An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs

M. V. Hermenegildo
Department of Electrical and Computer Engineering
The University of Texas at Austin; Austin, TX 78712

Abstract

Although the sequential execution speed of logic programs has been greatly improved by the concepts introduced in the Warren Abstract Machine (**WAM**), parallel execution represents the only way to increase this speed beyond the natural limits of sequential systems. However, most proposed parallel logic programming execution models lack the performance optimizations and storage efficiency of sequential systems. This paper presents a parallel abstract machine which is an extension of the **WAM** and is thus capable of supporting AND-Parallelism without giving up the optimizations present in sequential implementations. A suitable instruction set, which can be used as a target by a variety of logic programming languages, is also included. Special instructions are provided to support a generalized version of "Restricted AND-Parallelism" (**RAP**), a technique which reduces the overhead traditionally associated with the run-time management of variable binding conflicts to a series of simple run-time checks, which select one out of a series of compiled execution graphs.

KEYWORDS: LOGIC PROGRAMMING, PARALLEL PROCESSING, WARREN ABSTRACT MACHINE, RESTRICTED AND-PARALLELISM, PROLOG.

1 Introduction

The execution speed of sequential logic programming systems has been constantly improving since Warren's Prolog interpreter/compiler for the DECsystem-10 [14] proved the usefulness of logic as a practical programming tool [11]. Pipelined architectures [16] and microprogrammed Prolog machines [7] seem to be approaching the 1Mlips (Logic Inferences per Second) line. Most of these implementations are based on the Abstract Machine recently proposed by Warren [17] (the "**WAM**") which has made very fast and space efficient systems possible. Yet, in order to meet the requirements of applications as ambitious as those contemplated in next generation computer systems, vast improvements in performance are still needed. The source for performance improvement beyond the natural limits of sequential systems is *executing logic programs in parallel.*

Of the different sources of parallelism present in logic programs [4], in this paper we will study **AND-Parallelism** because, among other reasons, it offers promising results even for highly deterministic programs. Although the management of AND-Parallelism has traditionally involved excessive run-time overhead, we hope to show throughout this paper that it can in fact be implemented very efficiently by applying similar techniques to those brought by the **WAM** to the sequential logic programming implementation arena. We will present an abstract machine capable of AND-Parallel

execution while still supporting most of the optimizations present in current sequential systems. Its data areas, registers, operation, and instruction set will be described. The high overhead previously associated with the resolution of variable binding conflicts [5] will be greatly reduced in this model by providing special instructions to support a generalized version of Restricted AND-Parallelism (**RAP**) [6] [10]. However, other approaches can also be supported with the same basic instruction set.

Organization of the paper is as follows: in the next section we will explain the problems associated with variable binding conflicts and **RAP** will be presented as an efficient technique for detecting and dealing with them. In the following section we will review some of the concepts introduced by the **WAM**. We will then describe the extended AND-Parallel abstract machine, specifying data areas, instructions, and operation. An example of compiled parallel code will be fully commented on in order to further clarify the function of each instruction. Finally we will present some conclusions and suggestions for future work.

2 Towards AND-Parallelism: resolving binding conflicts

Consider the following clause:

```
child(X,Y,Z):- father(Y,X), mother(Z,X).
```

During the resolution of a query of the form `"?:- child(X, peter, mary)."` we cannot go ahead and evaluate `"father(peter,X)"` and `"mother(mary,X)"` in parallel (AND-Parallelism) because they will independently find a value for X but both values may not be the same, as needed by the semantics of the clause: a "binding conflict". The simplest course of action in this case is simply to evaluate the goals involved sequentially, with conventional backtracking.

Many approaches have been proposed in order to detect and deal with these variable conflicts either at compile-time or at run-time. In some of them 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 [15], Parlog [2], IC-Prolog [3], Delta-Prolog [13] etc. Other approaches attempt to solve binding conflicts without variable annotations and with minimal (or no) information from the user, using either a complex run-time system (such as Conery's [5]) or an extensive compile-time analysis (such as Chang's **SDDA** [1]).

Restricted And-Parallelism (**"RAP"**) [6] is a technique which deals with these conflicts by *combining a compile-time analysis of the clauses involved, with simple checks on variables at run-time.* While analyzing the example above, a **RAP** compiler would find that `"father(Y,X)"` and `"mother(Z,X)"` cannot in general run in parallel, but that it is possible to execute them concurrently if the clause happens to be called with the first argument (X) being "ground" (i.e. fully instantiated), and the other two (Y and Z) being independent (i.e. they do not "share"). This information can be encoded in the form of a "Conditional Graph Expression" (**CGE**), and the clause rewritten as shown below. This "rewritten" clause can represent a clause which was annotated by the user and/or an intermediate step of the compiler if it is capable of performing the analysis described above[2]:

```
child(X,Y,Z):-( ground(X), indep(Y,Z) | father(Y,X) & mother(Z,X)).
```

The declarative semantics of the clause above remains identical to that of the original clause, but the procedural semantics is now:

- Try to unify `"child(X,Y,Z)"` with the calling goal. If successful,
- Check if `"X"` is ground and if `"Y"` and `"Z"` are independent. In that case, start execution of `"father(Y,X)"` and `"mother(Z,X)"` in parallel.
- If the checks fail, execute `"father(Y,X)"` and `"mother(Z,X)"` sequentially.

---

[2]Note that the definition and syntax of **RAP** and **CGE**s are slightly different than DeGroot's. **CGE**s are shown here *embedded* within the original clause. See [10] for more details.

Thus, the **CGE** embedded within the clause above can generate (depending on the result of the "checks" *at run-time*) two execution graphs: a sequential and a parallel one. Nesting of Conditional Graph Expressions can generate more complicated execution graphs, and the run-time system, while executing the **CGE**, will select different branches of the graphs depending on the results of the checks.

It is interesting to compare **RAP** to other related solutions: Conery's approach would perform all the binding conflict analysis at run-time, with very high execution overhead, while Chang's would perform a data dependency analysis at compile-time but it would only select the worst of all possible cases due to the lack of run-time checks. The **RAP** compromise between run-time and compile-time analysis thus appears as a good choice for implementation and some instructions in the parallel abstract machine will be tailored to support it. However, we believe that the design of the machine and its instruction set is general enough that it can be used as a target by other approaches with only minor modifications.

Backtracking in AND-Parallel Execution

In DeGroot's model, only the forward execution semantics was specified. However, a backward execution semantics is clearly also needed for any implementation. The subject of backtracking in AND-Parallel systems is discussed in [10]. For the sake of completeness we include the following algorithm, taken from [10]. It turns out to be very simple to implement at the abstract machine level, and it offers *restricted intelligent backtracking* with very little overhead:

- *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**[3]*, 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).*

3 An Abstract Machine for AND-Parallelism

Although logic programs can present considerable opportunities for AND-Parallelism, there are always 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 support forward and backward execution of AND-Parallel programs through mechanisms which are extensions to the ones used in a high performance Prolog implementation: the **WAM**. This has several advantages: first, sequential

[3]This is obviously an arbitrary choice, but it is a simple way of keeping track of which goals have been "redone", in order to make sure that all "tuples" are generated. Also note that only goals which have alternatives (i.e. had a *choice point* available after return) need to be sent a redo.

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. Finally, the upward compatibility with **WAM** code makes it possible for a sequential program to run without modification on a single processor, and to make use of existing compiler technology.

3.1 Implementing Sequential Logic: the Warren Abstract Machine

The Warren Abstract Machine (**WAM**) [17] is a remarkably efficient execution model coupled with a host of compilation techniques leading to one of the highest performance implementations of Prolog today. The ideas it incorporates are believed to be a major breakthrough in the design of computational logic systems [12]. Lack of space prevents us from fully describing the **WAM** here. Instead we will only point out those basic concepts which are necessary for the understanding of our extensions. For a complete description of the **WAM** the reader is referred to Warren's original SRI report [17] or to the tutorial on the **WAM** available from Argonne Labs [8].

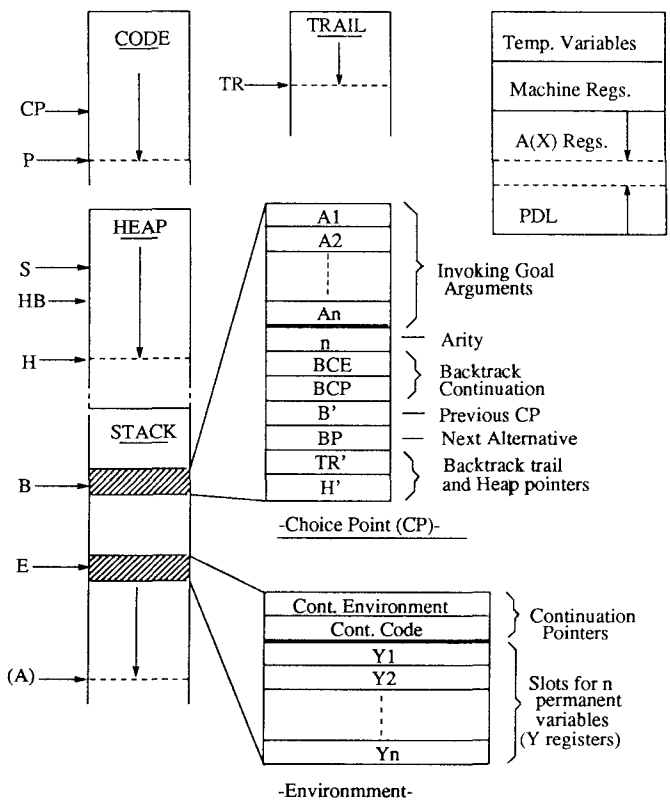

Figure 1: Data areas and registers for the **WAM**

Figure 1 shows a general view of the data areas of the **WAM**. They include the *Code* area, which contains the program *in compiled form*, and three areas operated as stacks:

- The *Heap*: where data structures and long-lived global variables are created, updated, and discarded (upon backtracking). Structure copying (rather than structure sharing) is used in the *Heap*: new structures are pushed on to the *Heap* explicitly, as modified copies of old ones.

- The *Stack*: which contains two types of objects: *environments* and *choice points*.

    - An *environment* is pushed on to the *Stack* every time a clause is entered[4]. It contains a number of value cells which are used to store variables which can be accessed by the goals within the body of the clause or by children clauses called by these goals. It also contains some continuation information which is equivalent to the return address in a subroutine call: it points to the instruction in the body of the calling clause where execution will continue after the called clause finally succeeds. Environments which are no longer needed (for example before the last call in a clause) can be discarded ("last call optimization").

    - A *choice point* is pushed on to the *Stack* when the first clause of a set of alternative clauses is entered. It contains all necessary information to restore the state of the machine and a pointer to the next alternative clause. Upon failure, backtracking is accomplished by simply finding the last *choice point* in the *Stack* (pointed to by register **B**), reloading all machine registers from its contents, and restarting execution at the alternative clause. Resetting the registers takes care of discarding the top of the *Heap* and *Stack* (i.e. discarding variables and structures created since the *choice point*), but there will still be one detail left: we might have done some *variable instantiations* deeper in the data areas which need to be undone upon backtracking. This is taken care of by

- The *Trail*: where variable instantiations which need to be undone are recorded. These entries are used on backtracking to restore the corresponding variables to "uninstantiated". This is called "detrailing" or "unwinding" the *Trail*.

In addition to the data areas (*Code/Stack/Heap/Trail*) there are other elements in the design of the **WAM**: a number of argument registers (called A or X registers) are used for passing arguments when a procedure (i.e. a collection of clauses with the same head functor and number of arguments) is called. There is also a small "Push-Down List" (*PDL*) which is used by the recursive general purpose unification routine.

Prolog programs are compiled into a series of instructions which perform different operations on the above mentioned areas. In order to broadly describe the function of some of these instructions, we will follow a normal procedure call ("goal invocation") sequence: the first step involves loading the argument registers (A1 through An, where n is the number of arguments in the call -the Arity of the procedure) with the appropriate values; *"put"* instructions are used for this purpose. The procedure is then called (*"call/execute"* instructions). Upon entry into a procedure, a *choice point* is created if it has more than one alternative (*"try"* instructions) and then each of the terms in the head of the clause is unified (*"get/unify"* instructions) with the corresponding argument loaded in (or pointed to by) the argument register. If unification does not succeed, failure occurs and backtracking to the last *choice point* will occur as described above. *"Get"* instructions are basically used to encode at compile-time cases where unification defaults to a simple assignment or a set of very simple determinate steps. Because the main activity of a Prolog program is centered around unification of goals with candidate clauses, the simplification of this step results in important performance improvements.

The **WAM** offers many other features designed towards improving speed and space economy, such as

---

[4]This is really only true if the clause has "permanent variables".

retrieval of all used space upon backtracking, last call optimization, "*environment trimming*" etc. Instructions are also provided for supporting the technique of indexing the clauses based on the first argument. This reduces the number of alternatives to be tried and has an important role in improving execution speed and detecting determinate cases.

3.2 Extending the WAM for Parallel Execution

Several issues have to be taken care of in order to extend the sequential **WAM** for AND-Parallel execution. Support has to be provided for the forward execution semantics described in section 2: upon arrival at a parallel call, some scheduling mechanism has to assign available work (i.e. the parallel goals) to the available processors. Also, some data structure has to be provided to keep track of the state of execution of parallel siblings. Of course, this has to be done in an as efficient and unobtrusive as possible way, so that all the performance advantages of the **WAM** are retained. Figure 2 shows one processor of the Restricted AND-Parallel Abstract Machine[5] . Clearly, each "processor" is equivalent to a standard **WAM** except for the addition of a "*Goal Stack*" and the inclusion of "*Parcall Frames*" in the local *Stack*, together with *environments* and *choice points*. These additions will be described in the rest of this section.

Support also has to be provided for the backward execution algorithm being used. Because of space limitations, in this paper we will be mainly concerned with forward execution. However, the basic elements for *local goals first* backtracking [10] are also included in the description for reference.

3.2.1 The Goal Stack

As seen in figure 2, each processor has a private *Goal Stack* (pointed to by **GS**) where goals which are ready to be executed in parallel can be pushed on to. Each entry in the *Goal Stack* is called a *Goal Frame*. A *Goal Frame* contains all necessary information for remote execution of the goal. In particular, it contains the following items:

- **Procedure_name**: points to the first instruction of the procedure to be executed.
- **P(1),...,P(n) registers**: Parameter Registers. They are a copy of the n argument registers for the procedure.
- **#of parameters**: this cell contains "n", the *Arity* of the procedure.
- *Parcall Frame* **Pointer (EPF)**: identifies which *Parcall Frame* this goal corresponds to.
- **Slot #**: identifies which slot in the *Parcall Frame* this goal corresponds to.

When a *parallel call* (a **CGE** whose "checks" succeed) is arrived at, all goals can be pushed on to the *Goal Stack*. Then a goal can be "stolen" from this stack by a "remote" processor, which will copy the parameter registers into its argument registers, load **P** with the address of "Procedure_name", and start execution from there. A goal can also be picked up from its own *Goal Stack* by the **local** processor (the one which just pushed it there), using the same technique.

3.2.2 The Parcall Frame

Entries in the *Goal Stack* completely disappear after they are "picked up" by remote processors. An additional data structure is thus needed in the local processor in order to:

1. keep track during forward execution of the parallel activities of the children processors which "picked up" the goals inside a parallel call,

---

[5] In this paper we will assume one process per processor for simplicity.

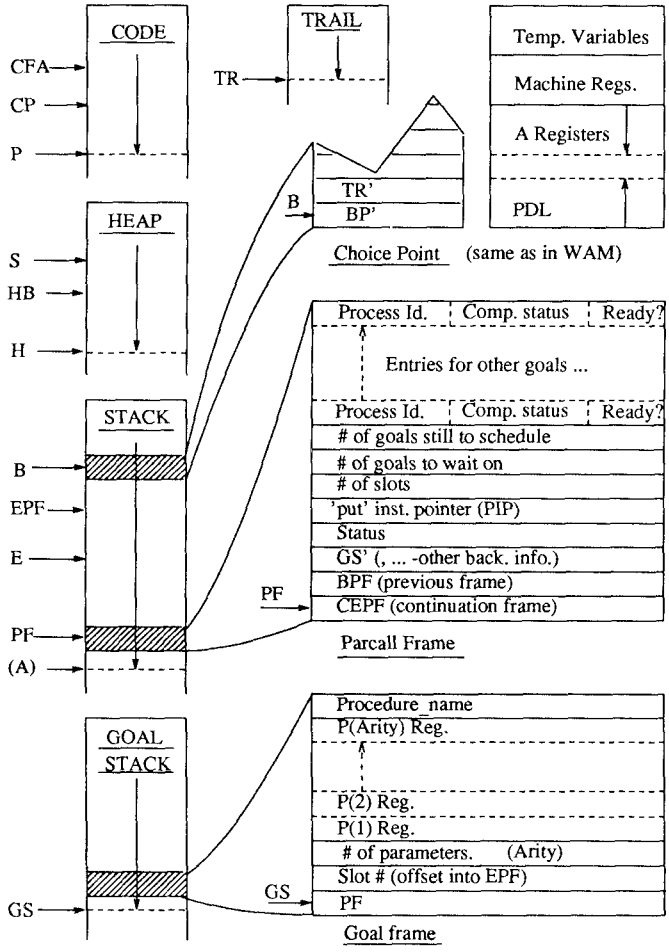

Figure 2: Data areas and registers for one processor of the Extended **RAP-WAM**

2. select the appropriate actions during backtracking.

We will call this structure a "*Parcall Frame*". One *Parcall Frame* is created for each parallel call. For each goal available for execution in parallel (i.e. for each goal pushed on to the *Goal Stack*) within this parallel call, there is one *slot* in the *Parcall Frame*. Each one of these slots has the following fields:

- **Process Id.**: this field contains the id. of the processor which picked up the corresponding goal. If it was the local processor, this field is marked accordingly ("*")[6].
- **Completion Status**: this is a one bit field, set by the corresponding processor when it returns, marking whether it still has alternatives or not.
- **Ready/NotReady**: this is also a one bit field, used (by the "check_ready" instruction) to select the goals that are actually going to be pushed on to the *Goal Stack*. It is used when

[6]If "local goals first" backtracking is used, the order in which the goals are picked up also has to be stored. A simple way of doing this is by recording the current value of the outgoing goals counter described below.

only *some* of the goals inside a parallel call need to be scheduled, as is the case during forward execution after backtracking. When a *Parcall Frame* is created, all Ready bits in all slots are initialized to ready.

In addition to a variable number of "slots", some fixed entries are needed in the *Parcall Frame*:

- **# of goals still to schedule**: this cell is initialized to the number of goals to be executed in parallel. Each time the local or remote processors take a goal from the *Goal Stack* this number is decremented.

- **# of goals to wait on**: this cell is incremented by a remote processor when it "steals" a goal from the local *Goal Stack*. It is decremented every time a processor returns with success.

- **Total # of slots in the** *Parcall Frame*: determines the size of the *Parcall Frame*.

- **Put instructions pointer (PIP)**: this cell contains the address of the first instruction of the first goal in the parallel call and is used to start pushing goals again on to the *Goal Stack* after backtracking. This time though, only those goals whose Ready field is set will be pushed, since all others are skipped by the "check_ready" instruction in front of them. The backtracking algorithm determines which Ready bits are to be set (i.e. which goals will be restarted) and reinitializes the values of the first two cells above to the appropriate value.

- **Status**: this cell marks whether execution of the parallel call corresponding to this *Parcall Frame* has already been completed once (**"outside"** status) or the first pass is still going on (**"inside"** status). This is used to select the type of backtracking [10].

- **GS' (, ...)**: the top of the *Goal Stack* upon entry to the parallel call is saved in this cell so that it can be restored during backtracking[7].

- **BPF**: this is a pointer to the previous *Parcall Frame* used to reset **PF** when the current *Parcall Frame* is discarded *as a result of backtracking.*

- **CEPF**: continuation **EPF**. The value of **EPF** before this *Parcall Frame* is created is saved here. It is used to reset **EPF** after exiting the parallel call.

Parcall Frames are just one more type of object which resides in the local *Stack*, together with *environments* and *choice points*. **PF** is an extra machine register which always points to the last *Parcall Frame*, i.e. the one which will be used for backtracking in the event of failure [10] (much in the same manner as **B** always points to the last *choice point*). **EPF** in turn, always points to the *current Parcall Frame*, i.e. the one being used for the management of scheduled goals.

3.3 General Operation of the Extended RAP-WAM

As stated before, each "processor" (figure 2) is equivalent to a standard **WAM** with a complete set of registers and stacks. This includes the new "*Goal Stack*" and the addition of "*Parcall Frames*" to *environments* and *choice points* in the local *Stack*. Note that there is also a new pointer into the *Code* area (**CFA** --"Check fail address") which points to the code which should be executed if the conditions in the **CGE** fail, i.e. the sequential code.

---

[7]Depending on the particular type of backtracking strategy being used, other backtracking information may also be saved (see [10]).

As soon as processor "steals" a goal (a *Goal Frame*) from another processor's *Goal Stack*, it will start working on it by loading its argument registers from the parameter registers in the *Goal Frame* and fetching instructions starting at the location (procedure address) received. The local stacks will then grow (and shrink) as indicated by the semantics of the standard **WAM** instructions it is executing. It will be the "local" processor for this instruction stream and its data areas will be the "local *Stack*", "local *Heap*", and "local *Trail*", etc. Note though, that the *environments* in its local *Stack* and the data structures in its local *Heap will contain references to the data areas of ancestor processors.* The character of these references will vary depending on the memory organization used in the underlying architecture (i.e. from absolute addresses for uniform addressing space, shared memory architectures to, for example, Pid./remote-address pairs for non-shared memories). Also note that although there might be reading conflicts (two or more processors trying to read the same memory location), there can be no writing conflicts if the **CGE**'s have been generated correctly! The ill-effects of reading conflicts on performance are much easier to avoid than those of writing conflicts, for example by using multiported memories and/or data caching. Also all synchronization is guaranteed by the wait instructions marking parallel call boundaries. This will become more clear after the instruction set has been introduced and an example commented on, but it shows how all program or data dependent control and synchronization issues are concealed within the semantics of the **CGE**'s.

Execution obviously differs from normal **WAM** execution when a parallel call is reached. In this case, a *Parcall Frame* is created in the local *Stack* and its goals are pushed on to the *Goal Stack*, ready to be picked up by the local processor or other remote processors. These remote processors will in turn work on their assigned goals growing their own stacks with references to ancestor stacks. Eventually all dependents of the processor we are looking at will terminate and, if no failures occur, success will be reported to the parent. However, there may be some entries (i.e. *choice points*, if the goal still has alternatives) left in the local *Stack*, some data structures in the local *Heap* that ancestors need to access (the "output" of the procedure), and also some entries in the *Trail*. This is left this way, and when the next goal is received its data structures can be grown above these[8]. This space is only retrieved upon local failure, or if a kill message is received from the parent processor (because of a failure there or in some other related processor), much in the same way as in the sequential **WAM**.

Note that if an appropriate ordering of events is chosen (for example, if processors which still have underlying *Stack* or *Heap* segments only take goals from siblings of their last goal or their dependents) then a kill message necessarily always refers to the last goal executed (i.e. to the last set of structures on the *Stack*) and space is always retrieved from the top of the *Stack* or *Heap* as in the sequential model. Local unwinding of the *Trail* will undo any bindings done outside the local data areas. This unwinding of the distributed *Trail* is done completely in parallel by all the AND-siblings which receive "kill" messages. Also note that with the above mentioned ordering of events, a "redo" message, when received, always refers to the last *choice point* in the local *Stack* and it can be executed just as if a local failure had occurred!

3.4 The Extended RAP-WAM Instruction Set

In the current version, all **WAM** instructions are supported in addition to the new instructions implementing AND-Parallelism, but we will not list the **WAM** instruction set here since it is widely available. Note how, although "check" instructions are somewhat particular to the implementation of **RAP**, all other instructions could be used in any AND-Parallel system.

---

[8]Note that the current pointers into the data areas should be saved at this point (see section 3.6), in order to detect, for example, goal failure (i.e. **B** $<$ top of the stack when the goal was "picked up")

3.4.1 Check Instructions

Check instructions are used to encode the "conditions" in a **CGE**. Two types of checks[9] ("ground" and "independent") and a branch instruction are provided. Note that by combining these, any kind of disjunctions or conjunctions of checks on any number of variables can be expressed:

check_me_else Label

- load check failure address with Label (**CFA**=Label).

check_ground Vn

- dereference register Vn and check to see if its contents are ground. If so, continue with next instruction; otherwise **P**=**CFA** (i.e. branch to Check Failure Address).

check_independent Vn,Vm

- dereference Vn and Vm. If they are independent, next instruction; otherwise **P**=**CFA**.

3.4.2 Goal Scheduling Instructions

These are the instructions used for pushing goals with their arguments on to the *Goal Stack* and for picking up these goals in the local processor:

push_call Procedure_name/Arity,Slot#

- request exclusive access to *Goal Stack*; push on to the *Goal Stack*: "Procedure_name", registers A_{Arity}, $A_{Arity-1}$, ...A_1, "Arity" ("n"), Slot# (i.e. offset from **EPF** for the slot corresponding to this goal), and current **EPF** pointer; release access to *Goal Stack*.

The arguments should be first loaded into the argument (A) registers using normal "put" instructions (as for a conventional "call"). Then, they will be transferred in one block to the *Goal Stack* with the push_call instruction. This eliminates the need for new "put" instructions and minimizes the time the *Goal Stack* is locked.

pop_pending_goal

- if no goals are pending to be scheduled ("# of goals to schedule" in *Parcall Frame* = 0), continue with next instruction; else pop a goal from the local *Goal Stack*.

This instruction is used by the local processor to pop a goal from its own *Goal Stack* for local execution. The corresponding slot in the *Parcall Frame* (as indicated by "Slot #" in the *Goal Frame*) is marked as "local" and the arguments are popped back from the *Goal Stack* to the local argument registers. Then **P** is loaded with the address of "Procedure_name" and execution continues from there. The continuation for forward execution is set to return to this instruction, so that when this goal finally succeeds, the next one can also be popped from the *Goal Stack*. This process continues until there are no more goals left (# of goals to schedule = 0). The next instruction is then executed.

3.4.3 Control Instructions

These instructions take care of the control issues involved in a parallel call: creating and deleting *Parcall Frames*, selecting the goals to schedule and waiting for children to report results.

---

[9] DeGroot's algorithms can be used to efficiently perform the checks. Note that any "conservative" algorithm can be used, i.e. one that never declares two dependent variables as independent although it may give up on complicated terms or long dereferencing chains.

allocate_pcall_frame #_of_slots,M

This instruction creates a properly initialized *Parcall Frame* in the local *Stack* with the correct number of slots. M, the number of "permanent variables" still needed in the *environment* is used to extend the concept of *environment trimming*. **EPF** and **PF** now point to the top of the stack.

check_ready Slot_#,Label

- Check that slot in **EPF**. If not ready, jump to Label; else, continue with next instruction.

Check_ready instructions are used to skip those goals whose slots are marked as "NotReady" in the *Parcall Frame* so that they are not pushed on to the *Goal Stack*. This is useful during backtracking, as only some of the goals inside a parallel call may need to be restarted after failure.

wait_on_siblings

- wait until "# of goals to wait on" in current *Parcall Frame* is 0; then, restore **EPF** from the *Parcall Frame* (**EPF**=**CEPF**), change status to "**outside**" (if it is "**inside**"), and go on to next instruction.

An extension of last call optimization can be implemented in the wait_on_siblings instruction by discarding the current *Parcall Frame* (**PF**=**BPF**) if all slots in it are either "local" or they have no alternatives: the frame is not needed in these cases because there are no goals to backtrack into inside it. However, the Pid's of the processors involved should be "trailed" so that the necessary "unwind" messages are sent to them during backtracking.

3.5 An Example

This example illustrates the code generated by the compiler for a simple clause. The comments provided explain the operation of the instructions involved. Suppose this is the original "Prolog" clause as written by the user in the source program[10] :

```
f(X,Y,Z):- a(X,Y), b(X,Y), c(X,Y), d(X,Y,Z), e(X,Y,Z).
```

The Graph Expression generated by the compiler after its analysis might be:

```
f(X,Y,Z):- a(X,Y), (ground(X,Y) | b(X,Y) & c(X,Y) & d(X,Y,Z)), e(X,Y,Z).
```

Obviously, in this graph expression it is expected that **a** will ground X and Y. In this case, "ground(X,Y)" will succeed and then **b**, **c**, and **d** can run in parallel. Otherwise they will run sequentially and the annotated clause will execute the same instructions as the original one would have in a conventional system. The code that the compiler would generate for the clause above follows. Since there is in general no point in pushing *all* goals in the parallel call on to the *Goal Stack* (the local processor is going to pick one up immediately) one of them (**d**) is called locally without going through the *Goal Stack*. In order to understand the first part of the example, note that at the point of entering this code the calling procedure has already loaded registers A_1, A_2, and A_3 with the arguments for **f**:

```
f/3:                          | (Entry point for procedure f)
     allocate                 | Push an environment on to the stack. It will
                              | have space for "X"(Y3), "Y"(Y2) and "Z"(Y1)
     ------------------------------------------------------------------------
                              | HEAD INSTRUCTIONS: f(X,Y,Z):- ...
     get_variable  "X",A1     | X <- (A1)  Unify (just "get" in this case) the
     get_variable  "Y",A2     | Y <- (A2)  arguments (X,Y,Z) from the parameter
     get_variable  "Z",A3     | Z <- (A3)  registers into the environment.
```

[10] This clause is purposedly chosen so that the code generated is as simple as possible (no "unsafe variables", no special unification instructions) in attention to the reader with no previous exposure to **WAM** code. Also some of the instructions are obviously unnecessary but leaving them there makes it easier to visualize the structure of the code.

```
                              | BODY INSTRUCTIONS: ... :- a(X,Y), ...
     put_value "X",A1         | (X) -> A1  Load argument registers from the
     put_value "Y",A2         | (Y) -> A2  environment for a.
     call  a/2,3              |            Call a.
     -------------------------------------------------------------------------
                              | ... ( ground(X,Y) | ...
     check_me_else SEQ_CODE   | Set the address to branch to in
                              | case the conditions fail (CFA).
     check_ground  "X"        | X ground?  if not go to SEQ_CODE
     check_ground  "Y"        | Y ground?  if not go to SEQ_CODE
     -------------------------------------------------------------------------
                              | The checks succeeded: parallel execution.
     allocate_pcall_frame 2,3 | First, create a Parcall Frame in the stack with
                              | 2 slots (slot 1 for c, slot 2 for b)
                              | (3 is # of perm. vars. -used for env. trimming)
     -------------------------------------------------------------------------
P_I_P:                        | ... | b(X,Y) & ...
     check_ready 2,PUSH_C     | See if slot 2 in Parcall Frame (i.e. "b") is
                              | ready (always true except when backtracking);
                              | else, jump to PUSH_C (skip this goal)
     put_value "X",A1         | (X) -> A1   Load argument registers from the
     put_value "Y",A2         | (Y) -> A2   environment for b.
     push_call b/2,2          | Push call to "b" with its arguments on to Goal
                              | Stack (it can now be "stolen" by another proc.)
PUSH_C:                       |
     check_ready 1,CALL_D     | ... & c(X,Y) & ...
     put_value "X",A1         | (same as b above)
     put_value "Y",A2         |
     push_call c/2,1          |
     -------------------------------------------------------------------------
CALL_D:                       | ... & d(X,Y,Z) ) ...
     put_value "X",A1         | (X) -> A1   Load argument registers from the
     put_value "Y",A2         | (Y) -> A2   environment for d.
     put_value "Z",A3         | (Z) -> A3
     call d/3,0               | "d" is executed locally (normal call)
     -------------------------------------------------------------------------
     pop_pending_goal         | If no goals are pending, next instruction;
                              | else execute remaining goals locally.
     wait_on_siblings         | Wait until all "remote" goals in the
                              | Parcall Frame have returned.
     execute  CALL_E          | Go on to execute "e" (CALL_E).
     -------------------------------------------------------------------------
SEQ_CODE:                     | Checks failed: sequential execution.
     put_value  "X",A1        | (X) -> A1   Normal WAM code for executing b,
     put_value  "Y",A2        | (Y) -> A2   c, and d sequentially.
     call  b/2,3              | call "b".
                              |
     put_value  "X",A1        | (X) -> A1
     put_value  "Y",A2        | (Y) -> A2
     call  c/2,3              | call "c".
                              |
     put_value  "X",A1        | (X) -> A1
     put_value  "Y",A2        | (Y) -> A2
     put_value  "Z",A3        | (Z) -> A3
     call  d/3,3              | call "d".
     -------------------------------------------------------------------------
CALL_E:                       | "Normal" WAM call to "e".
     put_value  Y3,A1         | (X) -> A1
     put_value  Y2,A2         | (Y) -> A2
     put_value  Y1,A3         | (Z) -> A3
     deallocate               | Discard environment: last call optimization.
     execute  e/3             | Execute "e".
```

3.6 Other Non-Instruction Related Actions: fail/ kill/ redo ...

In addition to the operations associated with particular instructions, each processor has to support other actions resulting from exceptions such as messages arriving from other processors or failure. These actions obviously differ somewhat from the corresponding ones in a sequential implementation. Due to space limitations we will only sketch some of them in this section:

- **failure**: in the Restricted AND-Parallel Abstract Machine, there are several cases of backtracking depending on the origin of the failure (the local processor or one of the remote processors) and also on the state of the computation ("**inside**" vs. "**outside**" backtracking) [10]. However, thanks to the existence of *Parcall Frames*, the operations involved remain similar to those in the sequential implementation: *local failure* is treated in the same way as in the **WAM** unless the "last" *choice entry* on the *Stack* is a *Parcall Frame* (**PF** > **B**) in which case the backtracking algorithm is applied to it, in order to select the goals to be "killed" or "restarted". *Remote failure*, i.e. failure coming from a different processor, basically involves undoing all local work done since the *Parcall Frame* associated with the failed goal was pushed on to the *Stack* (sending "kill" or "unwind" messages to all remote slots in *Parcall Frames* above it), and applying the backtracking algorithm to that *Parcall Frame*. These actions are explained in more detail in [9] and [10].

- **kill**: kill is a message which can arrive from the parent processor indicating that the goal being solved in the local processor is not useful any more and should be discarded: reset all registers to goal invocation point (i.e. throw away everything since execution of the goal was started), unwind the *Trail* until the last goal invocation point (i.e. undo all bindings in ancestors), reinitialize, go to idle. If there are any *Parcall Frames* in the *Stack*, all Pid.'s in non-local slots in them have to be sent kill messages also. Note how the values of all pointers before a goal is received (entries can still remain in the *Stack* or *Heap* from previous goals) have to be saved in order to do this. This remains an implementation issue but it can be solved using a small independent push-down list or "input goal markers" in the *Stack*.

- **redo**: redo is also received from the parent processor after reporting a solution which had a *choice point* available (i.e. after reporting "success with alternatives"). It is executed just as if local failure had occurred: go to the first *choice point* (or **PF**) on the *Stack*, continue with next alternative.

- **unwind**: this message is sent by the parent when backtracking, to a processor without alternatives but with a segment of the *Trail* pending. The *Trail* is unwound and the *Heap* is flushed to the point before the goal was received.

4 Conclusions

In the previous sections we have presented an AND-Parallel Abstract Machine level execution model for logic programs, based on combining the techniques used in the **WAM** with the advantages of **RAP** in dealing with variable binding conflicts. The same abstract machine and basic instruction set could also support with minor modifications many other AND-Parallel models and serve as a target for compilation of a variety of logic programming languages.

We feel that other solutions previously proposed lack the potential for storage efficiency and performance improvement that the Warren Abstract Machine has brought to the sequential logic programming arena. Conversely, we argue that this model is an attractive vehicle for the implementation of AND-Parallelism: the compatibility with conventional **WAM** code makes sequential speed almost identical to that of the **WAM** and permits the use of current **WAM** compiler technology. Simultaneously, most **WAM** optimizations are still supported, even during parallel execution. Also, a form of restricted intelligent backtracking is provided with virtually no additional overhead. "Soft" degradation of performance with resource exhaustion and user-transparent distributed control are attained as well.

The description in this paper has dealt mainly with *forward execution* at the abstract machine level. We have also covered other areas of the design, such as the backtracking algorithm [10], goal scheduling and memory management issues, and a more detailed system architecture. These results will be reported elsewhere. The reader can find more specific information regarding some of these subjects in [9]. Still, there are many areas in which work remains to be done, both in depth (i.e. further specification and implementation of the design) and breath (i.e. inclusion of other types of parallelism, support for a more sophisticated database interface etc.). Issues of interest which we are currently investigating are: a backtracking scheme which preserves the conventional ordering of alternatives, optimizations for determinate execution, development of better heuristics for the automatic generation of **CGE**'s, and treatment of cut and other side effects, etc.

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] T. P. Dobry, A. M. Despain, and Y. N. Patt.
Performance Studies of a Prolog Machine Architecture.
In *Proceedings of the 12 Int'l. Symp. on Computer Architecture*, pages 180-191. IEEE Computer Society Press, 1985.

[8] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek.
A Tutorial on the Warren Abstract Machine.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[9] 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.

[10] Manuel V. Hermenegildo and Roger I. Nasr.
Efficient Implementation of Backtracking in AND-parallelism.
In *Proceedings of the 3rd. Int'l. Conf. on Logic Programming.* Springer-Verlag, 1986.

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

[12] R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk.
Prolog on Multiprocessors.
Technical Report, Argonne National Laboratory, Argonne, Ill. 60439, 1985.

[13] 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.

[14] Pereira, L.M., F. C. N. Pereira, and D. H. D. Warren.
User's Guide to DECsystem-10 Prolog
Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

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

[16] E. Tick and D.H.D. Warren.
Towards a Pipelined Prolog Processor.
In *1984 International Symposium on Logic Programming, Atlantic City,* pages 29-42. IEEE Computer Society Press, Silver Spring, MD, February, 1984.

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