Design and Implementation of the BEAM, an Extended Andorra Model System

Ricardo Lopes, Vítor Santos Costa

Technical Report Series: DCC-2000-2

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Design and Implementation of the BEAM, an Extended Andorra Model System

Ricardo Lopes

LIACC and DCC-FCUP
Universidade do Porto
4150 Porto, Portugal
rslopes@ncc.up.pt

Vítor Santos Costa

COPPE-Sistemas and LIACC-UP
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil
vitor@cos.ufrj.br

May 15, 2000

Abstract

Logic programming provides a high-level view of programming, giving implementors a vast latitude in what techniques to research towards obtaining the best performance for logic programs. One of the holy grails of logic programming has been to design computational models that could be executed efficiently and would allow for both a reduction of the search space and for exploiting all the available parallelism in the application. These goals have motivated the design of the Extended Andorra Model. We report on the design of a first implementation for the Extended Andorra Model with Implicit Control, the BEAM.

keywords: Extended Andorra Model, Parallelism, Implementation.

1 Introduction

Logic programming provides a high-level view of programming where programs are fundamentally seen as a collection of statements that define a model of the intended problem, and questions may be asked against this model. Prolog is the most popular example of a logic programming language, and has been successfully used in applications such as artificial intelligence, database programming, circuit design, genetic sequencing, expert systems, compilers, simulation and natural language processing. The high-level nature of logic programs gives implementors a vast latitude in what techniques to research towards obtaining the best performance for logic programs. Advanced compilation techniques [13] have been used to obtain performance close to imperative languages for some benchmarks [12]. Parallelism [9] allows logic programs to transparently exploit the advantages of the new parallel machines that are more than ever gaining in popularity. Extensions such as co-routining, constraints, and tabulation allow for substantial reductions in search space over the traditional Prolog selection function, and are a basis on which to build advanced features and support novel applications.

Towards obtaining maximum performance, one of the holy grails of logic programming has been to design computational models that could be executed efficiently and would allow for both a reduction of the search space and for exploiting all the available parallelism in the application. These goals have been presented by David H. D. Warren [15] as, in order of priority:

- *Minimum number of inferences*: this is achieved by trying never to repeat the same execution step of one inference in different locations of the execution tree.

- *Maximum parallelism*: this is achieved by allowing goals to be executed as more independently as possible, and by doing the combination of all solutions as late as feasible.

These goals have motivated the design of the Extended Andorra Model. The model originates from the research on the Basic Andorra Model (BAM) [14]. The Andorra-I prototype [10] demonstrated that the BAM can successfully exploit two forms of parallelism, or-parallelism and dependent and-parallelism, and that it can reduce the search space for a large category of programs. On the other hand, Andorra-I depends on finding determinate applications.

The Extended Andorra Model addresses the major problems of the Basic Andorra Model, and approximates the goals of minimizing the search space and maximizing the parallelism. The original proposal was based in the concept of rewriting and- and or-boxes. Research then evolved in two directions.

One approach was followed by Haridi, Janson and researchers at SICS who concentrated on the AKL [5], the Andorra Kernel Language, based on the principle that the advantages of the Extended Andorra Model justified a new programming paradigm that could subsume both traditional Prolog and the committed choice languages. AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit. AKL also included novel features such as ports [4], an elegant approach towards manipulating streams. AKL was shown to obtain acceptable performance in a sequential and parallel implementations, such as Penny [8], had good results. Eventually, the AKL researchers have shifted their interest to Oz [11]. Oz provides some of the features of AKL such as thee logical variable and encapsulation, but control must be programmed explicitly.

In contrast, David H. D. Warren and researchers at Bristol concentrated on the *Extended Andorra Model with Implicit Control* [15], where the goal was to obtain the advantages of the Extended Andorra Model with the least effort from the programmer. Towards this goal Gupta designed a proof-of-concept interpreter [2]. Work in the interpreter allowed experimentation with novel ideas for the EAM, such as lazy copying and eager producers to give finer control over search and improve parallelism.

Gupta and Pontelli later experimented with an extension of dependent and-paralle-lism that provide some of the functionality of the EAM through parallelism [3]. The insight was that goals that share variables may execute in parallel and mutually constrain their search-space. In this case, the search space actually depends on how much parallelism is exploited.

The EAM offers great potential both for improving performance of existing applications and for obtaining more declarative formulations of search and database applications. In this work, we present a first implementation of the EAM with implicit control: the *BEAM* (Boxed EAM). Our initial goal in designing the BEAM was to prove the feasibility of the Warren's design for the EAM, and we believe that our initial results do confirm this.

We have designed the BEAM over a traditional Prolog system, YAP [1]. Initially, this reduces the amount of work we have to do in implementing our prototype. Ultimately, we aim at making the BEAM an extension of this Prolog system that the user can exploit towards maximum performance in declarative applications.

Next, we give an overview of the EAM including several extra simplification rules we implement. We next present the major data structures and execution algorithms that were required for efficient execution. We focus on the major issues in our implementation, such as forking and cut. Last, we present initial performance results and concluding remarks.

2 The Extended Andorra Model

The EAM was designed to generate all answers to problems coded as Horn clauses with a minimum number of inferences. The EAM tries to achieve this goal by performing as many steps in parallel as possible.

2.1 Execution Principles for the EAM

The EAM is formally defined using rewrite rules that manipulate and-or-trees. These are represented through *and*, *or* and *choice-boxes*.

- **And-box:** $[\exists X_1, ..., X_m : \sigma \& G_1 \& ... \& G_n]$

  And-boxes are created when launching a new clause with goals G_1 to G_n. The variables X_1 to X_n represent the variables created in the box and σ represents the constraints on external variables imposed by the and-box.

- **Or-box:** $\{C_1 \vee ... \vee C_n\}$

  Or-boxes are created when launching a new goal, each C_i represents an alternative clause for the goal.

- **Choice-box:** $\{G_1 \% C_1 \vee ... \vee G_n \% C_n\}$

  Choice-boxes are special or-boxes created when the clauses defining a procedure include a pruning operator, %. Each G_i represents the guard of the goal and each C_i represents the remainder of the clause. The original EAM supports two pruning operators, cut, !, and commit, |. AKL further supports a sequential conjunction, :.

We next concentrate on cut-free programs. In this case, there are four main operations in the Extended Andorra Model:

1. **Reduction:** $G \rightarrow \{[\exists Y_1 : \sigma_1 \& C_1] \vee \ldots \vee [\exists Y_n : \sigma_n \& C_n]\}$

   Reduction expands a goal G in an and-box into an or-box.

2. **Promotion:**

   $[\exists X, Z : A \& [\exists Y, W : Z = \theta(W) \& C] \& B] \rightarrow [\exists X, W, Z : Z = \theta(W) \& A \& [\exists Y : C] \& B]$

   Promotion promotes constraints from a inner-box to the outer and-box.

3. **Substitution:** $[\exists X, Z : Z = \theta(W) \& C] \rightarrow [\exists X : C\theta]$

   Substitution applies a constraint on a variable.

4. **Forking:** $[\exists X : \{C_1 \vee \ldots \vee C_n\} \ \& \ G] \rightarrow \{[\exists X : C_1 \& G] \vee \ldots \vee [\exists X : C_n \ \& \ G]\}$

   Forking distributes a conjunction across a disjunction.

2.2 Simplification and optimization rules

The basic EAM rules defined previously are constructive, so there is no way of simplifying the and-or tree and of recovering space by discarding boxes. The BEAM therefore includes additional simplification rules. The extra rules also allow us to optimize the computation process by early pruning of branches, resulting in early detection of failure. They are:

1. **Single And-box:** $\{[\exists X : \theta \& G]\} \rightarrow [\exists X : \theta \& G]$

   This rule simplifies an or-box that contains a single and-box. This rule allows discarding an or-box and applying promotion from the inner and-box to the upper and-box. Restrictions apply when the and-box contain a side_effect operator like `cut` (see section 6).

2. **Success Propagation:**

   The original EAM does not explicitly provide a notion of successful computation that has completed execution and that may now be discarded. The following rules identify success situations in the BEAM and allow the propagation of success towards the upper boxes. To implement these rules we use the notion of a *true-box*: an and-box is called true-box when all the local computations have been completed and the and-box does not contain constraints on external variables:

   $[\exists X :] \equiv \texttt{true}$

   Note that the and-box might have constrain external variables, but those constraints would have left the and-box after application of the promotion rule.

   The rules we implement are:

   (a) **true-in-and-simplification:**
   $[\exists X : \theta \& A \& \ldots \& \texttt{true} \& \ldots \& B] \rightarrow [\exists X : \theta \& A \& \ldots \& B]$
   Clearly, an occurrence of a true-box inside another and-box can be removed since the true-box is not adding any knowledge. Ideally, we would like all goals to reduce to true-boxes so that the initial box could be simplified itself.

   (b) **true-in-or-simplification:**
   $\{\ldots \vee \texttt{true} \vee \ldots\} \rightarrow \texttt{true}$
   This is the main source of pruning in our mode. In a pure logic language, the presence of a true-box in a branch of an or-box should allow immediate pruning of all the other branches in the or-box.
   If we are using Prolog as the user-level language, the user may want to have the other branches executed anyway. For this reason BEAM allows this simplification if and only if the other branches do not call built-ins with side_effects, such as `write`.

3. **Failure Propagation:** The symmetric problem to the propagation of success is the propagation of failure. It is quite important to identify failing situations in order to allow the propagation of failure towards the upper boxes and to recover space. Again, the basic EAM design does not contain explicit rules for propagating failure. First, we define a *fail-box* as an empty or-box:

   $\{\} \equiv \texttt{false}$

   We thus say that a failure occurs when we found a fail-box. The possible situations are:

   (a) **false-in-and-simplification:**
   $[\exists X : \theta \& A \& \ldots \& \texttt{false} \& \ldots \& B] \rightarrow \texttt{false}$
   If a failure occurs at any point of the and-box, the and-box can be removed and failure is propagated to the upper boxes.

   (b) **false-in-unique-or-simplification:**
   $\{\texttt{false}\} \rightarrow \texttt{false}$
   If an or-box contains an unique alternative and that fails, the or-box has failed.

   (c) **false-in-or-simplification:**
   $\{G_1 \vee \ldots \vee \texttt{false} \vee \ldots \vee G_n\} \rightarrow \{G_1 \vee \ldots \vee G_n\}$
   If one alternative of an or-box fails, the alternative is immediately removed from the or-box.

   Notice that these rules have priority over all the other rules to allow propagation of failure as fast as possible.

4. **Or-Compression:**

   $\{G_1 \vee \ldots \vee \{G_{i1} \vee \ldots \vee G_{im}\} \vee \ldots \vee G_n\} \rightarrow \{G_1 \vee \ldots \vee G_{i1} \vee \ldots \vee G_{im} \vee \ldots \vee G_n\}$

   We can remove a nesting of two or-boxes by promoting the inner box to the outer box. After application of this rule, an or-box is always followed by an and-box or a series of and-boxes.

5. **And-Compression:**

   $[\exists X : \theta_1 \& A \& [\exists Y : \theta_2 \& C] \& B] \rightarrow [\exists X, Y : \theta_1 \& \theta_2 \& A \& C \& B]$

   This rule removes a nesting of two and-boxes by promoting the inner box to the outer box. It thus generalizes on the promotion rule discussed previously. The rule is quite useful in simple recursive programs because it allows reusing the space of the original and-box for the inner and-box. It may be seen as a generalization of tail-recursion.

   The BEAM cannot apply this rule if there is a side_effect operator such as `cut` on the inner box (see section 6 for more detail on these cases). Note that if during the promotion there exists local variables on the inner box, they are changed to accepted the outer box as being their new home-box. On BEAM an and-box can be the leaf of the computation tree or it can be followed by and or-box or a series of or-boxes [6].

6. **Lazy propagation:**

   $[\exists X, W, Z : Z = \theta(W) \& \ldots \& \{\ldots \vee [\exists Y : C] \vee \ldots\} \& \ldots] \rightarrow$
   $[\exists X, W, Z : Z = \theta(W) \& \ldots \& \{\ldots \vee [\exists Y : Z = \theta(W) \& C] \vee \ldots\} \& \ldots]$

   As shown in Fig. 1 this rule is a symmetrical to the promotion rule. The promotion rule allows one to propagate locally generated constraints to the node where the variables were created. In contrast, the lazy propagation rule allows us to propagate the constraints in the opposite direction, from the root of a subtree downwards.

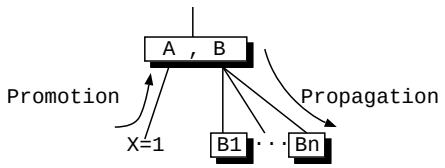

Figure 1: Promotion and propagation of constraints.

Executing this operation immediately after generation of a constraint in an and-box containing other boxes would have been very expensive. On the BEAM we adopt a lazy approach. Instead of having the propagation from the top to the bottom immediately after the binding on a variable, a `WAKE` signal is sent to all sub-branches that have constraints on the variable in question. All and-boxes that have received the `WAKE` signal now know that they must perform a consistency check of the constraints generated in the and-box with the constraints composing its environment (`Environment Synchronization`).

3 Execution control in BEAM

The BEAM uses a control scheme where an and-box suspends if it constrains an external variable. Fig. 2 shows the control scheme used in the BEAM. The systems first tries to apply a reduction or a promotion step. If it could, it will apply a sequence of *simplification* steps to maintain the data structures. If no reduction or promotion steps were possible, it will apply a forking step.

Forking is the operation that can lead to duplication of work, and thus is only applied when all the other operations cannot be applied further. In the BEAM, if several goals can produce a non-determinate binding for a variable, the leftmost one will be chosen to fork. Notice that a binding

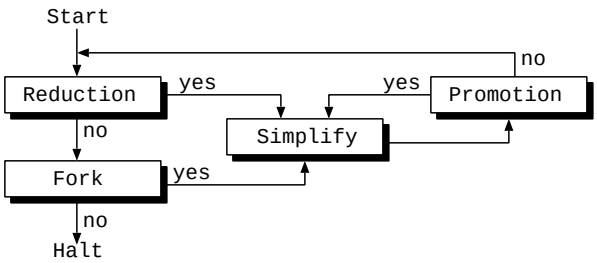

Figure 2: Execution control in the BEAM.

for a variable X has to be generated by one of the C_i before forking is initiated. After forking, this binding can be promoted in the newly created and-box. In the BEAM the forking step is always followed by a promotion. A forking step followed by a promotion is called *global forking*.

Favoring reduction steps over forking steps in the EAM allows early execution of non-determinate goals. In some cases, this may create speculative and-work, as a goal to the left might fail. In the worst case, speculation may result in a larger search space or even lead to non-termination, as the EAM allows any computation to go ahead as long as it does not bind variables. Ideally, one would like to provide an implicit control mechanism to prevent this problem. Unfortunately, detecting speculative work depends on knowing a priori whether the computation will terminate, an undecidable problem. On [7] we discuss how tabling can be used in a EAM model to prevent non-termination.

4 The BEAM Execution Model and Abstract Machine

The structure of the BEAM is shown in Fig. 3. The BEAM builds on top of the YAP [1] Prolog system. The shadowed boxes show where the EAM stores data. The *Code Space* stores the data-base, and specifically including the bytecode to be interpreted by the EAM emulator. The stack area stores dynamic data structures including the and-boxes, or-boxes, and compound terms.

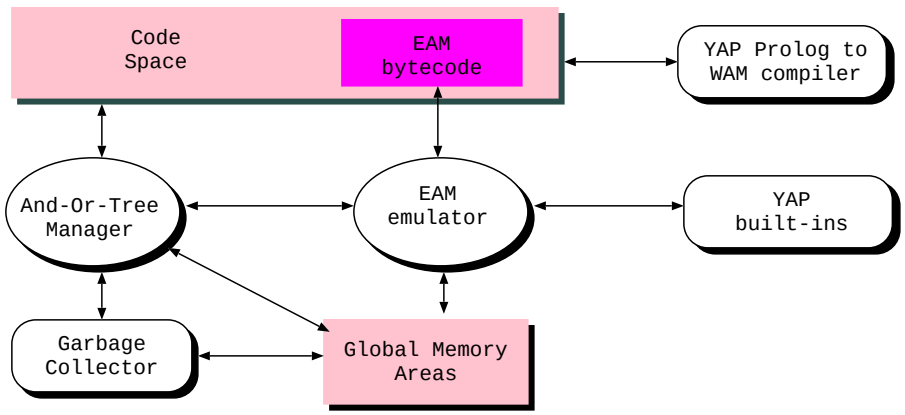

Figure 3: Execution Model.

The actual BEAM relies on two main components, shown as ovals in Fig. 3:

And-Or-Tree Manager: This module applies the EAM rewriting rules to the existing and-boxes and or-boxes until it can start WAM-like execution for the selected goal (see 4.3).

Emulator: This module runs WAM-like code to perform unification and setup goals. Unification code is similar to the WAM, as we explain next. Control instructions follow a compilation

scheme similar to the WAM but result in rather different execution. (see 4.2.1).

We next give a brief example of how the two components interact. Consider the goal `p(X,Y)` and the clause `p(X,Y):- g(X), f(Y)`. The manager transforms the `p(X,Y)` into one and-box, and calls the emulator to create the sub-goals and or-boxes for `g(X)` and `f(Y)`. Control then returns to the manager if the boxes need to suspend. The design of the emulator and of the manager are the main topic of this paper.

The BEAM does not implement backtracking, so space must be recovered by the *garbage collector*. The BEAM is built on top of the YAP system and reuses the YAP *compiler* and *built-in library*.

4.1 Representing the And-Or Tree

An efficient implementation of the EAM requires data structures capable of performing the main operations of the EAM quickly and efficiently. In this section we briefly describe how the BEAM represents or-boxes, and-boxes, local, and external variables.

Or-boxes have open alternatives, represented as an array of *alternatives*. Each alternative includes a pointer of an array of arguments, code to execute, a *state*, and a pointer to the and-box for the clause, if it exists. The or-box also points back to the and-box and goal that created this alternative.

And-boxes represent clauses. Clauses have *local* variables, and subgoal. Subgoals may point to a corresponding or-box, and always include code to execute and state information. And-boxes also mantain a depth-counter (`level`) that is used to classify variables. And-boxes also maintain a list of `external` variables and a pointer back to the parent alternative.

Local Variables may belong a single sub-goal, or may be shared by several goals. The *value* field is used to stores the current value for a variable. Terms are represented using a copying based scheme, as in the WAM. Unbound variables are represented as self-pointers. Variables also point to their *home* and-box, and maintain a list of and-box *suspensions* on them.

External Variables correspond to bindings for variables external to the current and-box. Each such binding is represented as an external variable data-structure that include a pointer to the variable definition, *local_var*, and the new value, *value*.

4.2 The Abstract Machine

The internal state of the BEAM abstract machine is stored in the following registers: `PC`: program counter; `H`: top of heap; `S`: structure pointer (in Heap); `Mode`: controls whether unification is in read or write mode; `X1,X2,...`: registers for temporary variables, also used as arguments registers; `OBX`: current or-box; `ABX`: current and-box; `SU`: pointer to the start of the suspensions list.

4.2.1 Abstract Machine Instructions

Code for the BEAM abstract machine very closely follows the WAM. The BEAM abstract machine instructions uses the WAM `get`, `put` and `unify` instructions, plus novel control instructions. The major difference between BEAM `get`, `put`, and `unify` and the corresponding WAM instructions is that in the BEAM permanent variable slots, or (`Y`) slots, refer in fact to the and-box local variables.

The main novel instructions are concerned with control:

`explore_alternative i:` explore the `i`th for the current or-box.

`prepare_calls n:` creates an and-box with `n` subgoals. Each subgoal record points to the start code for the call, and is initialized as `READY`, meaning that they are ready to be explored. Execution is then passed to the *manager*.

`call_pred:` creates one or-box with `n` branches, where `n` is the number of alternatives to `pred`. Each branch record points to the starting code of the corresponding alternative, and all branches are also initialized as `READY`, meaning that they are ready to be explored. Execution is then passed to the *manager*.

`proceed:` this instruction returns control from a clause to the *manager*. If the and-box does not have external variables, it has succeed, and enters through the `success` port. Otherwise, the and-box is marked as suspended, and execution enters the *manager* through the `suspend`.

Compiling Prolog clauses to the BEAM abstract machine instructions is very similar to WAM compilation. Fig. 4 graphically describes the process.

```
    a(...):- b(...), c(...).          a(...).
    explore_alternative 1             explore_alternative 2
    get's                             get's
    ...                               ...
    prepare_calls 2                   proceed
 -> put's
    ...
    call b
 -> put's
    ...
    call c
```

Figure 4: BEAM Abstract Machine Code.

Note that, unlike in the WAM, code for rules in the BEAM does not end with a `execute` instruction. In its current version, the BEAM abstract machine is goal based. The `prepare_calls` instruction creates an and-box with as many branches as calls, and initializes each branch to point the the start code of each call (marked with `->` in Fig. 4). It is up to the *manager* to decide how and when to execute the calls.

4.3 The And-Or-Tree Manager

The And-Or-Tree manager is the heart of our system. Its task is to decide which rewrite rule should be applied to the current tree, and then execute it. The rewrite rules are, the *Reduction* rule, the *Promotion* rule, and the *Forking* rule. Currently, the BEAM does not apply non-determinate promotion, hence *Forking* is only executed when no other rule can execute.

The *And-Or-Tree Manager* can be divided into eight different modules. Three modules correspond to the *entry ports*, that may be called from the emulator:

`success:` this module marks the current or-box as successful in its parent. The memory of the or-box is released. The parent and-box is checked and, if all calls have already reached `success`, the module is reentered for the upper or-box (*Success Propagation*). Otherwise, execution enters the `next_call` module.

`fail:` this routine marks the current and-box as failed in its parent. All the assignments made by the and-box are removed, and space for the and-box is reclaimed. If all the alternatives for the parent or-box have failed, the module is recursively called for the parent and-box (*Failure Propagation*). Otherwise, if there is only one more alternative, execution moves to `unique_alternative`. If there are several alternatives, execution continues to `next_alternative`.

`suspend:` this routine adds the and-box to the current suspension list. Next, the routine clears all assignments saved in the list of the external variables. Each external variable is also added to the suspension list included in the respective local variable. After that, the manager jumps to *next_alternative*.

The other modules select work to be executed:

`next_call:` this module searches for the next non-suspended call in the current and-box. If there is no ready call in the current and-box, execution moves to *next_alter-native*. Otherwise, the `PC` is set to point to the call's code, and the execution jumps to the emulator.

`next_alternative:` this routine searches for the next non-suspended alternative in the current or-box. If there is no such alternative, execution jumps to `next_call`. Otherwise, if the alternative is in the `WAKE` state, execution moves to `wake`, else execution sets the `PC` to the code for the alternative code, and enters the alternative.

`unique_alternative:` this module promotes the current and-box, since its parent or-box has a single alternative. All externals variables are checked, as after the promotion some external variables may have become local. If during the promotion of external variables unifications fail, execution moves to `fail`. If external variables still exist after the promotion, the and-box continues suspended, and execution moves to `next_call`. Otherwise, if the and-box has *suspended on end* execution can move to `success`; if goals are still left for running, the manager marks the and-box as *running* and continues its execution.

`wake:` this module wakes a suspended and-box. All externals variables are checked to look for changes. If an unification fails, execution jumps to `fail`. If externals variables still exist, execution continues to `top_of_tree`. If no more externals variables are left, and the and-box has *suspended on end*, it moves immediately to `success`, otherwise, it marks the and-box as *running* and continues its execution.

`top_of_tree (fork):` this modules looks for work in the suspension list. If no extra work is available in the suspension list, the execution can end. Otherwise, the `ABX` register is set to point to the next suspended box. If the current and-box is the single alternative available in its parent or-box, execution moves to `unique_alternative`. If there exist several alternatives, a *fork* is performed in the and-box. After the fork, one of the resulting and-box is waken, and its execution is restarted in `wake`.

5 The Implementation of Global Forking

Global forking is the most complex operation on the BEAM. We explain its implementation through a small example:

```
start:- a(X), b(...).      a(first).      a(second).
```

Fig. 5a presents the BEAM execution state, for the previous example, considering that the computation has suspended for both calls `a(X)` and `b(...)`. Note that the and-boxes (5) and (6) have tried to assign `first` and `second`, respectively, to `X` (local variable structure (3)), forcing both and-boxes to suspend. Note also that this fact have put both these and-boxes into the suspension list of the local variable `X`.

Fig. 5b presents the BEAM execution state after executing `fork`. The steps executed in `fork` are:

1. create a new or-box (8). The or-box is initialized as having a single alternative, pointing to the first suspended and-box (5). The branch of this and-box on the older or-box (4) is set to `FAIL` and removed.
2. create a new and-box (9) to the image of the older equivalent and-box (2). Set the first branch to point to the or-box (8) created in the previous step. Other branches are set to point to newly created or-boxes (10). These or-boxes are an exact copy of the branches in the older and-box, marked as (7).
3. add a new branch, representing a new alternative to the parent or-box (1). This new branch is set to point to the and-box (9) created in the previous step.

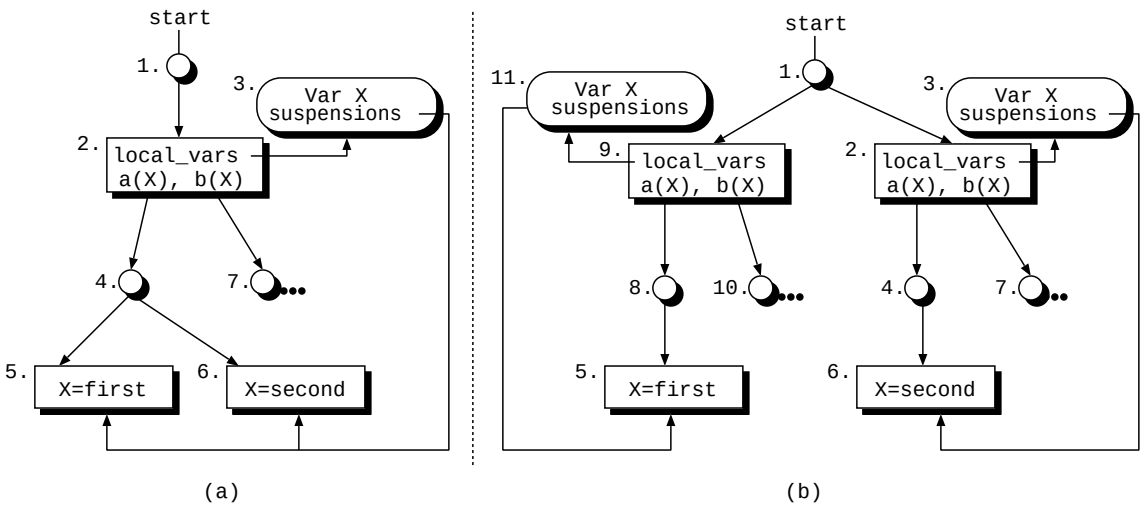

Figure 5: Fork example.

4. for all and-boxes created, create a new vector with new local variables. For instance, the `locals` pointer in and-box (9) is set to point to a new vector with new local variables (11) (in the example, the vector only has one variable).

5. set all external variables structures available on the and-box (9) and children to point to the new local variables, created in step 4. For example, the external variable structure in (5) is now set to point to the newly create local variable structure (11).

6. refresh the suspensions lists on the local variables. For example, the suspensions list for the local variables structure (3) of Fig. 5, had two elements, the and-box (6) and (5). After the fork, the variable structure (3) was set only to point the and-box (6). The and-box (5) is now included in the suspension list of the newly created local variable (11).

After these six steps, the and-box (5) and the and-box (6) are marked as `WAKE`, and both of them are ready to be promoted. The external variables will be promoted, and both and-box (5) and (6) will have their local variable `X` assigned to `first` and `second` respectively. The computation of `b(..)` on (10) and (7) will then be ready to continue.

6 The Implementation of Cut

Prolog programs often include side-effects predicates such as cut (!). On this section we first give an overview of some issues related with cut on the EAM and how we address them on the BEAM.

A cut in a Prolog program can prune work from two different areas of the and-or-tree (see Fig. 6):

$\{[\exists Y : \theta_1 \& \{[\exists X : \theta_2 \& X_1 \equiv \texttt{true}] \vee [X_2]\} \& \, ! \, \& \, Y] \vee W\} \rightarrow \{[\exists Y, X : \theta_1 \& \theta_2 \& Y]\}$

1. alternative clauses for the current predicate. On the and-or-tree the alternatives to be pruned are represented on the right side of the alternative containing the cut.

2. alternatives created by calls on the left side of the cut being executed.

On the BEAM we wanted the cut operation to perform as in Prolog. To archive this goal some execution principles of the EAM needed be restricted as they can lead the execution to a incorrect state in the presence of *cut*.

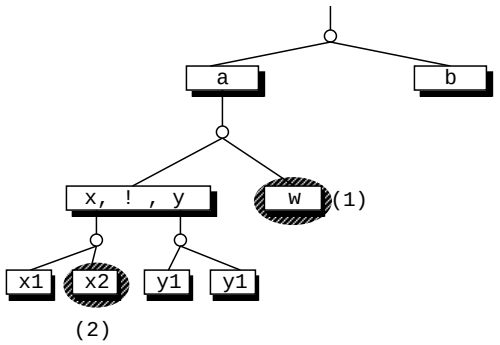

Figure 6: Cut scope.

6.1 Promotion and cut

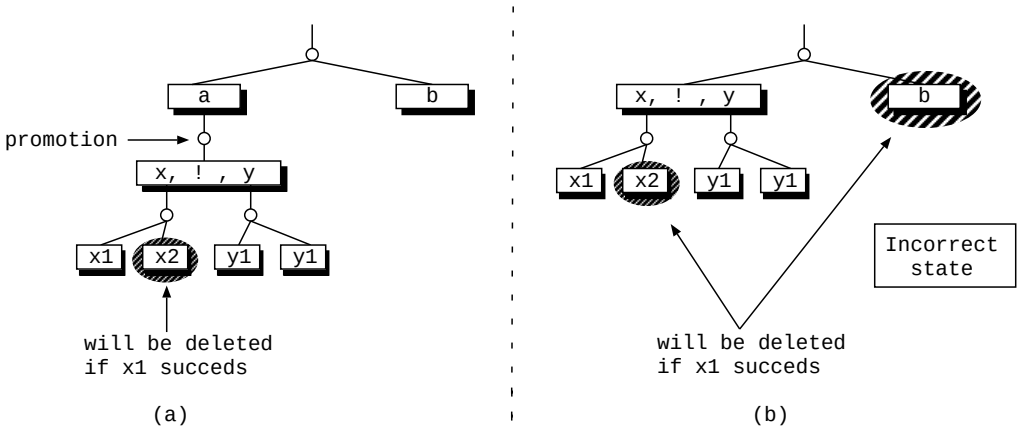

Figure 7: Incorrect use of promotion and cut.

Usually, an or-box with only one alternative can be removed and the and-box can be promoted. The problem is that the promotion of a clause that includes a cut can change the scope of the cut, allowing it to remove alternatives that were not supposed to be deleted. Fig. 7 shows an example of a promotion of an and-box containing a cut that leads to an incorrect state were the and-box b is in danger of being removed by the cut execution.

We address this problem by not allowing an and-box containing a cut to be fully promoted on BEAM. Instead, we allow the constraints to be promoted to the parent and-box but the and-box itself is kept separated.
$[\exists X, Z : A\&\{[\exists Y, W : Z = \theta(W)\&C]\}\&B] \rightarrow [\exists X, W, Z : Z = \theta(W)\&A\&\{[\exists Y : C]\}\&B]$

The and-boxes are kept separated, as shown on Fig. 7a, but any variable that is local to the parent and-box (a in the example) also becomes local to the and-box containing the cut. This is archived by changing the and-box parameter *level* [1] to be equal to the parent and-box. With this restriction the scope of the cut is not changed, and BEAM will never go into the incorrect state presented on Fig. 7b.

[1] each and-box maintains a depth counter (*level*) that is used to classify variables as local or external.

6.2 Fork and cut

On the BEAM a clause is suspended when it tries to bind variables that are not local. The clause will eventually be awaken and allowed to continue execution or it will be chosen to fork (if it is the leftmost).

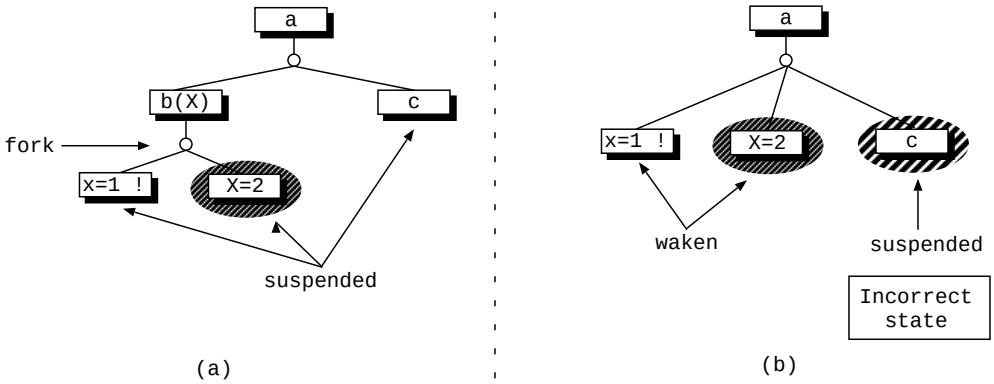

Figure 8: Incorrect use of fork and cut.

Consider the example presented on Fig. 8a. Suppose that all boxes are suspended and that the only available rule is fork. After forking the tree will look like in Fig. 8b. Of course this computational state is incorrect as the and-box c is in danger of being deleted by the cut execution.

This problem was solved in our EAM implementation by extending the scheduler with the following restrictions:

- a clause with a cut in its body is allowed to continue execution even if it binds external variables, although these binding are not not visible to the parent boxes. Normal scheduler rules apply to the and-box sons, as forks are safe on those (see example on Fig. 9).

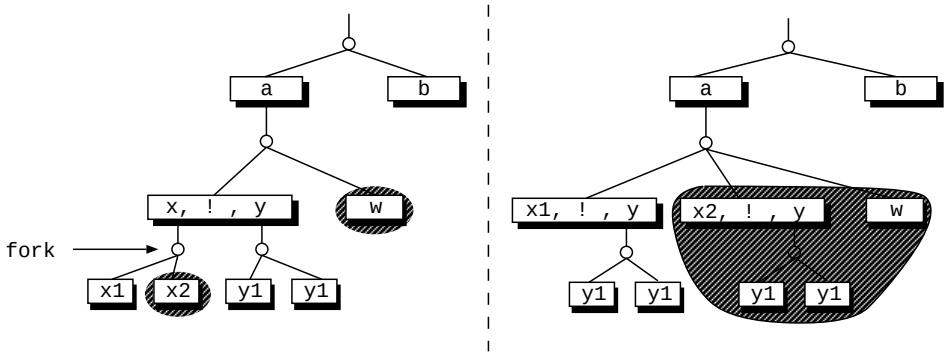

Figure 9: Correct use of fork and cut.

- a cut can be immediately executed if it is the first call in the and-box and if the and-box does not have external constraints.

- if an and-box containing a cut becomes leftmost in the tree, the cut can execute immediately when all the calls before it succeed (even if there are external constraints). For example, on Fig. 9b the cut is allowed to execute immediately when *X1* succeeds, even if there are external constraints.

Benchmark	BEAM	YAP98	YAP 4.3.0	Sictus 3.8.2
nreverse	0.51	0.18	0.15	0.43
qsort	1.15	0.36	0.32	0.91
query	13.41	2.15	0.86	2.14
ancestor	NA	0.08	0.06	0.14
ancestor (Eager Forking)	0.60	0.08	0.06	0.14
zebra	706	32	26	44
zebra (Eager Forking)	50	32	26	44
kkqueens (with cuts)	1,070	350	82	140
kkqueens (no cuts)	1,370	300	86	150
queens (9) (1st sol)	100	90	70	120
queens (10) (1st sol)	300	730	540	940
queens (11) (1st sol)	240	5,690	4,140	6,880
queens (12) (1st sol)	1,310	58,120	41,340	70,940
queens (13) (1st sol)	700	577,740	422,790	720,470
queens (14) (1st sol)	14,650	7,388,920	5,383,760	9,069,172
queens (15) (1st sol)	15,490		$>$ 20 hours	
queens (16) (1st sol)	140,180		$>$ 36 hours	
queens (17) (1st sol)	102,100			
queens (18) (1st sol)	967,110			
queens (19) (1st sol)	88,980			
queens (20) (1st sol)	9,743,890			
send_money (1st solution)	44	52,260	38,520	60,320
send_money (find_all)	242	380,000	285,260	426,790

Figure 10: Comparison between BEAM, YAP and Sicstus (time in miliseconds)

The extensions to the scheduler presented in this section allow early execution of cuts because and-boxes containing cuts are allowed to continue execution even if they have constraints on external variables.

These extensions go towards Warren's goal of obtaining a *Minimum number of inferences* because the early execution of cuts will in most cases delete alternatives and reduce the search space.

7 Early Results

This section presents results of the BEAM performance and compares them to the last version of Sicstus Prolog and YAP [1], an emulated Prolog system. The timings were measured running the benchmarks on a Pentium III 450MHz with 128Mb RAM running Red-Hat Linux 6.1.

Our initial goal is to detect what is the basic overhead for the BEAM. We could have arbitrarily better times for the BEAM by using benchmarks where the Andorra rule prunes the search space.

The results in fig. 10 are quite positive, considering that BEAM is still in prototype stage. The more impressive results of BEAM are archived with the *send_more_money* and *queens* benchmarks were BEAM archives a very good performance compared to traditional Prolog systems. These results are mainly due because EAM reduces the search space and explore reuse of goals, avoiding lots of recomputation. As an example consider the benchmark queens(15) where BEAM took about 15 seconds to find the first solution to the problem, and YAP and SICStus didn't find a solution after 20 hours of runtime (see fig. 10 and 11).

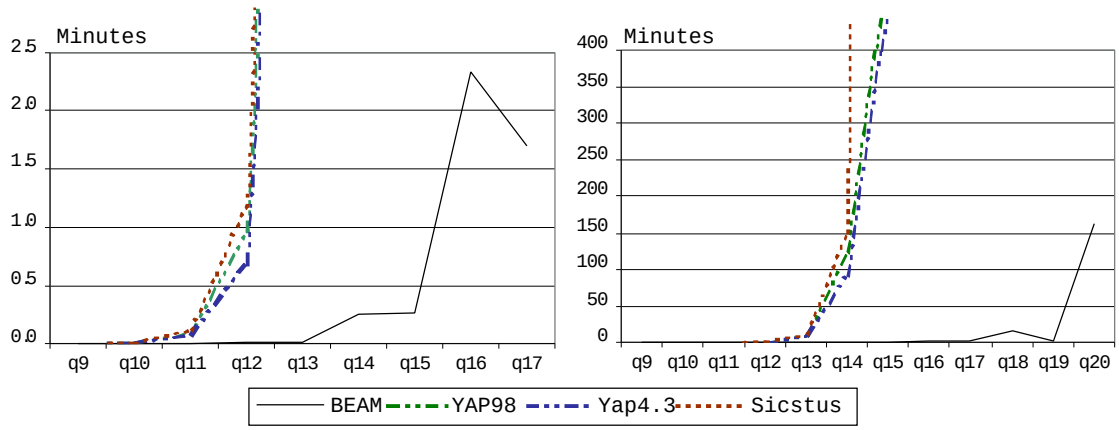

Figure 11: Time to find the 1st solution on the queens benchmark - BEAM vs others.

8 Conclusions

In this paper we have presented a first sequential implementation of the EAM, the BEAM. The current implementation handles the original rewrite rules of David Warren's Extended Andorra Model and also supports cut and arithmetic built-ins. BEAM includes additional rules that allow us to simplify the and-or-tree and to optimize the computation.

Our initial results show surprisingly good basic performance. The prototype is currently being improved by incorporating support for built-ins. We are also designing an implementation of parallelism for our system.

The BEAM provides the basic functionality of David H. D. Warren's EAM, and is therefore of interest by itself. The BEAM is also a step towards our ultimate goal of implementing the LIGHT-BEAM (Leftmost Innermost Goal Hopping Tabulation - Boxed EAM) that expands the BEAM with two main contributions: tabling, used to implement reuse and to control non-terminating computations [7], and lazy copying, used to increase the amount of parallelism [2].

Acknowledgments

The authors would like to acknowledge and thank the contribution and support that Fernando Silva gave to this work. The work has also benefitted from discussions with Ricardo Rocha, Gopal Gupta, and Enrico Pontelli. Our work has been partly supported by Fundação da Ciência e Tecnologia and JNICT under the Melodia project (JNICT contract PBIC/C/TIT/2495/95), and through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa PRAXIS. Ricardo Lopes thanks Fundação da Ciência e Tecnologia and PRAXIS (Praxis Grant BD/9277/96) for supporting his research.

References

[1] L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998.

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

[3] Gopal Gupta and Enrico Pontelli. Extended dynamic dependent And-parallelism in ACE. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic*

computation, July 20–22, 1997, Maui, HI, pages 68–79, New York, NY 10036, USA, 1997. ACM Press.

[4] Sverker Janson. *AKL – A Multiparadigm Programming Language*. SICS Dissertation Series 14, Uppsala University, 1994.

[5] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[6] Ricardo Lopes and Vítor Santos Costa. The BEAM: A First EAM Implementation. In *1999 Joint Conference on Declarative Programming (APPIA-GULP-PRODE) proceedings, L'Aquila, Italy*, September 1999.

[7] Ricardo Lopes and Vítor Santos Costa. The BEAM: Towards a first EAM Implementation. In *Parallelism and Implementation of Logic and Constraint Programming*, pages 87–106. Nova Sicence, 1999.

[8] Johan Montelius and Khayri A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 13(4), 1995.

[9] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[10] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[11] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[12] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[13] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[14] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[15] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.