A Partial Breadth-First Execution Model for Prolog*

Jordi Tubella and Antonio González

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona - Spain

Abstract

MEM (Multipath Execution Model) is a novel model for the execution of Prolog programs which combines a depth-first and breadth-first exploration of the search tree. The breadth-first search allows more than one path of the SLD-tree to be explored at the same time. In this way, the computational cost of traversing the whole search tree associated to a program can be decreased because the MEM model reduces the overhead due to the execution of control instructions and also diminishes the number of unifications to be performed. This paper focusses on the description of the MEM model and its sequential implementation. Moreover, the MEM execution model can be implemented in order to exploit a new kind of parallelism, called path parallelism, which allows the parallel execution of unify operations related to simultaneously traversed paths.

1 Introduction

Logic programming languages and, concretely, Prolog, as its most representative member, offer elegant features to write symbolic applications.

A Prolog program consists of a set of clauses and a query. From a declarative point of view [9], a program determines if the query can be inferred from the clauses for some variable bindings. From a procedural point of view [9], the execution consists of applying inference steps to a resolvent, which initially is the program query, until it becomes empty. An inference step tries to perform an SLD-resolution by unifying the left-most goal in the resolvent with a clause whose head has the same predicate name and arity. The SLD-resolution was described by Kowalski [8], and it is based on the resolution inference rule introduced by Robinson [10]. In case there are more than one clause to unify, the candidate clauses are tried in the order they are written. When the unification is successful, an inference has been performed and the left-most goal in the resolvent is substituted by all subgoals in the body of the clause. Then, execution attempts another inference step. When the unification fails, the backtracking process tries another inference step for the youngest goal that still has some candidate clauses to unify with. The above procedure constitutes the standard depth-first left-to-right sequential execution of a Prolog program. The sequence of inference steps is depicted using an SLD-tree.

Most sequential architectures oriented to Prolog are based on the Warren's Abstract Machine (WAM) [1]. This machine defines a set of data types, registers, memory areas, and instructions to implement the standard depth-first left-to-right execution model of Prolog. WAM instructions can be divided in control instructions, which are responsible for managing the traversal of the SLD-tree related to a program, and unify instructions, which perform the basic unification operation of Prolog semantics. The main contribution of this abstract machine has been the description of the implicit control component in a Prolog program in an imperative way. This allowed the change from an interpretative execution of Prolog programs to the execution of compiled WAM code. WAM implementations map the abstract machine into a general purpose computer architecture or design specific architectures to execute more efficiently WAM programs [7].

A disadvantage of the standard execution comes from the repeated computation of the same resolvent every time a new solution to a goal is found. These executions of the same resolvent operate on different binding environments corresponding to different paths of the search tree, but the repeated execution of the same control instructions is a source of overhead. Another disadvantage is the repeated execution of unifications with the same operands in different paths.

A novel model, called Multipath Execution Model (MEM), is proposed in this paper. MEM overcomes the sources of overhead exhibited by the standard

*This work has been supported by the Ministry of Education of Spain under grant CICYT TIC 91/1036.

model by allowing to explore more than one path of the search tree at the same time. In MEM, control instructions and unifications involving single binding variables are executed only once for all paths being simultaneously explored, while the rest of unifications are repeated for each path. The way to achieve the exploration of more than one path at the same time is allowing a goal to be traversed in breadth-first order. In this way, the breadth execution of a goal tries to find all solutions to that goal before proceeding with the next goal. The drawback of a breadth-first search is that the computation state related to all the paths being simultaneously explored must be stored in memory, and its management is more complex than in WAM. Due to the limitation of available memory, an exhaustive breadth exploration may not be feasible and a partial breadth-first search combined with a depth-first search and backtracking is more adequate. This is the choice of MEM.

Apart from the advantage of avoiding the execution of control instructions and certain unifications, an additional benefit of the MEM execution model is the possibility of processing in parallel unifications involving multiple binding variables. A parallel unification unifies two Prolog terms for all paths being simultaneously traversed. The individual unifications performed in each path are independent because each path has its own binding environment stored in memory. We call this type of parallelism path parallelism.

A preliminary definition of the MEM execution model was presented in [13]. In [11] D. A. Smith presents Multilog as a data parallel language that computes solutions to any arbitrary goal into a set of environments, and subsequent goals execute in this set of environments, with unification being performed in parallel. Multilog and MEM have been carried out concurrently but independently. They have similar objectives although their implementations are rather different.

Another related work is the DAP Prolog system proposed in [5], in particular its set mode operation. This proposal extends the standard implementation of Prolog in order to support sets of data and exploit parallelism for managing the different elements of a set. There are several important differences between DAP and MEM. First, DAP extends the semantics of Prolog with sets whereas MEM is transparent to the programmer (a standard Prolog program does not require any modification to be executed by MEM). Second, the source of parallelism in DAP is due only to facts while, in MEM, multiple bindings to the same variable can be obtained as a result of any type and number of clauses. Finally, the implementation of MEM is simpler since the different relationships that may happen to have different sets makes the management of sets very cumbersome in DAP.

In this paper, the MEM execution model and a sequential implementation are described. The rest of the paper is organized as follows. Section 2 reviews the standard depth-first execution model of Prolog and its implementation using WAM. Section 3 describes the Multipath Execution Model, along with several issues of its implementation. Section 4 compares a sequential execution of both models. Section 5 introduces the parallel processing capabilities of MEM. Finally, the main conclusions are described in section 6.

2 Depth-first execution of Prolog

In this section, the standard depth-first left-to-right execution model of a Prolog engine, and its implementation using the Warren's Abstract Machine (WAM) are reviewed.

2.1 Standard execution model

In the standard execution model, the computation state, which identifies the state of a Prolog engine at each moment, is represented by means of a resolvent R, a binding environment BE, and a stack of goals SG, that is,

$$CS_{std} = \{R, BE, SG\}$$

where R is a list of ordered goals containing the remaining goals to solve the program query, BE is a set that contains all bindings performed by unification operations since the beginning of the execution, and SG contains elements of the form $\{R, BE\}$, representing resolvents and their associated binding environments that can lead to alternative program solutions. Initially, R is the program query, and BE and SG are empty.

$$CS_{std,initial} = \{(program\ query), \emptyset, stack()\}$$

Given a resolvent $R = (G_1, G_2, \ldots, G_i)$, a solution to goal G_1 is found when the resolvent becomes $R = (G_2, \ldots, G_i)$. A solution to the program is found when R becomes empty. In that case, the substitutions to the variables are obtained through BE. All solutions are found when R and SG are empty.

A Prolog engine continually performs inference attempts on R in order to find a program solution, as shown in figure 1. The computation of all solutions

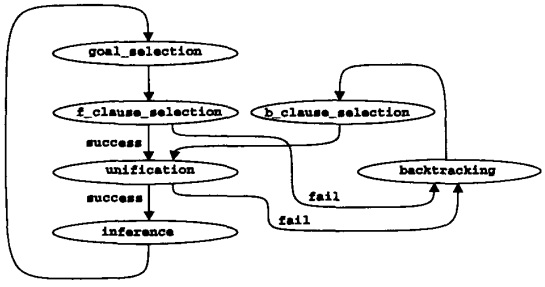


Figure 1: Basic computation loop of a Prolog engine based on the standard execution model.

to a program corresponds to a depth-first left-to-right traversal of the associated SLD-tree. OR-trees can also be used to depict the execution of a program. An OR-tree is an SLD-tree where nodes are only related to non-determinate goals. The update of the computation state at each inference attempt and the OR-tree traversal are performed as follows.

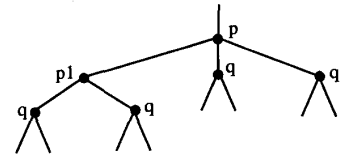
Given a resolvent $R = (G_1, G_2, \ldots, G_i)$, GOAL_SELECTION chooses the left-most goal in R, that is G_1, in order to attempt an SLD-resolution by means of its unification with a clause head having the same predicate name and arity. F_CLAUSE_SELECTION chooses the first written candidate clause. If there are no candidate clauses, the backtracking process is started. If there are more than one candidate clause, the resolvent R and the binding environment BE are pushed onto SG. This point corresponds to a node in the OR-tree, which has as many child branches as candidate clauses to unify with goal G_1. Then, execution continues unifying goal G_1 with the head of the chosen clause $G_1' : -G_{11}, G_{12}, \ldots, G_{1j}$.

The UNIFICATION operation determines whether these two first-order predicates match and, in that case, computes the most general unifier. If the unification succeeds, an INFERENCE has been performed.

```
?- p, q, r.

p :- p1, p2.          p11.    q.
p :- p3.              p12.    q.
p.                    p13.
                              r.
p1 :- p11, p12, p13.  p2.
p1.                   p3.
```

Figure 2: Sample Prolog program.



(a)

R = (p11, p12, p13, p2, q, r)
BE = {..., {*variable/binding*}, ...}
SG = stack({(p, q, r), BE'}, {(p1, p2, q, r), BE"})

(b)

Figure 3: (a) OR-tree for the sample program of figure 2; (b) computation state after the second inference (unification of goal `p1/0`).

Bindings in the most general unifier are added to BE and goal G_1 in R is substituted by all the subgoals in the body of the clause, that is the resolvent becomes $R = (G_{11}, G_{12}, \ldots, G_{1j}, G_2, \ldots, G_i)$. Then, execution continues attempting another inference in R.

If the unification of G_1 and the G_1' fails, the BACKTRACKING operation updates R and BE with the value of the topmost element in SG. Then, B_CLAUSE_SELECTION chooses the next candidate clause not tried yet in order to unify it with the left-most goal in R. In the case that this clause is the last candidate to unify with, the topmost element in SG is popped. Then, execution continues with another unification operation. Figure 3.a shows the OR-tree associated to the sample program of figure 2, while figure 3.b shows the computation state after the second inference has been performed.

2.2 Implementation based on the WAM

The Warren's Abstract Machine (WAM) has become a widely accepted architectural model to implement the standard Prolog depth-first left-to-right execution model [1]. The objective in this subsection is to review the most relevant aspects of WAM in order to compare it with the implementation of MEM to be presented in the next section.

WAM instructions for the goals in R are stored in the code area. Given a resolvent

$$R = (G_{111}, G_{112}, \ldots, G_{11k}, G_{12}, \ldots, G_{1j}, \ldots, G_2, \ldots, G_i)$$

register P points to the left-most goal in the resolvent (G_{111}); the rest of the goals belonging to the same

clause are sequentially accessed; register CP points to the goal to be executed when all the subgoals belonging to the current clause are solved (G_{12}); and **environments** in a **STACK** memory store the addresses of the remaining goals (G_2).

Bindings of variables in the BE of the computation state are stored in different memory areas depending on the type of the variable. There are three types of variables: *temporary*, which are variables used only by one goal of a clause, and stored in registers Ai; *permanent*, which are variables used by more than one goal, and stored in the environments of the STACK memory; and *unsafe* variables, which are permanent variables whose environment in the stack is deallocated, moved to a **HEAP** memory. Also note that when a variable binding is a compound term (list or structure), the elements of the compound term are stored in the heap. To obtain the substitution of a variable, a **dereference** operation follows the chain of bindings that unifications may have created.

Elements in SG of the form $\{R, BE\}$ are stored in **choice points** in the STACK memory. R is identified by storing in a choice point the contents of registers Ai, CP and E. Obviously, a complete copy of BE is not stored due to the amount of memory it needs. Only the necessary information to restore a BE in backtracking is stored in a choice point.

Unification is done argument by argument with different WAM instructions. Whenever an argument in the head of the clause is a ground term, WAM optimizes the unification with specific instructions. If the argument is a compound term, their elements are sequentially unified. When the operands to unify are variables, a call to the general unification procedure is performed. This procedure uses a small stack, called PDL, in case that both arguments are compound terms. Bindings obtained by the unification are stored directly in BE. To implement the restoring of BE in backtracking, addresses of variables to be bound are stored in a **TRAIL** memory if the variable may be later unbound, that is, if it is younger than the current choice point.

Backtracking updates R and BE in the computation state with the topmost element in SG. R is updated by getting registers Ai, CP and E from the current choice point. BE is restored by untrailing the variables bound since the creation of the current choice point. Backtracking also performs garbage collection. In the HEAP memory, it is done by updating register H to the value it had when the current choice point was created, while in the STACK and TRAIL memories is implicitly done by their LIFO structure.

3 Partial breadth-first execution

In this section, the Multipath Execution Model (MEM) is described. First, the main ideas are presented. A more detailed description along with several issues about its implementation can be found in the enclosed subsections.

Multipath does not impose any modification in the semantics of standard Prolog. It does not require neither any change in the syntax except for the inclusion of annotations that the programmer may use to give hints about the most convenient type of search. These hints are not mandatory since the compiler performs a static analysis to determine the type of search for each goal. However, since the programmer has a detailed knowledge of his program, these hints may improve the task of the compiler in some cases.

The main feature of Multipath is that it allows a given goal to be executed in depth-first, breadth-first or partial breadth-first order. The choice among these three options can be made by the programmer, the compiler and the execution model itself at run time.

In this context, a partial breadth-first execution of a goal means to explore in breadth-first order some (but not all) of the alternative clauses with the same name and arity, and go back to explore the remaining ones by means of backtracking. These remaining clauses may be then explored in any of the three possible orders (depth, breadth or partial breadth).

In general, those goals that have a large number of solutions are suitable to be explored in breadth-first order. This is determined at compile time or annotated by the programmer. The algorithm used by the compiler to detect the most convenient search for a goal is not described in this paper due to its complexity. In any case, the final decision about the execution of a goal will be taken at run time. A goal whose execution has been designated as depth-first will always be explored in this order. However, a goal for which the compiler or programmer has proposed a breadth-first search will be executed in that order only if at run time there are enough resources (especially memory) to support that type of execution. Otherwise, a partial breadth-first or even a depth-first search will be adopted again depending on the available resources.

A goal that is executed in breadth or partial breadth order may result in some variables bound to multiple values. For instance, in the following code:

```
p(Y) :- q(X),r(X,Y).
q(1).                r(2,3).
q(2).                r(3,4).
q(3).                r(4,5).
```

a breadth execution of `q/1` results in `X` being bound to three different values 1, 2 and 3.

The multiple bindings that result from a (partial) breadth execution of a goal are then processed all together afterwards. In the previous example, when `X` is dereferenced during the execution of `r/2` the result will be that it is bound to three different values: 1, 2 and 3. This is equivalent to say that the execution model explores several paths (branches) of the SLD-tree at the same time in a single flow of control.

3.1 Multipath execution model

A detailed operation of the Multipath execution model (MEM) is described below by explaining how the computation state of a program changes during its execution. Moreover, a MEM-tree is presented. This tree is equivalent to the SLD-tree and allows to represent the single flow of control when traversing more than one path.

In the MEM execution model there are two kinds of paths: CURRENT paths, which are those being traversed with the objective to become solutions to a breadth goal or to the program; and SOLUTION paths, which are those paths suspended as solutions to previous breadth goals. Note that breadth goals may be nested; that is, during the execution of a breadth goal, another breadth goal may be invoked (see figure 4). The representation of a MEM-tree is explained below when describing the Multipath operations.

The computation state in the MEM model is represented by the current breadth goal CBG, the current resolvent CR, a set of current binding environments $SCBE$, a set of breadth goals SBG, and a stack of goals SG, that is

$$CS_{MEM} = \{CBG, CR, SCBE, SBG, SG\}$$

The function of each element of the computation state is next defined.

CBG specifies the breadth goal being solved. A breadth goal is identified by two elements: the resolvent once the breadth goal is solved (NR), and the breadth nesting level (BNL). The objective of NR is to identify the point where the breadth goal is invoked, and the objective of BNL is to identify different instances of recursive breadth goals, when they are the last goal of a clause.

$$CBG = \{NR, BNL\}$$

CR specifies the remaining goals to solve CBG and it is represented by an ordered list of goals.

$$CR = (G_1, \ldots, G_i)$$

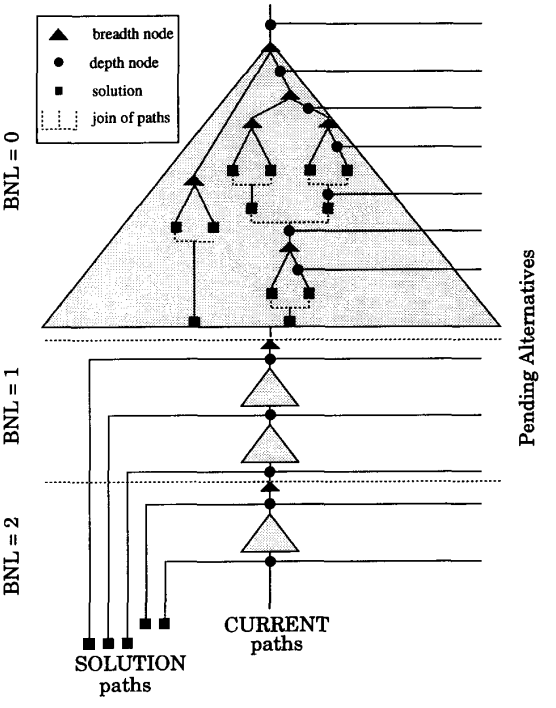


Figure 4: Shape of MEM-trees.

$SCBE$ contains the binding environments corresponding to the CURRENT paths. Each BE is a set of *variable/binding* elements. The representation of a binding environment is an implementation issue that is briefly sketched in the next subsection.

$$SCBE = \{BE_1, \ldots, \{\ldots, variable/binding, \ldots\}, \ldots, BE_j\}$$

SBG contains information about breadth goals with pending alternatives to explore. The information for each entry in SBG consists of three elements: $\{BG, SSBE, NBG\}$. BG is the identification of the breadth goal; $SSBE$ are all the binding environments associated with the SOLUTION paths already found to the breadth goal; and NBG is the next breadth goal to solve once BG is solved.

$$SBG = \{\ldots, \{BG, SSBE, NBG\}, \ldots\}$$

SG is a stack of goals with pending clauses to try. Each element is of the form $\{BG, R, SBE\}$. BG is a breadth goal with pending clauses; R is the resolvent to solve BG; and SBE are the binding environments associated to the paths that are visible for that goal.

$$SG = stack(\ldots, \{BG, R, SBE\}, \ldots)$$

Initially, one path is explored with its binding environment empty, and SG and SBG are empty.

$$
\begin{aligned}
CS_{MEM,initial} = \ & \{ \\
& CBG = \{(), 0\}, \\
& CR = (program\ query), \\
& SCBE = \{\emptyset\}, \\
& SBG = \emptyset, \\
& SG = stack() \\
& \}
\end{aligned}
$$

A breadth goal CBG is said to be solved when CR becomes empty. In this case, there are as many solutions to CBG as elements in $SCBE$. Each element in $SCBE$ when goal $CBG = \{(), 0\}$ is solved corresponds to a solution to the program. All solutions to a program are found when $CBG = \{(), 0\}$, $CR = ()$, $SG = stack()$, and $SBG = \emptyset$.

A MEM inference engine continually performs inference attempts that modify the computation state. The basic operations are summarized in figure 5 and explained in the next paragraphs.

GOAL_SELECTION selects the left-most goal in CR (as stated by the computation rule) in order to attempt an inference.

F_CLAUSE_SELECTION selects a clause to be unified with the left-most goal in the resolvent. If there are more than one candidate clause, they are tried in the order they are written. Thus, f_clause_selection always selects the first one. Note that *f_* stands for forward execution to differentiate it from the clause selection operation to be performed in backward execution (see below). Let us suppose that the resolvent of the computation state in a certain moment is $CR = (G_1, G_2, \ldots, G_i)$ and the first candidate clause to try is G_1' :- G_{11}, ..., G_{1k}. If G_1 has a breadth attribute and there are enough resources (memory) to allow its (partial) breadth execution, a new breadth goal is added to SBG with no solutions found so far, and CBG and CR are updated in the following way:

$$
\begin{aligned}
SBG &\leftarrow SBG \cup \{\{G_1, \emptyset, CBG\}\} \\
CBG &\leftarrow \{(G_2, \ldots, G_i \mid NR_{CBG}), BNL_{CBG} + 1\} \\
CR &\leftarrow (G_1)
\end{aligned}
$$

where NR_{CBG} and BNL_{CBG} are the identifiers of the current breadth goal CBG. If there are more than one candidate clause to unify with, $\{CBG, CR, SCBE\}$ is pushed onto SG.

$$push(SG, \{CBG, CR, SCBE\})$$

In the MEM-tree, depth goals are represented with circular nodes, while breadth goals are represented with triangular nodes. Execution continues unifying G_1 with G_1'.

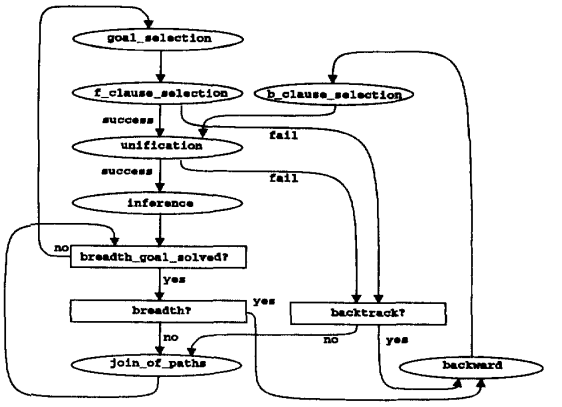


Figure 5: Basic execution loop of a Prolog engine based on MEM.

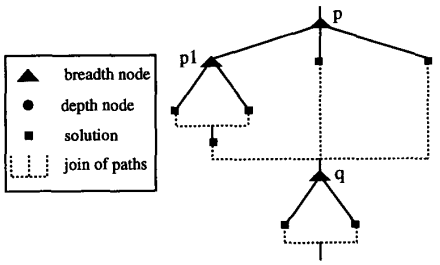
UNIFICATION is performed for every CURRENT path, that is, once for each element in $SCBE$. The unification operation fails if every individual unification in each BE fails, otherwise succeeds. BEs related to failed paths are eliminated from $SCBE$.

If unification succeeds, the left-most goal in the resolvent is substituted by all the subgoals in the body of the clause, and bindings representing the most general unifier performed in each path are added to the corresponding element in $SCBE$:

$$
\begin{aligned}
CR &\leftarrow (G_{11}, \ldots, G_{1k}, G_2, \ldots, G_i) \\
SCBE &\leftarrow \{BE_1 \cup \theta_1, \ldots, BE_j \cup \theta_j\}
\end{aligned}
$$

When eventually the current resolvent (CR) in the computation state becomes empty the breadth goal CBG is solved (this operation is called BREADTH_GOAL_SOLVED? in figure 5, and represented with a square in the MEM-tree). At this time, it is decided whether the execution continues with a breadth-first or a depth-first search, that is, with a backward or a join of paths operation, respectively.

A BREADTH-first search is possible if two conditions hold. The first condition is that the current breadth goal (CBG) has still more potential solutions. The second condition restricts the breadth-first search to a maximum number of paths. If this number is exceeded, the breadth-first search is turned into a depth-first search, and thus, a join of paths operation is performed. In this case, the current goal is executed in a partial breadth-first order. This restriction is introduced in the execution model for implementation



(a)

CBG = p1
CR = (p11, p12, p13)
SCBE = {BE_1}
SBG = {{ p,∅, '?-'}, {p1,∅, p}}
SG = stack ({p,(p),{BE_1'}}, {p1, (p1), {BE_1"}})
where breadth goals are
'?-' = {(),0}
p = {(q,r),1}
p1 = {(p2, q, r), 2}
q = {(r), 1}

(b)

Figure 6: (a) Example of MEM-tree; (b) Example of computation state.

reasons, mainly due to the requirement of storing in memory the BE of each path.

In the Multipath execution model, when a unification fails, execution does not continue immediately with a backtracking operation. There is a BACKTRACK condition that considers the existence in SBG of a breadth goal with SOLUTION paths that has no possibilities to obtain more solutions. In this case, a join of paths operation is executed. Otherwise, a backward operation is executed.

The JOIN_OF_PATHS operation joins all SOLUTION paths found so far to the youngest breadth goal of the SLD-tree. The SOLUTION paths become CURRENT paths. The join of paths also updates CR and CBG:

$$\begin{aligned} SCBE &\leftarrow SCBE \cup SSBE_{SBG(CBG)} \\ CR &\leftarrow NR_{CBG} \\ CBG &\leftarrow NBG_{SBG(CBG)} \end{aligned}$$

where $SBG(CBG)$ is the entry in SBG with its breadth goal equals to CBG, and $SSBE_{SBG(CBG)}$ and $NBG_{SBG(CBG)}$ are elements of that entry. In the MEM-tree, this operation is represented with dashed lines that collect all solutions.

In case of a BACKWARD operation to the youngest branch alternative of the SLD-tree, the main actions are to store every CURRENT path as a SOLUTION path in the entry associated to CBG in SBG, and to restore the computation state with the value of the topmost element in SG:

$$SBG \leftarrow \{\ldots, \{CBG, \{SSBE \cup SCBE\}, NBG\}, \ldots\}$$

$$\{CBG, CR, SCBE\} \leftarrow top(SG)$$

Entries in SBG are not needed when their associated breadth goals have no solutions already computed and there is no possibility to find more solutions to them. These entries are deleted from SBG when this condition is detected in the join_of_paths and backward operations.

B_CLAUSE_SELECTION selects the next candidate clause to be unified with the left-most goal in CR. In case this clause is the last one, the topmost element in SG is popped.

$$pop(SG)$$

Operations described above are continually repeated until all program solutions have been found. This is detected in the backtrack condition. Figure 6.a shows the MEM-tree associated to the sample program of figure 2, where goals `p/0`, `p1/0` and `q/0` are determined to be explored in breadth-first order. Figure 6.b shows the computation state after the second inference has been performed.

3.2 Implementation issues of the MAM

The implementation of the MEM is done by introducing some extensions to the WAM. This modified WAM is called Multipath Abstract Machine (MAM).

A difference between WAM and MAM is the existence of two types of variables: *single* and *multiple*. The former are the conventional variables used by WAM. These variables have a unique binding shared by all paths that can be simultaneously traversed. The latter are variables that may be bound to multiple values at the same time.

The type of a variable is determined at compile time. A variable that is instantiated inside the scope of a breadth goal is declared to be multiple. In other words, the compiler determines that a variable is single when it can assure that the variable will have at most one binding during execution. A variable never changes its type during execution time.

Single variables are stored as in the WAM. A new memory area, called **MBE** (Multi-Binding Environment), is added in order to store multiple variables. A multiple variable has the same address in all BEs where it is visible. A new register **HV** points to the top of MBE.

In MAM, there are two kinds of engines: a *Main Engine* (ME), which is responsible for controlling the traversal of the search tree, and several *Unification Engines* (UEs), which are responsible for performing unification operations on the BEs. Each UE manages one BE, and the maximum number of UEs is a parameter of MAM.

The MAM instructions are the same as in the WAM but their semantics are slightly modified in order to manage multiple BEs. Those instructions referencing a multiple variable must repeat its operation for every element in SCBE, and thus, they are executed by different UEs.

The implementation of the backward operation defined in MEM requires to allocate a number of new UEs, and to initialize their BEs to reflect the computation state at the time the last choice point was created. This is currently done by copying the contents of CURRENT or SOLUTION BEs to the new BEs and untrailing multiple variables bound since the creation of the choice point. Alternatively, other techniques similar to the ones used to exploit OR-parallelism could be used instead of copying (e.g., hash-windows, binding-arrays [6]).

4 Comparison MAM vs. WAM

We have implemented sequential emulators for both WAM and MAM abstract machines. A set of benchmark programs have been run on a DEC 3800 system whose CPU is an Alpha 21064 microprocessor. All benchmarks were taken from [12] excepting `bits-pal`, which is taken from [11].

Results are shown in figure 7. We can observe that the sequential breadth-first execution performed by MAM may be advantageous over a standard WAM implementation. Note that these figures compare two different execution models running on the same hardware. The speed-ups are between 1 and 4 except for `bits-pal`, which has an speed-up of 13. Therefore, the improvements achieved by MAM are significant since they are obtained without any additional hardware. Regarding the number of UEs, the best performance is achieved with 200 UEs.

The advantages of a sequential execution of MEM rely on avoiding the execution of control instructions

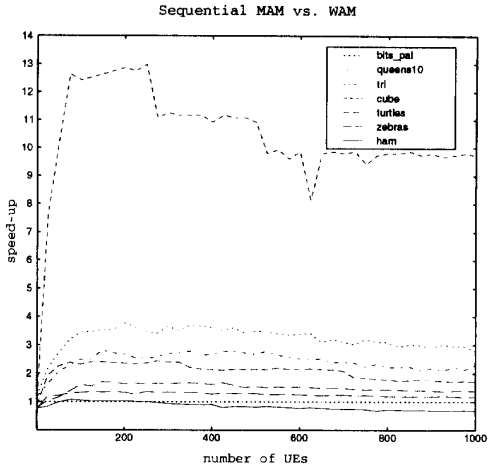


Figure 7: Speed-up of sequential MAM vs. WAM.

and unifications of single variables in the simultaneously explored paths. In `bits-pal`, performance is better because the average number of paths is very high compared with the rest of the benchmarks. A number of UEs greater than 200 is not beneficial due to the amount of parallelism exhibited by programs, which is related with the average number of CURRENT paths. Increasing the number of UEs requires more memory at run time. This results in a decrease of the cache hit ratio and an increase in the number of page faults, which in turn, increases the average memory access time. Results prove that the advantages overcome the overhead imposed by the breadth-first exploration and confirm the feasibility of an execution model based on a combined depth-first and breadth-first search.

5 Parallel execution of MEM

Another benefit of the MEM execution model is the possibility of executing in parallel the unification operations for each binding environment in the computation state. We call this kind of parallelism path parallelism. The sources of parallelism most related to path parallelism are unification parallelism and data parallelism.

Path parallelism is a particular case of data parallelism. Data parallelism consists in the concurrent treatment of multiple bindings of variables. In the literature, data parallelism has been exploited in the

context of OR-parallel systems [4]. In this approach, after binding a variable to multiple values, execution can continue in parallel exploring the subtrees related to each binding in an independent way. For each one of these subtrees, just one of the bindings is visible. In path parallelism, variables get multiple bindings as a result of the sequential execution of a non-determinate goal. When all bindings are collected, parallelism is exploited when a unify operation is performed on variables with multiple bindings.

Path parallelism is also different from unification parallelism [2]. In unification parallelism, parallel unifications are performed on different arguments of a goal for a single binding environment. In this case, there may be data dependences among the unifications. In path parallelism, the parallel execution corresponds to unifications of the same argument for different binding environments. In this case there are no data dependences.

In general, all operations to be performed by UEs (dereferences, unifications, BE copies) can be executed in parallel. In this way, tasks in path parallelism are fine-grained, but data sharing among UEs is not needed and synchronization with the ME is seldom required. In [3], a parallel implementation of MEM is described in more detail.

6 Conclusions

A novel execution model for Prolog programs (MEM) that combines a depth-first and a breadth-first exploration of the search tree has been presented. The main characteristics of MEM is the simultaneous traversal of more than one path of the SLD-tree. A modification of the Warren's Abstract Machine to accommodate the features of the MEM model has also been presented. Performance improvement of MEM over the standard depth-first traversal depends on three factors: the overhead added by the breadth-first search management, the ratio of the number of control instructions over the total number of executed instructions, and the average number of paths being simultaneously traversed. The results presented in this paper confirm that a combined depth-first and breadth-first search is advantageous over the standard depth-first search for usual Prolog programs.

The simultaneous traversal of more than one path enables the exploitation of a new kind of parallelism, called path parallelism. In path parallelism, accesses or unifications into the binding environments related multiple paths being simultaneously traversed may be executed in parallel. This type of parallelism, which is different from OR and unification parallelism, may contribute to increase substantially the performance of the system. Although path parallelism is fine-grained, it can be exploited very efficiently because the amount of synchronization and data sharing that it requires is rather low.

References

[1] H. Ait Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] W.V. Citrin. *Parallel Unification Scheduling in Prolog.* Technical report UCB/CSD 88/415, Berkeley University. 1988.

[3] A. González and J. Tubella. The Multipath Parallel Execution Model for Prolog. *In Proc. of PASCO'94.* World Scientific Pub. To appear.

[4] P. Heuze. *Using Data-Parallelism in Elipsys.* Technical report elipsys-003. ECRC, Munich (Germany), 1989.

[5] P. Kacsuk. DAP Prolog: A Parallel Array Extension of Prolog. *In Proc. of CONPAR'88,* British Computer Society (editor). 1988.

[6] P. Kacsuk and M. Wise (editors). *Implementations of Distributed Prolog.* John Wiley & Sons. 1992

[7] P.M. Kogge. *The Architecture of Symbolic Computers.* Mc-Graw-Hill, 1991.

[8] R.A. Kowalski. Predicate Logic as a Programming Language. *Information Processing 74,* pp. 569-574, Stockholm, North-Holland, 1974.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[10] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal ACM* 12, 1, pp. 23-41, Jan. 1965.

[11] D. A. Smith. Multilog: Data Or-Parallel Logic Programming. *In Proc. of ICLP'93.* MIT Press, 1993.

[12] E. Tick. *Parallel Logic Programming.* MIT Press, 1991.

[13] J. Tubella and A. Gonzalez. MEM: A New Execution Model for Prolog. *Microprocessing and Microprogramming,* vol. 39, pp. 83-86, North-Holland, 1993.