Efficient Management of Backtracking in AND-Parallelism

M. V. Hermenegildo[1]

R. I. Nasr[2]

Abstract

A backtracking algorithm for AND-Parallelism and its implementation at the Abstract Machine level are presented: first, a class of AND-Parallelism models based on goal independence is defined, and a generalized version of Restricted AND-Parallelism (**RAP**) introduced as characteristic of this class. A simple and efficient backtracking algorithm for **RAP** is then discussed. An implementation scheme is presented for this algorithm which offers minimum overhead, while retaining the performance and storage economy of sequential implementations and taking advantage of goal independence to avoid unnecessary backtracking ("restricted intelligent backtracking"). Finally, the implementation of backtracking in sequential and AND-Parallel systems is explained through a number of examples.

KEYWORDS: LOGIC PROGRAMMING, PARALLEL PROCESSING, INTELLIGENT BACKTRACKING, AND-PARALLELISM, PROLOG.

1 Introduction

The execution of logic programs [9] in parallel is a subject of great interest because of the dual relationship between logic and parallelism: parallel execution seems to be a promising way of increasing the execution speed of logic programs; logic programs in turn offer multiple sources of parallelism [4] so that concurrency can (ideally) be uncovered automatically (or expressed cleanly) and mapped onto parallel architectures.

Of the several types of parallelism present in logic programs, we are specially interested in **AND-Parallelism**, because it offers the advantage that, in general, all work done by a collection of AND-Parallel processes is "useful" for finding a particular solution to a query. If OR-Parallelism is supported in addition to AND-Parallelism, backtracking is not needed; a set of "solutions" is maintained instead for each goal invocation. While the relative simplicity of this solution and the additional source of parallelism make it attractive in principle, keeping multiple solutions around simultaneously obviously tends to complicate data storage management and use up excessive amounts of it. Moreover, the additional parallelism often leads to combinatorial explosion of the search space.

[1]Department of Electrical and Computer Engineering, The University of Texas at Austin; Austin, TX 78712.

[2]Digital Equipment Corporation, Assigned Representative, Microelectronics and Computer Technology Corporation, Artificial Intelligence Program; 9430 Research Boulevard, Austin, TX 78759.

As an alternative, we have presented a parallel abstract machine [7] [8] capable of implementing AND-Parallelism with very similar storage efficiencies and sequential-mode speed to that of the best sequential implementations. This is achieved in part by using backtracking rather than OR-Parallelism in the management of alternative paths in the search tree, and by implementing a stacking strategy with full space recovery on backtracking, as in sequential systems. In this paper we will deal with the problem of finding a suitable backtracking algorithm for this very general model of AND-Parallel execution, which can be implemented with minimum overhead, is compatible with the storage management strategy, and still takes advantage of the information available at run-time regarding goal independence in order to avoid unnecessary backtracking.

The organization of the paper is as follows: first, we will introduce "goal independence" models of AND-Parallelism and present a generalized version of Restricted AND-Parallelism (**RAP**) as a typical representative of this class. An efficient backtracking algorithm will then be elaborated for **RAP**. We will also study a suitable implementation strategy for this scheme capable of retaining most of the efficiency of sequential systems. Finally, some conclusions will be presented.

2 A General Model for AND-Parallelism: Goal Independence

Conery [5] showed how "brute force" exploitation of AND-Parallelism (i.e. the automatic scheduling of a process for every goal in the body of a clause) leads to binding conflicts if the goals involved have variables in common. This can occur even in cases where the goals appear not to share variables at all. Consider the following clause:

```
            crew(X,Y):- pilot(X), radio_operator(Y).
```

During the resolution of a query of the form "`?- crew(X,X).`" (looking for a person who is a pilot and can also operate a radio) X and Y in the clause above are coerced to be the same through unification. Thus, we cannot go ahead and evaluate "`pilot(X)`" and "`radio_operator(Y)`" in parallel (AND-Parallelism) because of the potential of conflicting instantiations for the identical variables X and Y.

Many approaches have been proposed in order to detect and deal with these variable binding conflicts either at compile-time or at run-time. Some of them, attempt to solve these conflicts without variable annotations and with minimal (or no) information from the user [5] [1] [6]. In other approaches, the user is required to annotate some variables or goals in the program in order to identify goals as "readers" or "writers" for each variable. This and other techniques are used in Concurrent Prolog [11], Parlog [2], IC-Prolog [3], Delta-Prolog [10] etc.

Although an interesting issue, we will not be concerned in this paper with the origin of these annotations. Instead, we will concentrate on dealing with how execution proceeds once a set of goals has been determined as being (variable-wise) independent, (i.e. after determining that they can be run in parallel with no conflicts) and, in particular, on how backtracking can still be efficiently supported in such an environment[4] . Consequently, rather than analyzing a particular source-level language, we will focus on an *intermediate code level* useful for a variety of programming languages, and we will pursue development of an efficient execution model for it. This level, which will be discussed in the next section, can be best described as *horn clauses augmented with literal-level conditional control expressions*. Such control expressions can, for example, be generated when a static analysis uncovers parallel execution potential. Alternatively, the source language could provide the user with the syntactic tools to explicitly trigger their generation.

[4]This is in contrast with other approaches [11] [2] where "don't know" non-determinism has been given up in order to improve efficiency or simplify the implementation.

Concerning the character of these expressions, note that in logic programs, the same clause can be used in various ways, depending on the run-time polarity (instantiation state) of interceding variables. Ideally, these expressions should be capable of dealing with the different cases involved, with a minimum of run-time overhead. *Restricted AND-Parallelism* (**RAP**) [6] is a technique which provides this capability by making it possible to choose at run-time between parallel and sequential execution (i.e. to generate several possible execution graphs) based on variable dependency checks. Such run-time determinations are embodied in what has been referred to as *Conditional Graph Expressions* (CGE's). In the next section we will present a generalized version of such a computation model which subsumes DeGroot's original definition of **RAP** and **CGE**'s. It will be the backtracking behavior of this generalized model that we will study in the subsequent sections.

3 Forward Execution

As explained above, **CGE**'s can be used for reducing run-time data dependency analysis overhead for AND-Parallel logic programming systems to a number of simple checks. Herein, a **CGE** is (informally) defined as a series of conditions followed by a conjunction of goals, i.e.:

```
( <CONDITIONS> | goal1 & goal2 & ... & goalN )
```

where "<CONDITIONS>" represents *any number of conjunctions or disjunctions of checks on a* **<variable_list>**. A **<variable_list>** is a collection of variable names which have their first occurrence before (i.e. "to the left of", in Prolog) the <CONDITIONS> field of the current graph expression[5]. In this definition **CGE**'s can appear in the body of a clause in any place a conventional goal may be placed. Therefore they can also appear in a goal position *inside* a **CGE** (nested **CGE**'s). Types of checks which can appear inside <CONDITIONS> are:

- **ground(<variable_list>)**: evaluates to ***true*** if and only if all variables in **<variable_list>** are ground, i.e. they are instantiated to a term with no uninstantiated variables.

- **independent(<variable_list>)**: We associate with each variable its "set of contained variables" (**SCV**), defined as follows: If the variable is instantiated to a fully ground term, the **SCV** is ***empty***. If the variable is uninstantiated, the **SCV** is the singleton containing the variable itself. If the variable is instantiated to a term, and some of its arguments are variables, the **SCV** is recursively defined as the union of the **SCV**'s for each of those variables. The **independent(<variable_list>)** check succeeds if and only if the intersection of all the **SCV**'s associated with each variable in **< variable_list >** is ***empty***[6].

- The logical values **true** and **false**.

Since each of the checks inside **<CONDITIONS>** will evaluate to **true** or **false**, **<CONDITIONS>**, being constructed as conjunctions and/or disjunctions of these checks, will also eventually evaluate to **true** or **false**. The forward semantics of **CGE**'s dictates that:

> *if <CONDITIONS> evaluates to true, then all expressions inside the* ***CGE*** ***can*** *execute in parallel. Otherwise, they* ***must*** *be executed sequentially and in the order in which they appear within the expression.*

[5] i.e. only those variables in the head or in goals to the left of the current **CGE** (including those in a **CGE** the current expression may be nested in) can be checked.

[6] Much more economical independence algorithms (such as DeGroot's [6]) can be used in practice, as long as they are conservative, i.e. they never declare a set of dependent variables as independent (although they may "give up" and declare some variables as dependent rather than traversing very complex terms).

An example will clarify this further. Suppose we have the following clause:

```
f(X,Y) :- g(X,Y), h(X), k(Y).
```

In general, the three goals in the body of **f** (**g**, **h** and **k**) cannot run in parallel because they have variables in common. Nevertheless, if both X and Y are ground when **f** is called, all goals can then run in parallel. This fact can be expressed by using the following **CGE**:

```
f(X,Y) :- ( ground(X,Y) | g(X,Y) & h(X) & k(Y) )
```

According to the forward execution semantics above, this means that X and Y should be checked and, if they are both ground, then **g**, **h**, and **k** can be executed in parallel and execution will proceed to the right of the expression only after all goals inside succeed. Note that this also means that if X and Y are ground but for some reason (for example, lack of free processors) **g**, **h**, and **k** are executed sequentially, this can be done in any order. Otherwise, if X and Y are not both ground, **g**, **h**, and **k** will run sequentially and in the order in which they appear inside the **CGE**. Selection between one mode of execution and the other is done by a simple run-time check. Of course the expression above only takes care of a rather trivial case. A more interesting execution behavior can be extracted from the following expression:

```
f(X,Y) :- ( ground(X,Y) | g(X,Y) & ( indep(X,Y) | h(X) & k(Y) ) ).
```

Now, if X and Y are not ground upon entry to the graph expression, **g** will be executed first. As soon as **g** succeeds, indep(X,Y) is checked in the hope that X and Y will be independent (either because one of them was ground by **g** or because they are still uninstantiated and do not "share" --as they would if **g** had matched against "**g**(X,X)."). If they are still independent then **h** and **k** can run in parallel. Note that if X and Y are ground upon entry of **f** then *all* goals will run in parallel as in the previous expression.

Sometimes it is necessary to express the fact that a number of goals can run in parallel, independently of any other consideration (perhaps because the programmer knows how a procedure is going to be used). This can be easily accomplished by writing **true** in place of **<conditions>** or eliminating the **<conditions>** field altogether. Thus, in the following expressions, g, h, and k can always run in parallel:

```
f(X,Y) :- ( true | g(X) & h(Y) & k(Z) ).

f(X,Y) :- ( g(X) & h(Y) & k(Z) ).
```

This illustrates how **CGE**'s are a superset of other control annotation schemes such as the parallel connective of Delta-Prolog ("/") [10].

4 Backward Execution

We refer to backward execution as the series of actions that follow failure in the head or body of a clause. In normal (i.e. sequential) Prolog execution this involves going back to the most recent point at which alternatives were still unexplored (most recent choice point). This definition is not directly applicable any more if some of the goals in the body of a clause have been executed in parallel: since execution of these goals was concurrent, there is no chronological notion of "most recent" to apply to the different choice points available. Although several sophisticated approaches have been proposed in order to solve this problem [5] [1] they are either not applicable to the semantics of **CGE**'s (and other Goal Independence models) or they involve too much bookkeeping overhead at run-time. In this section we will analyze the different cases involved in the backtracking of **CGE**'s and we will propose a general backtracking algorithm that will handle these cases efficiently, while taking advantage in some cases of goal independence in order to achieve limited intelligent backtracking.

Throughout this analysis we will consider the following annotated clause:

`f(..):- a(..), b(..),` (**<conditions>**| **c(..) & d(..) & e(..)**), `g(..), h(..).`

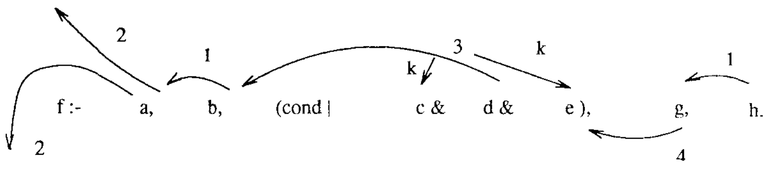

Figure 1: Backtracking cases for a **CGE**

In the trivial case when **<conditions>** is evaluated to false, execution defaults to sequential, and normal (Prolog) backtracking semantics can obviously be applied. We will therefore shift our attention to the cases where **<conditions>** evaluates to **true**. We illustrate in figure 1 the different backtracking situations through back arrows annotated by case numbers, where the cases are the subject of the following text.

Conventional Backtracking:

- **Case 1**- This is the trivial case in which backtracking still remains the same as for sequential execution. For example, if **b** fails and **a** still has alternatives, or if **h** fails and **g** still has alternatives.

- **Case 2**- This is also a trivial case: if **a** fails, the next alternative of **f** will be executed next. If there are no more alternatives for **f**, then **f** will fail in its parent and we recursively deal with the failure at that level.

Conjunctive failure; "inside" backtracking:

- **Case 3**- This is the case if **c**, **d**, or **e** fail while the body of the **CGE** is being executed the first time through (i.e. we are still "**inside**" the **CGE**).

  Suppose **d** fails. Since we are running in parallel, we know that **<conditions>** evaluated to true. This means that **c**, **d**, and **e** do not share any uninstantiated variables. Thus, the variable binding that caused the failure of **d** could not have been generated by **c** or **e**. Therefore it would be useless to ask **c** and/or **e** for alternatives and it is safe to kill the processes running **c**, **d**, and **e**, and to backtrack to the most recent choice point before the **CGE** (for example, **b** here). In this way this scheme very simply incorporates a restricted version of intelligent backtracking with only the overhead of remembering that we are "**inside**" the **CGE** when failure occurs.

"Outside" backtracking:

- **Case 4**- This is the most interesting case: we have already finished executing all goals inside the **CGE** -we are "**outside**" the **CGE**- and we fail, having to backtrack into the expression. This is the case if **g** fails.

  First, since this information will prove very useful, we will assume that processors not only report eventual goal resolution success, but also whether unexplored alternatives still remain for this goal. It will be shown how such information can be used in our context to simply extend the conventional backtracking algorithm to one that deals with **CGE**'s:

  - If **g** fails and none of the **CGE** goals has unexplored alternatives, we will backtrack to **b** just as we would in the sequential execution model.

- If **g** fails and one or more **CGE** goals still has unexplored alternatives, our object will be to establish a methodology whereby all the combinations of those alternatives will have a chance to be explored, if needed, before we give up on the whole **CGE** and backtrack to alternatives prior to it. The methodology we chose is one that will generate those alternatives *in the same order as that produced by naive sequential backtracking*. The idea is then to reinvoke the process which corresponds to the first goal with alternatives found when scanning the **CGE** *in reverse order* (i.e. reinvoking the "rightmost" goal with alternatives). This process will then, in turn, report either **success** (with or without pending alternatives) or **failure**.

    - If **failure** is reported, we simply perform the next invocation in the order described above. Of course when a **failure** is reported by the leftmost goal with alternatives in the **CGE**, we give up on the whole expression and backtrack as in **Case 1** above.

    - If **success** is reported, then we shift into forward AND-Parallel execution mode and *trigger the parallel evaluation of all the goals, if any exist, to the right of the succeeding one in the* **CGE**. Note that, if such goals do exist, they will be started from scratch since the last thing they would have reported would have been a **failure**, which we will assume here would have caused the termination of the corresponding process.

Note how the approach described above extends the "most recent choice point" backtracking model to a parallel execution model, preserving the generation of all "tuples" and offering parallel forward execution after backtracking. Also, goal ordering information provided by the user or by the compiler is preserved, and used in tuple generation. Alternatively, sometimes we might not be interested in generating all possible tuples for a conjunction of independent goals. Instead we might be interested in generating only one and "committing" to it. This can be easily annotated by including a "cut" after the **CGE** or by substituting the "|" in the **CGE** by "!".

In the above, we presented the AND-Parallel model backtracking algorithm through the use of a specific example. The general algorithm can be described as follows:

- Forward Execution: *During forward execution leave a choice point marker (***CPM***) at each choice point (traditional sequential mode) and a parallel call marker (***PCM***) at each* **CGE** *which evaluates to true (i.e. each* **CGE** *which can actually be executed in parallel). Mark each* **PCM** *as* **"inside"** *when it is created, trigger the parallel resolution of the* **CGE** *goals, and change the* **PCM** *mode to* **"outside"** *when all those goals report success.*

- Backward Execution: *When failure occurs, find the most recently created marker (***PCM** *or* **CPM***). Then:*

    - *If the marker is a* **CPM**, *backtrack normally (i.e. as in sequential execution) to that point.*

    - *If the marker is a* **PCM** *and its value is "inside", cancel ("kill") all goals inside the* **CGE**, *fail (i.e. recursively perform the Backward execution).*

    - *If it is a* **PCM** *and its value is "outside", find the first goal, going right to left in the* **CGE**, *with pending alternatives which succeeds after a "redo", and then "restart" all goals in the* **CGE** *"to its right" in parallel. If no* **CGE** *goal is found to succeed in this manner, fail (i.e. recursively perform the Backward execution).*

We have not mentioned nested **CGE**'s until now in order to make the discussion clearer. However, the algorithm works just as nicely with nested **CGE**'s, if it is applied recursively. A simple way of proving this intuitively is to "unravel" the recursive treatment of a nested **CGE** into treatment of a "dummy" goal whose corresponding clause simply embodies the nested **CGE**. The algorithm also turns out to be very simple to implement at the abstract machine level. This will be clear when we present the implementation scheme in the following section. Other special cases will be covered then. In particular we will see how backtracking in the case where some of the goals which could have been executed in parallel are executed locally in a sequential way (e.g. due to a lack of resources) fits trivially within the same scheme[7].

5 Efficient Implementation of the Algorithm

Although logic programs can present considerable opportunities for AND-Parallelism, there are always (determinate) code segments requiring sequential execution. A system which can support parallelism while still incorporating the performance optimizations and storage efficiency of current sequential systems is thus highly desirable. This is the approach taken in our design: to provide the mechanism for supporting forward and backward execution models for AND-Parallelism as extensions to the ones used in a high performance Prolog implementation. This has two clear advantages: first, sequential execution is still as fast and space efficient as in the high performance Prolog implementation (modulo some minimal run-time checks); second, the model is offered in the form of *extensions*, which are fairly independent, in spirit, of the peculiarities of that implementation. Therefore, the approach described here is applicable to a variety of compilation/stack based sequential models.

5.1 Implementing Backtracking in Sequential Systems

The Warren Abstract Machine (WAM) [12] is an execution model coupled with a host of compilation techniques leading to one of the most efficient implementations of Prolog today. Before we present our strategies for implementing **CGE** based AND-Parallelism with the associated backward execution mechanism, we will review here summarily the backtracking mechanism of the WAM since that will constitute our starting point.

In the WAM, backtracking is accomplished through the use of *choice point* frames. A *choice point* is created when the first of a sequence of alternative clauses is entered. It contains all the necessary information needed to restore the state of the machine and pick up the next alternative clause when it becomes necessary to do so. This is the case when we have a failure (e.g. when an invoking goal does not find a matching clause head). Backtracking at that point is accomplished by simply locating the most recent *choice point* (pointed to by register B), restoring the machine state from its content, and restarting from there with the next alternative. This can be seen in figure 2 where the following data areas are shown:

- The *Stack*: where *choice points* and *environments* are created, updated and discarded as needed. Only *choice points* will be shown and discussed here since we want to concentrate on the backward execution model.

- The *Heap*: where data structures and long-lived global variables are created, updated, and discarded (upon backtracking).

- The *Trail*: where variables getting instantiated, but potentially needing undoing such instantiations, are remembered (one entry per such variable).

[7] We call the approach described in this section "*point backtracking*". In "*streak backtracking*" literals to the right of the one being reinvoked are restarted in parallel with this reinvocation. Due to space limitation we will have to avoid discussing here *streak backtracking* or possible optimizations for these approaches.

Figure 2 corresponds to the execution of the clauses in the following example (labels have been given to the different clauses involved):

```
procedure a:                       procedure b:
a1:   a :- b, c, d, e.             b1:   b :- ..., ..., ... .
a2:   a :- b, c, d, e.             b2:   b :- ..., ..., ... .
a3:   a :- b, c, d, e.             b3:   b :- ..., ..., ... .

procedure c:                       procedure e:
c:   c :- ..., ..., ... .          e1:   e :- ..., ..., ... .
                                   e2:   e :- ..., ..., ... .
procedure d:                       e3:   e :- ..., ..., ... .
d:   d :- ..., ..., ... .
```

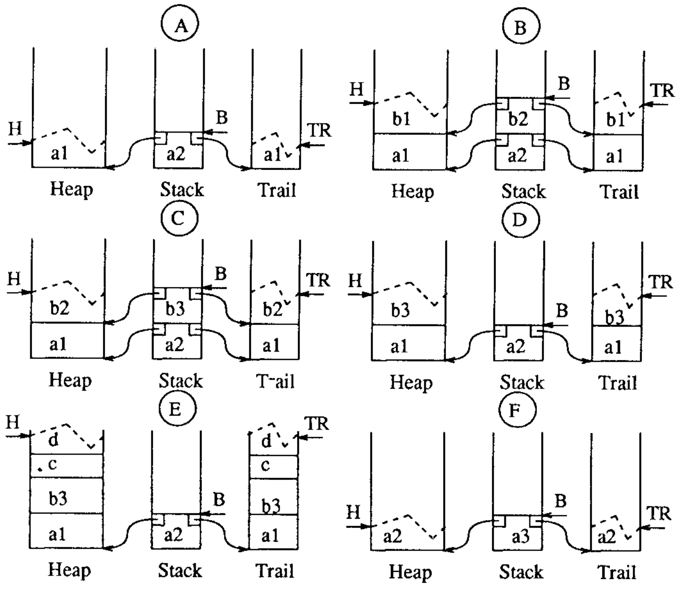

Figure 2: Choice Point Based Backtracking in Sequential Systems

Upon entering *procedure a:*, since **a** has alternatives, we create the corresponding *choice point* needed in the event of backtracking back to this point. Execution of **a** then starts with the first alternative *a1:*. This situation is depicted in figure 2-A. We show here only the following information included in the *choice point* (other information will be skipped for the sake of brevity):

- A pointer to the next unexplored alternative clause *a2:*.
- The value of the *Heap* pointer in register **H** at the time this *choice point* was created.
- The value of the *Trail* pointer in register **TR** at the time this *choice point* was created.

When the head of *a1:* unifies successfully with the invoking goal, *procedure b:* is entered. Again a *choice point* is created, since **b** also has alternatives (figure 2-B). Suppose now that some goal fails in the body of *b1:*, and that no more choice points have been created. The following sequence of actions takes place resulting in *backward execution* (this is illustrated in figure 2-C):

- The most recent *choice point* is fetched through register **B**'s content.
- The top of the *Heap* pointer (register **H**) is reset to the value saved in the fetched *choice point*. This will discard all the data just made obsolete by the failure that caused the backtracking.

- The variables remembered through entries located between the current top of the *Trail* stack and the *Trail* pointer saved in the fetched *choice point* are reset to uninstantiated. This is done because the instantiations being reset were made obsolete by the failure that caused the backtracking. Of course the top of the *Trail* pointer (register **TR**) is also reset appropriately.

- Finally, the next alternative *b2:* indicated in the *choice point* is picked up and execution proceeds from there. We also want at this point to indicate that the next alternative clause is *b3:* by updating the *choice point* appropriately.

If **b** should fail again, we would repeat the above sequence of actions, and start execution of *b3:*. However this time there are no more alternatives for *procedure b:*. This means that the *choice point* associated with *procedure b:* should be discarded and register **B** should be reset to the most recent one prior to the one being discarded. This is only possible if the *choice points* are chained together (This is one of the information items that we are not showing in the *choice point* frames illustrated in figure 2).

In figure 2-E we depict the situation after *b3:* and *c:* have succeeded, and we are executing *d:*. Note that since neither *c:* nor *d:* have alternatives, no more choice points have been created on the *Stack*. Therefore, if *d:* should fail at this point, the general backward execution model using the current most recent *choice point* (fetched through register **B**) would correctly take us to alternative clause *a3:*. This is shown in figure 2-F. Some interesting points to be noted are:

- This implementation achieves efficient garbage collection of *Heap* space upon backtracking: all data created there during forward execution are discarded automatically by appropriately resetting register **H**.

- Identifying the most recent *choice point* is immediate, since it is always pointed to by register **B**.

- *Choice points* are only created when they are needed (i.e., when the clauses have alternatives) and they are discarded efficiently when they are not needed any more.

5.2 Implementing Backtracking in AND-Parallel Systems

As stated before, our objective is to implement an AND-Parallel system with the associated backward execution mechanism presented earlier in section 4, while still preserving the efficiency present in sequential implementations similar to the WAM reviewed above. Our conceptual starting point is that a parallel execution of AND-siblings is going to correspond to a parent process controlling children processes handling independently the execution of the parallel siblings. Also processes have their own execution environments (*Stack*, *Heap*, *Trail*, as well as a machine state). The allocation of processes to processors is of course subject to the availability of such parallel system resources. One of the natural extensions to such a general model is that the parent process could execute one or more children processes instead of just idling while waiting for other children processes' responses: this will be discussed in more detail in section 5.3 on local execution of parallel goals, showing how the existing data areas (*Stack*, *Heap*, and *Trail*, etc.) can be shared for this purpose.

The control structure that the parent uses for its supervisory task will be referred to as a "parallel call" frame (*Parcall frame* in short) and will be located in the parent process's *Stack* (therefore three types of frames can now be found there: *Environments*, *Choice Points*, and now, *Parcall Frames*). The most recent *Parcall Frame* is pointed to by register **PF**. *Parcall Frames* are created when a **CGE** evaluates to **true**, hence clearing the way for the parallel execution of the **CGE**'s sibling literals. The Parcall frame, among other information, contains the following items important for our discussion here[8]:

[8]See [8] or [7] for other details on this subject.

- One slot for each of the sibling literals inside the **CGE**, consisting of the following fields:
    - the Id of the child process corresponding to this literal
    - completion status of the process (i.e. *processing, succeeded with pending alternatives, succeeded with no alternatives*, or *failed*).
- A flag indicating whether we have just entered the **CGE** or whether we are backtracking into it after the initial entry and at least one successful exit. This is a materialization of the "inside"/"outside" indication discussed in the backtracking algorithm in section 4.

In addition to these slots, a *Parcall Frame* contains other information needed for process synchronization, as well as in the event of backtracking out of a **CGE** to a preceding literal.

We will now show how the introduction of *Parcall Frames*, their relationship to *Choice Points*, and the manipulation of both types of frames will materialize the algorithms introduced in section 4 and make it possible to manage both forward and backward execution as a natural extension to the WAM model. First we will define two types of failure:

- *Local Failure*: the local processor fails while executing a goal, and
- *Remote Failure*: a "Failure" message is received from a child process.

Now our extended backward execution mechanism is based on recognizing, when either type of failure occurs, whether a *Choice Point* or a *Parcall Frame* is more recent (comparing registers **B** and **PF**). The algorithm then follows:

- If *Local Failure*, then:
    - If $\mathbf{B} > \mathbf{PF}$ then perform the normal *Choice Point* backtracking.
    - If $\mathbf{PF} > \mathbf{B}$ then find the first[9] *Parcall Frame* child process slot with pending alternatives to respond successfully to a "redo" message. When such a process is found, invoke the parallel execution of all the literals that correspond to the following slots, getting us back in (parallel) forward execution again. If none succeeds, fail by recursively performing this backward execution algorithm in a "local failure" mode.
- If *Remote Failure*, then, knowing definitely that $\mathbf{PF} > \mathbf{B}$ and that we are in the "inside backtracking" case (that is until we introduce next section's optimization):
    - "Kill" all goals in the *Parcall Frame*, fail by recursively performing this backward execution algorithm in a "local failure" mode.

The following example will illustrate the above algorithm. Suppose the clauses for "a" in the example in the previous section were annotated in the following way (with embedded **CGE**'s):

[9] Slots should always be scanned in the same order, e.g. from the higher addressed ones (hopefully corresponding to rightmost ones in the **CGE**) to the lower addressed ones. If parallel goals are allowed to execute locally (next section's optimization), then the order of scanning has to be from most recently executed goals (by remote processors) to less recent ones (see section 5.3).

```
procedure a:
a1:   a :- ( cond1 | b & c & d ), e.

a2:   a :- ( cond2 | b & c & d ), e.

a3:   a :- ( cond3 | b & c & d ), e.
```

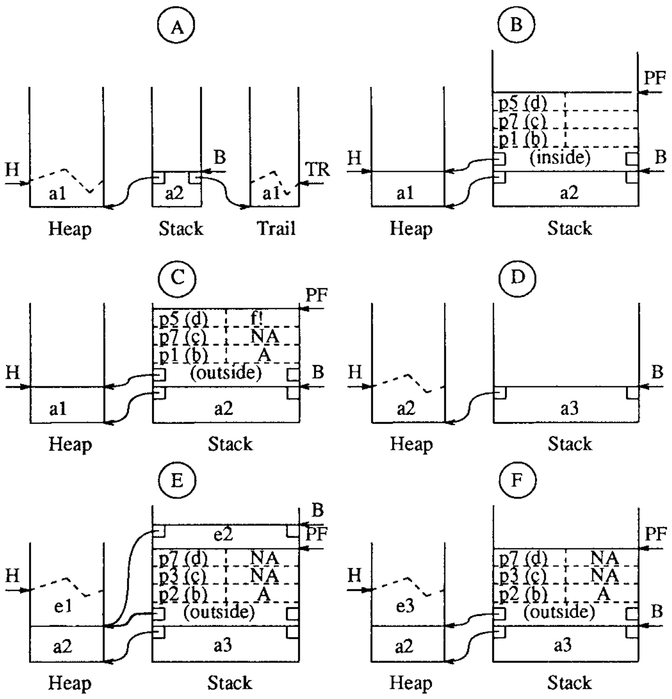

Figure 3: CP/Parcall Frame Based Backtracking in AND-Parallel Systems

Figure 3 illustrates the execution of this example in parallel. Execution of **a** in the "parent" process starts exactly as in the sequential case (figure 2-A vs. figure 3-A). If **cond1** failed, execution would proceed just as in figure 2. On the other hand, if **cond1** succeeds, a *Parcall Frame,* initialized to "inside" is created, with slots for **b**, **c**, and **d**. This is illustrated in figure 3-B where these goals have been "picked up"[10] by **p1**, **p7**, and **p5** respectively (the *Trail* is omitted in both the diagrams and the discussions for the sake of clarity). At this point the parent process simply waits for all goals to update their slot's *completion status* field . With the *Parcall Frame* still flagged as "inside", if one of the goals returns failure (*Remote* Failure)) we can backtrack "intelligently" to the last *Choice Point* before the *Parcall Frame.* In figure 3-C, **p5** returned failure for **d** (**p1** and **p7** returned with success, with **p1**'s success qualified as with pending alternatives, i.e. there is a *Choice Point* in **p1**'s *Stack*). Since the corresponding *Parcall Frame* is still flagged as "inside", an "unwind" message is sent to **p7** and **p1** (thus disregarding the alternatives in **b**), and execution is continued with the next alternative of **a** (figure 3-D).

The next two parts of figure 3 illustrate "outside" backtracking. In figure 3-E we have a situation similar to that in figure 3-B. Processors **p2**, **p3**, and **p7** "picked up" the goals but this time they all returned successfully (b still having alternatives). At this point we succeed the whole **CGE** by changing

[10]All goals in the parallel call are pushed on to a special stack as soon as the parallel call is entered (the checks succeed). From there they can be picked up by other processors or, as we will consider in the next section, by the local one.

the status of the *Parcall Frame* to "outside", and move on to goal **e**, pushing a choice point (since **e** has alternatives), and finally entering clause *e1:*. If *e1:* fails, we will then use the available choice point to try *e2:* (*Local Failure*; **B** $>$ **PF**). Figure 3-F illustrates the situation if *e2:* also fails: the *Choice Point* has been deallocated and we are executing *e3:*.

Note that in the event of a *local* failure now, the last *Parcall Frame* is more recent than the last *Choice Point* (PF $>$ B) and, since its status is "outside", the corresponding backtracking algorithm will be run on it: select the first goal with alternatives (**b**), send a "redo" to it (to **p2**, which will execute it by making use of the *Choice Point* on top of its local *Stack*, just as if a local failure had occurred). If **p2** now returns failure, since there are no more slots with alternatives in the *Parcall Frame*, we will deallocate it (after sending "unwind" messages to the child processes corresponding to all the slots, so that their *Heaps* will be deallocated and their *Trails* unwound) and use the next entry on the *Stack* (**a**'s choice point) to backtrack to *a3:*. If, on the other hand, **p2** had returned success, we would invoke the parallel execution of all the goals corresponding to the following slots, hence "shifting gears" to "Forward Execution". Note that we can safely assume that the **CGE** will be successfully exited at this point since those goals are being redone from scratch and we know that they have succeeded in the past!

5.3 Local Execution of Parallel Goals

One obvious optimization to the scheme above is to let the local processor pick up some of the goals in the *Parcall Frame* and work on them itself, instead of just idling while waiting for children processes responses. This is very important in that it allows the generalization of the architecture to any number of processors (including a single one). Such scalable systems could then run parallel code with "graceful" performance improvement or degradation depending on the available resources. Also, a single processor would run the parallel code at comparable speed to equivalent sequential code, while still taking advantage of the opportunity for "intelligent backtracking" present in "inside" backtracking.

In a multiprocessing system, local execution of parallel goals can be accomplished by creating a new process which will pick up the goal. Figure 4 shows a more efficient way of handling the execution of parallel goals locally, by stacking them on the local stack much in the same way as they would be in a sequential implementation. In figure 4-A, *b1:* has been immediately "picked up" by the local processor (and the corresponding slot has been marked accordingly -- "=*=") while **c** and **d** have been "picked up" by **p7** and **p5**, as in figure 3-B. Execution of the goal taken locally proceeds as normal (figure 4-B), but note that the *Parcall Frame* is still marked as "inside". In this figure **p5** has returned (with *no alternatives*) and **p7** is still working on its goal. In the event of either a local or a remote failure now, "inside" (i.e. "intelligent") backtracking would occur (as in figure 3-D). However, this would only be triggered locally if **b** runs out of alternatives. A first failure in *b1:* in figure 4-B would use the *Choice Point* and continue with *b2:*, just as if it were being executed remotely.

If all goals succeed, we will continue with **e**, data structures and *Choice Points* being again simply pushed on top of their respective areas (*Heap* and *Stack*, figure 4-C). "Outside" backtracking will work in a similar way as before, but with the difference that *goals executed locally will always be backtracked first*: in figure 4-C, if **e** runs out of alternatives, we will try all the alternatives of **b** before using the *Parcall Frame*. This is perfectly valid, as long as it is used consistently, since the order of execution is immaterial inside a parallel call. The *Stack* status of figure 4-C is therefore equivalent to the one found while executing the following clause using the scheme described in the previous section:

```
a :- ( c & d ), b, e.
```

Figure 4-D depicts "outside" backtracking after all goals executed locally have run out of alternatives. We are executing *e3:* after *b3:* (both choice points have been discarded). If failure occurs in *e3:*, we will find the *Parcall Frame* above any choice points, and execute the "outside" algorithm. In this case, since no goals in the *Parcall Frame* have alternatives, we will simply discard the *Parcall Frame* itself (sending "unwind" messages to **p5** and **p7**) and try the next alternative of **a** (*a2:*) as in figure 3-D.

An interesting situation occurs if external failure arrives while the local processor is executing a goal

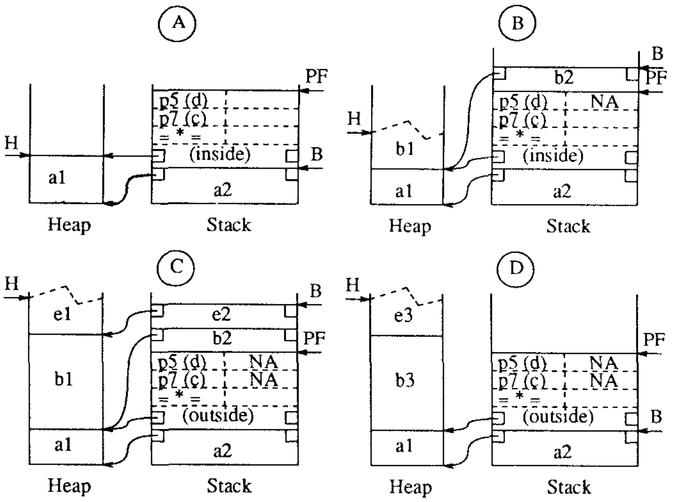

Figure 4: CP/Parcall Frame Based Backtracking With "Local Goals"

from the parallel call, and this goal in turn has generated other *Parcall Frames*. Suppose that in figure 4-B execution of *b1:* has pushed other *choice points* and *Parcall Frames* on the *Stack*. If **p7** (**c**) returns at this point with failure, all those entries, and their corresponding data structures (in the *Heap*) have to be deallocated. This turns out to be simple if **p7** provides the value of the PF pointer for the *Parcall Frame* containing the goal failing (it can be "picked up" with the goal). Then we only need to use the backtracking information in that Frame to recover all space (i.e. just above *a1:* for the *Heap* in figure 4-B). Of course, all processes started by the execution of **b** need to be cancelled. This is also easily accomplished by following the chain of *Parcall Frames*, from the one on top to the one given by **p7**, sending "kill", "unwind" etc. messages to all slots that are not marked local ("==*=="). This is very similar to what a processor has to do when it *receives* a "kill" message.

In summary, an algorithm along the same lines as the one presented in the previous section can be used when **CGE** goals are executed locally, provided it is adapted to handle the extra special cases involved:

- If *Local Failure*, then:
    - If **B** $>$ **PF** then perform the normal *Choice Point* backtracking.
    - If **PF** $>$ **B** and the status of the *Parcall Frame* is "inside", "kill" all goals in the *Parcall Frame* (by sending "kill"/"unwind" messages to all non-local slots in this *Frame*; local goals will be deallocated automatically by the local trimming of the stacks) and fail by recursively executing this algorithm in a *Local Failure* mode.
    - If **PF** $>$ **B** and the status of the *Parcall Frame* is "outside", then find the first[11] parcall frame child process slot with pending alternatives to respond successfully to a "redo" message. When such a process is found, invoke the parallel execution of all the literals that correspond to the following slots, and of all those literals which were executed locally[12] . If none succeeds, fail by recursively executing this algorithm in a *Local Failure* mode.

[11]The correct scanning order now is *opposite to that in which the goals were picked up by remote processors*. A simple way of following this order by making use of an extra field in the child process slot is shown in [8].

[12]Note that all local goals have been completely backtracked before we arrive at this point.

  - If there are no choice points or *Parcall Frames* available, report failure to parent.

- If *remote failure*, then:
  - If the **PF** value received is the same as the current one: we are in a similar case to the second one above.
  - If the **PF** value received is lower than the current one: follow chain of *Parcall Frames* "killing" dependent processes up to and including referred Frame; fail by recursively executing this algorithm in a *Local Failure* mode.

Note that although the description is lengthy because of the many cases involved, the abstract machine can select the appropriate case using extremely simple arithmetic checks ($\mathbf{B} > \mathbf{PF}$ or $\mathbf{B} < \mathbf{PF}$; Status = 1 or 0) and the actions are in any case very simple and determinate. Backward execution can be performed in parallel (i.e. unwinding of *Trails*, killing of descendants etc.) with very little overhead. Then forward execution is resumed also in parallel. Also note that, since in this model local goals are backtracked first, and there is no a priory knowledge of which goals will be executed locally, the order in which solutions are produced depends on run-time factors, even though *all* solutions will still be produced. We are also considering a slightly different model where any compile-time established order can be preserved. A description of such a model will be reported elsewhere.

6 Conclusions

In the previous sections we have presented an approach to AND-Parallel execution of logic programs, goal independence, which characterizes models such as our generalized version of restricted AND-Parallelism. We have then proposed a series of algorithms for management of backtracking in this class of AND-Parallel execution models, offering an efficient implementation scheme and some examples to illustrate its operation. We argue that this solution cleanly integrates one form of AND-Parallelism with the implementation technologies of high performance Prolog systems with efficient data storage management similar to the Warren Abstract Machine. Also, a form of restricted intelligent backtracking is provided with virtually no additional overhead. "Soft" degradation of performance with resource exhaustion is attained: even a single processor will run any parallel program while still supporting restricted intelligent backtracking when goals are independent.

The discussions in this paper concentrated on the backtracking algorithms. We have also covered other areas of the design of an AND-Parallel implementation, such as an instruction set and Abstract Machine [8], goal scheduling and memory management issues, and a more detailed system architecture. The reader can find more detailed information regarding some of these subjects in [7].

References

[1] J.-H. Chang, A. M. Despain, and D. DeGroot.
AND-parallelism of Logic Programs Based on Static Data Dependency Analysis.
In *Digest of Papers of COMPCON Spring '85*, pages 218-225. 1985.

[2] K. Clark and S. Gregory.
PARLOG: A Parallel Logic Programming Language.
Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, May, 1983.
University of London.

[3] Clark, K.L. and G. McCabe.
The Control Facilities of IC-Prolog.
Expert Systems in the Micro Electronic Age.
Edinburgh University Press, 1979.

[4] J.S. Conery and D.F. Kibler.
Parallel Interpretation of Logic Programs.
In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture.*, pages 163-170. October, 1981.

[5] J.S. Conery.
The AND/OR Process Model for Parallel Interpretation of Logic Programs.
PhD thesis, The University of California at Irvine, 1983.
Technical Report 204.

[6] Doug DeGroot.
Restricted And-Parallelism.
Int'l Conf. on Fifth Generation Computer Systems , November, 1984.

[7] Manuel V. Hermenegildo.
A Restricted AND-parallel Execution Model and Abstract Machine for Prolog Programs.
Technical Report PP-104-85, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, 1985.

[8] Manuel V. Hermenegildo.
An Abstract Machine for Restricted AND-parallel Execution of Logic Programs.
In *Proceedings of the 3rd. Int'l. Conf. on Logic Programming.* Springer-Verlag, 1986.

[9] Kowalski, R.A.
Predicate Logic as a Programming Language.
Proc. IFIPS 74 , 1974.

[10] Luis M. Pereira and Roger I. Nasr.
Delta-Prolog: A Distributed Logic Programming Language.
In *Proceedings of the Intl. Conf. on 5th. Gen. Computer Systems.* 1984.
Japan.

[11] E. Y. Shapiro.
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003, ICOT, January, 1983.
Tokyo.

[12] David H. D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.