Analysis of Dependent And-Parallelism

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

Abstract

We consider the problem of exploiting *non-deterministic dependent and-parallelism (DAP)* from Prolog programs. The main issues that arise in designing a parallel Prolog system for exploiting DAP are discussed. Three criteria that an ideal implementation of DAP should satisfy are developed. We prove that an ideal system that satisfies all three criteria cannot be constructed. These criteria are then used to evaluate and classify existing execution models proposed for exploiting DAP, and to inspire new schemes for exploiting DAP that are arguably better than the existing ones. One such scheme, termed the *Filtered Binding Model* is presented. The Filtered Binding model has been incorporated in the ACE and-or parallel Prolog system, running on a Sequent Symmetry and Sun Sparc Multiprocessors. Performance results from this implementation are presented and shown to be better than the results achieved by other systems.
Keywords: Prolog, And-parallelism, Dependent And-Parallelism.

1 Introduction

Logic programming is a popular computer programming paradigm that has been used in a wide variety of 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*.

This paper is concerned with the parallel execution of logic (Prolog) programs. Parallelism is automatically extracted from Prolog programs, resulting in faster execution, with very limited burden on 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* [16] 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 [12]); such independent goals can be run in parallel. &-Prolog [12] and &ACE [8] are two example 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, are executed in parallel. Dependent and-parallelism is readily found, for example, in applications that involve producer-consumer interactions.

So far researchers have mainly focused their attention on or-parallelism and independent and-parallelism. This is because of the apparent simplicity of the problems that have to be tackled in exploiting these two forms of parallelism[1].

Our purpose in this paper is to identify the main issues in designing an implementation scheme for a nondeterministic dependent and-parallel system (supporting Prolog's computational model). This allows us to analyze the weaknesses of the previously proposed models (DASWAM [24], as well as others [27, 17]) and the

[1]It should be noted, though, that strong relationships between or-parallelism and *dependent* and-parallelism have been identified [20].

limits of any implementation scheme for exploiting dependent and-parallelism. It also allows us to propose new models which are arguably superior in terms of amount of parallelism exploited and, especially, efficiency of execution. To achieve this objective, we identify three basic criteria that an ideal implementation of DAP should satisfy. Unfortunately, it turns out that an ideal implementation of DAP doesn't exists. Thus, at least one criteria has to be sacrificed in order to build a DAP system. Depending on which criteria one chooses to sacrifice, different models can be developed. This observation is used to create a taxonomy of dependent and-parallel systems, and to design new models.

Several new models have emerged from our rigorous analysis of dependent and-parallelism [19]. In this paper we present one such model—the *Filtered Binding Model*. The Filtered Binding model has been implemented within the ACE [9] system, and its performance results are presented. Even in its prototypical, unoptimized version, the performance results are significantly better than those of other systems, and underscore the advantages of adopting an analytical approach to designing models for exploiting parallelism.

The paper makes several novel contributions: (i) it proposes 3 criteria that an ideal implementation of DAP should satisfy; (ii) it shows that an ideal implementation of DAP cannot be achieved; (iii) it uses this theoretical result to develop a classification of existing DAP systems, which in turn is used to predict the existence of better, more efficient models; (iv) it describes one such model called the Filtered Binding Model, and present performance figures for it on a Sequent Symmetry, which demonstrates its superiority in efficiency w.r.t. the best DAP implementations realized to date.

We are assuming the reader to be familiar with the general ideas of Prolog and logic programming.

2 Non-deterministic DAP

Dependent And-parallelism is a generalization of independent and-parallelism. Much of the machinery needed for implementing IAP is also needed for DAP. For this reason, the presentation is articulated in two parts. First IAP and its implementation requirements are analyzed; next, the discussion is extended to encompass the most general form of DAP. Readers familiar with IAP and its implementation can directly skip to Section 2.2.

2.1 Independent And-Parallelism

A Prolog goal is composed of a conjunction of literals,

$$: -G_1, \ldots, G_n$$

Computing an answer to this goal involves successively solving each of the literals, starting with the leftmost one and progressively moving to the right until the goal G_n has been completely solved (*forward execution*). Formally, we can denote the process of performing one step of resolution (assuming Prolog semantics, i.e., left-to-right exploration of the derivation tree), reducing a subgoal q using a clause $p : -B$ and a unifier σ, as follows:

$$\langle (q, q_1, \ldots, q_n), \theta \rangle \vdash_P \langle (B, q_1, \ldots, q_n)^\sigma, \theta \circ \sigma \rangle$$

where the notation $\langle R, \theta \rangle$ represents the current resolvent (R) and the current computed substitution (θ) w.r.t. program P. A sequence of n consecutive resolution steps is indicated by $\vdash_P^n$. The computation terminates when no more subgoals are left, i.e. the *terminal* configuration $\langle \Box, \sigma \rangle$ is reached (wher $\Box$ denotes an empty resolvent). When a failure occurs, backtracking (*backward execution*) is performed, which involves choosing an untried alternative in the subgoals to the left of the failing one.

The sequential computation model above can be extended to exploit IAP [5, 14]. Subgoals are considered independent if the bindings produced by one of them do not affect the computation of the others[2]. Formally, two subgoals p, q are said to be *independent* iff

$$\forall R, \sigma, n\, (\langle p, \emptyset \rangle \vdash_P^n \langle R, \sigma \rangle \rightarrow$$
$$\forall R', \theta, m\, (\langle q, \emptyset \rangle \vdash_P^m \langle R', \theta \rangle \leftrightarrow \langle q^\sigma, \sigma \rangle \vdash_P^m \langle R'^\sigma, \sigma \circ \theta \rangle))$$

and vice versa (i.e. with p and q interchanged). Note that $\emptyset$ denotes an empty substitution. Intuitively, the sequence of steps that a subgoal (e.g. q) performs in its reduction is not affected by the presence of the bindings (σ) generated by the other subgoal (p).

If subgoals in $\{G_1, \ldots, G_n\}$ are known to be independent, we can allow their unrestricted—i.e., without any requirements for synchronization—execution in and-parallel. During forward execution, when the goals to be executed in parallel are reached, n independent tasks are started for the execution of the independent subgoals. The continuation of this set of goals (i.e., whatever comes after such set of subgoals) is not started until all the n subgoals have reported completion of their executions (i.e., the different parallel computations are "joined").

Backward execution is more complex: if a subgoal fails, the computation should move to the rightmost subgoal with untried alternatives to the left of the failing one. The main problem is that the subgoal on the left may have been executed by some other computing agent (necessitating communication between processing agents). It is also likely that this subgoal may still be executing. Further problems occur due to the possible non-conformance between logical order of execution and

[2]For the sake of simplicity, we will focus on data dependencies only. The discussion can be extended in a straightforward manner to include control dependencies due to side-effects. The reader is referred to [13] for a more detailed discussion on the various degrees of independence.

physical distribution of computations on the processors' stacks.

An independent and-parallel system that supports the above operations should be able to implement the same pattern of forward and backward execution found in sequential Prolog (i.e. depth-first, left-to-right exploration of the solution space). &-Prolog [12] and &ACE [8] are two such systems.

2.2 Dependent And Parallelism

The "classical" example of Dependent And-Parallelism (DAP) is represented by a conjunctive goal of the form: $p(X), q(X)$ where the two subgoals "compete" (in parallel) to construct the binding for the common (or *shared*) variable X.

Unrestricted parallel execution of the above query (in Prolog) is likely to produce non-deterministic behaviour: the result computed will depend on the order in which the two subgoals access X. Thus, the first aim of any system exploiting dependent and-parallelism is to ensure that the operational behavior of dependent and-parallel execution is consistent with (sequential) Prolog semantics. This can be realized only by guaranteeing that order-sensitive operations (e.g., bindings to shared variables, execution of side-effects) are performed in the same order as in a corresponding sequential execution.

The question then arises whether dependent and-parallelism is fruitful at all. Because typically in a query such as above, p will produce a binding for X while q will process (or consume) it. If this order between production of binding and its consumption is to be preserved, q will be suspended until execution of p is over. However, this is not always the case, and execution of p and q can be overlapped in certain situations: *(i)* q may first perform significant amount of computation before it needs the binding of X; this computation can be overlapped with computation of p, because it doesn't depend on X; *(ii)* p may first partially instantiate X. In such a case q can start working with the partially instantiated value, while p is busy computing the rest of the binding for X.

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 [23, 2]. This is largely due to the complexity of dealing with distributed backtracking. Nevertheless, it has been shown [24] 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.

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. Such common variables are usually called *shared* or *dependent* variables. Given two goals p, q with a *shared variable* X, then the following holds:

- $X \in vars(p) \cap vars(q)$;
- $\langle p, \emptyset\rangle \vdash^n_P \langle R, \theta\rangle$ such that
  - $X^\theta \neq X$ (but see footnote [3] below);
  - the binding for X affects the computation of q, that is

    $$\exists m\,(\langle q, \emptyset\rangle \vdash^m_P \langle R', \eta\rangle \wedge \langle q^\theta, \theta\rangle \not\vdash^m_P \langle R'^\theta, \theta \circ \eta\rangle)$$

    that is there is a step in the execution of q where the presence of the binding for X leads to a different action (e.g., selection of a different clause, or failure).

The objective of maintaining the same sequential view of execution as in traditional Prolog programming, imposes the requirement of generating variable bindings in the *same order* as in a sequential execution. This boils down to deciding which subgoals are entitled to bind each variable. This concept is commonly expressed through the notion of *producers* and *consumers* of a variable—a producer is a subgoal which is entitled to bind the variable, while a consumer is only supposed to read the value of the variable and not change it. More formally, given a collection of parallel subgoals $q_1, \ldots, q_k$, and given a dependent variable $X \in vars(q_1) \cap \ldots \cap vars(q_k)$, subgoal q_i is the producer for X iff

- for $1 \leq j \leq i-1$, if

  $$\langle q_j^{\sigma_1 \circ \ldots \circ \sigma_{j-1}}, \sigma_1 \circ \ldots \circ \sigma_{j-1}\rangle \vdash^n_P \langle \Box, \sigma_1 \circ \ldots \circ \sigma_j\rangle$$

  then $X^{\sigma_1 \circ \ldots \circ \sigma_j} = X$;
- $\langle q_i^{\sigma_1 \circ \ldots \circ \sigma_{i-1}}, \sigma_1 \circ \ldots \circ \sigma_{i-1}\rangle \vdash^n_P \langle \Box, \sigma_1 \circ \ldots \circ \sigma_i\rangle$ and $X^{\sigma_1 \circ \ldots \circ \sigma_i} \neq X$.

The different schemes to handle DAP proposed in the literature differ in the way in which this concept has been realized (e.g., a-priori detection of consumers with suspension if they attempt binding, a-posteriori comparison of bindings, etc.). It is important to observe that determining the actual producer of a variable is an undecidable property. Therefore, static analysis is of limited help, rather the producer or consumer status of a subgoal has to be determined dynamically to exploit maximum parallelism efficiently.

In terms of backward execution, a further degree of complexity is added: whenever a binding to a shared variable is undone, any subgoal that may have "consumed" that binding needs to be rolled back to the point at which the consumption took place.

[3] If $X^\theta = X$ then the goal q_1 does not produce a binding for X. In such a case q_1 and q_2 can be regarded as independent and executed in parallel without any restriction. This case is known as non-strict independence and-parallelism (NSIAP) [13].

Implementation Mechanisms

All the machinery that is needed for supporting IAP (described earlier) is also needed for DAP (since former can be viewed as a subset of the latter). Extra machinery is needed to handle new concerns that arise in DAP, namely, the management of shared variables:

1. Implementation of *shared variables*: Various issues need to be considered for representing and implementing shared variables:

(a). *identification:* note that a normal variable can become dependent (i.e. change its "status") during the execution—e.g. when it gets aliased to a shared variable.

(b). *race condition*: the construction of a binding for a variable may require a potentially unbound number of instructions (i.e. when a variable gets bound to a complex structure), and given that concurrent read/write accesses to the same variable are admissible, there is a risk of a consumer goal reading a binding that is still being constructed.

(c). *aliasing*: different paths leading to the same variable should be organized in such a way to give the same 'view' of the variable, in order to avoid inconsistent behaviour.

Issues related to the first and third point will be analyzed in detail in the rest of this work. A new and more efficient technique for dealing with race conditions has been implemented in the ACE system, but its description is omitted due to lack of space and also because it doesn't affect the main results of this paper [19].

2. *Binding Validation*: a shared variable can be only bound to bindings that have been *validated* (i.e., bindings which have been certified as coming from a current producer for the variable). Binding which cannot be validated must be either temporarily suspended (as long as the subgoal is certified not to be a producer) or rejected (and the corresponding computation rolled back). The main aspect of this problem is the actual identification of the producer of a shared variable. Since this property is undecidable, it is typically approximated by treating the currently leftmost subgoal that can access the shared variable as the producer. Note that this approximation may require refinement during execution: if the leftmost goal that was deemed to be the producer of a shared variable X terminates without instantiating X (i.e. our approximation was incorrect), then the next leftmost active subgoal that can access X (and was earlier a consumer) is designated as X's next producer.

3. *Implementation of extended backtracking*. The presence of dependencies between computations may require complex *roll-back* patterns, in order to maintain consistency of the execution [24, 11]. In the rest of this work we will not deal with this issue as backtracking behavior does not influence our main result.

Degrees of DAP Exploited

Another question that has to be considered in designing execution models for DAP is "How much parallelism should be exploited?" Different models can be distinguished depending on the degree of DAP exploited, i.e., the "size" of the set of programs that they can parallelize. Clearly, the higher the degree of exploitation, the more complex the mechanisms required for DAP execution (and the higher the corresponding overhead introduced). Three degrees of DAP can be typically distinguished:

- *Determinate DAP*: DAP execution is allowed only for subgoals that are known to be determinate (their derivation tree contains at most one successful refutation). This is adopted in Andorra-I [3], Reform [2] and Parallel NU-Prolog [18].
- *Unrestricted*: no restrictions are imposed on the classes of programs that are subject to DAP execution.
- *Restricted DAP*: DAP execution is allowed between any two subgoals except when its exploitation leads to excessive overhead. We have left this class vague on purpose, to include all the approaches that fall in between the two extremes described above. The best examples are DASWAM [24] and Attributed Variables [17]: both control excessive overhead by restricting general aliasing between shared variables (this restriction is of considerable importance and will be further analyzed later in this paper).

Curative vs. Preventive Approach

Two approaches can be taken to synchronize producer and consumer computations.

1. *Curative Approach*: nothing is done to prevent either a producer or a consumer from binding shared variables. Special actions are taken *a posteriori* to validate the bindings either when a conflict occurs [27, 6] or at the end of the parallel execution [28]. It should be noted that in models which delay validation of binding until the end of the parallel conjunction (e.g., based on back-unification [28]), it is extremely difficult to enforce Prolog semantics. Furthermore, a consumer subgoal may produce unbounded redundant computation due to the lack of communication with the producer.

Curative models have also been designed that allow producers and consumers to freely bind the shared variable if it is unbound [27, 6]. Binding validation is done *during* the parallel computation when the producer accesses a shared variable and finds it bound. In such a case, the consumer (which bound the variable erroneously) has to be rolled back and its binding retracted. In this case too unbounded redundant computation may be performed. In these models validation requires being able to detect whether the generator of the binding is the leftmost active subgoal accessing the shared variable (i.e., given two subgoals, being able to tell whether one is on the left of the other in the current computation tree).

2. *preventive approaches*: in this class of models, no subgoal is allowed to bind a shared variable unless there is a guarantee that it is effectively the producer of that shared variable. Since the producer subgoal cannot be determined a-priori, this information needs to be approximated. The best approximation is represented by "the leftmost active subgoal accessing the shared variable", and is the only safe approximation (unless additional knowledge about the binding ability of other subgoals is known). Preventive model are typically preferred, as they do not perform redundant computation.

3 Analysis of Implementation Schemes

3.1 Criteria for DAP

The following three basic operations need to be efficiently implemented for realizing fast, forward DAP execution of goals:

1. Procedure Call: in presence of and-parallelism the procedure call operation involves performing all the actions related to creation of a parallel call (spawning new computations, setting up environments, etc.).

2. Unification: which involves:

(a). access to variables: finding variable's location, dereferencing, and obtaining its value;

(b). Binding: assigning value to an unbound variable;

(c). Validation: verifying if a goal is allowed to bound a shared variable or not (only a producer goal can bind a shared variable).

3. Detection of continuation: determining the point from where execution should continue after a goal has been solved by a processor. This involves:

(a). *Goal scheduling*: picking another goal once a processor finishes executing the current one. Terminating an execution may require special actions connected with the management of shared variables (e.g., refining the approximation of "producers").

(b). *Completion of parallel calls*: this involves taking appropriate actions when all goals in a parallel conjunction are solved, including starting the execution of the continuation goal after the parallel conjunction.

These three operations can also be dubbed *task creation*, *variable access*[4], and *task switching*. Any model for exploiting DAP can be characterized by the way in which these three operations are performed.

[4]Note that both for variable binding and validation the variable needs to be accessed first.

In an ideal system, all three operations should be executed as efficiently as possible—with a *constant time* ($O(1)$) overhead. None of the models that exploit a sufficiently general form of DAP (i.e., at least achieving the *Restricted DAP* stage in our parallelism-exploitation scale) have achieved the ideal. The ideal constant-time nature has been achieved (at least theoretically) only in the Basic Andorra Model [3] and the Reform system [2]—the restriction to determinate parallelism leads to considerable ease of implementation (but severely limits the class of programs that can be executed in parallel). Our analysis, presented later, shows that it is possible to achieve the ideal execution behaviour for restricted DAP. One such scheme, called the Filtered Binding Model, is described in Section 5.

In the case of unrestricted DAP, where no restrictions are imposed on subgoals that can be run in parallel, it turns out that it is not possible to perform all three operations in constant time. The intuition behind this is the following. In order to keep track of whether a variable is accessed in a producer or consumer position (to determine if a goal can bind a variable or not) a certain amount of information needs to be stored with the variable itself (e.g., in case of DASWAM this information consists of pointers to the parallel call descriptors, in Optimistic Parallel (OP) model it is the id of the subgoal which generated the binding, etc.). If a succession of bindings between shared variables are performed, each intervening element in this succession needs to be maintained, i.e., *no short-circuiting of the chain of variables can be done.* In the pathological case, this will eventually produce a tree structure of bindings, in which the nodes are the shared variables and the links are created between those variables that have been aliased. For example, consider the program shown below:

```
?- (k(X1) , p(.., X1..) , q(.., X1, ..)).
p(.., X1, ..) :- ..., (s(X1, X2) , t(X2)), ..
p(.., X1, ..) :- ...
s(X1, X2) :- X1 = X2, ....
```

Initially, `k` is a producer of `X1` and `p` and `q` are its consumers. Likewise, the goal `s` is a producer for the shared variable `X2`, and `t` is its consumer. The goal `s` inherits the consumer status for `X1` from `p`. `s` aliases `X2` (a variable for which it is a producer) to `X1` (a variable for which it is a consumer). If we simply alias `X2` to `X1`, then if `t` became the producer for `X2` (because `s` didn't bind `X2`), then due to the aliasing, `t` will be wrongly allowed to bind `X1`. This is because, although `t` is now a producer for `X2`, `X2` is aliased to `X1` for which `t` is a consumer (`k` is still `X1`'s producer). So even though `X1` and `X2` are aliased, `t` has to know whether it is a consumer or a producer for each of them. This, information has to be maintained, and explicitly checked for each variable encountered in the dereference chain. This can be an

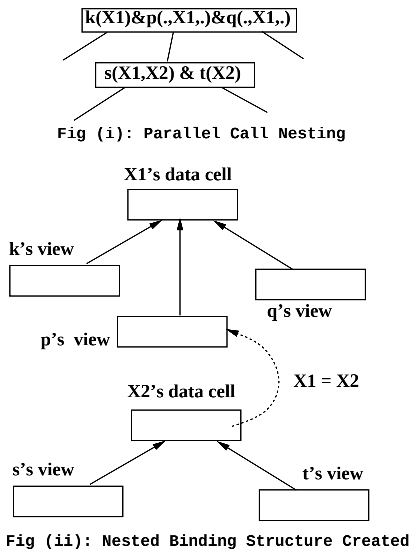

Figure 1: Nested Parallel Calls with Variable Aliasings

expensive (non constant-time) operation, as arbitrarily long chains of shared variables can be produced in pathological cases.

3.2 The View Forest

The producer of a shared variable is the leftmost subgoal which actually binds it. The exact producer subgoal cannot be determined *a priori* and has to be approximated at runtime in both the curative as well as preventive models by the leftmost goal that has access to the shared variable. Thus, independent of the model adopted, identification of the producer goal requires knowing whether a node (corresponding to a subgoal) is leftmost in the computation tree that is dynamically growing/shrinking.

We formalize the problem of managing producers and consumers through the notion of *Views Forest*. Intuitively, views forest explicitly describe the different access paths to a given shared variable that may exist during execution. Each shared "object" corresponds to a tree in the view forest. The views forest is constructed as follows:

- whenever a new shared variable is created, a new tree is created in the forest. This tree has that variable as its root node and no other nodes.
- given a parallel call $G_1, \ldots, G_n$ with X as a shared variable (i.e. different subgoals may access X), then a new leaf is added below the node for X for each subgoal G_i that can access X. Figure 2 illustrates this step. Every reference to the variable X within the computation tree rooted in G_i will be treated as a reference to X_i (X_i represents G_i's view of X).
- whenever a parallel subgoal G_i is completed, the corresponding X_i is removed from the forest;
- when a variable X_i in goal G_i is aliased to another shared variable Y, then the root node of the tree corresponding to Y is made to point to X_i.

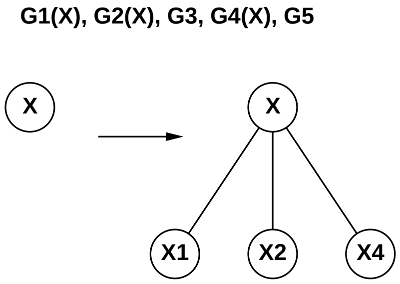

Figure 2: Views Forest

An example view forest for an actual program is shown in figure Figure 1(ii). A goal can bind a shared variable X only if its corresponding X_i (its view) is in the leftmost branch of the tree for X. Observe that, in principle, validating a binding will require also identification of the specific tree in which the search has to be performed (since, a-priori, we do not have any knowledge about which tree contains which nodes). This suggests that a better formulation of the problem is as follows: given a node $\mathcal{B}$, find out the root node $\mathcal{A}$ of the largest subtree in the view forest such that $\mathcal{B}$ is leftmost in the subtree rooted at node $\mathcal{A}$. We term this the *leftmostness problem*.

We next show that the leftmostness problem cannot be solved in $O(1)$ time. The proof proceeds by first assuming that the leftmostness problem can be solved in $O(1)$ time, and then showing that this allows solving the union-find problem over disjoint sets [26, 7] in time less then its proved lower bound—a contradiction.

The disjoint-set problem is as follows. A a collection $\mathcal{S} = \{S_1, \ldots, S_n\}$ of disjoint sets is given, where each set is identified by an element belonging to it, called *canonical element* of the set. The sets are dynamic, and they can be manipulated through three operations: *makeset(x)* that creates a new set containing the single element x; *find(x)* that identifies the (unique) set containing the element x; *union(x,y)* which unites the two sets containing x and y. The operations are assumed to be performed *on-line*, that is each operation must be completed before the next one is known. Tarjan [26], and later Fredman and Saks [7], showed that under some simple technical conditions, a lower bound of $\Omega(m\alpha(m,n))$ time is required for operations on any disjoin-set data structure, where n is the number of *makeset* operations and m is the total number of operations (*makeset*, *union*, and *find*) performed. The notation $\alpha(m,n)$ denotes the functional inverse of Ackermann's function; α is a *very* slowly growing function.

3.3 The Main Result

Lemma 3.1 *Given a dynamic forest F of dynamic trees and a node n, it is not possible to determine in $O(1)$ time the root node n' of the largest subtree of a tree in F such that n is in the leftmost branch of that subtree.*

Proof Sketch: Consider the following representation of a collection $\mathcal{S} = \{S_1, \ldots, S_n\}$ of disjoint sets: Each S_i is represented by a tree with exactly one branch where each node corresponds to an element of set S_i and the canonical element is the root node. The root nodes of all S_i's are joined in a chain that forms the rightmost branch of the final tree created. Thus, all canonical elements can be collected by visiting the rightmost branch of the tree. (See example in Figure 3).

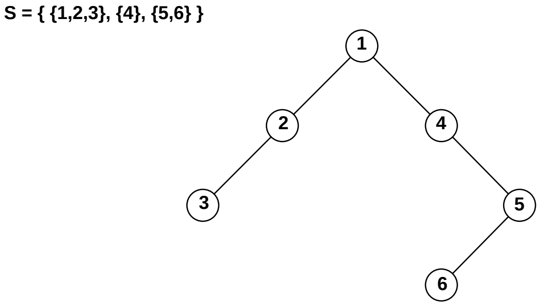

Figure 3: Tree representation of Disjoint Sets

Suppose the leftmostness problem can be solved in $O(1)$ time. It is straightforward to implement all the three operations defined above for a disjoint set in $O(1)$ time. It is easy to see that *makeset* and *union* operations can be realized in $O(1)$ time on any pointer machine. The operation *find* requires determining the canonical representative of the set to which the element belongs. This element is precisely the root node of the subtree in which the given element is leftmost, and given the assumption that the leftmostness problem can be solved in $O(1)$ time, the *find* can also be done in $O(1)$ time.

Putting everything together, it is straightforward to show that this representation of disjoin-sets will allow any arbitrary sequence of m operations to be performed in $\Theta(m)$ time. However, this is better than the lower bound proved by Tarjan—a contradiction. Therefore, the leftmostness problem cannot be solved in $O(1)$ time. □

Theorem 3.2 *Unrestricted DAP execution incurs non-constant time overhead, i.e., at least one of task creation, task switching, or variable access is a non-constant time operation.*

Proof Intuition:[5] Unrestricted execution results in generation of arbitrary view forest. The ability to validate a binding generated by a given goal G for a shared variable X requires that we find G's view corresponding to X, and then check if the node corresponding to that view is leftmost in the tree in which X occurs. Clearly, by the lemma above this cannot be done in $O(1)$ time. □

It is possible to perform the leftmost check in constant time but then non-constant cost will be incurred elsewhere (in task creation or switching). (For instance in the union-find problem, *finds* can be made $O(1)$ by incurring overhead in the union operation.)

Note that the non-constant cost comes from the fact that when shared variables are aliased, the aliasing cannot be short circuited as in the case of aliasing of normal variables. The structure of the aliasing has to be maintained as the producer/consumer status is encoded in it. On the other hand, if no aliasing between shared variables takes place, then we are guaranteed that the tree can be kept at depth 1, allowing constant time validation of bindings. Not surprisingly, many implementation models (e.g. DASWAM) impose this restriction to avoid arbitrarily long chains of shared variables.

4 Classification of Models for DAP

The above result (which pairs up with a similar result [10] proved for or-parallelism) gives us possible criteria for comparing different models for DAP. The similarity between this result and the similar one proved for or-parallelism [10] is not surprising: or-parallelism and dependent and-parallelism have been shown to share similar characteristics [20]. Given that at least one non-constant component must be present, we should prefer models in which such overhead is placed in operations whose frequency is not program dependent and can be controlled by the system (e.g., the overhead is incurred during task switching rather than variable access). Furthermore, situations that lead to non-constant behavior (creation of chains of dependent variables) do not always arise, and for such programs we want a model to not incur excessive overheads.

Given that for unrestricted DAP, at least one of the three criteria has to be sacrificed, we can classify the various schemes that have been proposed in the literature based on which criteria they choose to sacrifice. Thus, schemes that incur non-constant time overhead in a given operation, will all be placed in one class. This classification scheme is shown in Figure 4. This classification also suggests new models for DAP: For each of the empty leaf nodes in the classification tree (Figure 4) a model can be designed with those characteristics.[6] For example, a new model may be obtained by shifting the non-constant time overhead from one of the 3 operations in an existing model to another operation. It

[5] Formal proofs can be found elsewhere [19].

[6] This method of designing models has similarities to Mendeleyev's prediction of existence and properties of yet-undiscovered chemical elements with the help of his periodic table.

should be noted that frequency of the variable access and environment creation operations during execution is dependent on the program, whereas the frequency of task switching is in control of the scheduler. Thus, if the overhead is shifted from variable access to task switching, then by carefully building the scheduler a better parallel system can be obtained. This is, in fact, how the *Linked Binding Model* [19] was derived. The above analysis also shows that if unrestricted DAP is not allowed, then it is possible to eliminate non-constant time overhead. However, this is not the case with any of the existing models.

4.1 Proposed Models for DAP

The models for DAP found in the research literature can be divided into two categories, depending on whether validation takes place at the *goal-level* or at the *binding-level*. Each class is further refined depending on whether validation is *preventive* or *curative*. Table 1 summarizes the main characteristics of the various models proposed for DAP, with respect to the criteria described in the previous section. The classification of these schemes is also shown in figure 4.

1. Goal-Level Validation: validations are performed rarely, typically leading to excessive sequentialization of the computations or generation of excessive speculative work.

- *preventive:* execution of a subgoal is not started until the validation conditions are met. Independent and-parallelism and the Basic Andorra Model are examples in this class. They guarantee high efficiency, at the price of severe restrictions on the amount of parallelism exploited.

- *curative:* Each and-parallel computation develops its computation on local copies of the environments, introducing an additional "merging" step at the end of the parallel call—to verify consistency of the values produced by the different computations for the shared variables. This approach, adopted mainly in some of the older process-based models, like Epilog [28] and ROPM [22], has the advantage of being extremely simple, but it suffers some serious drawbacks: *(i)* it produces highly speculative computations (due to the lack of communication between parallel subgoals); *(ii)* it may produce parallel computations that terminates in a time longer than the corresponding sequential ones; *(iii)* it is extremely difficult to enforce Prolog semantics.

2. Binding-Level Validation: validation is associated to the operation of either accessing or binding a variable. Clearly this implies higher overhead (due to the higher frequency of these operations) but also higher parallelism and lower speculative work.

- *preventive:* bindings are preempted until validation is guaranteed to be successful. This typically involves suspension of not-validated computations. Various models fall in this class:

  DASWAM [24] represents shared variables through a new class of tagged locations (*dependent cells*), in which the information required to manage the shared objects is stored (e.g., pointer to the parallel call to which the shared variable belongs to, etc.). DASWAM incurs a non-constant overhead exclusively at the level of binding validation—this requires traversal of a potentially unbound chain of parallel call descriptors. This is a direct consequence of the fact that different views of a shared variable corresponding to its producer and consumer goal are not maintained separately. As a result, every shared variable access requires the computation of the actual position of the subgoal (from where the access request originated) within the (nested) computation tree—which cannot be realized in constant-time.

  Attributed Variables: Attributed Variables are an extension to basic Prolog implementations, originally suggested by Hoitouze [15]. An attributed variable is a standard variable with which an attribute (represented by a standard term) can be associated. Whenever unification leads to the binding of an attributed variable, a *daemon* is automatically invoked and the (user) definition associated to such daemon is executed. This mechanism, coupled with the ability to allow creation of parallel threads, is sufficient to implement a DAP model as shown by Hermenegildo [17]. Each dependent variable X is split into multiple instances, one for each subgoal belonging to the parallel call. Explicit procedures are introduced to handle unification and transfer bindings to the different instances of each shared variable. The idea behind this model is attractive, and it shares some points with our Filtered Binding model presented later. Unfortunately, non-constant time overhead is introduced during variable binding because when a variable is successfully bound, the binding has to be propagated to all the copies of that variable.

- *curative:* validation does not preempt bindings from taking place (i.e. any goal can bind a shared variable), but special rollback actions are needed whenever a violation of program semantics is detected. The two most significant proposals where this strategy is adopted are those made by Tebra [27] and by Drakos [6]. They can be both identified as instances of a general scheme, named *Optimistic Parallelism (OP)*. In OP, validation of bindings is performed not at binding time (i.e., the time when the shared variable is bound to a value), but only

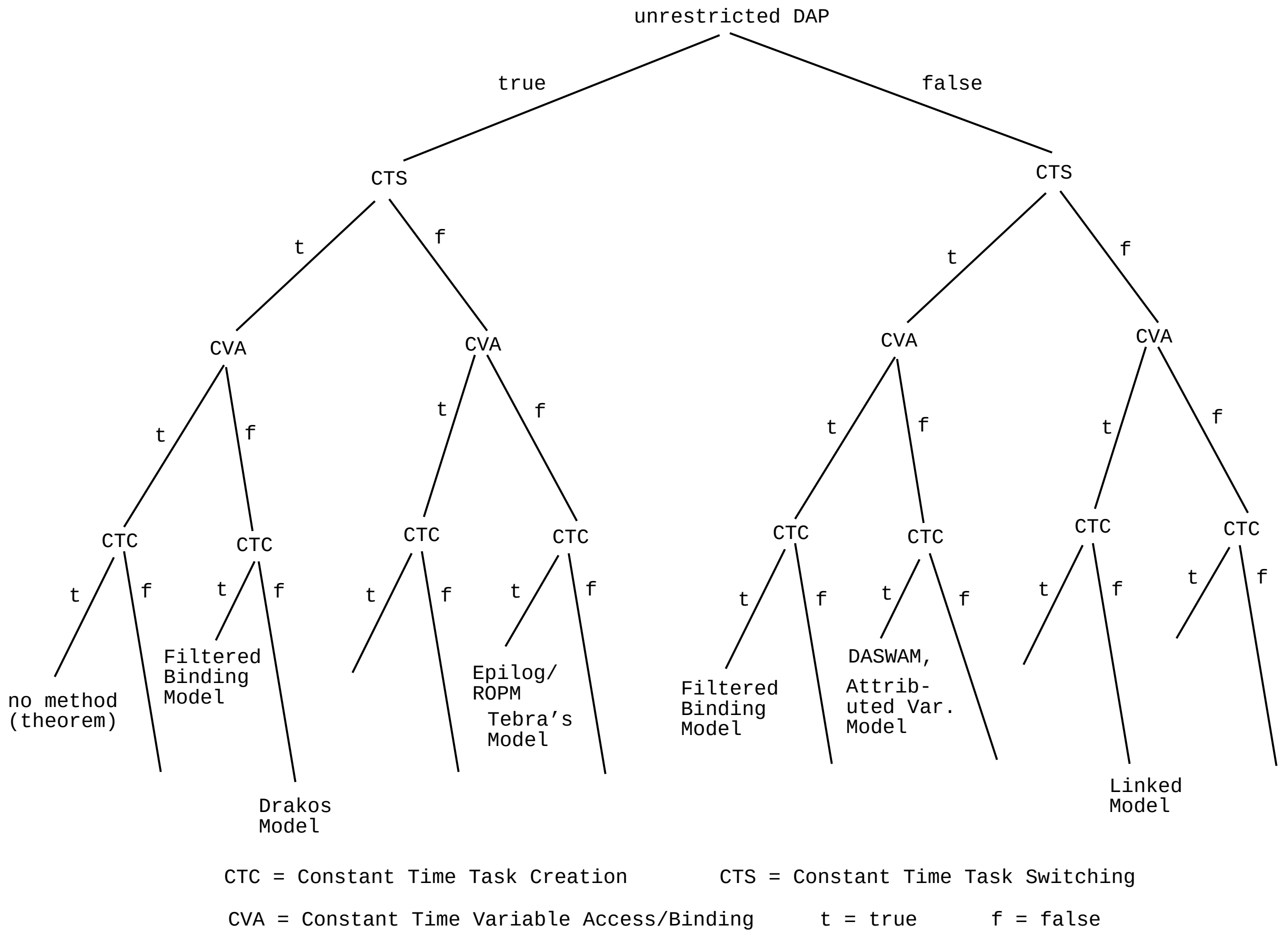

Figure 4: Classification of DAP Methods

when a conflict occurs (i.e. when a producer attempts to bind a shared variable that had already been bound earlier by a consumer goal.) In case of a conflict, the lower priority binding (made by the consumer), has to be undone, and the consumer goal rolled back to the point where it first accessed the shared variable. These model have various drawbacks, mainly related to their high speculative nature, to the limitations of some of the mechanisms adopted (e.g. labeling schemes to record binding priorities), and to the high costs of rolling back computations[7].

5 Designing New Schemes for DAP

With the aid of the criteria developed and the observations made above, we have developed alternative, more efficient implementation schemes for DAP Prolog systems. In particular, the main goal of our effort was to study the effect of moving the non-constant overheads in different operations, while attempting to reduce the overhead as much as possible.

A common principle has been used in the design of these new implementation schemes: 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 $\exists X(p(X) \wedge q(X)) \not\equiv \exists X(p(X)) \wedge$

[7] Tebra's model, in its original definition, generates constant-time task-switching, but limits the maximum depth of nesting of parallel calls. Unbound nesting of parallel calls requires non constant-time activities during task-switching.

Model	Environment Creation	Task Switching	Var. Access/Binding Access	Validation	Degree of DAP
Basic Andorra model	constant	(?)constant	constant	constant	determinate
Epilog/ROPM	constant	non-constant	constant	non-constant	unrestricted
DASWAM	constant	constant	(?)constant	non-constant	restricted
Attrib. Vars.	constant	constant	non-constant	constant	restricted
Drakos' model	non-constant	constant	non-constant	constant	unrestricted
Tebra's model	constant	non-constant	non-constant	non-constant	unrestricted
Filtered Binding model	constant	constant	constant	constant	restricted
Filtered Binding model	constant	constant	constant	non-constant	unrestricted

Table 1: Evaluation of DAP Models

$\exists X(q(X)))$. 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).

The above guiding principles have inspired the *Linked Binding Model* [19] and the *Filtered Binding Model* so far. We breifly describe the Filtered Binding Model next. We also present performance figures from its implementation.

5.1 Filtered Binding Model for DAP

The Filtered Binding model arose from our analysis that models exploiting *restricted* DAP can deliver constant-time performance on all three basic operations. The Filtered Binding Model is an instance of the class of models which use *binding-level* validation and are *preventive*. The specific approach assumes a program statically annotated to identify the promising sources of parallelism (in the same style as DASWAM [24]—details on how to generate annotations are omitted due to lack of space [19]). Each subgoal maintains an independent access path to the shared object. The idea of 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.

Figure 5 presents an intuitive schema of this idea. 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).

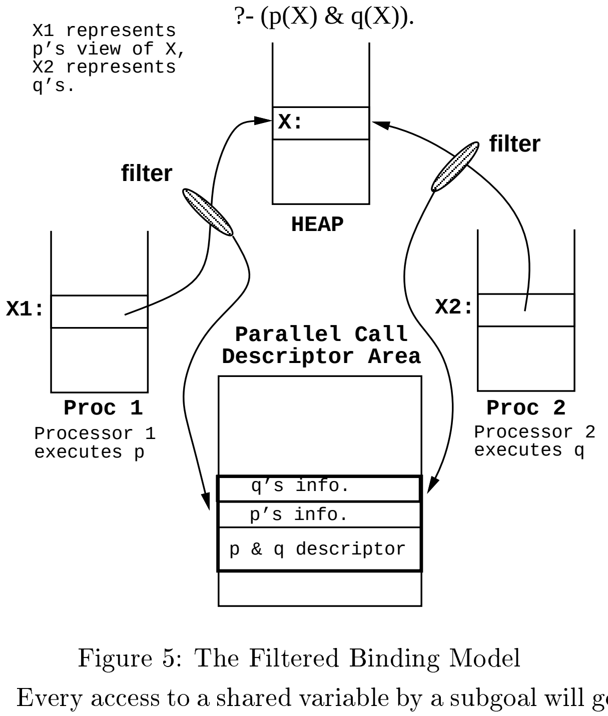

Figure 5: The Filtered Binding Model

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" of the access (producer or consumer).

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 is possible to prove that at any time a variable access will require traversal of at most one filter—which means constant-time validation of an access. The setup of a parallel call and the detection of continuation also do not require any non constant-time operation (cost is always bounded by the number of dependent variables detected by the compiler in that parallel call[8]). Details of the Filtered Model are not reported here for lack of space. The interested reader is referred to [19] 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.

Thus, the Filtered Binding model is a model that exploits restricted DAP and *performs all operations in constant-time* (see table 1). 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). If this restriction is relaxed then a non-constant overhead will be produced in the variable access operation. In the implementation, in fact, this restriction is not imposed, since, in practice, the pathological case that leads to the non-constant overhead case seldom arises. 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* [4].

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

5.2 Performance Results

As mentioned earlier, the Filtered Binding model for DAP has been recently incorporated in the ACE system, an and/or parallel implementation of Prolog developed at New Mexico State University in collaboration with UPM, Madrid. The current version of the ACE system is running on both Sequent Symmetry and Sun Sparc multiprocessors.

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 implementation (and many optimizations [21] can be applied). Implementation of garbage collection and various other optimizations is under way. Table 2 presents the execution times and speedups obtained on various benchmarks for the Sequent[9]. Table 4 describes the benchmarks used. 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.

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.

Comparison with Other Models

We have compared Filtered Binding Model as implemented in ACE to the three models that we consider the best implemented to date: DASWAM [25], Andorra-I [3], and Reform Prolog [2]. Of these, both Andorra-I and Reform limit exploitation of parallelism to very specific classes of programs (determinate parallelism for Andorra-I, determinate-bindings DAP for Reform) which allows in principle a considerably higher efficiency of implementation, while DASWAM exploits the fully general form of DAP. Even so, on a majority of programs ACE shows performance that is at par with that of Reform on deterministic programs and 2 to 5 times faster compared to Andorra-I. On most of the benchmarks DASWAM and ACE achieves very similar speedups; occasionally the speedups on DASWAM are slightly better. On the other hand, and as can be observed in Table 3, the efficiency of ACE is considerably superior to that of DASWAM (and the high overhead of DASWAM on certain benchmarks can be seen as partial explanation for the better speedups). (Note that in Table 3 each column labeled X/Y presents the overhead introduced when the program was run on Y and

[8] 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.

[9] 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.

Goals	&ACE agents				
Query	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 2: Execution times in msec (Speedups are shown in parenthesis)

then X and the timings were compared.) It should be emphasized that the results presented for ACE come from the first prototype implementation. We hope to considerably improve these speeds and speedups in the future.

6 Conclusions

We presented an overview of the problems encountered in incorporating DAP in a parallel Prolog system. The main characteristics common to all the implementation schemes were identified, and major weaknesses of the existing models were analyzed. Three criteria for an ideal implementation of DAP were presented and discussed. It was proved that no ideal system exists that satisfies all three criteria. The criteria were used to classify and critique existing schemes for exploiting DAP as well as for proposing new implementation models for efficiently exploiting DAP. These schemes are more efficient than any of the schemes proposed/implemented for exploiting non-deterministic DAP to date. Of these models proposed by us, performance results from an implementation of the Filter Binding Model on a Sequent Symmetry Multiprocessor were reported. These results demonstrate that indeed DAP can be fruitfully and efficiently exploited to speed-up execution of large programs. Work is in progress to further optimize our models and implementation, as well as to combine DAP with or-parallelism.

Acknowledgements

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 92-11732, HRD 93-53271, and INT 95-15256, by NATO Grant CRG 921318, and a fellowship to Enrico Pontelli by Phillips Petroleum.

[10](for program Micro in Table 4): This program is composed of a loop whose iterations have the typical producer/consumer pipelined organization—a very nice structure for exploiting DAP. However, the degree of speedup decreases as the number of processors increase due to the progressively decreasing size of the iterations of the loop. The size of the first iterations tends to dominate the parallel execution time.

References

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

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

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

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

[5] D. DeGroot. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems*, pages 471–478. Tokyo, Nov. 1984.

[6] N. Drakos. Unrestricted And-Parallel Execution of Logic Programs with Dependency Directed Backtracking. In *Proc. of the International Joint Conference on Artificial Intelligence*, 1989.

[7] M.L. Fredman and M.E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Procs. of 21st ACM Symposium on Theory of Computing*. ACM, 1989.

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

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

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

[11] G. Gupta and E. Pontelli. An overview of the ace project. In *Workshop on Parallelism and Implementation Technologies*. ESPRIT Compulog Net, 1995.

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

Query	DASWAM		ACE	
	WAM/DASWAM$_{seq}$	DASWAM$_{seq}$/DASWAM$_{par}$	WAM/ACE$_{seq}$	ACE$_{seq}$/ACE$_{par}$
Pqsort	54.5%	95.8%	4.1%	20%
kkqueen	40.5%	22.7%	8.2%	15%
nrev	44.3%	139%	1%	5.2%
Bt_cluster	39.3%	0%	8.6 %	0%
Pascal	47.8%	6.9%	10.7%	3.2%

Table 3: Comparison DASWAM/ACE

Goals	Description	Max. Speedup	No. of Lines
Genetic	Genetic Algorithm applied to solve a TSP problem	n/a	450
Disj	Part of the Aquarius Compiler (remove disjunctions)	2.0	300
Dynam	Dynamic Programming algorithm to compare DNA sequences	n/a	150
Micro	Micro-code compactor for 8600 microcode	see note [10]	720
Peephole	Peephole Optimizer used by Aquarius Prolog	n/a	680
Theorem	Simple arithmetic theorem prover	3.0	135
TSP	Direct solution to the TSP problem (parameter 24)	n/a	145
Pqsort	Quicksort	n/a	13
Cls	Simulation of a client-server interaction	n/a	35
Difference	Recursive difference between two lists	n/a	15
Nrev	Naive reverse	n/a	3
Primes	Prime numbers computed using Eratosthenes' Sieve	n/a	35

Table 4: Description of Benchmarks

[13] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1995. to appear.

[14] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proc. 3rd ICLP, LNCS 225*, pages 25–40. Springer-Verlag, 1986.

[15] S. Le Houitouze. A new data structure for implementing extensions to prolog. In *Symp. on Progr. Languages Implem. and Logic Programming*. Springer Verlag, 1990.

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

[17] D. Cabeza M. Hermenegildo and M. Carro. Using Attributed Variables in the Implementation of Parallel and Concurrent Logic Programming Systems. In *Proc. of the International Conference on Logic Programming*, 1995.

[18] L. Naish. Parallelizing NU-Prolog. In *Proc. of 5th International Conference and Symposium on Logic Programming*. MIT Press, 1988.

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

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

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

[22] B. Ramkumar and L.V. Kale. Machine Independent AND and OR Parallel Execution of Logic Programs. Part i and ii. *IEEE Transactions on Parallel and Distributed Systems*, 2(5).

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

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

[25] K. Shen. Initial Results from the Parallel Implementation of DASWAM. In *Proc. of the Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[26] R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18), 1979.

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

[28] M.J. Wise. *Prolog Multiprocessors*. Prentice-Hall, 1986.