The BEAM: A first EAM Implementation

Ricardo Lopes, Vítor Santos Costa

Technical Report Series: DCC-98-13

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

The BEAM: A first EAM Implementation

Ricardo Lopes, Vítor Santos Costa
{rslopes,vsc}@ncc.up.pt

LIACC, Universidade do Porto,
Rua do Campo Alegre, 823, 4150 Porto, Portugal

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.

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 [12] have been used to obtain performance close to imperative languages for some benchmarks [11]. Parallelism [8] 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 [14] 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) [13]. The Andorra-I prototype [9] 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. If determinacy cannot be found efficiently, applications with strong parallelism may need to be run sequentially.

The Extended Andorra Model was proposed as an extension of the Andorra Model that could address the major problems of the Basic Andorra Model, and approximate 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. David H. D. Warren and researchers at Bristol concentrated on the *Extended Andorra Model with Implicit Control* [14], where the goal was to obtain the advantages of the Extended Andorra Model with least effort from the programmer. Haridi, Janson and researchers at SICS concentrated on the AKL [4], 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.

Research in AKL concentrated on including novel features such as ports [3], important for concurrent programming, and on obtaining good performance through sequential [5] and parallel implementations, such as Penny [7], with good results. More recently, the AKL researchers have shifted their interest to Oz [10], a language developed at DFKI which provides similar features to AKL.

Research on the Extended Andorra Model with Implicit Control, that will refer to more shortly as *EAM*, followed a rather different approach. Research started by trying to prove the feasibility of the model by developing an interpreter. Gupta's interpreter [2] showed that the EAM was indeed feasible. Towards obtaining maximum performance, Warren proposed solutions such as lazy copying and eager producers to give finer control over search when required.

We report on the design of a first implementation for the EAM. As a first step we concentrate on the implementation main rewriting rules of the EAM. This subset of the EAM was implemented on our first prototype, the *BEAM*. We present the major data structures and execution algorithms that were required for efficient execution, with initial performance results. The paper is organized as follows. We first give an overview of the EAM, and present an example for better understanding of the model. Next, we discuss the basic data structures required to implement the model, and the BEAM execution model. Last, we present initial performance results.

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 which 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 shall not discuss choice-boxes further here.

There are four main operations in the Extended Andorra Model:

- **Reduction:** $G \rightarrow \{[\exists Y_1 : \sigma_1 \& C_1] \vee ... \vee [\exists Y_n : \sigma_n \& C_n]\}$
  Reduction expands a goal G in an and-box into an or-box.
- **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.
- **Substitution:** $[\exists X, Z : Z = \theta(W) \& C] \rightarrow [\exists X : C\theta]$
  Substitution applies a constraint on a variable.
- **Forking:** $[\exists X : \{(C_1 \vee ... \vee C_n) \ \& \ G] \rightarrow \{[\exists X : C_1 \& G] \vee ... \vee [\exists X : C_n \ \& \ G]\}$
  Forking distributes a conjunction across a disjunction.

2.2 An Example of EAM Execution

To better understand the essential ideas of how the computation of a logic program proceeds in the EAM, we next present a very simple example:

```
g(X):- a(X),b(X).     b(4).            a(1).     c(1).
                      b(X):- c(X).     a(2).     c(3).
```

Figure 1 shows several execution steps for this program and the query:

```
?- g(X).
```

Or-boxes are represented as black dots. And-boxes are represented as a rectangle divided in 3 areas. The left area is reserved to the local variables, the middle area to the conjunction of goals, and the right one is for the external variables. On the first step, after starting execution, the EAM would perform a reduction on `g(X)` followed immediately by a promotion since `g(X)` only has one alternative. On the second step, we perform a reduction on both `a(X)` and `b(X)`. Notice now there are two alternatives to `a(X)`. The first one tries to perform `X=1`, leading to suspension because `X` is an external variable. The second alternative will also suspend when trying to make `X=2`. Again there are also two alternatives to `b(X)`. The first one suspends when trying to make `X=4`. The next step (3) can only work on `c(X)` because all the others are suspended. A reduction is applied, creating two more alternatives. Both of them suspend immediately because they also try to assign a value to `X`. At this point, all possible alternatives are suspended, which leaves applying a nondeterminate promotion as the only rule. We decided to choose the or-box that is the leftmost on the tree. After forking, we can see the tree divided in two, with a or-box on top. The left side has `X` bound to 1. The right side has `X` bound to 2. After reactivating all the suspended and-boxes, most of them fail immediately, leaving just one solution, which is `X=1` on the left side of the tree.

2.3 Lazy Copying

In the original EAM a goal will always suspend when nondeterminately trying to bind an external variable, irrespective of whether it is the producer (the goal that generates bindings for) or the

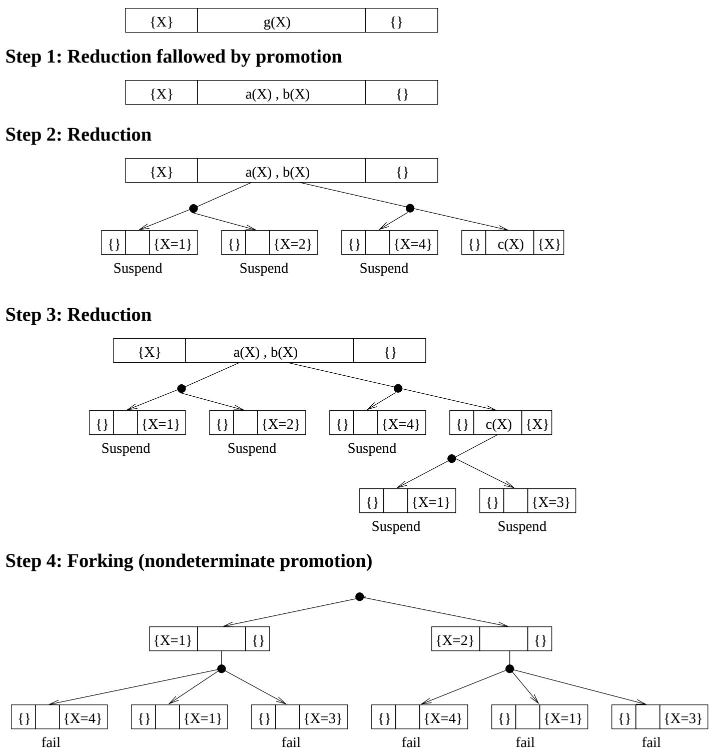

Figure 1: Execution in EAM.

consumer (the goal that reads bindings for) that variable. One important observation is that if the goal is the producer, there is no need for it to suspend. The EAM does in fact try to obtain extra parallelism by letting the producer continue. To avoid duplicating work, this is implemented by creating pointers from the producer to the consumer goal and not by creating a physical copy for every alternative of the producer. This copy of the consumer is performed lazily, hence the name *lazy copying*. The forking rule now becomes:

$$[\exists X : \{C_1 \vee ... \vee C_n\} \& G] \rightarrow \{[\exists X : C_1 \& G_p] \vee ... \vee [\exists X : C_n \& G_p]\}$$

where G_p is a reference to a single goal. One can do even better, by copying parts of G_p as soon as they suspend. To support this scheme the EAM now includes a classification scheme for variables as guessable or non-guessable. This classification might be provided by the programmer, or generated automatically. A variable is annotated as *guessable* if the annotator is certain it is going to be guessed non-determinately. In this case, as soon as a binding is generated by a producer for a guessable variable, global forking with lazy copying is performed. In contrast, global forking on non-guessable variables is only performed if there is no other way to restart the computation.

3 Representing the And-Or Tree

An efficient implementation of the EAM requires data structures capable of performing the four main operations of the EAM quickly and efficiently. In the following subsections we present the data structures that we have used in BEAM: or-boxes, and-boxes, local and external variables.

Or-Box : Or-boxes represent open alternatives. Figure 2 presents the structure of an or-box. Each or-box refers to its parent and-box through the *parent* pointer and the number of the call in the parent and-box that created this or-box, *nr_call*. *nr_all_alternatives* gives the number of alternatives available. Last, the box points to an array of *alternatives*, where each element in the array includes:

- a pointer to a corresponding and-box, *alternative*, initially empty, and initialised when the alternative is explored;
- a pointer to the goal arguments, *args*;
- a pointer to the code for the alternative, *code*; and,
- the *state* of the alternative. Initially, alternatives are in the *ready* state. They then move to the *running* state, from where they may reach the *success* or *fail* states, or they may enter the *suspend* state. Suspended alternatives will eventually move to the *wake* state. As an optimisation, if no more goals need to be executed for the goal but the alternative is suspended, the alternative enters a special *suspended_end* state.

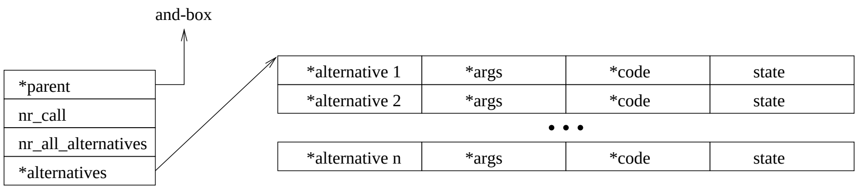

Figure 2: Or-Box representation.

Note that if *nr_all_alternatives* equals one, there is no need to keep the or-box, and determinate promotion can be applied to that only choice that has not failed. If *nr_all_alternatives* equals zero the box has failed.

And-Box : And-boxes require a more complex structure, as they must store information on what goals are active, and on external and internal variables. Figure 3 shows an and-box.

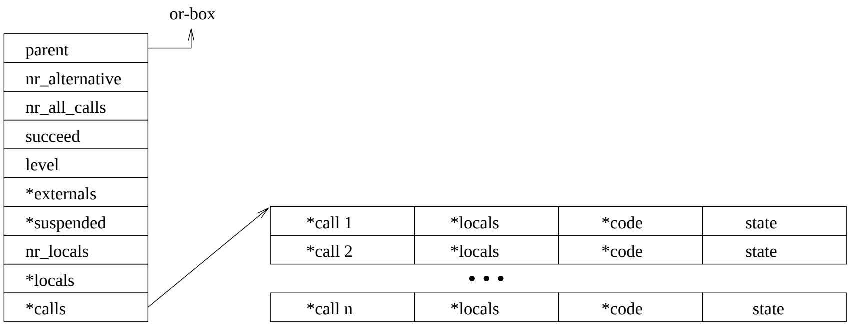

Figure 3: And-Box representation.

And-boxes represent clauses. Access to the parent is performed through the *parent* pointer to the parent or-box, and through the *nr_alternative*, indicating to which alternative it belongs. Call management, when performed, requires knowing the number of subgoals in the clause, *nr_all_calls*, and the number of goals which have already *succeed*. Variable control is discussed in more detail next. Variables local to this and-box have their home in the *locals* vector (the *nr_locals* field registers how many such variables exist). A list of external variables is accessed through the *externals* field. The and-box may have suspended on some variables, if so this is registered in the *suspended* field. Last, each subgoal or call has separate information:

- a pointer to a corresponding or-box, *call*, initially empty, and initialised when the call is open;

- a pointer to the *locals* vector;
- a pointer to the *code* for the subgoal;
- *State* information says whether the goal is *ready* to enter execution, is *running*, or has entered the *success* or *fail* states. Goals may also be *waiting* on some variable, upon which they will enter the *wake* state.

Each and-box also maintains a *depth* counter. This is required to classify variables.

Local Variables : these are variables new to a clause. They may belong a single sub-goal, or may be shared by several goals.

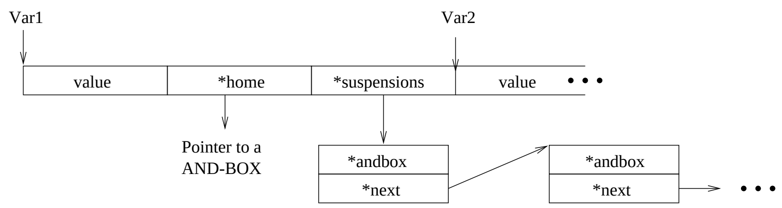

Figure 4: Local Variables representation.

Local variables are represented with the attributes shown in Figure 4. The *value* field 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-pointing. Variables also point to their *home* and-box, and maintain a list of and-box *suspensions* on them.

External Variables : goals may bind variables external to the current and-box. Each such binding is represented as an external variable data-structure. The attributes are shown in Figure 5. They

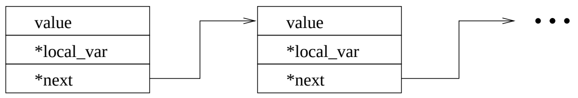

Figure 5: External Variables representation.

include a pointer to the variable definition, *local_var*, and the new value, *value*.

Note that whenever a goal binds an external variable, the assignment is recorded in *both* the external variable structure and in the local variable itself. This way, whenever a descendent and-box wants to use the value of this external-variable, it can do it simply by accessing to the local variable directly. The external_variable data structure therefore works as a trail, allowing to unwind and rewind the bindings performed from the current and-box.

4 BEAM Execution Model and Abstract Machine

There are five main components in the execution model of BEAM, as shown in Figure 6. They are:

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 4.1.2).

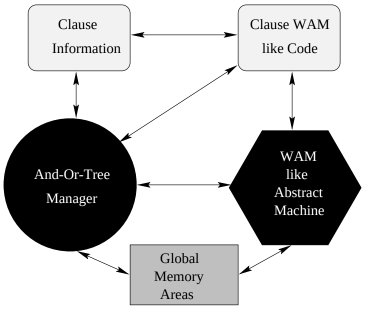

Figure 6: Execution Model.

Abstract Machine : 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 4.1.1).

Clause Information : This area contains the program's code and support information. It contains the next area, the

WAM-like code : this forms a sequence of abstract machine instructions for each clause.

Global Memory Areas : These operates like the heap in the WAM. They are used for saving variables values, compound terms like lists and structures, and even Prolog atoms. The area further serves requests for and-or-boxes. They include the next area,

The execution model presented has two main components. The And-Or-tree manager handles most of the complexity in the EAM. It uses the Clause information area to know how many alternatives one clause have and how many goals one clause calls. Using this information it constructs the tree and uses the EAM rewriting rules to manage it. It request memory for the boxes in the Global Memory Areas. The WAM-like Abstract Machine is called by the And-Or-tree manager to execute and unify the arguments of the goals and clauses. As an example consider the clause: `p(X,Y):- g(X), f(Y).` When running this clause the And-Or-Tree Manipulation transforms the `p(X,Y)` into one and-box, and calls the the WAM like Abstract Machine 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.

In the following subsections we will describe in more detail the implementation of the *WAM like Abstract Machine*, the *And-Or-Tree Manager*.

4.1 The Abstract Machine

The following registers describe the internal state of the BEAM abstract machine:

- `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.

4.1.1 Abstract Machine Instructions

Intermediate code for the BEAM abstract machine very closely follows the WAM style. The BEAM abstract machine instructions uses the WAM `get`, `put` and `unify` instructions, and implements novel control instructions.

The `get`, `put`, and `unify` instructions have small differences from the corresponding WAM instructions. In the BEAM, permanent variable slots, or (`Y`) slots, refer in fact to the and-box local variables.

The main new instructions are concerned with control:

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

`prepare_calls n` : creates a 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 *And-Or-Tree Manager* that will decide which call to execute.

`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 are branches also initialized as `READY`, meaning that they are ready to be explored. Execution is then passed to the *And-Or-Tree Manager*, that will decide which alternative to execute.

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

Compiling Prolog clauses to the BEAM abstract machine instruction is very similar to WAM compilation. Figure 7 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 7: BEAM WAM like Abstract Machine code.

Note that, unlike in the WAM, code for rules in the BEAM does not end with a `proceed` 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 Figure 7). It is up to the *And-Or-Tree Manager* to decide how and when to execute the calls.

4.1.2 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. There are three *entry ports*, that may be called from the abstract machine:

`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 routine jumps to the *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_alternative*. Otherwise, the `PC` is set to point to the call's code, and the execution jumps to abstract machine 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 there is no more work 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).
```

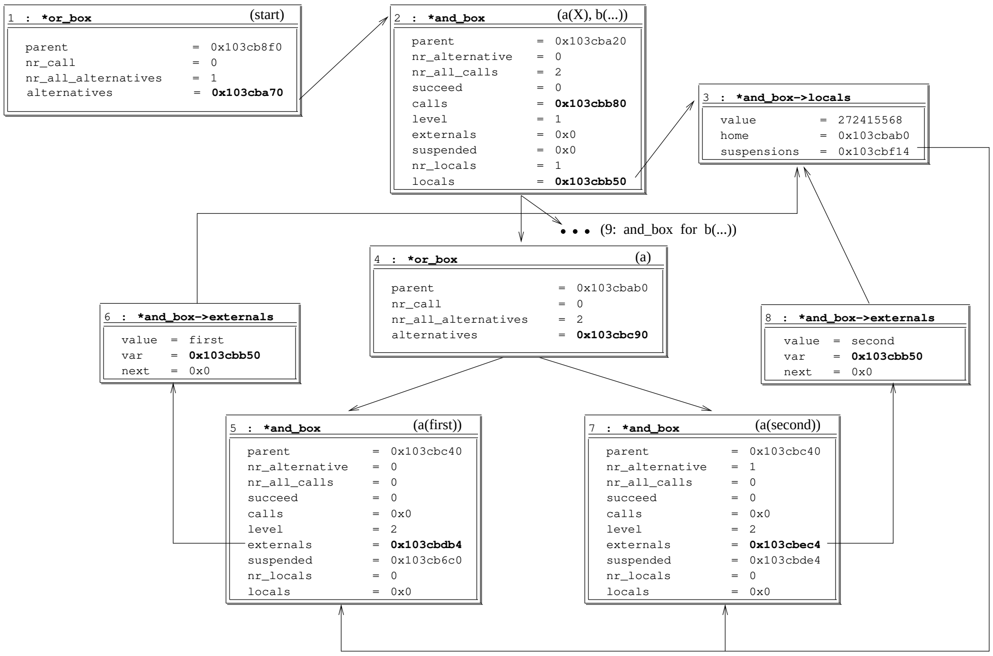

Figure 8: Example before using the fork rule.

Figure 8 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 (7) 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`.

Figure 9 presents the BEAM execution state after the executing `fork`. The steps executed in `fork` are:

1. create a new or-box (10). 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 (11) to the image of the older equivalent and-box (2). Set the first branch to point to the or-box (10) created in the previous step. Other branches are set to point to newly created or-boxes (12). These or-boxes are an exact copy of the branches in the older and-box, marked as (9).
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 (11) created in the previous step.
4. for all and-boxes created, create a new vector with new local variables. For instance, the `locals` pointer in and-box (11) is set to point to a new vector with new local variables (13) (in the example, the vector only has one variable).
5. set all external variables structures available on the and-box (11) and children to point to the new local variables, created in step 4. For example, the external variable structure (6) is now set to point to the newly create local variable structure (13).
6. refresh the suspensions lists on the local variables. For example, the suspensions list for the local variables structure (3) of figure 8, had two elements, the and-box (7) and (5). After the fork, the variable structure (3) was set only to point the and-box (7). The and-box (5) is now included in the suspension list of the newly created local variable (13).

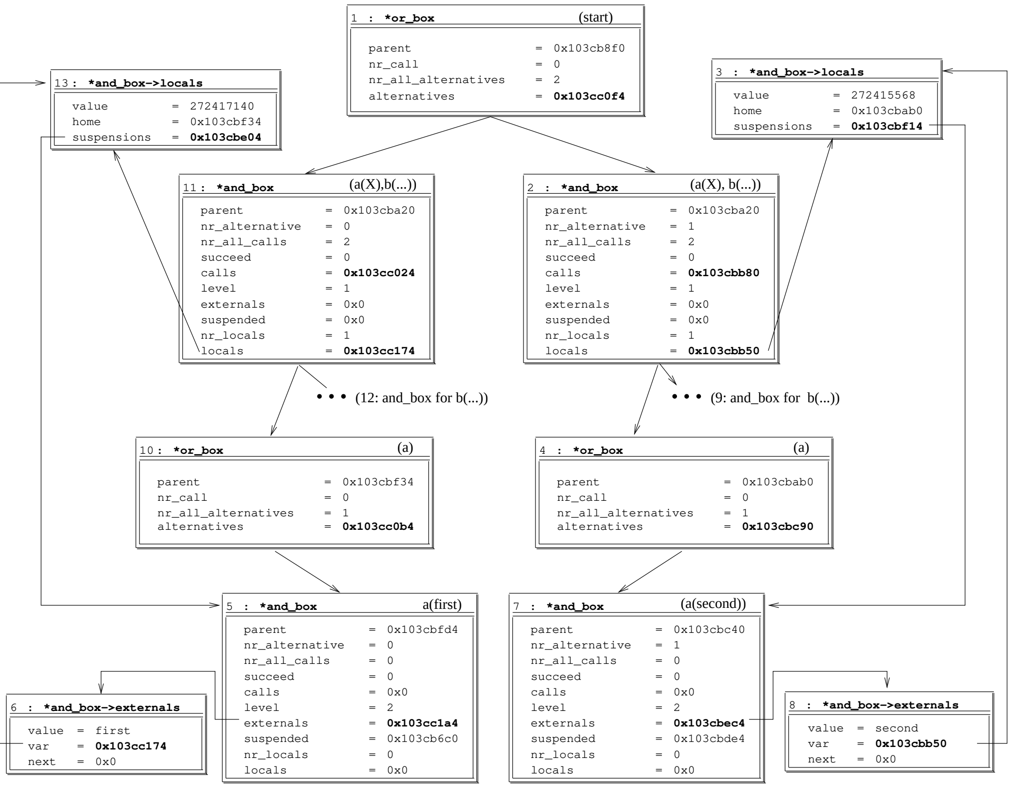

Figure 9: Example after using the first steps of the fork rule.

After these six steps, the and-box (5) and the and-box (7) 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 (7) will have their local variable `X` assigned to `first` and `second` respectively. The computation of `b(..)` on (12) and (9) will then be ready to continue.

6 Early Results

This section presents first results of the BEAM performance and compares them to YAP98 [1], an emulated Prolog system (YAP98 performance in these benchmarks outperforms emulated SICStus Prolog).

These are very preliminary results. Our initial goal is to detect what is the basic overhead for the BEAM, so we are using a deterministic benchmark. We could have arbitrarily better times for the BEAM by using benchmarks where the Andorra rule prunes the search space. naïve reverse, for two small queries:

- *Bench1*: Naive reverse of a 5 elements list.
- *Bench2*: Native reverse of a 10 elements list.

The results in Figure 10 are quite positive, considering that the BEAM is still in prototype stage and does not yet support indexing. Also, the current version of the BEAM is not optimised for deterministic computations: for each reduction the BEAM creates an or-box, a new and-box, just to do a determinate promotion in the sequence. We are designing a special rule for deterministic computations that simply rewrites the current and-box.

Benchmark	**BEAM**	**YAP98**
Bench1 (nreverse5)	1	2.07
Bench2 (nreverse10)	1	4.16

Figure 10: Speedup of YAP98 over BEAM

7 Conclusions

In this paper we have presented a first sequential implementation of the EAM, the BEAM. The current implementation of the BEAM handles the original rewrite rules of the David Warren's Extended Andorra Model for pure logic programs. Our initial results shows surprisingly good basic performance.

Our prototype is currently being improved by incorporating support for built-ins and for deterministic computations. To support built-ins we will include new abstract machine instructions that will block part of the and-box if some variables are unbound, or if the box is not leftmost. To support deterministic computations, we will include indexing and the new optimised deterministic rewrite rule.

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 [6], and lazy copying, used to increase the amount of parallelism.

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

[4] 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.

[5] Sverker Janson and Johan Montelius. Design of a Sequential Prototype Implementation of the Andorra Kernel Language. Sics research report, in preparation, Swedish Institute of Computer Science, 1992.

[6] Ricardo Lopes and Vítor Santos Costa. The BEAM: Towards a first EAM Implementation. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Port Jefferson*, October 1997.

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

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

[9] 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.

[10] 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.

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

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

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

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