benefit from extended dynamic DAP.

We have tested our prototype implementation of EDDAP on a collection of benchmarks. Most of the benchmarks belong to the Andorra-I benchmark suite. The only exceptions are Metallel, a large NLP application used to automatically detect metonymies in a text corpus, and Graduation, an expert system for checking University graduation requirements.

In all cases we have observed monotonically increasing speedup curves. The table compares the best speedups achieved in the case of simple depen-

| **Program** | **DAP Execution** | | | **Best EDDAP Execution** | | |
|---|---|---|---|---|---|---|
| | Exec. Time | Speedup | No. Procs. | Exec. Time | Speedup | No. Procs. |
| Crossword | 9700 | 1.55 | 5 | 429 | 35.05 | 15 |
| Money | 2728 | 1.10 | 9 | 649 | 4.20 | 5 |
| Map | 19200 | 1.1 | 10 | 529 | 39.92 | 5 |
| Magic | 22248 | 1.1 | 10 | 1690 | 14.63 | 10 |
| Nqueen | 1546 | 1.5 | 10 | 610 | 3.8 | 6 |
| Metallel | 6443 | 1.9 | 10 | 1549 | 7.9 | 10 |
| TreeInsert | 3180 | 1.0 | 10 | 970 | 3.3 | 10 |
| Graduation | 788 | 2.2 | 10 | 142 | 12.2 | 10 |

Table 3: Performance Results for EDDAP (exec. time in msec.)

dent and-parallel execution and in the case of EDDAP executions. Execution times using number of processors larger than those indicated in the table are negligibly different from the best case reported. This can be easily seen also from Figure 8. The executions under EDDAP have produced a performance which is constantly better than regular DAP, and in over 70% of the cases within a 10% variation from the best results reported in table 3.

A lot of the speed-up comes from pruning of search space. However, early deterministic binding of the dependent variable by the consumer goals, can also lead to more parallelism, as it may wake-up other consumer goals to the right of the deterministic consumer sooner. The source of parallelism for various benchmarks is discussed next.

In the *Crossword* program, the super-linear speedup comes almost exclusively from the excellent pruning obtained by starting more subgoals concurrently and letting deterministic bindings take place. In the case of *Money*, the speedup is a mixture of pruning and parallelism (i.e. other consumer goals being woken up sooner due to the deterministic binding); in fact, we estimated that, if only pruning was present, then the maximum speedup would have been 3.0. Thus EEDAP allowed in this case also a greater degree of parallelism to take place. In the case of *Magic* most of the speedup is obtained by avoiding suspensions; in fact, avoiding the creation of many choice-points produces a super-linear speedup. The program *Nqueen* shows a modest speedup; nevertheless, the presence of EDDAP allows for a greater degree of parallelism to take place (with a good number of alternatives getting pruned), leading to a speedup that is more than twice of that achievable using standard DAP. *Map* is another program in which the speedup is achieved

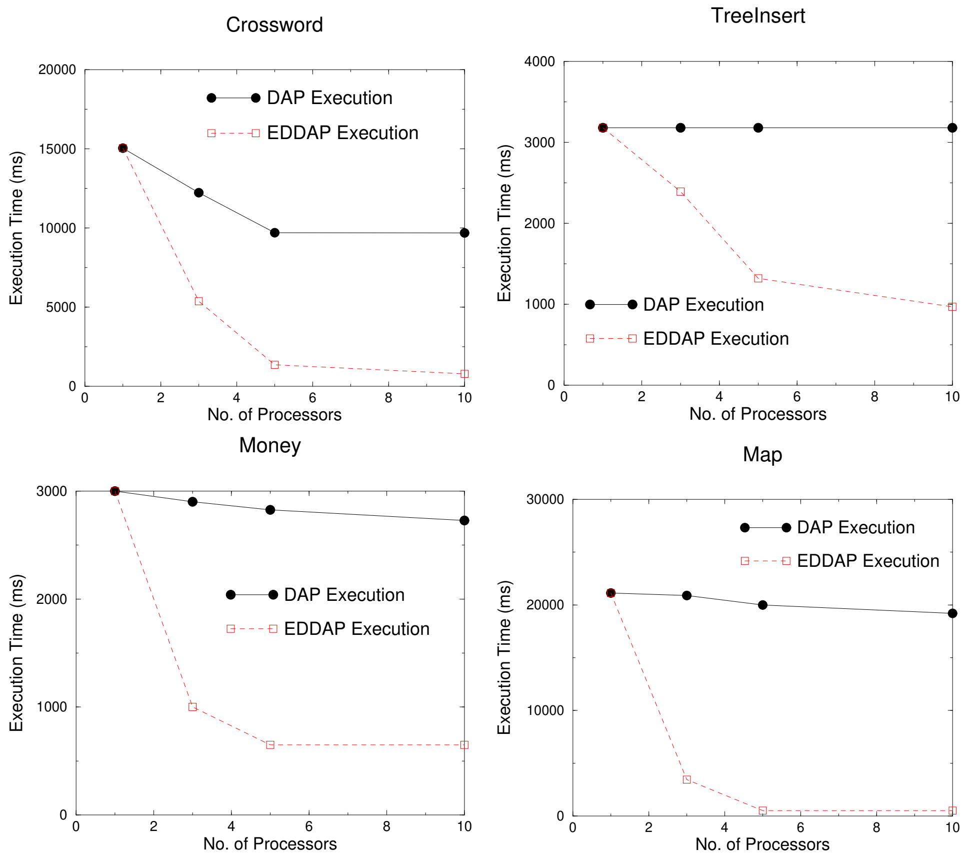


Figure 8: Improvements in Execution Speed

not by real parallelism, but essentially due to the drastic reduction in search space obtained from the ahead-of-time bindings performed by the consumers.

The benchmarks presented in table 3 are well-known and good representatives of classes of programs that will take advantage of the techniques described in this work.

Figure 8 compares the execution times for some of the benchmarks in presence of varying number of processors.

# 8 Conclusions

In this paper we presented an extension of Dynamic Dependent And-parallel scheme in which deterministic consumer goals are allowed to bind the dependent variable. The extended scheme, called the *Extended Dynamic Dependent And-Parallel* scheme, leads to considerable pruning of the search space, and thus to improved efficiency. It results in more overlapping of execution of dependent subgoals, and thus to more parallelism being exploited. In a normal dynamic dependent and-parallel execution, given a parallel conjunction with a shared variable `X`, the leftmost goal is normally designated as the producer of the binding for `X`, all others goals are designated as consumers. If the producer goal finishes execution without binding `X`, then the leftmost consumer goal becomes the next producer and so on. In the extended scheme a deterministic consumer goal is also allowed to bind the dependent variable. The general idea is to take advantage of the determinacy of the bindings to avoid suspensions and improve the overlapping between producer and consumer computations. This allows for improved speedups—due to the greater overlapping of computations—as well as for significant pruning of the search space. Our extension can be regarded as incorporating elements of the Extended Andorra Model. Thus, the extended scheme can be seen as a realization of a weaker form of the Extended Andorra Model. Extended Dynamic DAP permits both one way and two way communications between goals, allowing us to simulate coroutining and reactive computations. Thus, EDDAP can be easily extend to allow subsumption of different computation strategies (e.g., committed-choice languages). The extended scheme has been implemented in the ACE system on a Sequent Symmetry using the *Filtered Binding Model* and very encouraging results obtained. These performance results were also reported.

# Acknowledgments

Thanks are due to Eduardo Correia, Vítor Santos Costa, Kish Shen, and Manuel Hermenegildo and his CLIP group, for various discussions. We would also like to thank the anonymous referees, their comments have helped in greatly improving this work. This work has been partially supported by NSF grants CCR 96-25358, HRD 96-28450, CDA 97-29848, EIA 98-10732, and INT 95-15256.

# References


[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, 1990.

[2] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the Language and its Implementation. In Dale Miller, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 283–298. MIT Press, 1993.

[3] S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368. MIT Press, 1989.

[4] K. L. Clark and S. Gregory. Notes on the Implementation of Parlog. *Journal of Logic Programming*, 2(1), April 1985.

[5] 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.

[6] J. Crammond. The Abstract Machine and Implementation of Parallel Prolog. Research report, Dept. of Computing, Imperial College of Science and Technology, July 1990.

[7] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In David S. Warren, editor, *Proceedings of the International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.

[8] N. Drakos. Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking. In N. Sridharan, editor, *Proc. of the International Joint Conference on Artificial Intelligence*, pages 157–162. Morgan Kaufmann, 1989.

[9] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[10] S. Gregory and R. Yang. Parallel Constraint Solving in Andorra-I. *Proceedings of the Fifth Generation Computer Systems Conference*, pages 843–850, June 1992.

[11] D. Gudeman, K. De Bosschere, and S. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 399–413. MIT Press, 1992.

[12] G. Gupta and V. Santos Costa. Cuts and Side-effects in And/Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.

[13] 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.

[14] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 93–109. MIT Press, 1994.

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

[16] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In David H. Warren and Peter Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.

[17] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems*, pages 831–840. Tokyo, November 1988.

[18] M. Hermenegildo. CIAO, a Generic Parallel Concurrent Constraint Logic Programming System. In Alan Borning, editor, *Principle and Practice of Constraint Programming*, pages 123–133. Springer Verlag, 1994.

[19] M. Hermenegildo. Towards Efficient Parallel Implementation of Concurrent Constraint Logic Programming. In T. Chikayama and E. Tick,

editors, *Proceedings of ICOT/NFS Workshop on Parallel Logic Programming and its Programming Environments*. University of Oregon, 1994.

[20] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Parallel and Concurrent Logic Programming Systems. In Leon Sterling, editor, *Proceedings of the International Conference on Logic Programming*, pages 631–645. MIT Press, 1995.

[21] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In David H. Warren and P. Szeredi, editors, *Proceedings of the International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[22] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[23] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Symposium on Programming Languages Implementation and Logic Programming*, pages 260–268. Springer Verlag, 1992.

[24] S. Le Houitouze. A new data structure for implementing extensions to prolog. In *Proceedings of the Symposium on Programming Languages Implementation and Logic Programming*, pages 136–150. Springer Verlag, 1990.

[25] S. Janson. *AKL: a Multiparadigm Programming Language*. PhD thesis, Swedish Institute of Computer Science, 1994.

[26] R.S. Lopes and V.Santos Costa. The BEAM: Towards a first EAM Implementation. In I. Dutra et al., editor, *Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*. Nova Science, 1999. (to appear).

[27] E. Lusk et. al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[28] J. Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala University, 1997.

[29] J. Montelius and K. Ali. A Parallel Implementation of AKL. *New Generation Computing*, 14(1):31–52, 1996.

[30] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In G. Levi and M. Martelli, editors, *Proceedings of the International Conference on Logic Programming*, pages 80–97. MIT Press, June 1989.

[31] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In K. Furukawa, editor, *Proceedings of the International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[32] L. Naish. Parallelizing NU-Prolog. In R. Kowalski and K. Bowen, editors, *Proceedings of 5th International Conference and Symposium on Logic Programming*, pages 1565–1577. MIT Press, 1988.

[33] D. Palmer. A Parallel Implementation of the Andorra Kernel Language. Technical Report TR-97/23, University of Melbourne, September 1997.

[34] E. Pontelli and G. Gupta. On the Duality Between And-parallelism and Or-parallelism. In *Proceedings of Euro-Par'95*, pages 43–54. Springer Verlag, 1995.

[35] E. Pontelli and G. Gupta. Implementation mechanisms for dependent and-parallelism. In L. Naish, editor, *Proceedings of the International Conference on Logic Programming*. MIT Press, 1997.

[36] E. Pontelli and G. Gupta. Efficient backtracking in and-parallel implementations of non-deterministic languages. In T.H. Lai, editor, *Proceedings of the International Conference on Parallel Processing*, pages 338–347. IEEE Computer Society, 1998.

[37] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–571. IEEE Computer Society, Santa Barbara, CA, April 1995.

[38] E. Pontelli, G. Gupta, F. Pulvirenti, and A. Ferro. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In L. Naish, editor, *International Conference on Logic Programming*, pages 108–122. MIT Press, 1997.

[39] E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of Parallel Prolog Implementations. In L. Sterling, editor, *Proceedings of the International Conference on Logic Programming*, pages 615–629. MIT Press, 1995.

[40] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137. Springer Verlag, 1997.

[41] A. Roychoudhury, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. A Conservative Technique to Improve Deterministic Evaluation of Logic Programs. In *International Conference on Computer Languages*. IEEE Computer Society, 1998.

[42] 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 *Proceedings of the 3rd ACM SIGPLAN Conference on Principle and Practice of Parallel Processing*, pages 83–93. ACM Press, 1991.

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

[44] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-parallel Scheme. In Krzysztof Apt, editor, *Proceedigs of the Joint International Conference and Symposium on Logic Programming*, pages 717–731. MIT Press, 1992.

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

[46] K. Shen. Improving the Execution of the Dependent And-parallel Prolog DDAS. In *Proceedings of PARLE 94*. Springer Verlag, 1994.

[47] A. Takeuchi. *Parallel Logic Programming*. Kluwer, 1992.

[48] H. Tebra. Optimistic And-Parallelism in Prolog. In *Proceedings of PARLE 87*. Springer Verlag, 1987.

[49] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[50] D. H. D. Warren. The Andorra Principle. Presented at Gigalips workshop, 1987. Unpublished.

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