Extended Dynamic Dependent And-parallelism in ACE

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

Abstract

We present an extension of Dynamic Dependent And-parallel scheme in which deterministic consumer goals are allowed to bind the *dependent variable*. 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 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* [15] 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:

   1) Independent And-Parallelism (IAP): arises when run-time bindings for the variables in two or more goals are such that they are *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 [13]); such independent goals can be run in parallel. &-Prolog [13] and &ACE [21] are two systems that exploit IAP.

   2) 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 [27] and ACE [19] 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:

1. The two subgoals can be executed independently until one of them accesses/binds the common variable `X`. Note that it is also possible to continue executing the two goals independently in parallel (i.e., executing each without regard to the other goal) even after the common variable has been accessed; in such a case, after the two goals finish, the bindings produced by each will have to be checked for compatibility (this compatibility check at the end is called *back unification*); unrestricted execution of dependent goals is not recommended because it can lead to redundant execution [19].

2. Once the common variable is accessed by one of the goals, if it is bound to a structure (the goal generating this binding is called the *producer*), and this structure is read as an input argument of the other goal (called the *consumer*) then parallelism can be further exploited by having the consumer goal compute with one element of the structure (typically this structure is a list, or a *stream* of elements) while the producer goal is computing the next element.

The first case is very similar to independent and-parallelism. The second one is sometimes also referred to as *stream-parallelism* and is useful for speeding up producer-consumer interactions (e.g., those found in system programs).

Determining a-priori which goal is going to bind the variable first is an undecidable problem. In the description of stream-parallelism 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. 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)* [26, 3]. 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 (earlier implementations also exist such as DASWAM [26]). In this paper we 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 [28], and adds the ability to prune search space. It also incorporates coroutining into our system, allowing for two way communication. In fact, this scheme of allowing consumers to make deterministic bindings is a weak form of the Extended Andorra Model (EAM) [29] 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 implemented with the help of 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 Model, 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 5 describes how reactive computations can be realized in EDDAP, while Section 6 describes performance results of implementation of coroutining in the ACE system on a Sequent Symmetry.

2 Dependent And-parallelism

As mentioned earlier, 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. 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.

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

[1] Following [27] we term `X` the dependent variable.

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 w.r.t. sequential execution. That is, parallel execution takes an amount of time bounded by the time taken for sequential execution. This, implies that 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) bind 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 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.

2.1 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 [25]. Considerable amount of work, including by us, has been done in designing models and building systems where the current active leftmost goal containing a dependent variable is designated as its producer. All other goals are consumers. This form of DAP has been termed Dynamic DAP in the literature, 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. 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 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 [25, 2]. This is largely due to the complexity of dealing with distributed backtracking. Nevertheless, it has been shown [27] 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.

2.2 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 not be delayed. 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 will 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 following simple program:

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

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.

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 finds application in the Basic Andorra Model. 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 to the Basic Andorra Model, deterministic goals are never executed in parallel with non-deterministic goals.
The Extended Andorra Model 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 1). 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 1 (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 [29, 11, 10]. 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 [14], suffers the drawback of being based on a rather different language and semantics w.r.t. Prolog. A parallel implementation of AKL, called *Penny*, has been recently completed with good performance results [17, 16].

3.1 EDDAP: A Step Towards EAM

Dynamic DAP extended with allowing deterministic consumers to become a producer (*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 [8] 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.

The Extended Dynamic DAP has better termination properties than the Basic Andorra Model. This is because in the Basic Andorra Model, the goals are reordered (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:

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

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

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. In fact, EDDAP execution has better termination properties than

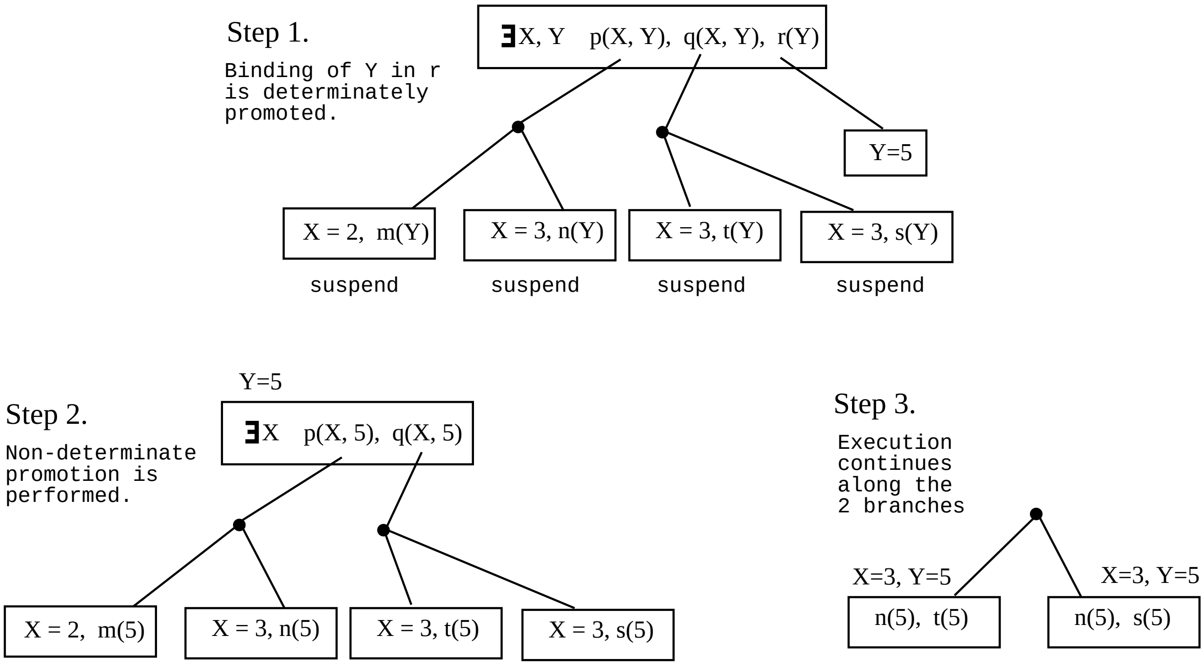

Figure 1: Execution in EAM

that of sequential Prolog execution. A program that is non-terminating under sequential execution may terminate under extended dynamic DAP.

Reordering of subgoals in the current goal-list adds considerable overhead to the implementation, as any reordering done should be backtrackable. It is our belief that it is a major source of overhead in the Andorra-I system, a system based on the Basic Andorra Model [4]. Also, deterministic goals are not allowed to execute in parallel with non-deterministic goals. 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 both deterministic goals can run in parallel with non-deterministic goals. Thus, if only 1 processor is available, then execution is exactly like Prolog, no pruning of search space takes place. The Andorra-I system may uncover more determinacy due to the re-ordering. It is our belief, however, that this re-ordering can be accomplished quite effectively at compile-time using techniques developed by Ramakrishnan et al [24]. 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 [27]—details on how to generate annotations are omitted due to lack of space [22]). 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 [20]. Implementation of or-parallelism has to deal with dependencies between parallel threads because different or-parallel threads may produce distinct bindings for the same (conditional) 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 $\exists X(p(X) \wedge q(X)) \not\equiv \exists X(p(X)) \wedge \exists X(q(X)))$. Nevertheless we do not need to go to the extreme of keeping a unique location for these bindings, as done in most of the implementations (e.g., DASWAM), 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: $\exists X(p(X) \wedge q(X)) \equiv \exists X_1 X_2 (p(X_1) \wedge q(X_2) \wedge X_1 = X_2)$ where the equality $X_1 = X_2$ is used to maintain consistency between the bindings produced by $p(.)$ and $q(.)$.

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., $p(X_1)$ and $q(X_2)$) 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

Thus, 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 [27].

Figure 2 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 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[2]). Details of the Filtered

[2]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 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.

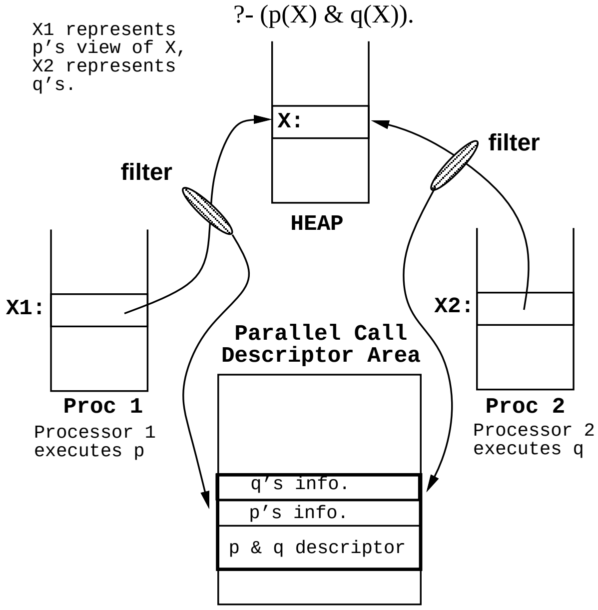

Figure 2: The Filtered Binding Model

Model are not reported here for lack of space. The interested reader is referred to [19, 22] 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.

The current implementation, implements filters 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* [5].

To the best of our knowledge, the Filtered Binding Model is the most efficient model for exploiting dynamic DAP proposed to date [19].

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 [19]—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. 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.

Determinacy is detected in our current implementation dynamically, i.e., during the execution itself. The test is performed by verifying that no choice points have been generated between the current point of execution (where the binding of the dependent variable is being attempted) and the parallel call where this dependent variable was introduced. 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. This is illustrated in figure 3.

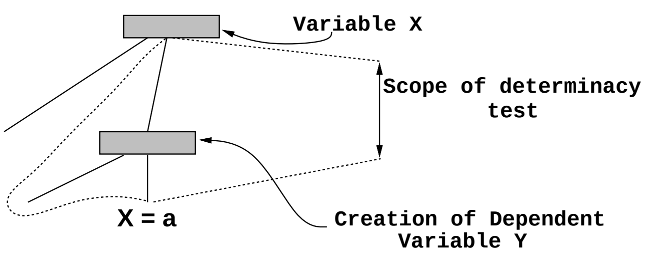

Figure 3: Determinacy Test

The determinacy is detected by:

- determining the parallel call τ representing the highest point in the determinacy scope. This information is readily available through the filter stored with each dependent variable (which, 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.
- 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.
- a scan of the parallel calls between the current execution point and τ 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 τ is reached without finding any flag set, then the computation is declared determinate.
- observe that the determinacy should hold also on all the branches on the left of the current one up to the parallel call τ (excluded). This also means that the process may require temporary suspensions whenever some computations on the left are still active and determinate.

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].

[3] To our knowledge, DASWAM [26] is the only other model that will allow a similar operation with similar performance.

Additionally, the ACE analyzer [22] is occasionally capable of discovering the determinacy of certain subgoals by using the sharing and freeness information [22]. 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 [18] 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 execution.

Backtracking occurs when a failure is encountered. Backtracking in a DAP system has been analyzed in detail in the literature [27]. 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 a rather straightforward pattern. (Nevertheless, implementing it is quite complex due to the intense interaction between different computations). When a failure occurs we know that the cure for the failure (if any) will be in a branch to the left of the point of failure. This essentially means that backtracking will explore choice points in the same order as in a sequential execution. The only added complexity arises due to the need to kill the computations on the right which may have consumed bindings produced by the failed computation.

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 4(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 4(ii). 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;
3. the early deterministic binding of a dependent variable by a consumer goal allows for a limited amount of *intelligent backtracking*. In fact, 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 on backtracking may become smaller than that explored in the sequential execution.

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 Reactive Computations in EDDAP

The Extended Dynamic DAP model can be used for programming reactive computations with ease. Both unidirectional (1-way) and bidirectional (2-way) communication can be obtained.

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 [6], 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).

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

When this goal 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. 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).

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

This example is very similar to the previous example, except that now each element in the stream `X` is a structure that

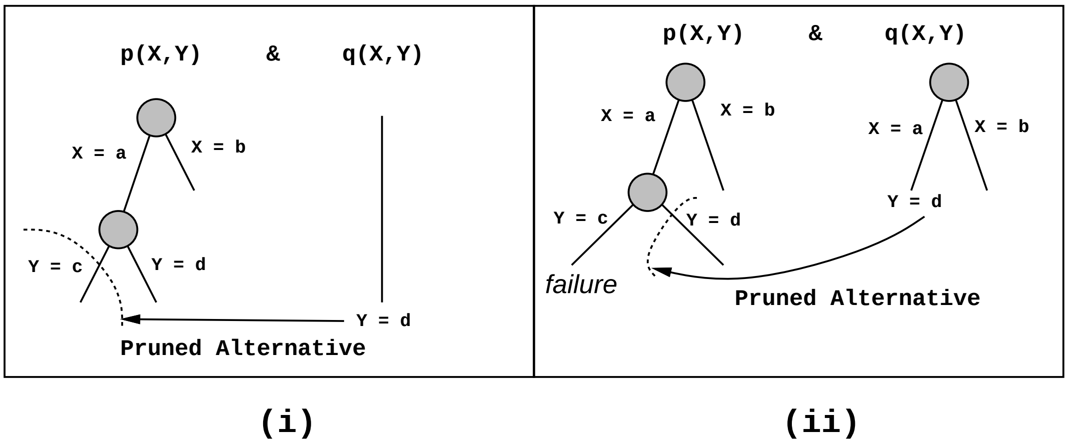

Figure 4: Possible situations during backtracking

contains an unbound variable called `Envelope`. `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 (e.g., 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 (like the one in the previous example). Extended Dynamic DAP can thus constrain the search space of both the producer and the consumer goals.

5.1 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 [12] can also be incorporated to further prune the search space [7]. This is an area of current investigation.

5.2 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 variable. 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.

6 Performance Results

As mentioned earlier, the Filtered Binding model for implementing Dynamic DAP has been recently incorporated in the ACE system [9], 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 and Sun Sparc multiprocessors. We first present some performance figures for Dynamic DAP, followed by figures for Extended Dynamic DAP.

6.1 Dependent And-Parallelism

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

Goals *Query*	&ACE agents 1	2	4	6	8
Genetic	17779	9419 (1.89)	4810 (3.70)	3269 (5.44)	2540 (7.0)
Disj	6100	3190 (1.91)	3100 (1.97)	3100 (1.97)	3100 (1.97)
Dynam	58939	29469 (2.0)	14730 (4.00)	9989 (5.9)	7429 (7.93)
Micro	14450	7579 (1.91)	4110 (3.52)	3179 (4.55)	2609 (5.54)
Peephole	2949	1620 (1.82)	1299 (2.27)	870 (3.39)	809 (3.65)
Theorem	1590	1000 (1.59)	690 (2.30)	690 (2.30)	690 (2.30)
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)
Difference	11090	5610 (1.98)	2860 (3.88)	1960 (5.66)	1519 (7.30)
Nrev	8120	4580 (1.77)	2659 (3.05)	2020 (4.02)	1674 (4.85)
Primes	3199	1639 (1.95)	899 (3.56)	659 (4.85)	640 (5.00)
Pascal	32918	16460 (2.0)	8300 (3.97)	5878 (5.6)	4840 (6.8)

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

implementation (and many optimizations [23] 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[4]. The benchmarks have been chosen from the benchmark suite of the Aquarius Prolog system. In many benchmarks the amount of DAP present is limited. The maximum speed-up for such benchmarks is indicated. 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 ≥ 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.

6.2 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 [22]. Programs which are based on certain programming styles, such as *generate and test*, typically are unable to 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 2 shows performance results for a set of benchmark that benefit from extended dynamic DAP. 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 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.

7 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

[4] Some recent technical problems, specifically the loss of two memory boards, forced us to limit our experiments to a maximum of 8 processors. In the final version of the paper, if accepted, we hope to report more comprehensive results.

Program	Best DAP Execution			Best EDDAP Execution		
	Exec. Time	Speedup	No. Procs.	Exec. Time	Speedup	No. Procs.
Crossword	9700	1.55	5	429	22.6	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
TreeInsert	3180	1.0	10	970	3.3	10

Table 2: Performance Results for EDDAP (exec. time in msec.)

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 Kish Shen of University of Manchester and Manuel Hermenegildo and his CLIP group at UPM, Spain, for various discussions. This work has been partially supported by NSF grants CCR 96-25358, HRD 93-53271, and INT 95-15256, by NATO Grant CRG 921318.

References

[1] Ali, K., and Karlsson, R. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.* (1990), MIT Press.

[2] Bevemyr, J., Lindgren, T., and Millroth, H. Reform Prolog: the Language and its Implementation. In *Proc. of the 10th Int'l Conference on Logic Programming* (1993), MIT Press.

[3] Conery, J. S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, Ma 02061, 1987.

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

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

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

[7] Gregory, S., and Yang, R. Parallel Constraint Solving in Andorra-I. *Proceedings of FGCS'92* (June 1992).

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

[9] Gupta, G., Hermenegildo, M., Pontelli, E., and Costa, V. S. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94* (1994), MIT Press, pp. 93–109.

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

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

[12] Hentenryck, P. V. *Constraint Handling in Prolog.* MIT Press, 1988.

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

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

[15] Lusk, E., and al. The Aurora Or-parallel Prolog System. *New Generation Computing 7*, 2,3 ('90).

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

[17] Montelius, J., and Ali, K. A Parallel Implementation of AKL. *New Generation Computing 14*, 1 (1996).

[18] Muthukumar, K., and Hermenegildo, M. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming* (June 1991), MIT Press.

[19] Pontelli, E., and Gupta, G. Dependent And Parallelism in Logic Programming. Tech. rep., Laboratory for Logic, DB, and Advanced Programming, 1995. Internal Report.

[20] Pontelli, E., and Gupta, G. On the Duality Between And-parallelism and Or-parallelism. In *Proc. of Euro-Par'95* (1995), Springer Verlag.

[21] Pontelli, E., Gupta, G., and Hermenegildo, M. &ACE: A High-performance Parallel Prolog System. In *IPPS 95* (April 1995), IEEE Computer Society, Santa Barbara, CA.

[22] Pontelli, E., Gupta, G., Pulvirenti, F., and Ferro, A. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In *International Conference on Logic Programming* (1997), MIT Press.

[23] Pontelli, E., Gupta, G., and Tang, D. Determinacy Driven Optimizations of Parallel Prolog Implementations. In *Proc. of the Int'l Conference on Logic Programming 95* (1995), MIT Press.

[24] Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I., and Sekar, R. Making a Success out of Early Failures. Tech. rep., SUNY-Stony Brook, Dept. of Computer Science, 1997.

[25] Shapiro, E., Ed. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

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

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

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

[29] Warren, D. H. D. The Extended Andorra Model with Implicit Control. In *Parallel Logic Programming Workshop* (Box 1263, S-163 13 Spanga, SWEDEN, June 1990), Sverker Jansson, Ed., SICS.