Implementação de Linguagens de Programação Lógica

Extended Andorra Model

Ricardo Lopes
rslopes@ncc.up.pt
DCC-FCUP

Tópicos Avançados de Informática
Mestrado em Informática 2005/06
The Andorra Principle

- Much research on Logic Programming has been performed on improving the performance of Prolog through And/Or parallelism and concurrent execution.
Much research on Logic Programming has been performed on improving the performance of Prolog through And/Or parallelism and concurrent execution.

The work on these areas is substantial and already mature, however the combination of this work has introduced new difficulties affecting both language design and implementation.
The Andorra Principle

- Much research on Logic Programming has been performed on improving the performance of Prolog through And/Or parallelism and concurrent execution.
- The work on these areas is substantial and already mature, however the combination of this work has introduced new difficulties affecting both language design and implementation.
- In order to tackle this problem, several languages were proposed more recently, namely the Andorra-based languages.
The Andorra Principle

In 1987 D.H.D Warren proposed the Basic Andorra Model.

- A goal is *determinate* if it has at most one candidate clause.
The Andorra Principle

In 1987 D.H.D Warren proposed the Basic Andorra Model.

- A goal is determinate if it has at most one candidate clause.
- Determinate goals are executed first.
The Andorra Principle

In 1987 D.H.D Warren proposed the Basic Andorra Model.

- A goal is *determinate* if it has at most one candidate clause.
- Determinate goals are executed first.
- When no determinate goal exists, a non-determinate goal is selected for execution.
Advantages of the Basic Andorra Model:

- May reduce the search space.
Advantages of the Basic Andorra Model:

- May reduce the search space.

- Deterministic goals need to be tried only once, rather than re-executed at different branches of the search space.
Advantages of the Basic Andorra Model:

- May reduce the search space.
  - Deterministic goals need to be tried only once, rather than re-executed at different branches of the search space.
  - Constraints from the deterministic goals may reduce the number of alternatives for other goals, and even make them deterministic as well.
Advantages of the Basic Andorra Model:

- May reduce the search space.
  - Deterministic goals need to be tried only once, rather than re-executed at different branches of the search space.
  - Constraints from the deterministic goals may reduce the number of alternatives for other goals, and even make them deterministic as well.

- All deterministic goals can execute in parallel. The model supports two forms of parallelism extracted implicitly from the program:
  - **And-Parallelism**, by running deterministic goals in parallel.
Advantages of the Basic Andorra Model:

- May reduce the search space.
  - Deterministic goals need to be tried only once, rather than re-executed at different branches of the search space.
  - Constraints from the deterministic goals may reduce the number of alternatives for other goals, and even make them deterministic as well.

- All deterministic goals can execute in parallel. The model supports two forms of parallelism extracted implicitly from the program:
  - And-Parallelism, by running deterministic goals in parallel.
  - Or-Parallelism, by exploring different alternative clauses to a goal in parallel.
Advantages of the Basic Andorra Model:

- May reduce the search space.
  - Deterministic goals need to be tried only once, rather than re-executed at different branches of the search space.
  - Constraints from the deterministic goals may reduce the number of alternatives for other goals, and even make them deterministic as well.

- All deterministic goals can execute in parallel. The model supports two forms of parallelism extracted implicitly from the program:
  - **And-Parallelism**, by running deterministic goals in parallel.
  - **Or-Parallelism**, by exploring different alternative clauses to a goal in parallel.

- Programs are less execution order sensitive than in Prolog.

```prolog
?- member(X,L), L=[1,2,3]. is finite in the Andorra Model.
```
The Basic Andorra Model

An Andorra implementation has to deal with novel problems inherent to the model:
An Andorra implementation has to deal with novel problems inherent to the model:

- Determine which goals are deterministic can be hard.

```prolog
boss(mary, john). boss(mary, mary). ?- boss(X, X).
```
The Basic Andorra Model

An Andorra implementation has to deal with novel problems inherent to the model:

- Determine which goals are deterministic can be hard.
  ```prolog
  boss(mary, john). boss(mary, mary). ?- boss(X, X).
  ```

- Concurrency can break Prolog semantics.
  ```prolog
  ?- write(Solutions), fail. ?- var(X), X=a.
  ```
The Basic Andorra Model

An Andorra implementation has to deal with novel problems inherent to the model:

- Determine which goals are deterministic can be hard.
  ```prolog
  boss(mary, john).  boss(mary, mary).  ?- boss(X, X).
  ```

- Concurrency can break Prolog semantics.
  ```prolog
  ?- write(Solutions), fail.  ?- var(X), X=a.
  ```

- Programs may do more work.
  ```prolog
  a(1).  a(2).  b(3).  b(4).  ?- a(X), b(X), lots_determ_work.
  ```
The Andorra-I is the most well-known implementation of the Basic Andorra Model. It exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog.
Andorra-I

- The Andorra-I is the most well-known implementation of the Basic Andorra Model. It exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog.

- The Andorra-I system consists of three main components:
  1. **Preprocessor**: responsible for generating the determinacy code and the sequencing information necessary to maintain the correct execution of programs.
Andorra-I

- The Andorra-I is the most well-known implementation of the Basic Andorra Model. It exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog.

- The Andorra-I system consists of three main components:

  1. Preprocessor: responsible for generating the determinacy code and the sequencing information necessary to maintain the correct execution of programs.
  2. Engine: responsible for executing the Andorra-I programs. The engine consists of teams of workers, where each worker normally corresponds to a CPU.
The Andorra-I is the most well-known implementation of the Basic Andorra Model. It exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog.

The Andorra-I system consists of three main components:

1. **Preprocessor**: responsible for generating the determinacy code and the sequencing information necessary to maintain the correct execution of programs.
2. **Engine**: responsible for executing the Andorra-I programs. The engine consists of teams of workers, where each worker normally corresponds to a CPU.
3. **Scheduler**: responsible for finding new tasks (work) for workers that have completed their tasks.
The Andorra-I is the most well-known implementation of the Basic Andorra Model. It exploits or-parallelism and determinate dependent and-parallelism while fully supporting Prolog.

The Andorra-I system consists of three main components:

1. **Preprocessor**: responsible for generating the determinacy code and the sequencing information necessary to maintain the correct execution of programs.
2. **Engine**: responsible for executing the Andorra-I programs. The engine consists of teams of workers, where each worker normally corresponds to a CPU.
3. **Scheduler**: responsible for finding new tasks (work) for workers that have completed their tasks.

Despite the excellent results that the Andorra-I attained, the system is limited by the fact that coroutining and and-parallelism can only be exploited between determinate goals.
EAM Motivation: Attain Maximum Performance

In 1990 D.H.D Warren proposed the Extended Andorra Model (EAM)

- **Minimum number of inferences:**
  - try to never repeat the same execution step in different locations of the execution tree.
In 1990 D.H.D Warren proposed the Extended Andorra Model (EAM)

- **Minimum number of inferences:**
  - try to never repeat the same execution step in different locations of the execution tree.

- **Maximum parallelism:**
  - allow goals to be executed as independently as possible, and combine solutions as late as feasible.
EAM Motivation: Attain Maximum Performance

In 1990 D.H.D Warren proposed the Extended Andorra Model (EAM)

- **Minimum number of inferences:**
  - try to never repeat the same execution step in different locations of the execution tree.

- **Maximum parallelism:**
  - allow goals to be executed as independently as possible, and combine solutions as late as feasible.

- **Implicit control:**
  - ideal behavior without requiring too much reliance on user annotation.
EAM Characteristics

- **Extends the Basic Andorra Model** by allowing non-deterministic goals to execute in and-parallel as long they do not bind external variables.
  - **BAM** clause determinacy – don’t guess a clause till you have to.
  - **EAM** binding determinacy – don’t guess a variable binding till you have to.
EAM Characteristics

- **Extends the Basic Andorra Model** by allowing non-deterministic goals to execute in and-parallel as long they do not bind external variables.
  - **BAM** clause determinacy – don’t guess a clause till you have to.
  - **EAM** binding determinacy – don’t guess a variable binding till you have to.

- **Explore the three main forms of Parallelism:**
  - **Or-Parallelism:** between alternatives.
EAM Characteristics

- **Extends the Basic Andorra Model** by allowing non-deterministic goals to execute in and-parallel as long they do not bind external variables.

  - **BAM** clause determinacy – don’t guess a clause till you have to.

  - **EAM** binding determinacy – don’t guess a variable binding till you have to.

- **Explore the three main forms of Parallelism:**

  - **Or-Parallelism**: between alternatives.
  - **Independent And-Parallelism**: between goals that do not share variables.
EAM Characteristics

- **Extends the Basic Andorra Model** by allowing non-deterministic goals to execute in and-parallel as long they do not bind external variables.
  - **BAM clause determinacy** – don’t guess a clause till you have to.
  - **EAM binding determinacy** – don’t guess a variable binding till you have to.

- **Explore the three main forms of Parallelism:**
  - **Or-Parallelism**: between alternatives.
  - **Independent And-Parallelism**: between goals that do not share variables.
  - **Dependent And-Parallelism**: between goals that share variables.
EAM different approaches

- One approach was followed by researchers at SICS who concentrated on the AKL, 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.
One approach was followed by researchers at SICS who concentrated on the AKL, 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.

- **Explicit Control Scheme:** AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit.
One approach was followed by researchers at SICS who concentrated on the AKL, 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.

- **Explicit Control Scheme**: AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit.

In contrast, David H. D. Warren and researchers at Bristol and UP concentrated on the Extended Andorra Model with Implicit Control.
EAM different approaches

One approach was followed by researchers at SICS who concentrated on the AKL, 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.

- **Explicit Control Scheme:** AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit.

In contrast, David H. D. Warren and researchers at Bristol and UP concentrated on the Extended Andorra Model with Implicit Control.

- The goal was to obtain the advantages of the Extended Andorra Model with the least effort from the programmer.
EAM different approaches

One approach was followed by researchers at SICS who concentrated on the **AKL**, 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.

- **Explicit Control Scheme**: AKL programs were constructed from guarded clauses, where the guard could be separated with a sequential conjunction, cut, or commit.

In contrast, David H. D. Warren and researchers at Bristol and UP concentrated on the **Extended Andorra Model with Implicit Control**.

- The goal was to obtain the advantages of the Extended Andorra Model with the least effort from the programmer.
- **BEAM** was the first sequential implementation of the EAM.
The EAM is formally defined through rewrite rules that manipulate and-or trees.

- **And-boxes** – clause with subgoals $G_1, \ldots, G_n$; include:
  - local variables $X_1, \ldots, X_m$ created in the box,
  - and constraints, $\sigma$, on external variables imposed by the box.
The EAM is formally defined through rewrite rules that manipulate and-or trees.

- **And-boxes** – clause with subgoals $G_1, \ldots, G_n$; include:
  - local variables $X_1, \ldots, X_m$ created in the box,
  - and constraints, $\sigma$, on external variables imposed by the box.

- **Or-boxes** – alternative clauses for a goal, $C_1, \ldots, C_n$. 
The EAM is formally defined through rewrite rules that manipulate and-or trees.

- **And-boxes** – clause with subgoals $G_1, \ldots, G_n$; include:
  - local variables $X_1, \ldots, X_m$ created in the box,
  - and constraints, $\sigma$, on external variables imposed by the box.

- **Or-boxes** – alternative clauses for a goal, $C_1, \ldots, C_n$.

- **Rewrite rules:**
  - reduction, promotion, substitution (propagation) and forking (splitting).
The EAM is formally defined through rewrite rules that manipulate and-or trees.

- **And-boxes** – clause with subgoals $G_1, \ldots, G_n$; include:
  → local variables $X_1, \ldots, X_m$ created in the box,
  → and constraints, $\sigma$, on external variables imposed by the box.

- **Or-boxes** – alternative clauses for a goal, $C'_1, \ldots, C'_n$.

- **Rewrite rules:**
  → reduction, promotion, substitution (propagation) and forking (splitting).

- **Simplification and optimization rules:**
  → to simplify the and-or tree and to discard boxes.
The EAM is formally defined through rewrite rules that manipulate and-or trees.

- **And-boxes** – clause with subgoals $G_1, \ldots, G_n$; include:
  - local variables $X_1, \ldots, X_m$ created in the box,
  - and constraints, $\sigma$, on external variables imposed by the box.

- **Or-boxes** – alternative clauses for a goal, $C_1, \ldots, C_n$.

- **Rewrite rules:**
  - reduction, promotion, substitution (propagation) and forking (splitting).

- **Simplification and optimization rules:**
  - to simplify the and-or tree and to discard boxes.

- **Control strategies**
  - define how and when to apply the rules.
EAM Basics

- The computation is represented as an And-Or Tree.
- Execution in the EAM proceeds as a sequence of rewrite operations on a And-Or Tree.
Main Operations of BEAM I

*Reduction:* expands a goal $G$ into an or-box.
Main Operations of BEAM II

**Promotion**: promotes constrains from a inner-box to the outer-box.
Main Operations of BEAM III

**Propagation:** propagates constrains from an outer-box to the inner-boxes.
Main Operations of BEAM IV

Splitting: distributes a conjunction across a disjunction.
**EAM pruning rules**

- **Cut**
  - ![Diagram](CutDiagram.png)
- **Commit**
  - ![Diagram](CommitDiagram.png)
- **Implicit Pruning**
  - ![Diagram](ImplicitPruningDiagram.png)
Simplification Rules and Implicit Pruning I

false-in-or & true-in-or Simplifications:

```
C_1 \cdots C_{i-1} \text{false} \quad C_{i+1} \cdots C_n
C_1 \cdots C_{i-1} \text{true} \quad C_{i+1} \cdots C_n
```

```
C_1 \cdots C_{i-1} \text{true} \quad C_{i+1} \cdots C_n
```
true-in-and & false-in-and Simplifications:

\[
\begin{align*}
\text{X} & \quad \sigma \\
G_1 \ldots G_i-1 \quad \text{true} \quad G_{i+1} \ldots G_n \\
\end{align*}
\]

\[
\begin{align*}
\text{false} \\
G_1 \ldots G_{i-1} \quad G_{i+1} \ldots G_n \\
\end{align*}
\]
David Warren presented a set of intuitions on how to optimally evaluate $[\exists X : (P_1 \& \ldots \& P_n)]$:
Execution Control

David Warren presented a set of intuitions on how to optimally evaluate \[ \exists X : (P_1 \& \ldots \& P_n) \] :

- if we have no solution for \( X \), we only want to work on goals which will lead to failure;
David Warren presented a set of intuitions on how to optimally evaluate \(\exists X : (P_1 \& \ldots \& P_n)\):

- if we have no solution for \(X\), we only want to work on goals which will lead to failure;
- if we have one or more solutions to \(X\), it is safe to work on all goals;
**Execution Control**

David Warren presented a set of intuitions on how to optimally evaluate $[\exists X : (P_1 \& \ldots \& P_n)]$:

- if we have no solution for $X$, we only want to work on goals which will lead to failure;
- if we have one or more solutions to $X$, it is safe to work on all goals;
- if there are multiple solutions for $X$, we want to perform all work that does not depend on $X$ before installing different instantiations for $X$;
Execution Control

David Warren presented a set of intuitions on how to optimally evaluate \([\exists X : (P_1 \& \ldots \& P_n)]\):

- if we have no solution for \(X\), we only want to work on goals which will lead to failure;
- if we have one or more solutions to \(X\), it is safe to work on all goals;
- if there are multiple solutions for \(X\), we want to perform all work that does not depend on \(X\) before installing different instantiations for \(X\);
- if one goal can generate a unique value for \(X\), allow that goal to be the *producer* and others to be *consumers*. Otherwise we need to select one goal to be a *non-determinate producer* and let other goals be consumers;
Execution Control

David Warren presented a set of intuitions on how to optimally evaluate $[\exists X : (P_1 \& \ldots \& P_n)]$:

- if we have no solution for $X$, we only want to work on goals which will lead to failure;
- if we have one or more solutions to $X$, it is safe to work on all goals;
- if there are multiple solutions for $X$, we want to perform all work that does not depend on $X$ before installing different instantiations for $X$;
- if one goal can generate a unique value for $X$, allow that goal to be the producer and others to be consumers. Otherwise we need to select one goal to be a non-determinate producer and let other goals be consumers;
- use splitting solely to transmit non-determinate bindings from producer to consumers, making copies of consumers in the process.
Execution Control

The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.
Execution Control

- The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.

- The unrestricted application of the described rules will surely lead to a completely unpredictable and undesirable computation.
Execution Control

- The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.

- The unrestricted application of the described rules will surely lead to a completely unpredictable and undesirable computation.

- The original EAM design tries to keep the control implicit, as much as possible, that is, it does not rely on information supplied by the programmer.
**Execution Control**

- The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.

- The unrestricted application of the described rules will surely lead to a completely unpredictable and undesirable computation.

- The original EAM design tries to keep the control implicit, as much as possible, that is, it does not rely on information supplied by the programmer.

- The control decisions are based exclusively on information implicitly extracted from the program.
Execution Control

- The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.

- The unrestricted application of the described rules will surely lead to a completely unpredictable and undesirable computation.

- The original EAM design tries to keep the control implicit, as much as possible, that is, it does not rely on information supplied by the programmer.

- The control decisions are based exclusively on information implicitly extracted from the program.

- One of the aims of the Extended Andorra Model is to perform the least number of reductions to obtain the solutions for a goal.
BEAM Execution Control

The Extended Andorra Model gives us a set of basic operations which are logically valid, and which can be controlled in a variety of ways.

- Failure propagation rules have priority over all the other rules to allow propagation of failure as fast as possible.
- Success propagation and and-compression should always be done next, because they simplify the tree.
- Promotion and propagation should follow, because their combination may force some boxes to fail.
- Splitting is the most expensive operation, and should be used when no deterministic reductions are available.
**BEAM Execution Control**

**Control strategies:** define how and when to apply the rules.

![Control Flow Diagram]

- **Start**
  - **Simplify**
    - **Promotion**
      - yes
    - **Propagation**
  - yes
- **Reduction**
  - yes
  - no
- **Splitting**
  - yes
  - no
- **Halt**
false-in-and-simplication:

true-in-or-simplification:
The implicit pruning mechanisms we provide are not always sufficient for managing the search space.

The BEAM therefore supports two explicit pruning operators:

★ Cut (!)
★ Commit (|).

Both operators disallow goals to their right from exporting constraints to the goals to the left, prior to execution.

There are several issues to consider when using splitting or promotion within boxes that contain a cut.
EAM Explicit Pruning

► We next discuss some issues in the design of explicit pruning for the BEAM.

► In the following discussion we refer mainly to cut, but similar restrictions apply to the usage of the commit operator.
Correct use of fork/splitting and cut.
Incorrect use of fork/splitting and cut.

- This problem is solved in our BEAM by disallowing splitting to an and-box containing a cut.
- The scheduler allow clauses with a cut to continue execution even when head unification constrains external variables (note that these bindings may not be made visible to the parent boxes).
- In this regard the BEAM is close to the AGENTS, but differs in the fact that the cut does not provide sequencing on BEAM.
Correct use of splitting and cut.

Splitting can be applied freely to stable goals within the guard of the cut, as in this case splitting may not export constraints for variables external to the guard, or change the scope of the cut.
**Cut Leftmost example**

A cut can always execute immediately if it is the leftmost call in the and-box and if the and-box does not have external constraints (fig a).

In fig b the cut would not be allowed to execute since the alternative restricts the external variable $X$ to the value 3.
Cut in the leftmost box in the tree.

The cut is allowed to execute immediately when $X$ succeeds even if there are external constraints.
Incorrect use of promotion and cut.

We address this problem by disallowing and-compression when the inner and-box has a cut. **Promotion is still allowed.**

The and-boxes are kept separated (as fig a), but any variable that is local to the parent and-box \(A\) is also local to the and-box containing the cut (and vice versa).
**Prunning**

- We have presented how to address pruning in an implementation of the Extended Andorra Model with Implicit Control.
- Our approach contrasts with the AKL design, where pruning is embedded in the language.
- The EAM offers natural support for implicit pruning, but explicit pruning is more complex than in Prolog.
- Our results show that both implicit and explicit pruning are required in actual logic programming systems.
Non-Termination

As long as it does not bind variables, the EAM allows the parallel early execution of non-determinate goals. In some cases, this may create speculative and-work, as some other goals might fail. This may result in a larger search space or even lead to non-termination.
Non-Termination

As long as it does not bind variables, the EAM allows the parallel early execution of non-determinate goals. In some cases, this may create speculative and-work, as some other goals might fail. This may result in a larger search space or even lead to non-termination.

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).

parent(a,fa).  parent(a,ma).  parent(b,fb).  parent(b,mb).
parent(ma,mma). parent(ma,fma). parent(mb,c).  parent(mb,d).

?- ancestor(a,Z), ancestor(b,Z).
```
Implementação de Linguagens de Programação Lógica

1. \( \{ Z \} \quad a(Z) \quad a(b,Z) \quad \{ \} \)

2. \( \{ \} \quad p(a,Z) \quad Z \)
   - \( Z=fa \) (suspend)
   - \( Z=ma \) (suspend)

3. \( \{ X1 \} \quad p(a,X1) \quad a(X1,Z) \quad \{ Z \} \)
   - \( X1=fa \) (suspend)
   - \( X1=ma \) (suspend)

4. \( \{ X2 \} \quad p(X1,X2) \quad a(X2,Z) \quad \{ Z,X1 \} \)
   - \( X2=fa \) (suspend)
   - \( X2=ma \) (suspend)

Computation will never stop
Dealing with Non-Termination

- The solution that we proposed is the combination of:
  - Eager Non-Determinate Promotion
Dealing with Non-Termination

The solution that we proposed is the combination of:

- Eager Non-Determinate Promotion
- Tabling
Dealing with Non-Termination

The solution that we proposed is the combination of:

- Eager Non-Determinate Promotion
- Tabling

This solutions is ideal in two contexts:

- First it guarantees that the computation end in programs that have finite solutions.
- Second, it allows the reuse of goals.
On the End we would have for $a(a,Z)$ the solutions:
$Z=fa$ or $Z=ma$ or $Z=c$ or $Z=d$ or $Z=mma$ or $Z=fma$

On the End we would have for $a(b,Z)$ the solutions:
$Z=fb$ or $Z=mb$ or $Z=c$ or $Z=d$ or $Z=mfb$ or $Z=ffb$
Improving Memory Usage

A critical issue in the design of logic programming systems is their memory performance, both in terms of total memory usage and locality.
Improving Memory Usage

- A critical issue in the design of logic programming systems is their memory performance, both in terms of total memory usage and locality.

- BEAM is a sequential implementation of the Extended Andorra Model and as most modern Prolog systems, requires both good emulator design and good memory performance for best performance.
Improving Memory Usage

- A critical issue in the design of logic programming systems is their memory performance, both in terms of total memory usage and locality.

- BEAM is a sequential implementation of the Extended Andorra Model and as most modern Prolog systems, requires both good emulator design and good memory performance for best performance.

- Next we report on a detailed study of the memory management techniques used on BEAM.
Improving Memory Usage

- A critical issue in the design of logic programming systems is their memory performance, both in terms of total memory usage and locality.

- BEAM is a sequential implementation of the Extended Andorra Model and as most modern Prolog systems, requires both good emulator design and good memory performance for best performance.

- Next we report on a detailed study of the memory management techniques used on BEAM.

- We also propose a finer variable allocation scheme to reduce memory overheads that is quite effective at reducing memory pressure, with only a small overhead.
BEAM Memory Architecture

BEAM stores the And-Or Tree in two main areas:

- **Box Memory**: stores dynamic data structures including boxes and variables.
- **Heap**: uses term copying to store compound terms (similar to the WAM).
BEAM Memory Architecture

- BEAM stores the And-Or Tree in two main areas:
  - **Box Memory**: stores dynamic data structures including boxes and variables.
  - **Heap**: uses term copying to store compound terms (similar to the WAM).

- Standard WAM implementations follow a stack discipline, where space can be recovered during backtracking.
BEAM Memory Architecture

- BEAM stores the And-Or Tree in two main areas:
  - **Box Memory**: stores dynamic data structures including boxes and variables.
  - **Heap**: uses term copying to store compound terms (similar to the WAM).

- Standard WAM implementations follow a stack discipline, where space can be recovered during backtracking.

- The EAM control strategy is much more flexible, thus BEAM must carefully detect the points where to recover space.
BEAM Memory Architecture

- BEAM stores the And-Or Tree in two main areas:
  - **Box Memory**: stores dynamic data structures including boxes and variables.
  - **Heap**: uses term copying to store compound terms (similar to the WAM).

- Standard WAM implementations follow a stack discipline, where space can be recovered during backtracking.

- The EAM control strategy is much more flexible, thus BEAM must carefully detect the points where to recover space.

- We have two techniques to recover space:
  - Reuse space for pruned boxes.
  - Garbage collect useless data.
Reusing Space in the And-Or-Tree

The *Box Memory* must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).
Reusing Space in the And-Or-Tree

- The *Box Memory* must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).

- Objects are small and most will have short lifetimes. Minimizing allocation and deallocation overheads is crucial.
Reusing Space in the And-Or-Tree

- The *Box Memory* must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).

- Objects are small and most will have short lifetimes. Minimizing allocation and deallocation overheads is crucial.

- Unfortunately, we cannot recover space through backtracking. Instead we rely on a hybrid algorithm to allocate space:
The *Box Memory* must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).

Objects are small and most will have short lifetimes. Minimizing allocation and deallocation overheads is crucial.

Unfortunately, we cannot recover space through backtracking. Instead we rely on a hybrid algorithm to allocate space:

1. *We keep a top of stack and an array of* \( n \) *buckets*
The Box Memory must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).

Objects are small and most will have short lifetimes. Minimizing allocation and deallocation overheads is crucial.

Unfortunately, we cannot recover space through backtracking. Instead we rely on a hybrid algorithm to allocate space:

1. We keep a top of stack and an array of $n$ buckets
2. To allocate a block we check whether bucket $i$ has a free block. Otherwise we allocate the block from the top of the stack.
Reusing Space in the And-Or-Tree

- The *Box Memory* must satisfy intensive requests for the creation of objects (boxes, variables, external references and suspension lists).

- Objects are small and most will have short lifetimes. Minimizing allocation and deallocation overheads is crucial.

- Unfortunately, we cannot recover space through backtracking. Instead we rely on a hybrid algorithm to allocate space:
  1. We keep a top of stack and an array of $n$ buckets
  2. To allocate a block we check whether bucket $i$ has a free block. Otherwise we allocate the block from the top of the stack.
  3. To release a block of size $i$ we add it to the front of the list for bucket $i$. 

## Box Memory Reuse

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Requested</th>
<th>Local Vars</th>
<th>Reuses</th>
<th>Memory used</th>
</tr>
</thead>
<tbody>
<tr>
<td>nreverse</td>
<td>33Kb</td>
<td>84%</td>
<td>4.13%</td>
<td>32Kb</td>
</tr>
<tr>
<td>qsort</td>
<td>82Kb</td>
<td>47%</td>
<td>59.51%</td>
<td>33Kb</td>
</tr>
<tr>
<td>kkqueens</td>
<td>41005Kb</td>
<td>46%</td>
<td>49.27%</td>
<td>20800Kb</td>
</tr>
<tr>
<td>tak</td>
<td>24599Kb</td>
<td>45%</td>
<td>72.47%</td>
<td>6773Kb</td>
</tr>
<tr>
<td>houses</td>
<td>558Kb</td>
<td>49%</td>
<td>84.02%</td>
<td>89Kb</td>
</tr>
<tr>
<td>query</td>
<td>2108Kb</td>
<td>7%</td>
<td>85.40%</td>
<td>308Kb</td>
</tr>
<tr>
<td>query-ES</td>
<td>435Kb</td>
<td>32%</td>
<td>90.14%</td>
<td>43Kb</td>
</tr>
<tr>
<td>zebra</td>
<td>5806Kb</td>
<td>38%</td>
<td>94.41%</td>
<td>325Kb</td>
</tr>
<tr>
<td>zebra-ES</td>
<td>2296Kb</td>
<td>38%</td>
<td>91.82%</td>
<td>188Kb</td>
</tr>
<tr>
<td>scanner</td>
<td>2687Kb</td>
<td>75%</td>
<td>64.86%</td>
<td>944Kb</td>
</tr>
<tr>
<td>queens-9</td>
<td>180140Kb</td>
<td>70%</td>
<td>98.34%</td>
<td>2982Kb</td>
</tr>
</tbody>
</table>
Boxes and Variables Longevity
Recovering Heap Space

- The algorithm used to reuse memory space in the *Box Memory* does not work for the *Heap* because the BEAM releases memory eagerly, and the released objects in the *Heap* tend to be very small.
Recovering Heap Space

- The algorithm used to reuse memory space in the *Box Memory* does not work for the *Heap* because the BEAM releases memory eagerly, and the released objects in the *Heap* tend to be very small.

- We implemented a *copying* garbage collector for the BEAM: living data structures are copied to a new memory area and the old memory area is released.
Recovering Heap Space

- The algorithm used to reuse memory space in the *Box Memory* does not work for the *Heap* because the BEAM releases memory eagerly, and the released objects in the *Heap* tend to be very small.

- We implemented a *copying* garbage collector for the BEAM: living data structures are copied to a new memory area and the old memory area is released. The garbage collector starts by replicating the living data in the root of the And-Or Tree and then follows a top-down-leftmost approach.
Recovering Heap Space

- The algorithm used to reuse memory space in the *Box Memory* does not work for the *Heap* because the BEAM releases memory eagerly, and the released objects in the *Heap* tend to be very small.

- We implemented a *copying* garbage collector for the BEAM: living data structures are copied to a new memory area and the old memory area is released. The garbage collector starts by replicating the living data in the root of the And-Or Tree and then follows a top-down-leftmost approach.

- The advantage of a *stop-and-copy* GC is that the execution time for the garbage collection is proportional to the amount of data in use by the system.
Recovering Heap Space

- The algorithm used to reuse memory space in the Box Memory does not work for the Heap because the BEAM releases memory eagerly, and the released objects in the Heap tend to be very small.

- We implemented a copying garbage collector for the BEAM: living data structures are copied to a new memory area and the old memory area is released. The garbage collector starts by replicating the living data in the root of the And-Or Tree and then follows a top-down-leftmost approach.

- The advantage of a stop-and-copy GC is that the execution time for the garbage collection is proportional to the amount of data in use by the system.

- The main disadvantage is that the system can only use part of the total memory available.
Performance Analysis

We have experimented running different Prolog programs with different memory configurations:
Performance Analysis

We have experimented running different Prolog programs with different memory configurations:

- **BEAM-OnlyGC**: uses the garbage collector to recover memory in the Box Memory and Heap.
Performance Analysis

We have experimented running different Prolog programs with different memory configurations:

- **BEAM-OnlyGC**: uses the garbage collector to recover memory in the Box Memory and Heap.

- **BEAM-HybGC**: uses the hybrid algorithm to reuse memory in the Box Memory and the garbage collector to recover memory in the Heap.
Performance Analysis

We have experimented running different Prolog programs with different memory configurations:

- **BEAM-OnlyGC**: uses the garbage collector to recover memory in the Box Memory and Heap.

- **BEAM-HybGC**: use the hybrid algorithm to reuse memory in the Box Memory and the garbage collector to recover memory in the Heap.

- **BEAM-GCHybGC**: use our hybrid algorithm to reuse memory in the Box Memory and the garbage collector to recover memory on Heap. Moreover, in this version, whenever the garbage collector is recovering Heap memory, it also compacts the data on the Box Memory.
**BEAM running queens-9 with different memory configs**

We have chosen the queens-9 benchmark for a deeper analysis of BEAM memory management.

<table>
<thead>
<tr>
<th>queens9</th>
<th>BEAM-OnlyGC</th>
<th>BEAM-HybGC</th>
<th>BEAM-GCHybGC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heap Mem</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC</td>
<td>GC</td>
</tr>
<tr>
<td></td>
<td>1Mb</td>
<td>1,572 (382 BM)</td>
<td>1,320 (253 H)</td>
</tr>
<tr>
<td>1</td>
<td>2Mb</td>
<td>1,492 (182 BM)</td>
<td>1,290 (122 H)</td>
</tr>
<tr>
<td></td>
<td>4Mb</td>
<td>1,429 (89 BM)</td>
<td>1,293 (60 H)</td>
</tr>
<tr>
<td>0</td>
<td>8Mb</td>
<td>1,396 (44 BM)</td>
<td>1,289 (30 H)</td>
</tr>
<tr>
<td>2</td>
<td>16Mb</td>
<td>1,373 (22 BM)</td>
<td>1,288 (14 H)</td>
</tr>
<tr>
<td>4</td>
<td>32Mb</td>
<td>1,349 (11 BM)</td>
<td>1,282 (7 H)</td>
</tr>
<tr>
<td>Kb</td>
<td>64Mb</td>
<td>1,310 (5 BM)</td>
<td>1,282 (3 H)</td>
</tr>
<tr>
<td>L2</td>
<td>128Mb</td>
<td>1,271 (2 BM)</td>
<td>1,272 (1 H)</td>
</tr>
<tr>
<td></td>
<td>256Mb</td>
<td><strong>1,240</strong> (1 BM)</td>
<td>1,265 (0)</td>
</tr>
<tr>
<td></td>
<td><strong>400Mb</strong></td>
<td><strong>1,228</strong> (0)</td>
<td>1,265 (0)</td>
</tr>
</tbody>
</table>
Local Variables Representation

- Variables are a major source of memory pressure (boxes have a very short lifetime, while variables live longer).
Local Variables Representation

- Variables are a major source of memory pressure (boxes have a very short lifetime, while variables live longer).

- In the initial BEAM implementation all variables were processed the same way.
Local Variables Representation

- Variables are a major source of memory pressure (boxes have a very short lifetime, while variables live longer).

- In the initial BEAM implementation all variables were processed the same way.

- Every and-box maintains a list of its local variables, and every variable would be in some and-box (*permanent* variables).
Local Variables Representation

- Variables are a major source of memory pressure (boxes have a very short lifetime, while variables live longer).

- In the initial BEAM implementation all variables were processed the same way.

- Every and-box maintains a list of its local variables, and every variable would be in some and-box (permanent variables).

- Processing all variables the same way has major drawbacks. specially sinced there is a large portion of memory that can only be released when the and-boxes fail or succeed.
Classification of Variables at Compile and Run-Time.

Therefore, we would like to classify some variables as *temporary*: vars used to create an and-box and that do not need support for suspension.
Classification of Variables at Compile and Run-Time.

Therefore, we would like to classify some variables as temporary: vars used to create an and-box and that do not need support for suspension. Unfortunately, in general we do not know beforehand if we will need to suspend on a variable.
Classification of Variables at Compile and Run-Time.

Therefore, we would like to classify some variables as *temporary*: vars used to create an and-box and that do not need support for suspension. Unfortunately, in general we do not know beforehand if we will need to suspend on a variable.

We proposed a WAM-inspired scheme, the **BEAM-Lazy**.

- Like in the WAM, variables that appear only in the bodies of clauses or in queries are classified at compile time as *permanent* variables, meaning that all data-structures required for suspension are created for them.
Classification of Variables at Compile and Run-Time.

Therefore, we would like to classify some variables as *temporary*: vars used to create an and-box and that do not need support for suspension. Unfortunately, in general we do not know beforehand if we will need to suspend on a variable.

We proposed a WAM-inspired scheme, the **BEAM-Lazy**.

- Like in the WAM, variables that appear only in the bodies of clauses or in queries are classified at compile time as *permanent* variables, meaning that all data-structures required for suspension are created for them.

- All others variables are classified on compile time as *temporary*. 
Classification of Variables at Compile and Run-Time.

Therefore, we would like to classify some variables as *temporary*: vars used to create an and-box and that do not need support for suspension. Unfortunately, in general we do not know beforehand if we will need to suspend on a variable.

We proposed a WAM-inspired scheme, the **BEAM-Lazy**.

» Like in the WAM, variables that appear only in the bodies of clauses or in queries are classified at compile time as *permanent* variables, meaning that all data-structures required for suspension are created for them.

» All others variables are classified on compile time as *temporary*.

» Moreover, variables that are needed to create compound terms in write mode need not to be *permanent*. 
Classification of Variables at Compile and Run-Time.

As an example, consider the second clause of the nreverse:

\[
nreverse([X|L_0],L) :- nreverse(L_0,L_1), \text{concatenate}(L_1,[X],L).\]

\[\Rightarrow\] L1 is the only variable that is classified as \textit{permanent} at compilation time. The other variables are classified as \textit{temporary}. 
Classification of Variables at Compile and Run-Time.

As an example, consider the second clause of the nreverse:

\[
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
\]

- L1 is the only variable that is classified as permanent at compilation time. The other variables are classified as temporary.

- The system may need to create two more permanent variables, X and L0, when the clause is called with the first argument as variable.
Classification of Variables at Compile and Run-Time.

The advantage of this classification, is that:

* classifying a group of variables as temporary requires less memory
The advantage of this classification, is that:

★ classifying a group of variables as *temporary* requires less memory
★ improves performance since we avoid managing the more complex structure of the *permanent* variables.
Classification of Variables at Compile and Run-Time.

The advantage of this classification, is that:

- classifying a group of variables as *temporary* requires less memory
- improves performance since we avoid managing the more complex structure of the *permanent* variables.
- *temporary* variables can be immediately released after executing all the *put* instructions in the clause body, unlike *permanent* variables that can only be released when the and-box succeeds or fails.
Classification of Variables at Compile and Run-Time.

The advantage of this classification, is that:

- classifying a group of variables as temporary requires less memory
- improves performance since we avoid managing the more complex structure of the permanent variables.
- temporary variables can be immediately released after executing all the put instructions in the clause body, unlike permanent variables that can only be released when the and-box succeeds or fails.

One disadvantage of this classification of variables, is that the implementation of the instructions in the abstract machine is more complex, with instructions requiring extra tests to determine if the system needs to create new permanent variables.
## Deterministic Benchmarks

<table>
<thead>
<tr>
<th>Benchs.</th>
<th>BEAM</th>
<th>AGENTS</th>
<th>Andorra-l</th>
<th>YAP 4.2</th>
<th>YAP 5.0</th>
<th>SICStus 3.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal</td>
<td>0.0035</td>
<td>0.0051</td>
<td>0.0050</td>
<td>0.0021</td>
<td>0.0007</td>
<td>0.0010</td>
</tr>
<tr>
<td>deriv</td>
<td>0.0320</td>
<td>0.1190</td>
<td>0.0300</td>
<td>0.0139</td>
<td>0.0078</td>
<td>0.0100</td>
</tr>
<tr>
<td>qsort</td>
<td>0.2000</td>
<td>0.3280</td>
<td>0.1900</td>
<td>0.0510</td>
<td>0.0440</td>
<td>0.0450</td>
</tr>
<tr>
<td>serialise</td>
<td>0.1130</td>
<td>0.1970</td>
<td>0.2000</td>
<td>0.0290</td>
<td>0.0280</td>
<td>0.0300</td>
</tr>
<tr>
<td>reverse_1000</td>
<td>0.1830</td>
<td>0.4200</td>
<td>0.3400</td>
<td>0.1190</td>
<td>0.0430</td>
<td>0.0500</td>
</tr>
<tr>
<td>nreverse_1000</td>
<td>99</td>
<td>214</td>
<td>100</td>
<td>20</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>kkqueens</td>
<td>112</td>
<td>151</td>
<td>150</td>
<td>52</td>
<td>19</td>
<td>30</td>
</tr>
<tr>
<td>tak</td>
<td>51</td>
<td>48</td>
<td>70</td>
<td>10</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
## Non-deterministic Benchmarks

<table>
<thead>
<tr>
<th>Benchs.</th>
<th>BEAM</th>
<th>AGENTS</th>
<th>AND.-I</th>
<th>YAP</th>
<th>SICStus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default</td>
<td>ES</td>
<td></td>
<td>4.2</td>
<td>5.0</td>
</tr>
<tr>
<td>ancestor</td>
<td>NA</td>
<td>0.0369</td>
<td>0.2470</td>
<td>0.0900</td>
<td>0.0101</td>
</tr>
<tr>
<td>houses</td>
<td>0.47</td>
<td>0.45</td>
<td>0.43</td>
<td>0.80</td>
<td>0.28</td>
</tr>
<tr>
<td>query</td>
<td>11.00</td>
<td>2.55</td>
<td>14.80</td>
<td>1.05</td>
<td>0.35</td>
</tr>
<tr>
<td>zebra</td>
<td>22.9</td>
<td>9.5</td>
<td>18.6</td>
<td>23.4</td>
<td>7.71</td>
</tr>
</tbody>
</table>
| puzzle4x4    | 619   | -     | 877    | 910    | 198     | 165     | 200
# Reduced Search Benchmarks

<table>
<thead>
<tr>
<th>Benchs.</th>
<th>BEAM</th>
<th>AGENTS</th>
<th>ANDORRA-I</th>
<th>YAP 5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_money</td>
<td>7</td>
<td>8</td>
<td>0.8</td>
<td>7,767</td>
</tr>
<tr>
<td>scanner</td>
<td>20</td>
<td>39</td>
<td>3</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>queens-9</td>
<td>2</td>
<td>8</td>
<td>0.9</td>
<td>16</td>
</tr>
<tr>
<td>queens-13</td>
<td>23</td>
<td>58</td>
<td>3.4</td>
<td>93,343</td>
</tr>
<tr>
<td>queens-15</td>
<td>456</td>
<td>1,042</td>
<td>50</td>
<td>15,287,308</td>
</tr>
<tr>
<td>queens-16</td>
<td>3,958</td>
<td>8,363</td>
<td>396</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>queens-20</td>
<td>138,799</td>
<td>302,048</td>
<td>10,680</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>check_list-10</td>
<td>0.09</td>
<td>0.09</td>
<td>209,140</td>
<td>34,910</td>
</tr>
<tr>
<td>check_list-11</td>
<td>0.10</td>
<td>0.11</td>
<td>6,165,824</td>
<td>987,243</td>
</tr>
<tr>
<td>check_list-15</td>
<td>0.17</td>
<td>0.18</td>
<td>&gt;12 hours</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>ppuzzle-A</td>
<td>8</td>
<td>5</td>
<td>134,417</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>ppuzzle-B</td>
<td>18</td>
<td>10</td>
<td>&gt;12 hours</td>
<td>&gt;12 hours</td>
</tr>
<tr>
<td>ppuzzle-C 1st</td>
<td>1.9</td>
<td>1.3</td>
<td>2,059,329</td>
<td>&gt;12 hours</td>
</tr>
</tbody>
</table>
Conclusions

- The EAM with Implicit Control can be implemented efficiently.

- The model performs well, even when just using implicit control.

- Simple programmer annotations and pruning can often lead to a better performance.

- The EAM can extend logic programming for applications where Prolog would not cope well.
Current and Future Work

- We are working in integrating the BEAM with the current version of the YAP Prolog system so that it can be made more widely available.
Current and Future Work

- We are working in integrating the BEAM with the current version of the YAP Prolog system so that it can be made more widely available.
  - Use the new just-in-time indexing scheme implemented on Yap 5.0
  - Improve how to exchange solutions between BEAM and YAP.
- Work on the PBEAM (Parallel-BEAM)
- Integrate the EAM with Tabling (IMPACT)