UNIVERSIDADE DE ÉVORA

Escola de Ciências Exactas

Departamento de Informática

An Abstract Model for Parallel Execution of Prolog

Pedro José Grilo Lopes Patinho

Informática

Dissertação

Agosto de 2016

To my grandpa José Manuel, who taught me how to read and write,
to my grandpa Manuel, who taught me that family is what matters most
and
to my mom, Cármen, who taught me how to live.

Wherever you are, thank you.

Acknowledgements

First of all, I would like to thank Salvador Abreu for his help, his permanent encouragement and his pressing for me to complete this thesis. During all my academic path, his teachings have been invaluable.

A big thanks to Irene Rodrigues for her slaps on the wrist in order to keep me on track, and for her invaluable help along my academic course.

To all my colleagues in the Informatics Department of the Universidade de Évora, who all have helped me, in some phase of my work.

Finally, I would like to thank my family for their constant support and encouragement, especially my wife, Sónia, who is always there for me, and my children, Ana and Afonso, for bringing extra joy to my life.

Contents

List of Figures

List of Tables

List of Acronyms

APGAS Asynchronous Partitioned Global Address Space

BAM Basic Andorra Model

BEAM Basic implementation of the Extended Andorra Model

CLP Constraint Logic Programming

DSM Distributed Shared Memory

EAM Extended Andorra Model

ECT Escola de Ciências e Tecnologia

GAS Global Address Space

GPGPU Geral-Purpose computing on Graphics Processing Units

HPC High Performance Computing

IIFA Instituto de Investigação e Formação Avançada

IL Intermediate Language

IR Intermediate Representation

JVM Java Virtual Machine

MIC Intel®'s Many Integrated Core architecture

PGAS Partitioned Global Address Space

SMP Symmetric Multiprocessing

SIMD Single Instruction Multiple Data

SPMD Single Program Multiple Data

STEAM Scalable, Transparent Extended Andorra Model

UE Universidade de Évora

WAM Warren Abstract Machine

Abstract

Logic programming has been used in a broad range of fields, from artifficial intelligence applications to general purpose applications, with great success. Through its declarative semantics, by making use of logical conjunctions and disjunctions, logic programming languages present two types of implicit parallelism: and-parallelism and or-parallelism.

This thesis focuses mainly in Prolog as a logic programming language, bringing out an abstract model for parallel execution of Prolog programs, leveraging the *Extended Andorra Model* (EAM) proposed by David H.D. Warren, which exploits the implicit parallelism in the programming language. A meta-compiler implementation for an intermediate language for the proposed model is also presented.

This work also presents a survey on the state of the art relating to implemented Prolog compilers, either sequential or parallel, along with a walk-through of the current parallel programming frameworks. The main used model for Prolog compiler implementation, the *Warren Abstract Machine* (WAM) is also analyzed, as well as the WAM's successor for supporting parallelism, the EAM.

Keywords: Prolog, Logic Programming, Extended Andorra Model, Parallelism

Sumário

Um Modelo Abstracto para Execução Paralela de Prolog

A programação em lógica tem sido utilizada em diversas áreas, desde aplicações de inteligência artificial até aplicações de uso genérico, com grande sucesso. Pela sua semântica declarativa, fazendo uso de conjunções e disjunções lógicas, as linguagens de programação em lógica possuem dois tipos de paralelismo implícito: ou-paralelismo e e-paralelismo.

Esta tese foca-se em particular no Prolog como linguagem de programação em lógica, apresentando um modelo abstracto para a execução paralela de programas em Prolog, partindo do *Extended Andorra Model* (EAM) proposto por David H.D. Warren, que tira partido do paralelismo implícito na linguagem. É apresentada uma implementação de um meta-compilador para uma linguagem intermédia para o modelo proposto.

É feita uma revisão sobre o estado da arte em termos de implementações sequenciais e paralelas de compiladores de Prolog, em conjunto com uma visita pelas linguagens para implementação de sistemas paralelos. É feita uma análise ao modelo principal para implementação de compiladores de Prolog, a *Warren Abstract Machine* (WAM) e da sua evolução para suportar paralelismo, a EAM.

Palavras chave: Prolog, Programação em Lógica, Extended Andorra Model, Paralelismo

1

Introduction

Logic programming languages provide a high-level method of programming, where programs are based on a set of facts and rules that model the actual problem, instead of modeling the solution of the problem (as in imperative programming languages, for instance). By using declarative semantics, logic programming languages have been being used successfully in several applications where logic is intimately involved, such as artificial intelligence, expert systems, compilers, simulators, natural language processing, automatic timetable generation and theorem proofing.

Many research efforts have been devoted to implementation of the logic programming paradigm in modern hardware, which design is more appropriate for imperative-style programming. The logic approach, by describing the problem instead of describing an

algorithm to solve it, implies the creation of *abstract machines*, which, in turn, have to be emulated in an imperative fashion.

One of the common pitfalls associated with logic programming languages has been their relative poor performance, when compared to languages using other paradigms (e.g., imperative languages). This fact has stimulated research on improving the performance of logic programs. This research has focused mainly in four distinct (but combinable) alternatives: coroutining, tabling, parallelism and constraint logic programming.

Coroutining is a means to having logic goals previously scheduled for execution as certain conditions are met. The most common of those conditions is the binding of one variable. With coroutining, one can, for instance, only allow certain values to be bound to a specific variable, thus reducing (sometimes drastically) the search space, allowing logic programs to run much faster.

We can think of constraint logic programming (CLP, for short) as a specific form of coroutining. CLP is an extension over traditional logic programming which allows the programmer to place constraints in the body of clauses. These constraints set the allowed domain for variables, and can also be used to instantiate variables with allowed values (labeling).

Tabling deals with saving intermediate solutions for a recursive goal, so that those solutions can be retrieved later, without recomputing them.

Parallelism is a feature that naturally arises from logic programming languages. Their declarative syntax, consisting of clauses with alternative rules that have bodies with a conjunctive set of goals, allows us to observe the inherent possibility of testing those rules and those goals in parallel. The exploitation of this implicit parallelism is an attractive field of research, as it means we can have the same logic programs running at greater speeds, without any added complexity on the program itself.

Some authors argue that in order to get the best outcome from parallelization in logic programming languages, the programmer have to be provided with mechanisms to explicitly declare which parts of the program can and shall be executed in parallel. However, this often comes with a cost of increased complexity in the program's code, as well as in the programmer's effort to create an effective, bug free program. With this in mind, logic programs can express parallelism in one or both of two different

ways:

- **explicitly**, by adding special constructs to the source language, in order to describe parallel computations. This means that it's up to the programmer to decide if and when the execution of the program is to be performed in parallel. It may also be necessary for the programmer to provide the mechanisms for proper synchronization between those parallel tasks;
- **implicitly**, when the program is provided in it's normal form, without any extra information from the programmer. This means that the compiler has to identify the sub-tasks that can be parallelized, as well as ensuring that parallel running tasks get their results communicated correctly to each other.

The advantages of implicit parallelism are obvious from the programmer point of view: there is nothing for him to do, in order to parallelize a program. In contrast, the compiler has an increased work, by having to correctly split the work in tasks, generate the parallel execution model for the program and ensure that the results match those one would get with sequential execution.

There have been many research efforts over implementation of parallel logic programming systems, some with explicit parallelism (AKL [JH91], OZ [Smo95], GHC [Ued86], KL1 [CFS94]) and some with implicit parallelism (Aurora [LBD$^+$90], Andorra-I [CWY91a] , &-prolog [HG91], Muse [AK94]).

In this thesis the focus gears towards taking advantage of implicit parallelism in Prolog [CKPR73], as it is arguably the most popular logic programming language and, while not being fully declarative (by allowing procedural semantics and extra-logical predicates), the language is suitable for a vast number of problems, either logic-based, or general purpose.

Prolog relies on Horn clauses and SLD resolution [Gal85, Chapter 9]. The search algorithm is based on a left-to-right selection function and depth-first search. Prolog programs present two main forms of implicit parallelism:

- **or-parallelism**: when a rule has two or more alternatives, they can be evaluated in parallel, instead of sequentially. Thinking in terms of logic programming,

executing a goal is the same as finding proof(s). The different ways (alternative clauses) of finding that proof can be run in parallel. Of course, if we intend to preserve the same evaluation order (which we normally do), some cautionary measures have to be applied.

- **and-parallelism**: most Prolog queries consist in a set of conjunctive goals that must be satisfied. These goals can be evaluated in parallel, speeding up the resolution process. And-parallelism can be further divided in two different types:
  - ***dependent* and-parallelism**: when different goals in a query share variables (thus depend on each other's bindings). This form of and-parallelism can be used in a form that each dependent goal constrains the others' variables;
  - ***independent* and-parallelism**: when the goals don't share variables, i.e., the execution of one goal doesn't affect the execution of the other one.

The majority of Prolog implementations target the WAM (Warren Abstract Machine) [War83], designed by David H.D. Warren in 1983. The WAM consists in both a memory architecture and an instruction set for implementation of Prolog interpreters and compilers. Being a sequential abstract machine, several efforts have been made to parallelize the execution of WAM code, aiming to take advantage of the inherent parallelism of the Prolog language.

The Basic Andorra Model[1] (BAM) [War88] was proposed by David H.D. Warren to enable the exploitation of dependent and-parallelism between deterministic goals and or-parallelism between alternatives of non-deterministic goals. Through coroutining, dependent conjunctive goals can constrain the search space of each other, achieving and-parallelism. The Andorra-I prototype [CWY91a] showed the applicability of the BAM over a large set of problems, but it also showed the BAM's limitations, namely the need to find deterministic rules to achieve improvements over the sequential execution of Prolog programs. Those limitations, inherited from the BAM, led Warren to develop the Extended Andorra Model (EAM) [War89].

The EAM extends the BAM by allowing the parallel execution of non-deterministic conjunctive goals as long as they don't need to bind external variables, or, if they do,

[1] Or simply Andorra Model.

performing a *split* on that goal's computation process. The EAM proposal is based on a set of rewriting rules for And/Or trees, the *split* being one of those rules.

While addressing the former BAM limitations, the EAM further reduces the search space, by allowing and-parallelism in non-deterministic goals. Research based on the EAM led to two implementations: a proof-of-concept interpreter by Gopal Gupta [GW91]; and a sequential interpreter by Ricardo Lopes, the BEAM [Lop01], which proposes some optimizations over the original EAM proposal, having achieved good performance results.

1.1 Motivation

Although much research has been made over the parallel execution of logic programs, there is still work to do in what comes to exploiting simultaneously and- and or-parallelism with implicit control. In fact, there are already good implementations of and- and or-parallel Prolog systems, most of them built over the WAM structures, but it has been proven difficult to integrate both and- and or-parallelism in a single system. We believe that the EAM establishes a good starting point for achieving good performances, either at execution speed, or at memory consumption. Based on the performance results for the sequential implementation of the BEAM, we believe that there's room for improvement, in terms of the base EAM model.

The recent developments over parallel programming frameworks give us optimal tools for developing a model for parallel execution, by taking advantage of the transparency those frameworks provide. We believe that the EAM provides a stable base for the implementation of parallel Prolog engines running in the new distributed hardware paradigms, either multi-core, multi-computer or hybrid systems.

Recently, the bulk of research in parallel logic programming have been leaning towards explicit parallelism, by adding special constructs to the language, whereas our goal is to exploit the parallelism that is implicit into the language, by not burdening the programmer with new operators, constructs or programming methodologies.

1.2 Contributions

The main contribution of this thesis is STEAM, a model for transparent parallel execution of Prolog programs, taking advantage of the implicit parallelism that Prolog provides, using the EAM base model as a starting point. Specifically, a design for running STEAM over a Partitioned Global Address Space (PGAS) model is presented.

A STEAM Intermediate Language (IL) is also proposed, in order to allow the compilation of Prolog code to machine code, using the EAM model. A reference compiler that translates WAM-IL to STEAM-IL, `wam2steam`, is also presented in this work.

1.3 Thesis outline

In chapter 2, we present a brief survey over the state of the art of Prolog implementations, covering both sequential and parallel implementations. Also some other relevant logic programming languages are covered, as well as parallel programming frameworks that facilitate the implementation of parallel abstract machines.

Chapter 3 explores the common abstract models for Prolog implementation, namely the WAM, still the de facto standard in Prolog implementation, and the EAM, arguably the natural evolution of the WAM to support parallelism.

In chapter 4, a design for a WAM-to-EAM translator is presented, where a Prolog program passes through a number of intermediate steps until it becomes an EAM-based executable.

Chapter 5 describes the STEAM model in detail, and a proposal for implementing STEAM in a PGAS programming model is discussed in chapter 6.

The conclusions and future work are discussed in chapter 7.

2

Prolog and Parallel Logic Programming

This chapter presents analysis on the current state of the art in terms of Prolog and related logical programming languages, as well as the most used frameworks for parallel application development. Any of these frameworks can be used as a backend to an implementation of a Prolog compiler, with parallelism in mind.

As we look into the state of the art in Prolog and logic programming in general, it is useful to get acknowledged with the history of Prolog.

2.1 Historical background

Prolog was invented in the early 1970s by Alain Colmerauer and Robert Kowalski, at the University of Marseille, following the failing attempts to develop a computational system based on deduction [CR96]. The first Prolog system was an interpreter written in Algol-W, followed by a much refined interpreter written in Fortran, both developed under supervision of Colmerauer. This second system was very similar to modern Prolog systems in operational semantics, and its reasonable performance has contributed to the idea that logic programming was indeed viable.

Based on this idea, David H.D. Warren developed the first Prolog compiler, DEC-10 Prolog [War78], circa 1977. This compiler's syntax and semantics became the de facto standard, the "Edinburgh standard", and defined the main principle in compiling Prolog, still valid today, which is to increase the efficiency of each occurrence of unification and backtracking operations, the core of Prolog engines, and the most expensive operations.

Further explorations led David H.D. Warren to develop the WAM (Warren Abstract Machine) in 1983 [War83], which defines a high-level instruction set into which Prolog source code can be mapped (almost) directly.

Efficient Prolog implementations use one of two main approaches when it comes to running the Prolog program: emulated or native code. Emulated code is Prolog code compiled to an abstract machine (bytecode), being interpreted at run time. Native code is Prolog code compiled to the target machine and is executed directly (natively), although most compilers use intermediate representations based on the WAM (or other abstract machine). It is usual for a native code Prolog compiler to also support an emulated mode, to allow either consulting user predicates (via `consult/1` or derivatives), or dynamic predicates (created with `assert*/1`).

Most modern Prolog compilers use the sequential engine of the WAM or a variation of it. The WAM is further described in this thesis in 3.2.

2.2 Sequential Prolog implementations

The WAM specifies a model to sequentially run interpreted code, stored in a "code area" in memory. However, there are implementations that successfully translate WAM code into native code, being full-blown Prolog compilers.

Previous research on improvements over the WAM have led to two different outcomes: improved or extended WAM designs [gprolog, yap, swi-prolog, sicstus, xsb, binprolog] and to different abstract machines (e.g., the Berkeley Abstract Machine [VR84] in Aquarius Prolog [vR92], the TOAM-Jr [Zho07] from B-Prolog [Zho12] , or the Vienna Abstract Machine [Kra96]]. These sequential implementations still fill the largest share of Prolog compilers used in real-world applications.

Relatively to WAM-based implementations, existing research led to different paths, which contributed to various incompatible Prolog implementations. This incompatibility doesn't reflect so much on the language itself (most implementations are ISO-Prolog compatible), but are more visible when it comes to module support or foreign language interfaces, which means that Prolog programs made for one implementation may not run in another one. Some efforts have been made to achieve convergence between implementations [SD08], the most relevant being various compatibility enhancements between YAP Prolog [CRD12] and SWI-Prolog [WSTL12].

SICStus Prolog [CM12] is a reference commercial Prolog implementation, supporting both emulated and native code compilation. SICStus' engine, while based on the WAM, has been subject to numerous optimizations, mostly related to instruction merging and specialization [NCS01, Nä01], as well as an efficient coroutining implementation. The engine has served as the base for various Prolog systems, including parallel systems (e.g., Aurora [LBD$^+$90], Muse [AK90], both referred in section 2.3.3) and constraint logic programming systems (for instance, the QD-Janus system [Deb94] is an efficient sequential implementation of Janus, a flat committed-choice language, on top of SICStus Prolog).

Tabling [War95] is another optimization over Prolog systems that has been actively researched. The first well-known implementation came from David S. Warren and Terrance Swift, on the XSB Prolog system [SW12]. Other current Prolog systems already support tabling [RSC05, Zho12].

Recently, research efforts have gathered up around constraint logic programming (CLP) and extensions for Prolog to support CLP. GNU Prolog [DC00b] has incorporated CLP as a simple extension to the WAM [DC93]. Other systems have integrated CLP by leveraging their coroutining implementations or by interfacing with external solvers.

2.3 From sequential to parallel logic programming

A Prolog program is a set of Horn clauses. A Horn clause is a term of the form:

$$H \leftarrow G_1 \wedge \ \ldots \wedge \ G_n$$

Where H is the *head* of the clause and $G_1, \ldots, G_n$ are the goals that form the body of the clause, which is a logical conjunction of the goals. The head can consist of a single goal or can be empty. If it's empty, the clause is called a *query*. The above query is represented in Prolog as:

$$H :- G_1, \ldots, G_n.$$

Execution of a query involves using a *resolution* rule, by which the query is matched with the head of a clause. Given a clause and a query:

$$H :- P, Q.$$

$$:- H', R.$$

The resolution rule will unify H with H', resulting in a new clause: $:- P, Q, R.$

Prolog extends pure Horn clause logic by incorporating features that are not purely logic, but are essential to enable a Prolog program to answer real world problems. We can sum up these features in three categories:

- Meta-logical predicates: these predicates can't be modeled in first-order logic, but provide the programmer with extra flexibility, by allowing to inquire about the state of the computation (`var/1, nonvar/1, ground/1, …`).

- Side-effect predicates: these predicates are used to perform I/O or to alter the program being executed, by adding (`assert/1`) or removing (`retract/1`) clauses from the program database.
- Control operators: these operators allow the programmer to dynamically reduce the execution tree. The most popular control operator is *cut* (`!`), which tells the program to discard all alternative clauses to the clause being executed (by discarding backtracking data).

Prolog uses SLD-resolution [Gal85, Chapter 9] with a simple left-to-right, depth-first selection function, which, by other words, means that the order of the clauses in a Prolog program influentiates the results of evaluating a query.

This selection function has been the main source of criticism about the Prolog language, as some point out that it is too restrictive. From another point of view, it may be hard to parallelize a Prolog program, while keeping the selection function's order of evaluation.

A great part of research on parallel logic programming implementation has historically been driven away from Prolog, into more specific languages which favor some kind of explicit parallelism, mainly by adding special constructs that allow the programmer to specify where the parallel evaluation should occur.

2.3.1 Committed-choice languages

Also named concurrent logic programming languages [Sha89], the committed-choice languages allow the reading of a program as a network of concurrent processes, with shared logical variables as the interconnection points. The main difference between committed-choice languages and Prolog is that clauses are *guarded*. Parlog [CG83] was one of the first of these languages.

Arguing that Horn clauses weren't sufficient for describing parallel programs, Ueda proposed Guarded Horn Clauses (GHC) [Ued86], later simplified to KL1 [UC90].

A guarded Horn clause is a term of the form:

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n. \ \ (m \geq\ 0, n \geq\ 0)$$

Where $G_1, \ldots, G_m$ define the guard and $B_1, \ldots, B_n$ define the body of the clause. The evaluation of a goal involves matching that goal with the head and then the goals on the guard. If both succeed, the system *commits* with this clause (discarding all the others, even if they would also succeed), and then evaluating the body.

Although this kind of languages is well suited for controlled parallelism, they lack the natural simplicity of Prolog when it comes to describe search problems.

2.3.2 The Andorra Principle and the Basic Andorra Model

Following the need to support (implicit) parallelism in Prolog, while keeping some of the features of committed-choice languages, David H.D. Warren proposed the Basic Andorra Model (BAM) [War88], following the directives of his **Andorra Principle**:

- Determinate goals can be executed in And-parallel;
- When there is no selectable determinate goal, non-determinate goals' alternatives can be explored in Or-parallel.

Research over the BAM, mainly pursued at Bristol, led to the implementation of the Andorra-I prototype [CWY91a], which uses the Andorra Principle to execute Prolog programs exploiting Or- and And-parallelism. The system uses a *preprocessor* to detect determinate goals through abstract analysis. The preprocessor is also responsible for guaranteeing the correct order of execution for goals with side-effects, cuts and other order-sensitive Prolog builtins.

Other implementations based on the BAM include Pandora [BG89], which extends Parlog [CG83] to support non-determinism, and NUA-Prolog [PN91], a parallel version of NU-Prolog [RSB+89] that includes sophisticated determinacy checking.

Besides work on simultaneous And- and Or-parallelism in Prolog, there are intermediate approaches, exploiting only one kind of parallelism, either And- or Or-parallelism.

2.3.3 Or-parallel Prolog Implementations

Preceding the work on the BAM, Warren's research on parallel Prolog execution led to the Aurora Or-parallel system [LBD+90]. Based on the SICStus Prolog engine, Aurora used different schedulers, until it settled with the Bristol Scheduler [BRSW91], which uses shared memory between workers and a bottom-most dispatching strategy. This scheduler was later adopted in Andorra-I, previously referred in section 2.3.2.

Aurora uses Warren's SRI-model [War87b], where *binding arrays* are used to store conditional variable bindings across different or-branches. The Bristol scheduler uses a demand driven approach to scheduling, where each worker asks the scheduler for more work. The scheduler then tries to find work in the search-tree and, if it finds some, give it to the worker.

Another approach was followed at SICS, with Muse [AK90]. Instead of sharing the memory among workers, Muse uses a stack-copying approach, where a processor P_1 shares work with (i.e., steals work from) another processor P_2 by copying P_2's entire stack and then backtracks up to the choice-point it's going to work on, cleaning (undoing) the conditional bindings that P_1 had already made. This approach guarantees that parallel execution is completely independent, apart of a mechanism to share the choice-points. To allow the stack copying to be somewhat transparent, the system ensures that all processors are working on identical logical memory address spaces. This way, when a stack is copied to another processor, there is no need to relocate any pointer. This approach has proved to be efficient and relatively easy to implement over an existing sequential system (SICStus). Muse is currently integrated into SICStus Prolog, as an optional way of running Prolog programs, giving the programmer the ability to execute his program(s) in or-parallel or in the standard sequential method.

Another or-parallel system, YapOr [RSC99], was developed over YAP Prolog and extends the WAM with new pseudo-instructions and an extended memory organization in order to have a full, independent WAM in each worker's private memory space and to allow incremental stack copying between workers. YapOr has shown good performance when compared to Muse and has been further extended to support tabling [RSSC00].

An interesting system is Multilog [Smi96], a logic programming system that uses multi-SLD resolution [SH94], a data-centered or-parallel implementation strategy. In Mul-

tilog, a single Prolog WAM-based engine solves annotated goals in parallel, saving each alternative binding in a *multi-variable*. A multi-variable is like a standard Prolog variable, but instead of a single binding it contains a disjunction of all the possible bindings for that variable, in order to constrain the solutions given by the following goals in a Prolog query. In contrast with control or-parallelism used by Muse or Aurora (multiple Prolog engines solving disjunctive alternatives in parallel), Multilog uses a single Prolog engine aided by a (massively-) parallel unification engine, used to perform unification between multi-variables and other variables in a goal. In benchmarks over generate-and-test programs, Multilog has reported impressive speedups [Smi94] over traditional Prolog engines. One of the drawbacks has to do with the semantic (multi-variables) and syntactic (user annotations) differences from traditional Prolog.

2.3.4 And-parallel Prolog Implementations

Although Or-parallelism has shown generally good speedups relatively to sequential execution, there are still highly-deterministic programs that can't benefit from Or-parallelism.

Hermenegildo proposed the RAP-WAM [Her86b], an extended WAM which can deal with parallel execution of conjunctive goals by using Restricted And-Parallelism (RAP), a technique that combines compile-time analysis of the clauses with simple run-time checks on the variables. A new memory area, the Goal Stack, stores the goals that are ready for parallel execution.

Using the RAP-WAM, Hermenegildo implemented &-Prolog [HG91], an and-parallel Prolog system that exploits independent And-parallelism by doing compile-time analysis on the program to decide which goals to run in parallel. Later efforts by Hermenegildo and Gupta used the &-Prolog And-parallelism approach joined with a Muse-inspired refined stack-copying scheme in ACE [GH91].

Further research from Hermenegildo evolved into the Ciao multiparadigm programming system [HBC+12]. Although the first stages of the Ciao system leaned towards the Prolog language, it was further developed to support functional, higher-order logic and object-oriented programming styles, as well as constraint programming.

Another approach was suggested in Reform Prolog [Mil93], by supporting dependent And-parallelism across recursive invocations of a procedure [BLM93]. Developed from scratch, the compiler changes the control-flow of a recursive procedure, in order to take advantage of parallel evaluation.

Working at Bristol, Shen proposed modifications to the WAM to exploit Dynamic Dependent And-parallelism (DDA) in his DASWAM prototype [She96]. While having initial good performance results, there isn't further available documentation on this system.

2.3.5 Extending the Andorra Model

In what concerns exploitation of both And- and Or-parallelism, the Andorra Model[1] [War88] gathered a meritorious share of research effort. The main advantage of the Andorra Model was allowing determinate goals to execute in (and-) parallel and before the other goals in the same clause (i.e., instead of the Prolog's left-to-right selection rule). The Andorra Model specifies that a goal is determinate if only one of it's clauses matches a goal, being otherwise non-determinate. This reordering gives a direct advantage over traditional, WAM-based Prolog implementations, as determinate goals are executed only once, instead of being recomputed when backtracking reaches them. Another advantage is that the search space of the other goals is also constrained by this determinate goals, which may lead to:

1. Early failing, avoiding the search for alternatives which would fail later;
2. Reducing the number of alternatives for other goals, allowing, in some cases, non-determinate goals to become determinate.

Although various implementations of the Andorra Model exist (briefly discussed in section 2.3.2), Andorra-I was arguably the system at which most research efforts were targeted, making it the reference BAM implementation for many researchers. The Andorra-I system is subdivided in three components: the *preprocessor*, which performs analysis over the Prolog program in order to detect determinacy and guarantee the

[1]The Andorra Model is frequently referred as the Basic Andorra Model, to contrast with the Extended Andorra Model.

correct order of solutions for the program; the *engine*, which consists of teams of (parallel) workers that execute parts of the program; the *scheduler*, which is responsible to find tasks and give them to idle workers. It is further subdivided into the *and-scheduler*, the *or-scheduler* and the *reconfigurer*, which, respectively, distribute and-work among workers in the same team, distribute or-work between teams and rearrange the workers in teams according to the available work.

The preprocessor is crucial to allow the correct execution of Prolog programs, being responsible for ensuring that solutions are presented in the same order as a sequential Prolog engine, as well as generating the runtime determinacy code which will allow the engine to detect when a clause is determinate.

Andorra-I relies on *binding arrays* [War87b] to achieve or-parallelism, using a scheme very similar to the one used by the or-parallel Aurora system (see section 2.3.3). This similarity has allowed schedulers developed for Aurora to be easily adapted to work in Andorra-I.

The Andorra-I system evolved through different versions, being tested with different schedulers. The initial versions of Andorra-I [CWY91a] used a fixed configuration of workers. Dutra designed the Andorra-I *reconfigurer* [dCD95], which could dynamically adapt the configuration of workers to the available forms of parallelism and obtained very good results.

Although the Andorra Model allows the parallel execution of standard Prolog programs, David H.D. Warren continued developing the model in order to allow further parallelism and address some of the inherent limitations of the BAM, namely the need for the determinacy check, which can be costly to the system in runtime. Further, not all determinate goals can be safely executed before the other goals (imagine a query `?- p(X), write(X), fail.` If the determinate goal `fail` is executed first, the program will not try to find a solution, nor it will display it. Also, there are cases when the determinate goal comprises a big load of work that would not be done if the non-determinate goals that precede the determinate goal would be executed first (imagine a query `?- …, parent(X,X), …, determinate_but_heavy(…).` If `parent/2` is not determinate but there are no solutions for `parent(X,X)`, the work done by `determinate_but_heavy` will have been wasted).

Andorra-I also suffers from this problem, as it is unable to bind external variables to detect determinacy. By only allowing determinate goals to execute in and-parallel, one of the bigger limitations of the Andorra-I system is that the performance depends on finding determinate work in the program being executed, but several programs don't have determinate work, or determinism can't be found by the preprocessor.

This limitation led David H.D. Warren to develop a model in which dependent and-parallelism between non-determinate goals could be exploited. To allow non-determinate goals to also run in parallel, David H.D. Warren proposed the Extended Andorra Model (EAM) [War89], a model based in representing logical computations in the form of And-Or trees and providing logically correct rewrite rules over nodes of such trees. The rewrite rules reflect the properties between logical conjunctions and disjunctions. Briefly, the EAM rewrite rules are:

- *local forking* → unfolds an atomic goal into an *or-box* containing one *and-box* for each of the alternatives in its definition.
- *determinate promotion* → if an *or-box* contains only one child alternative (*and-box*), this child and it's bindings can be safely *promoted* to the grandparent *and-box*.
- *non-determinate promotion* → one child of an *or-box* is chosen to be split from it's sibling, by splitting the grandparent *and-box* into two copies of itself under an *or-box*: one containing the split child, the other containing the *or-box* with the remaining children.
- *explicit pruning* → *cut* and *commit* operators allow removal of boxes that fall on the scope of the operator.
- *implicit pruning* → when an alternative under an *or-box* succeeds, the other alternatives can be pruned from the And-Or tree.

As a model, the EAM presents two immediate advantages over traditional Prolog evaluation, that contribute to reducing the search space:

1. By following the Andorra Principle, determinate goals are executed as soon as possible;

2. By using a tree to represent computation, the scope of non-deterministic computations is reduced.

The EAM extends the BAM parallelism, by allowing non-determinate goals to execute in parallel as long as they don't bind external variables. As the EAM presents a base for this thesis, it will be further dissected in section 3.3.

While the EAM addresses some of the limitations concerning the BAM, regarding execution of Prolog programs, researchers at SICS argued that the EAM needed a programming paradigm that could merge traditional Prolog with committed-choice languages, resulting in the development of the Andorra Kernel Language (AKL) [JH91].

AKL, later renamed to Agents Kernel Language [Jan94b] is a programming language that tries to merge Prolog and concurrent logic programming languages (committed-choice languages), based on Kernel Andorra Prolog [HJ90]. AKL leverages the Extended Andorra Model, but keeps the control explicit, in contrast with [War90], where the focus gears towards implicit control.

AKL programs are sets of guarded Horn clauses, with three distinct guard operators: *conjunction*, *cut* and *commit*. The use of guards extends the determinacy test of the Andorra Model, as a goal is determinate when a single guard check succeeds between all the alternative clauses for that goal, avoiding the limitation of relying only in head unification. The guard can, thus, be viewed as a helper test to allow the system to chose the correct goal to execute. A restriction is that AKL only allows quiet pruning, i.e., a pruning guard operator (cut or commit) can only be used if the head and the guard don't make any external bindings. Also, all clauses belonging to the same predicate must use the same guard operator.

AKL makes use of the same concepts as the EAM, by defining a computation as a set of rewrite rules over an And/Or tree. Nodes in the tree can be one of three types: *and-boxes*, *or-boxes* or *choice-boxes*. For contextualization purposes, we briefly describe the concepts behind the rewrite rules of AKL:

1. *local forking* $\rightarrow$ unfolds an atomic goal into a *choice-box* containing all the alternatives in its definition.

2. *failure propagation* $\rightarrow$ an *and-box* containing a failed goal (or box) is rewritten

as *fail*.

3. *choice elimination* → a failed goal (or box) inside a *choice-box* is eliminated.

4. *environment synchronization* → converts an *and-box* containing one goal into a failed box, if the goal's constraints are incompatible with the *and-box*'s bindings. Represents unification failure.

5. *non-determinate promotion* → selects a guarded goal (with a solved guard) from a non-determinate *choice-box* and splits the parent box in two copies of the same configuration, one with the solved goal and the other with the remaining alternatives. This creates a determinate branch in the parent box.

6. *determinate promotion* → is similar to the previous rule, but is applied when the *choice-box* contains only one goal.

7. *choice splitting* → is similar to the non-determinate promotion, but focuses on splitting a *choice-box* inside an *and-box* into a *choice-box* of two *and-boxes*: one with the chosen goal and the other with the remaining choices.

8. *pruning rules* → handle the effect of pruning (cut and commit) guard operators, by removing the remaining alternatives from the *choice-box*.

The AKL rewrite rules are described in detail in [JH91]. The rules are classified either as *guessing* (non-determinate promotion and pruning) or *guess-free* (all the other) rules. Guess-free rules are always applicable, whereas guessing rules can only be applied to *stable* goals or, in the case of pruning operators, iff the pruning operator is quiet. The *stability* of a goal is an AKL concept that derives from the Andorra Principle, which states that determinate work is performed before non-determinate work. The stability concept is somewhat expensive to compute, which led AKL implementations to use a simplified condition: an *and-box* is marked as not stable if it suspends on external variables.

Various implementations of AKL were developed:

- AGENTS [JM92], a sequential AKL implementation that uses a WAM-style stack to execute programs compiled into a WAM-style abstract machine. The implementation waived *or-boxes*, as AKL predicates must have one guard operator.

- Penny [MA95], a parallel implementation of AKL using a copy-based approach for parallel bindings.

- ParAKL [MD93], another parallel implementation which differs from Penny (which uses a sequential copying approach) by using a parallel implementation of the choice splitting rule (thus with parallel copying) and by using the PEPSys [BdKH$^+$88] hashing scheme for parallel bindings.

Although they're members of the logic programming language family, AKL and Prolog differ both in syntax and semantics. Hermenegildo proposed an automatic translator from Prolog to AKL [BH92], but this was not enough to gather sufficient interest into AKL.

While AKL has shown acceptable performance both in sequential and parallel implementations, the researchers have shifted their research into Oz [Smo95], a system that merges concurrent logic programming with object oriented features.

At Bristol, David H.D. Warren and other researchers built upon the EAM, insisting in implicit control [War90], in order to achieve as much compatibility with traditional Prolog as possible.

A proof-of-concept interpreter was developed by Gupta [GW91], which made experimentation on new ideas for the EAM possible. Some ideas allowed finer control over search and improved parallelism (e.g., lazy copying, eager producing of bindings in a producer-consumer environment, etc.).

Building on top of the EAM and AKL, Abreu proposed the OAR model and language [Abr00]. This model uses rewrite rules for And-Or trees to achieve parallelism in mixed shared- and distributed shared-memory environments, relying on a language for contextual logic programming (CxLP). The OAR model presents a set of rewrite rules that are classified either as determinate or non-determinate, as well as whether they perform forward (expanding the code that makes up the clauses) or backward execution (transitions that occur as consequence of factors other than code execution). Previous works by Abreu [APC92, AP93, Abr94] have also contributed to improve the AKL execution model.

Leaning towards the EAM initial goals, further research at University of Porto by Lopes

led to the implementation of the "Basic design for Extended Andorra Model" (BEAM) [Lop01], an EAM sequential interpreter implemented on top of YAP Prolog [CRD12]. Lopes also proposed a parallel model of execution for the BEAM, called the RAINBOW [LSCA00]. The BEAM proposes various optimizations over the EAM model, namely simplification rewrite rules derived from the AKL rewrite rules, as well as using concepts from Gupta's EAM interpreter, as eager forking and lazy copying, allowing goals that produce bindings for an external variable to execute, instead of suspending. In the original EAM specification, all goals that constrain external variables must suspend. The results presented by the sequential implementation of the BEAM led us to trow that the EAM is indeed a feasible model for parallel Prolog execution.

Recent breakthroughs in hardware for parallelism (e.g., General Purpose computing on Graphics Processing Units – GPGPU, Many Integrated Core – MIC, etc.) have made possible to employ different paradigms to implement parallel engines for logic programming.

In the next section we analyze parallel programming models and frameworks which can be used to implement a parallel Prolog engine, based on the EAM.

2.4 Parallel programming models

As single-processor performance becomes nearer to its theoretical maximum, hardware implementers have been focusing into producing multi-processor alternatives, which can be effectively harnessed by software that takes advantage of the parallelism in the hardware. This almost always means that the programmer has to write his code specifically to run on parallel hardware, or else the software would be limited to run in a single processor of a multi-processor machine.

As stated earlier, logic programming languages exhibit implicit parallelism, and it's up to the execution engine to exploit the presence of hardware parallelism, which allows older programs, written without parallel hardware concerns, to run efficiently in parallel hardware.

There are many different types of parallel machines, but we can generally divide them in three main categories:

- **Symmetric multiprocessing (SMP) machines** - This is the most common type of machine in use today, where multiple identical processors connect to a single, shared main memory and share access to all I/O devices. A single operating system is used to control all processors equally. In today's multi-core systems, each core is treated like a separate processor. In SMP systems, each processor can run different programs and work on different data at the same time, while sharing access to the memory and I/O system. The most common way of harnessing SMP parallelism is by *multi-threading*, dividing a program in multiple *threads* that can be run in parallel.

- **Distributed systems** - This category comprises clusters of machines (possibly and probably SMP machines) connected through a bus (normally network-based). In this systems, each machine has it's private memory and I/O resources and is controlled by an operating system. The sharing of data is usually performed via *message passing*. One of the most common frameworks to support software running on DSM systems is MPI [Pac97]. There has been some effort in making this distributed memory sharing transparent to the programmer, by the means of programming frameworks. We'll visit some of these frameworks in section 2.4.

- **Micro-core systems** - These systems are based in specialized chips that include several processor cores, usually performing the same function over large sets of data, with each core processing a different part of the data. The most well-known micro-core systems are GPGPU (General Purpose computing on Graphics Processing Units), which make use of the several cores of a traditional GPU to process large data volumes; and MIC (Many Integrated Core Architecture), from Intel, which takes the GPGPU paradigm, incorporating several micro-cpus in a single expansion board, or co-processor. Also, various frameworks exist (e.g., OpenCL, CUDA) in order to harness the parallelism in micro-core systems.

With the rapid growth of hardware power, especially the trend of *High Performance Computing* (HPC), there's an accompanying need for programming models that can provide implementers a level of abstraction on top of different parallel hardware architectures. To allow development for parallel hardware, there are three main programming models that provide a logical interface between the application and the architecture:

- **Message passing** - in this model, different processes or threads run with their private address space, possibly in different machines. The synchronization and sharing of data between processes/threads occurs explicitly, via a message passing interface.

- **Shared Memory** or **Global Address Space (GAS)** - this model provides a virtual address space that transparently maps to the private address spaces of the components of the parallel system. Generally, there's no distinction between local and remote memory addresses, making possible to use remote memory transparently.

- **Partitioned Global Address Space (PGAS)** - this is a specialized class of shared memory systems, where the whole memory is shared, but also *partitioned*, meaning that there's a distinction between local and remote addresses. This allows the programmer to exploit locality and avoid unnecessary overheads in remote communication.

These models have been materialized in programming frameworks, ranging from the lower-level to the higher-level languages.

The message passing model has been around for a long time, mostly materialized by the *MPI* (Message Passing Interface) standard [Hem94], widely used in multi-computer systems. As it is a mature standard, many implementations are available, for languages like C, C++, Java, Python, etc. It is commonly used along with other high- and low-level frameworks in order to extend their multiprocessing capabilities with distributed computing capabilities.

Some arguing about MPI's restrictive message semantics led to the development of alternative message passing interfaces, namely GASNet [BJ], a communication interface that aims to be high-performance and network-independent, in order to allow transparent access to both local and remote (distributed) memory addresses. GASNet has been used in the implementation of Unified Parallel C (UPC), an extension to the C language to provide a PGAS programming model.

For shared memory approaches, OpenMP [DE98] is an standardization API for parallel computing on shared memory systems. The standard is managed by a consortium of

manufacturers that includes AMD, Intel, HP, IBM, Nvidia and others. OpenMP is not really a framework, but rather a specification for compiler directives and library routines that C, C++ and Fortran compilers should implement in order to support transparent parallelism. It has been widely used either on academic or commercial parallel software.

In order to take advantage of locality, the PGAS model has lately gathered a great deal of research interest. *X10* [CGS+05, SBP+11] is a language with a Java-like syntax augmented with primitives to generate parallel computing tasks. Developed at IBM, it has been used to support several large scale projects, including city traffic simulation, graph processing, etc. X10 uses a *Asynchronous Partitioned Global Address Space* (APGAS) programming model, which means it can create asynchronous local and remote tasks and manage synchronization between them. X10 also offers GPGPU (General-Purpose computing on Graphics Processing Units) backends.

Built on past work in X10 v1.5[2], *Habanero-Java* [CZSS11] is a framework that provides a language, compiler and runtime environment for extreme scale systems. Heavily linked with Java, it works on top of the JVM. It was developed with educational purposes, but has been used in real-world scenarios. The Habanero developers at Rice University have also developed other alternative frameworks, as *Habanero-C++* (still in development), which uses C++ and doesn't depend on the JVM.

Another novel framework is Chapel [Cha13], developed at Cray Research. It uses a PGAS model and a language in the family of C++ and Java. Rather than extending an existing language, Chapel provides a new language designed from first principles and allows the integration with previously existing code. As X10, Chapel also provides an asynchronous layer over the PGAS model.

On the low-level side, Cilk [BJK+95], developed at the Massachusetts Institute of Technology (MIT), presents a C-like language and has show very good performance results, in comparison with other frameworks [NWSDSM13]. Cilk is a mature framework (development started at the 1990s), having been successfully used in a wide area of large scale systems as protein folding, graphic rendering, etc.

UPC (Unified Parallel C) [CDC+99a] is an extension for the C language to enable par-

[2]At the time of this writing, the current X10 version is 2.5.3

allelism over a PGAS model. Developed at Berkeley, UPC focuses on a SPMD (Single Program Multiple Data) model, which means that a single algorithm can perform the same operation on a large set of data in parallel. Recent efforts have successfully extended UPC to work over the Nvidia CUDA platform [CLT+10] and over the Intel Many Integrated Core Architecture (Xeon Phi coprocessor) [LLV+13].

2.5 An Abstract Model for Parallel Execution of Prolog

This thesis focuses in STEAM, an abstract model for parallel execution of Prolog programs, focusing in maintaining either the original Prolog syntax, either the expected output semantics. While several research efforts have been made into the exploitation of implicit parallelism in Prolog, only a few focus in both forms of parallelism (and- and or-parallelism), and even fewer focus in the Extended Andorra Model.

We believe that the EAM is the natural successor for the WAM when targeting parallel hardware, and there is still a gap to be fulfilled in regard to research over the EAM. The successful results obtained by Lopes in the BEAM [Lop01] have encouraged us to build upon the EAM in order to accommodate the newer hardware developments and parallel programming frameworks.

Specifically, STEAM consists in the following key components:

- STEAM uses a tree-rewriting system, based on the EAM;
- STEAM relies on a multi-step compilation scheme, by transforming WAM code into EAM code, which can then be either interpreted or compiled to native code;
- STEAM takes advantage of current parallel programming paradigms to allow the parallel execution of the aforementioned EAM code.

2.6 Concluding remarks

This chapter brought a survey on logic programming implementations, with emphasis in the Prolog language, giving focus to parallel implementations and their origins. We can observe that there hasn't been recent significant developments regarding parallel Prolog execution, mainly because the focus has arguably moved to constraint logic programming (CLP).

Programming models and lower-level frameworks for developing parallel applications are briefly presented, in order to explore alternatives that allow us to build a prototype for the model described in this thesis.

Also, the difference between Prolog and the committed-choice (concurrent) logic programming languages is noted, showing that it's no easy task to adapt programs from one to other of these two approaches.

In the next chapter we describe the main abstract models for executing Prolog programs.

3

Abstract Machines for Prolog

When Colmerauer and Kowalski invented Prolog in the 1970s there was no abstract model to support execution of Prolog programs. They supervised the implementation of the first Prolog compilers, which directly influenced the creation of the WAM by David H.D. Warren.

Although many derivatives of the WAM exist, the WAM continues to establish the *de facto* standard in sequential Prolog implementation. In what concerns parallel implementation, there is still no standard, but this thesis focuses on the EAM as a starting point for parallel Prolog implementation.

In this chapter, a more in-depth study of the WAM is presented, as well as an extended description of the EAM.

3.1 Formal definitions

For the rest of this thesis, the Edinburgh syntax [BBP+81] will be used for terms, predicates and logical variables:

- A *term* is either a variable or a function symbol of *arity* $n \geq 0$ applied to n terms (e.g., p, $f(X)$, $g(f(X), Y, c)$).
- An *atom* is a formula of the form $p(T_1, \ldots, T_n)$, where p is a predicate of arity n and $T_1, \ldots, T_n$ are terms.
- A *definite clause* (or simply *clause*) is a formula of the form

$$H \leftarrow B_1, \ldots, B_n \ (n \geq 0)$$

  where H is an atom and is called the clause's *head*, and $B_1, \ldots, B_n$ is a sequence of atoms, called the clause's *body*.

3.2 The Warren Abstract Machine (WAM)

As stated earlier, the Warren Abstract Machine was created by David H.D. Warren in 1983. The report where the WAM was introduced [War83] was very abstract, without many technical or implementation details, so not many people were able to understand it. This led some authors to write tutorials about the WAM [GLLO85, Kog90, AkF99], contributing to a widespread growing interest in the WAM implementation and in the development of optimizations for the WAM, some of which, in turn, led to implementations somewhat different from the original Warren's machine [BAM, TOAM-Jr].

Nevertheless, the WAM is still the basis for the most part of the current Prolog implementations, as it can be efficiently implemented in modern computer architectures. Although the WAM was initially seen as a model for specialized hardware, it has been proved that there is no need for such hardware, as compilers based on the WAM have achieved comparable results to imperative lower-level languages (e.g., C).

The WAM organizes the memory as an array of *cells*. Each cell has a *tag* that specifies

the type of that cell. The basic WAM tags are shown in table 3.1.

REF	Reference	contains a variable
STR	Structure	contains a compound term
CON	Constant	contains a term with arity=0
LST	List	contains a (part of) a list
INT	Integer	contains an integer (constant) value
FLT	Float	contains a floating point value

Table 3.1: Types of tags for WAM cells

A variable is a reference pointer to an address in the WAM memory, thus containing a tag of REF and a memory address. Unbound variables are REF cells with their own address (a REF cell that points to itself). References from other cells to these cells also represent unbound variables (the last cell in the chain is obtained through *dereferencing*), although the cells are efectively bound to each other.

Terms (structures) are represented in the WAM as a consecutive set of cells: a term of the form $f(t_1, \ldots, t_n)$ will consist of $n+2$ consecutive memory cells (more precisely, the first two cells don't need to be consecutive, and in most cases, they aren't). Figure 3.1 shows an example representation of the term $f(A, h(A, B), g(B))$. The term starts at address 7, which point at address 8 where the name and arity of the term reside, and the next 3 cells point to the addresses of the argument terms.

Figure 3.1: Example representation of the term $f(A, h(A, B), g(B))$

3.2.1 Registers and memory organization

The WAM was mainly designed for emulated execution and comprises six distinct logical memory areas:

- **The heap** or **global stack** is used to store compound terms and lists as they are created by goal evaluation;
- **The local stack** (or simply **stack**) stores choice-points and environment frames. An environment frame stores values for permanent variables across conjunctive goals, while a choice-point records variable bindings that can be reset upon goal failure, to allow other alternative clause to be tested (i.e., to backtrack);
- **The trail** is another stack used to store the addresses of the variables that must be unbound upon backtracking. The *trail condition* dictates if a variable has to be *trailed*;
- **The push-down list (PDL)**. A stack used for the unification of nested compound terms. As the PDL doesn't need to be permanent, in real implementations it is common the use of the local stack for unification purposes.
- **The code area** is where the code of the program resides. In native code implementations, this area contains dynamic predicates (created via `assert/1`) and user code (generally inserted via `consult/1`);
- **The symbol table** stores information about the symbols used in a Prolog program (e.g., atom names, variable printing names, etc.).

As a register-based architecture, the WAM uses a set of internal registers, where the execution state is stored. The WAM's registers are shown in table 3.2.

A sample illustration (taken from [AkF99]) of the memory architecture for the WAM can be seen in figure 3.2 (page 32). Many variations on this scheme have been implemented in current Prolog systems.

`P`	Program counter
`CP`	Continuation pointer (top of return stack)
`E`	Environment pointer (current environment in local stack)
`B`	Most recent choice point
`B0`	Cut pointer
`A`	Stack pointer
`TR`	Trail pointer
`H`	Heap pointer
`HB`	Heap backtrack pointer
`S`	Structure pointer
`mode`	`read` or `write` mode for unification
$\texttt{A}_1$`, …, `$\texttt{A}_n$	Argument registers
$\texttt{X}_1$`, …, `$\texttt{X}_n$	Temporary variables (generally overlap with A registers)
$\texttt{Y}_1$`, …, `$\texttt{Y}_m$	Permanent variables (in the stack)

Table 3.2: Internal WAM registers

3.2.2 WAM instructions

In a WAM-based system, compiling a Prolog program consists in translating that program to equivalent WAM instructions. As already discussed, the WAM-compiled Prolog program can be seen as a program written in a *WAM language*, which, albeit not a standard language (syntactic differences occur among different compilers), stays close to an Intermediate Language (IL), to which we generally refer as *WAM code*. Following the compilation to WAM code, the resulting set of WAM instructions can be used in two different forms:

- translated into native code, targeting a specific hardware (or low-level assembly) in order to be executed directly, or
- maintained in WAM-form, going through some form of compression (byte-code), in order to be interpreted by an emulator.

Many compilers use both forms, in order to allow the emulated execution of dynamic and consulted predicates (not available at compilation time). After all, Prolog is meant to be an interactive language.

WAM instructions can be grouped in different types:

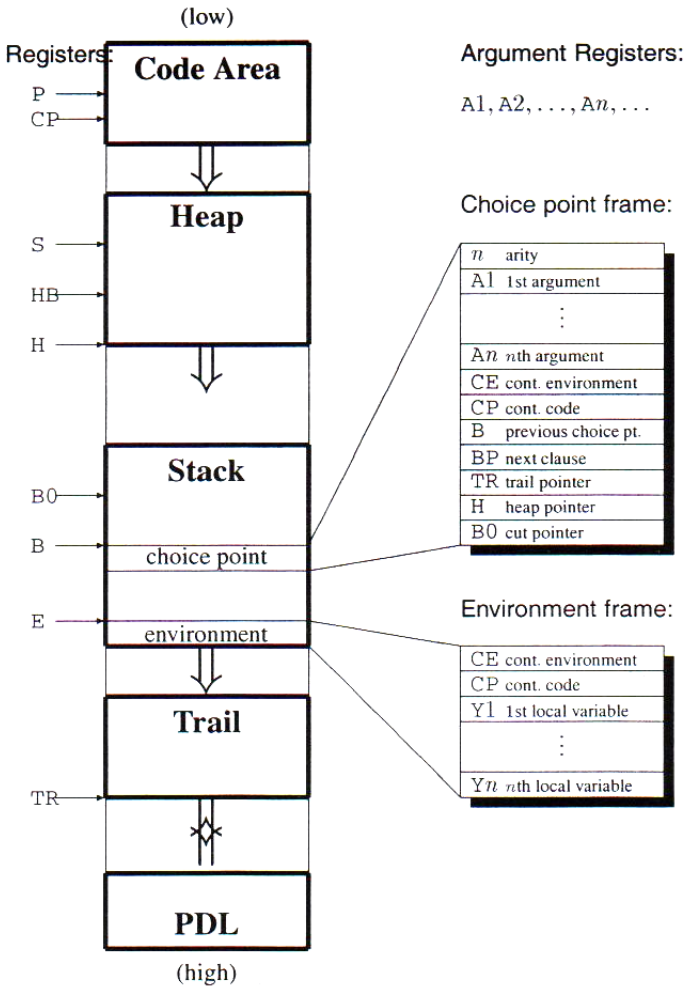

Figure 3.2: Example WAM memory layout and registers

1. **put_*** instructions: used to create elements in the heap (or in the stack, when used on permanent variables), are mainly used to load argument registers before a call;

2. **get_*** instructions: used to perform unification, are mainly used in the execution of a goal;

3. **unify_*** instructions: used to perform unification with structure arguments;

4. **procedural control** instructions: used to control the course of the program, comprise unconditional jumps and allocations in the stack (**call**, **execute**, **proceed**, **allocate**, **deallocate**);

5. **switch_*** instructions: used for indexing, by making conditional branches re-

garding the type of the (first) argument;

6. **`try`**, **`retry`** and **`trust`** instructions: used for choice-point management, setting the alternative code to jump to when the current goal fails.

7. **search space puning** instructions: used to remove choice-points from the search (**`cut`**, **`neck_cut`**).

Figure 3.3 shows an example Prolog program. We can see the WAM instructions for the same program in figure 3.4, as generated by **`pl2wam`** [DC00b], the GNU Prolog component that converts Prolog code to WAM code.

```
p(X,Y) :- q(X), q(Y).

q(1). q(2).
```

Figure 3.3: An example prolog program

```
predicate(p/2,1,static,private,monofile,global,[
    allocate(1),
    get_variable(y(0),1),
    call(q/1),
    put_value(y(0),0),
    deallocate,
    execute(q/1)]).

predicate(q/1,3,static,private,monofile,global,[
    switch_on_term(2,fail,1,fail,fail),
label(1),
    switch_on_integer([(1,3),(2,5)]),
label(2),
    try_me_else(4),
label(3),
    get_integer(1,0),
    proceed,
label(4),
    trust_me_else_fail,
label(5),
    get_integer(2,0),
    proceed]).
```

Figure 3.4: WAM code for the example Prolog program.

3.2.3 Limitations and relating optimizations

There are many issues in the initial Warren's specification for the WAM. Fortunately, most of these issues have been object of study and improvement over the years.

- The implementation of cut requires an additional register in the WAM. There are proposed optimizations on this matter [MD89]
- Unification modes require different paths of execution for different modes (and thus jumps, which are expensive). This can be optimized by using specialized unification instructions when we know if the arguments are previously instantiated or not.
- Memory management is fairly good in the WAM, as it allows fast recovery of space in the stack on backtracking. However, long computations may require more space, and then there is a need for space recovery between backtracking requests. Some algorithms for garbage collection [ACHS88, DET96a] have been proposed along the years.

3.3 The Extended Andorra Model (EAM)

The Extended Andorra Model (EAM) [War89] was presented by David H.D. Warren as a set of rewriting rules over nested conjunctions and disjunctions (i.e., And-Or trees), in order to allow parallel resolution of logic programs. It is *extended* in the part that the EAM is an extension to the Andorra Model, allowing parallelism between independent non-determinate and-goals.

The first proposal for the EAM led researchers to believe there was a need for a new logical programming language, in order to fully take advantage of the EAM design. In order to adapt the model to existing languages (e.g., Prolog), Warren then proposed the Extended Andorra Model with Implicit Control [War90], where the control part is managed by the implementation of the EAM and not by the programmer.

We focus our research in this implicit control model, and for the remaining of this thesis, all references to the EAM consider the implicit control model.

Sections 3.3.1 and 3.3.2 describe the EAM in further detail.

3.3.1 EAM base constructs

The EAM defines two base constructs, which form the nodes of the And-Or tree: the *and-box*, which represents conjunctions, and the *or-box*, representing disjunctions (alternative clauses for the same predicate).

Formally, an **and-box** (figure 3.5) corresponds to a clause $G_1, \ldots, G_n$ which creates variables $X_1, \ldots, X_m$ and impose constraints (σ) on external variables.

$$\exists X_1, \ldots, X_m : \sigma \ \wedge \ G_1 \wedge \ \ldots G_n$$

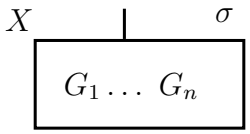

Figure 3.5: Graphical representation for an and-box

An **or-box** (figure 3.6) represents the different alternative clauses $C_1, \ldots, C_n$ for a specific goal. Each clause C_i is represented by an and-box.

$$C_1 \vee \ \ldots \ \vee \ C_n$$

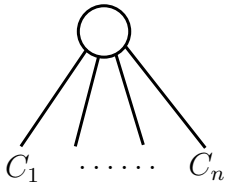

Figure 3.6: Graphical representation of an or-box

3.3.2 Rewriting rules

The EAM rewriting rules are the engine that allows the computation to advance. We apply the rules to simplify the tree and propagate results, until a solution is available.

Local forking (figure 3.7) is the substitution of a goal G in an and-box for an or-box corresponding to the unfolding of G's alternatives $C_1, \ldots, C_n$. Each alternative C_i, creating variables Y_i and generating external constraints σ_i, is placed in an and-box.

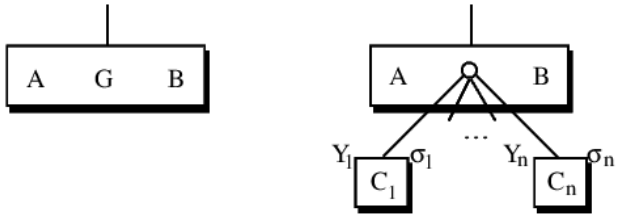

Figure 3.7: EAM local forking rule.

Determinate promotion (figure 3.8) is the substitution of an or-box with a single alternative (and-box) for that single alternative. This normally means the end of a computation, when lower level results are propagated to the upper level.

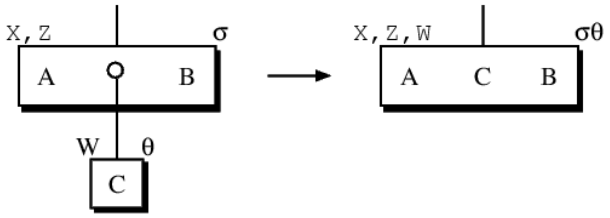

Figure 3.8: EAM determinate promotion rule.

Nondeterminate promotion (figure 3.9) is the promotion of an and-box that has alternatives (siblings) to its (grand) parent and-box, by creating an or-box with two alternatives: one is the original parent and-box with the promoted alternative; the other is the original parent and-box with the remaining alternatives (which remain unpromoted under the or-box). As this rule creates two copies of the parent and-box, it is sometimes referred as the *splitting* rule.

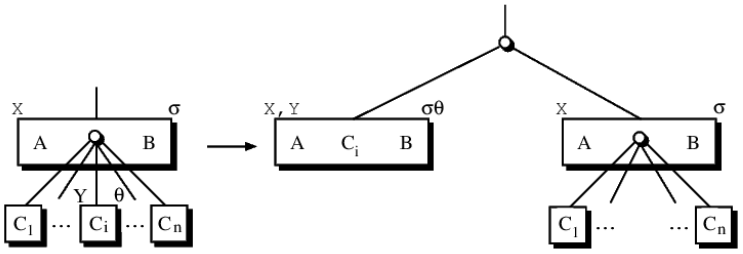

Figure 3.9: EAM nondeterminate promotion rule.

Besides this rules, Warren also provides rules for simplifying the And-Or tree when pruning operators are present. These rules facilitate reducing the search space under

special conditions, namely when the and-box containing the pruning operator doesn't contain constraints on external variables.

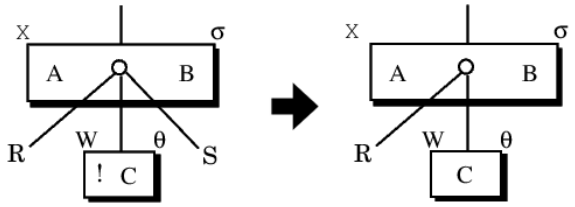

Figure 3.10: EAM handling of cut.

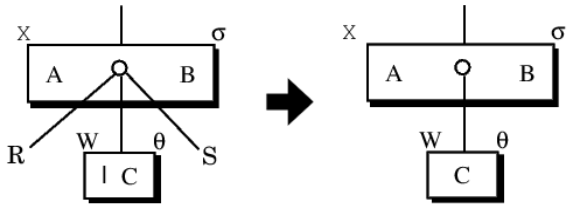

Figure 3.11: EAM handling of commit.

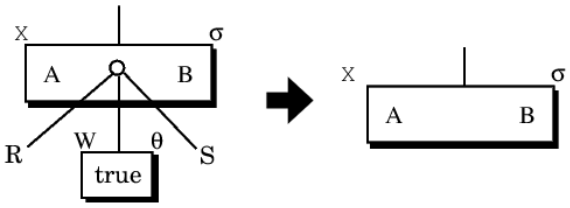

Figure 3.12: EAM handling of implicit pruning.

Although the latter two rules are more appropriate for committed-choice languages, they can be used in Prolog, when extended with annotations for when the programmer wants to explicitly control parallelism. They are, however, optional.

Through simple, logically correct rewriting rules, the EAM makes possible the parallel execution of Prolog goals. Along with these rules, Warren also proposed some guidelines to control the execution on the EAM. Warren proposed the following execution strategy for the EAM:

1. Perform *evaluation* (by applying all rules except non-determinate promotion), as long as there is no production of non-determinate bindings.

2. *Suspend* evaluation of any and-box containing a *test goal* over an external variable.

3. When execution of an and-box can't proceed, allow non-determinate promotion, but only on one selected goal, for instance, the leftmost goal (in order to mimic Prolog traditional execution behavior).

4. In non-determinate promotion, use *lazy copying* of suspended evaluations, i.e., always complete the evaluation in the higher environment, before copying to the lower environment.

In order to better understand the inner workings of the EAM, figures 3.13 to 3.15 show an example of EAM execution for a simple Prolog program. Consider the program

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

and the query

```
?- q(X,X).
```

To prepare for the resolution of the query, the EAM creates an and-box for the query. The only internal variable is X (figure 3.13, step 1). The second step is to perform a local fork, unfolding the alternatives for the predicate q (figure 3.13, step 2).

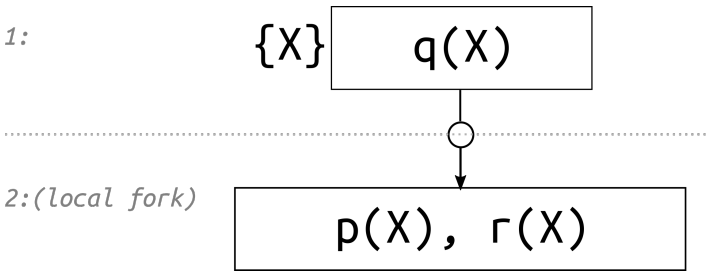

Figure 3.13: Example of EAM execution, steps 1 and 2.

As there's only one alternative clause for q (i.e., the or-box has only one child), the EAM proceeds with the determinate promotion rule (figure 3.14, step 3), followed by a local fork for each call in the and-box (figure 3.14, step 4). As the newer and-boxes all try to bind an external variable (X), the EAM suspends them.

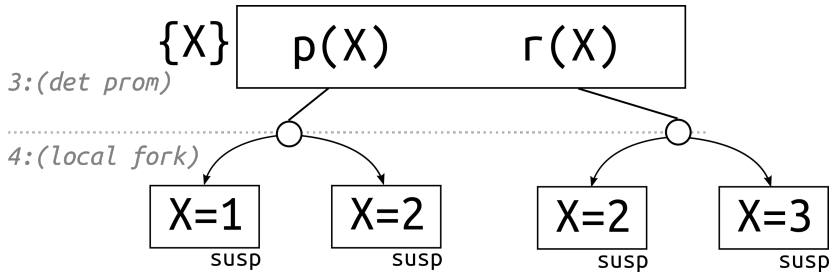

Figure 3.14: Example of EAM execution, steps 3 and 4.

As there's no other applicable rule, the next step is to perform a nondeterminate promotion. To keep the semantics of Prolog, the leftmost or-box is chosen to be split (figure 3.15). The suspended and-boxes are then awakened, and the ones that try to bind X with incompatible values fail, leaving only one possible solution.

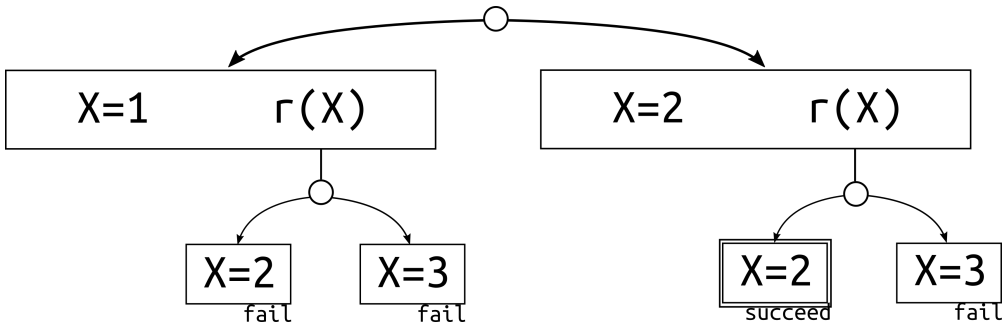

Figure 3.15: Example of EAM execution, step 4.

Although very simple, this example shows how the EAM works with pure logic Prolog programs. The EAM specification doesn't consider how to deal with meta-logical predicates and, although there are rules to deal with pruning operators, caution has to be made in order to keep Prolog semantics intact, especially when dealing with parallel execution.

3.4 An EAM-based scalable model for parallel Prolog

The main object of this thesis is stEAM, the Scalable, Transparent Extended Andorra Model, a model for parallel execution of Prolog programs which can take advantage of the PGAS programming model. stEAM aims to exploit the implicit parallelism in

the Prolog language, without relying in user annotations or semantic differences from standard Prolog.

STEAMuses a pluggable compilation scheme:

- Using a WAM → STEAM translator, we can perform abstract analysis and pre-optimize the code with parallelization in mind, as well as being able to support WAM code from different Prolog-to-WAM compilers;
- Producing STEAM Intermediate Language code, it's possible to implement different runtimes to explore alternative programming models (e.g., PGAS, GPGPU, SMP) or even to decide if the code is to be compiled or interpreted;

Although based on the EAM, STEAM diverges in some significant parts, in order to allow the exploitation of parallelism in recent programming models. When there's a conflict between maximum parallelism or minimum inference, STEAM favors maximum parallelism.

As the next chapter focuses in the translation of WAM code to STEAM code, this section is meant as a form of introductory context to the model. The STEAM model will be presented in detail in chapter 5.

3.5 Closing remarks

The WAM is still the most used model when compiling Prolog programs. By providing an almost direct-mapping of Prolog code to WAM instructions, it's both an efficient and effective model to execute Prolog programs. Over the years, several optimizations over the original model of the WAM were proposed and implemented in real-world compilers, making the WAM the de facto standard in Prolog compiler implementation. In fact, many of the existing Prolog compilers use the WAM as the basis for an intermediate language, which is frequently referred as WAM-code, used in intermediate phases of the compilation. Although there's no universal WAM-code specification, the WAM instructions used by most of these compilers can be thought as a "WAM programming language" by itself.

The EAM can be seen as an evolution of the WAM, in the way that it provides a model that enables the parallel execution of Prolog programs. With this model, it is possible to achieve **minimum inference**, by never repeating the same inference in distinct branches of the And-Or tree[1] and **maximum parallelism**, by allowing to perform parallel search on each goal, before making a global choice.

The STEAM model aims to extend the EAM in order to take advantage of novel parallel programming models, by using a pluggable compilation scheme and focusing in maximum parallelism, by making use of an Intermediate Language (IL), STEAM-IL, which well be further discussed in the next chapter.

[1] To be strictly correct, the nondeterminate promotion rule can produce repeated inferences in the And/Or-Tree

4

Designing a WAM→EAM translator

We can argue that the biggest drawback of the EAM (with implicit control) is the lack of real implementations. The BEAM [Lop01] was the first and only implementation of the EAM and, while being a sequential implementation, it showed that the EAM is indeed a feasible model for Prolog compiler implementation.

The BEAM showed us that by leveraging the EAM constructs and rules, it is possible to achieve both performance and parallelism.

Drawing over the BEAM's encouraging results, Andre proposed wam2eam [AA, AA10], a translator from WAM IL[1] to C, by using the EAM And-Or Tree as a base for Prolog execution in a parallel engine. While translation from WAM code to executable code using the EAM as a base for the runtime seems like a good starting point, we think that an intermediate approach can be more flexible, by translating WAM code to EAM code. This intermediate approach allows us to introduce optimizations in two phases:

1. By performing abstract analysis over the WAM code, generate EAM code optimized for parallel execution;
2. By running EAM specific code, the runtime can be focused on the underlying target architecture and be optimized accordingly.

Although there is already a "pseudo"-standard WAM language, there isn't a counterpart for the EAM. This chapter presents a WAM-to-EAM compiler, '`wam2steam`', further introducing the `ST`EAM-IL, an Intermediate Language that reflects the EAM machinery, just as the WAM language reflects the WAM's.

Most of the current Prolog compilers use the approach of precompiling the Prolog code to WAM code, which will then be compiled either to bytecode or to native code. Some of them enable the developer to export that WAM code to a file, a feature we will leverage in order to build our translator to EAM code. The current implementation of `wam2steam` uses the WAM code generated by `pl2wam` as input, producing `ST`EAM-IL code as output.

4.1 The `pl2wam` translator

The GNU Prolog [DC00b] compiler uses an interesting approach to the compilation of Prolog programs. By using a multi-step compilation scheme, a Prolog program is sequentially processed by a chain of independent translating programs:

1. `pl2wam`, entirely written in Prolog, compiles a Prolog source file into WAM code. This compiler uses many of the standard WAM optimizations, like indexing, last

[1] Intermediate Language

call optimization, as well as some optimizations that are specific to the GNU Prolog system internals.

2. `wam2ma` then translates the WAM code to mini-assembly (MA) [DC00a], a low-level language designed by Diaz specifically for GNU Prolog. The main purpose of the MA language is to be as much low-level as possible, while being simple to translate to assembly languages for different architectures (e.g., Intel 386, Intel 64bit, PowerPC, etc.).

3. The mini-assembly is then translated to architecture-specific assembly language by the `ma2asm` program.

Having obtained the target architecture assembly code, the remaining of the process is achieved by the standard GNU assembler and linker (i.e., `as` and `ld`).

Before using this compilation scheme, Diaz used a different compilation scheme, where the WAM code was directly translated to C code, via the wamcc compiler [CD95]. The change from compiling directly to C code to the current approach led to a performance increase of GNU Prolog compiled code.

4.2 `wam2steam` - compiling WAM to STEAM

The flexibility achieved by this multi-step model of compilation, allied to Andre's work led us to develop a multi-step model of compilation, using WAM code as a starting point, but diverging to EAM intermediate code afterwards.

In this multi-step model, the `pl2wam` compiler is used to transform a Prolog program into WAM code, which we will further process, in order to obtain STEAM-IL code. With that in mind, we started by defining the base STEAM-IL instructions.

4.2.1 STEAM-IL instructions

The EAM proposes a very different model of execution from the WAM. By using or-boxes instead of choice-point frames in the stack, the EAM doesn't indeed need a

stack. It does, still, need a heap, in order to allow the building of composite terms. The execution on the EAM is done inside the and-boxes, possibly in parallel, while execution in the WAM is done sequentially, using the stack to store temporary results that possibly will be discarded later.

Despite these structural differences, the execution of a specific goal is very similar across the two approaches:

- Argument terms must be built before a call (`put_*` instructions);
- Unification is the standard mechanism for resolving Prolog goals (`get_*` and `unify_*` instructions);
- Indexing is useful, as it allows to reduce the search space (`switch_*` instructions).
- Also, explicit pruning gives us opportunities to reduce the search space (`cut`, `neck_cut`).

With this being said, `ST`EAM-IL can reuse most of the WAM instructions, providing we take care of the cases where variables were stored on the WAM stack (i.e., in Y registers). The `try*`, `retry*` and `trust*` instructions don't have any meaning in the EAM, as choice-point management is ensured by the creation of or-boxes. However, in the context of translating WAM to `ST`EAM, those instructions will be useful to identify the alternatives which will be put under an or-box.

Following the WAM definitions in section 3.2.2 (chapter 3), we propose the following base instruction set for the `ST`EAM-IL:

Our main goal is to provide a set of instructions that will be executed in an EAM context, while producing the same results that the original WAM code would produce (in a WAM context). The `ST`EAM-IL code, when executed, will produce the and- and or-boxes that will form the And-Or Tree. It will be up to the runtime, though, to decide where and when to use parallelism.

In order to translate the original WAM code to `ST`EAM-IL code, the `wam2steam` compiler starts by performing an abstract analysis over the WAM code. In the current prototype implementation, we used the traditional compiler construction flex+bison pair.

put_* instructions	used to create elements on the heap or in the and-box
get_* and **unify_*** instructions	used to perform binding of variables, by means of unification
call	triggers the execution of a goal, or the creation of its or-box with children and-boxes
proceed	indicates the successful end of a computation
fail	indicates that the computation has failed
allocate_or $L_1, ..., L_n$	creates an or-box with n alternatives, each with code starting at label L_i
allocate_and $L_1, ..., L_n$	creates an and-box with n conjunctive goals, each with code tarting at label L_i

Table 4.1: Initial instruction set for STEAM-IL.

4.2.2 Abstract analysis of the WAM code

The `wam2steam` compiler starts by constructing an APT (Abstract Parse Tree) that represents the original WAM code. This APT is the used to perform abstract analysis, from which we firstly get the following (almost) direct translations:

- **choice-point instructions** (`try_*`, `retry_*`, `trust_*`) determine how many alternatives an or-box will have. We can then use the labels associated with the different choice-points as arguments to the `allocate_or` instructions, signaling where the code for each alternative is located.
- **call** and **execute** instructions are used to determine the conjunctive goals belonging to a single and-box, as well as the location of their respective code. They provide the locations of the code for `allocate_and` instructions, but have no labels, which means we have to create those labels. Also, as the difference between `call` and `execute` is based on the existence of the WAM stack, which doesn't exist in the EAM, we can merge both instructions into the STEAM-IL `call` instruction.
- **put_* instructions** allow us to determine the arguments, the local and external variables of an and-box, by looking at the argument registers and the arity of the predicate.
- **switch_* instructions** generate indexing information, which will later be used to reduce the search space (e.g., by allocating smaller or-boxes).

Besides these translations, there is some more information that we can obtain from the abstract analysis, which can later be useful for the runtime to make decisions regarding the execution of the code. The information we are retrieving is the following:

- **Determinate predicates** - As determinate predicates can be promoted in the STEAM execution context, we'll take note of determinacy in order to allow the early promotion of such predicates. This is accomplished by annotating as determinate a predicate that has only one alternative.

- **Facts** - Facts are simple predicates that don't implicate further calls and are usually the source for variable bindings. We may want these predicates to execute earlier (or later, for that matter) than others, so we also annotate facts.

- **Rules with a single goal in the body** - These are also useful to differentiate, as they don't require and-parallelism.

- **Control predicates** - Control predicates (e.g., *`cut (!)`*, `repeat`) change the flow of the program, by adding or removing alternatives from the search space. Parallel execution of these predicates, when possible, must be performed with extra care.

- **Non-logical predicates** - Here we include extra-logical and meta-logical predicates, which can trigger side-effects or can perform functions that must be executed at a specific time, thus not able to be run in parallel with other dependent goals.

- **Independent variables** - Although not always possible, in some cases we can detect if a variable is independent. By annotating it, we can allow the STEAM runtime to execute goals that use independent variables in and-parallel, without restrictions.

- **All-solution predicates** - This kind of predicates ask for all the solutions available, allowing us to search for solutions in parallel, without concerns over speculative work, maximizing the potential parallelism.

Control predicates

In the current implementation of `wam2eam`, we are only handling the *cut* operator (!), but we have plans to implement others (e.g., the controversial `repeat, ..., fail` loops).

Cut instructions are very important in an EAM context, as they can effectively reduce the search space. Consider the query $a(X), b(X), !, \ldots$: the cut means that we only want the first value for X that satisfies both $a(X)$ and $b(X)$, which means we can delete all non-solved or-alternatives for $a(X)$ and $b(X)$ as soon as we find one that succeeds. Further, if the previous query is the body of a clause in our Prolog program, we can also delete the other alternative clauses.

In a WAM context, the cut is executed as soon as a valid X is found. In an EAM context, though, it's possible that when the cut is performed, all the or-alternatives for $a(X)$ and $b(X)$ have already been calculated. However, we may in part be able to prevent this if we annotate the code that calls $a/2$ and $b/2$, letting the runtime know that only one solution is needed.

A special case of the cut operator is when it appears as the first goal in the body of a clause. This is generally called a *neck cut*, and represents a good opportunity to early pruning of unneeded nodes in the STEAM tree. In short, neck cuts allow us to ignore all alternative clauses for a predicate as soon as the clause's head is matched.

By looking into the generated WAM code, we can easily detect each form of cut:

- A `cut` instruction appearing before all `call` and `execute` instructions is a neck cut and we can safely remove other alternative clauses from the And-Or Tree as soon as the execution reaches this cut operator;
- A `cut` instruction appearing after a `call` instruction has already been executed can be signaled beforehand, so that the runtime knows that only one solution is to be found and, as soon as that happens, the other alternatives can be pruned.

For this we will include three more instructions in the STEAM-IL:

[2] Previous alternatives are kept, as we're mimicking sequential Prolog.

cut_scope i	Initiates scope i for a `cut` instruction
neck_cut	Performs a neck cut, removing the remaining alternatives[2] from the And-Or Tree
cut i	Performs a cut in scope i, i.e., discards all alternatives but the first that produces a valid binding.

Table 4.2: STEAM-IL explicit pruning instructions.

Figures 4.1 and 4.2 show an example of how **wam2steam** deals with **cut** instructions.

```
1 p(X,Y) :- q(X),r(Y),!.
2 p(X,Y) :- !, r(X), r(Y).
```

Figure 4.1: Example Prolog program with cuts.

```
1  nondetpredicate p/2:
2    get_current_choice x2
3    allocate_or (2) L000001 L000003
4  L000001:
5    cut_scope C001
6    allocate_and (3) LA1 LA2 LA3
7  LA1:
8    get_variable y0, 1
9    get_variable y1, 2
10   call q/1
11 LA2:
12   put_value y0, 0
13   call r/1
14 LA3:
15   cut C001
16   proceed
17 L000002:
18 L000003:
19   allocate_and (3) LA4 LA5 LA6
20 LA4:
21   get_variable y0, 1
22   neck_cut
23 LA5:
24   call r/1
25 LA6:
26   put_value y0, 0
27   call r/1
28   proceed
```

Figure 4.2: Resulting STEAM-IL.

Meta- and extra-logical predicates

In presence of an extra-logical predicate, we must be very cautious, as either early or late evaluation of side-effects can be disastrous, albeit we don't want to lose parallelism in the presence of such predicate. Meta-logical predicates are also of concern, as they generally require that the execution has already reached a specific state when they are called. Although we leave the execution worries to the runtime, we can identify these predicates (and predicates that call them) and advert the runtime that those predicates are to be handled carefully.

To identify predicates which can trigger side-effects, we perform a sweep test on all the predicates that call side-effects builtins (e.g., `assert`, `retract`, etc.) and the ones that call them. To allow the runtime to distinguish between calls to predicates with side-effects and predicates that haven't side-effects but (can) call those predicates, we mark the former in "red" and the latter in "yellow". Meta-logical predicates are marked "orange", and the predicates that call them are also "yellow". We then generate different instructions for "red", "orange" and "yellow" calls. Figure 4.3 shows a simple algorithm that shows the concept behind this "coloring" of the calls.

```
do:
  marked = false
  for pred in not_marked_preds:
    for call in pred.calls:
      if call in {'assert', 'retract', ...}:
         mark_red(call)
         mark_yellow(pred)
         marked = true
      else if call in {'var', 'nonvar', ...}:
         mark_orange(call)
         mark_yellow(pred)
         marked = true
      else if call is marked:
         mark_yellow(call)
         mark_yellow(pred)
         marked = true
         break
while marked
```

Figure 4.3: Resulting STEAM-IL.

We mark both the predicate and the call, so that we can still allow the not marked calls within a predicate to run without concerns of triggering side-effects. After marking all the "dangerous" predicates, we can then generate appropriate code that effectively

adverts the runtime that those predicates are *fragile* and must be handled with care. Thus, we added five new instructions to STEAM-IL to handle predicates with side-effects:

call_fragile	Calls a predicate which include calls to (calls to...) meta- or extra-logical predicates
call_sidefx	Calls a extra-logical predicate
call_meta	Calls a meta-logical predicate
wait i n	Waits (suspends) until semaphore i reaches level n
signal i	If the current call succeeds, increase semaphore i

Table 4.3: STEAM-IL extra-logical handling instructions.

To cope with left-to-right evaluation needs, we simulate a n-level *semaphore*, with n corresponding to the number of calls before the side-effects call. Each (succeeding) call preceding the side-effects call will increase the semaphore level by one, using the `signal` instruction. The side-effects call will have a `wait` instruction which forces it to suspend until the semaphore reaches level n and, as soon as that level is reached, the side-effects call can be executed.

The `wait` and `signal` instructions work in the following way: the `wait` instruction creates a semaphore with n states, and waits for the semaphore to reach zero. Each `signal` instruction decreases the semaphore by one. Once the semaphore reaches the level zero, the `wait` instruction let the control pass for the next instruction, as it means that the side effects have been executed.

After the generation of the APT, the system performs an analysis of the program, focusing on finding the blocks which will later generate and- and or-boxes.

After the abstract analysis, the system proceeds to the generation of STEAM-IL code.

4.3 STEAM-IL code generation

As soon as the semantic information is deducted, we can now generate STEAM-IL instructions that correspond to the initial Prolog source code.

Figure 4.4 shows an example Prolog program, `'append.pl'`. This example is translated to WAM code by `pl2wam` (figure 4.5), which is then translated to STEAM-IL code (figure

4.6).

```
append(X, [], X).
append([H|T], X, [H|T2]) :- append(T, X, T2).
```

Figure 4.4: Example Prolog program (`'append.pl'`).

```
predicate(append/3,1,static,private,monofile,global,[
    try_me_else(1),
    get_nil(1),
    get_value(x(2),0),
    proceed,

label(1),
    trust_me_else_fail,
    get_list(0),
    unify_variable(x(3)),
    unify_variable(x(0)),
    get_list(2),
    unify_value(x(3)),
    unify_variable(x(2)),
    execute(append/3)]).
```

Figure 4.5: Resulting WAM code, generated by `pl2wam`.

```
nondetpredicate append/3:          ||   L3:
                                   ||        alocate_and L4
                                   ||   L4:
    allocate_or L1, L3             ||        get_list 0
L1:                                ||        unify_variable x3
    allocate_and L2                ||        unify_variable x0
L2:                                ||        get_list 2
    get_nil 1                      ||        unify_value x3
    get_value x2, 0                ||        unify_variable x2
    proceed                        ||        call append/3
                                   ||        proceed
```

Figure 4.6: Resulting STEAM-IL code.

4.3.1 Detecting patterns

While analyzing the WAM source code, we can observe recurring patterns, which can be translated to equivalent STEAM-IL code:

- **'`try_me_else` … `retry_me_else` … … `trust_me_else_fail`' patterns** are converted to `allocate_or` instructions (figure 4.7). As the `pl2wam` translator (almost) always sets a label for the instruction following the `try*/retry*/trust*`

instruction, we can use that labels in the `allocate_or` instruction. When the following instruction doesn't have a label, we simply add a new one.

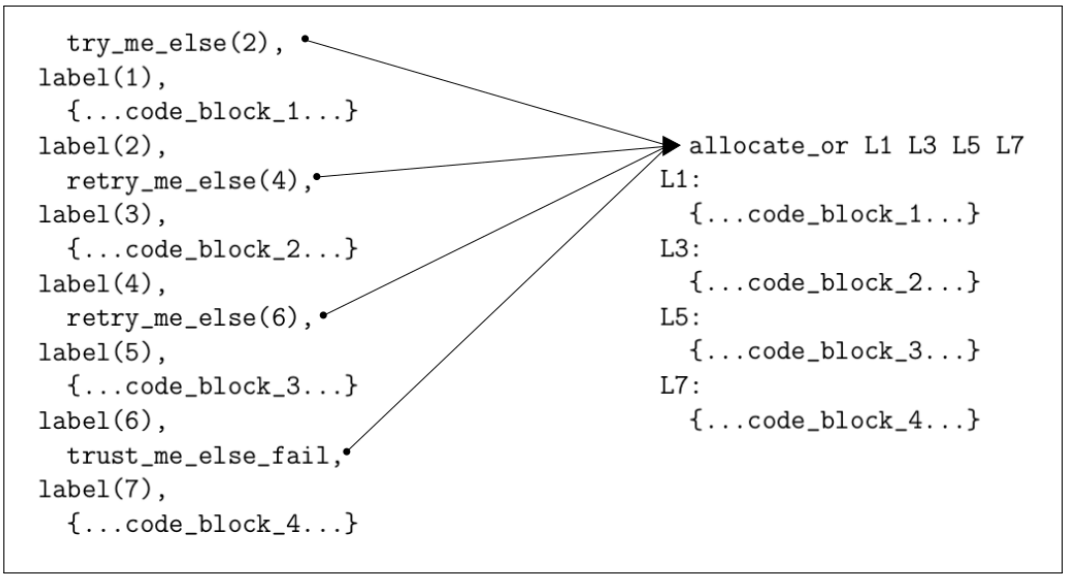

Figure 4.7: '`try_me_else` … `retry_me_else` … … `trust_me_else_fail`' pattern translation.

- **'`try` … `retry` … … `trust`' patterns** are also translated to `allocate_or` instructions (figure 4.8). However, these WAM instructions differ from the previous ones, as they simply jump to the label and set the continuation pointer to the next instruction, instead of continuing with the execution of that next instruction. This makes our job easier, as we simply need to generate an `allocate_or` instruction with the same labels that the original instructions were jumping to.

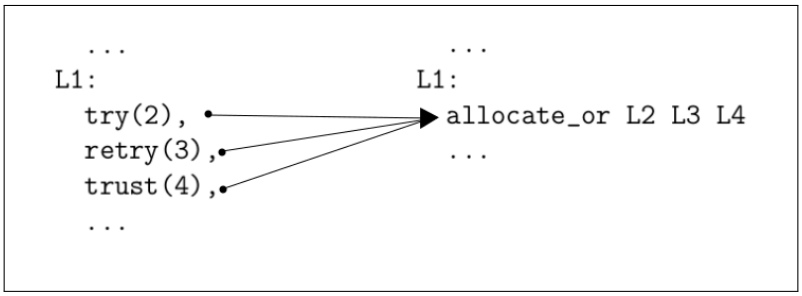

Figure 4.8: '`try` … `retry` … … `trust`' pattern translation.

- `call` and `execute` instructions are always preceded by a group of instructions that build the argument terms on the heap. We can then put a label before each of those instruction groups, defining an *and-node* or call inside an *and-box* (figure 4.9).

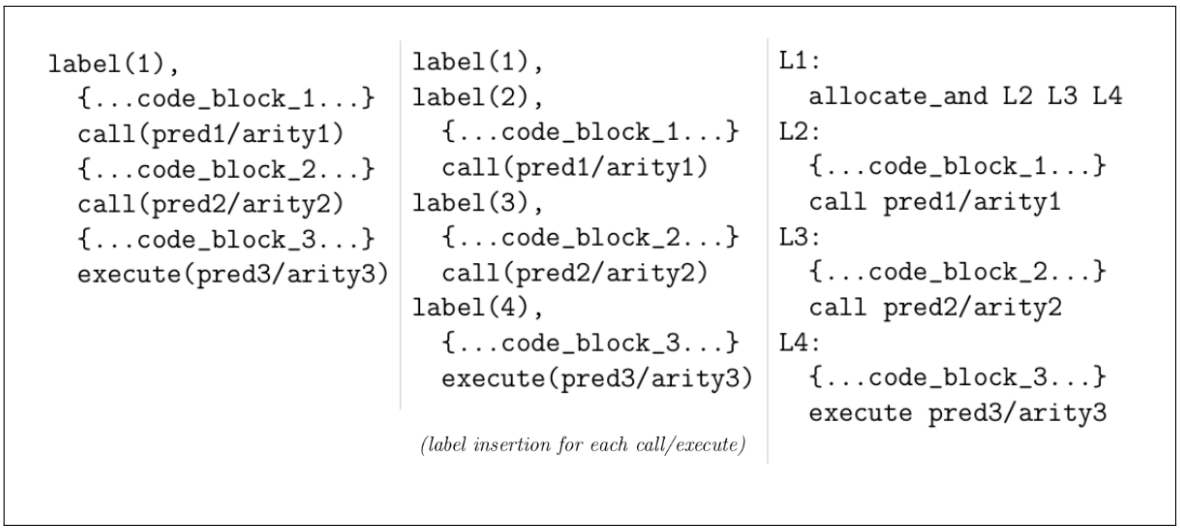

Figure 4.9: 'conjunctive `call`/`execute`' pattern translation.

4.3.2 Optimizing STEAM-IL execution

One of the issues with STEAM-IL is the fact that some instructions have a variable number of arguments, which makes it difficult to, for instance, generate bytecode for a STEAM interpreter to run, as we cannot easily detect the number of arguments associated with the instruction. One solution that would work would be passing the arguments on a stack, as shown in the example in table 4.4.

Old form	New form
`allocate_or` $L_1, ..., L_n$	`push` L_n `push` ... `push` L_1 `allocate_or` n
`switch_on_integer` $(1, L1), (2, L2), (3, L3)$	`switch_on_integer` $1, L1$ `switch_on_integer` $2, L2$ `switch_on_integer` $3, L3$

Table 4.4: Converting to a fixed number of arguments

This will allow to have a predefined size for each instruction, making it easier to generate a compatible bytecode for the runtime, either an interpreter or a compiler[3]. Also, this scheme can also be a base for adding foreign predicates, by allowing the arguments to be passed on the stack.

Dealing with `repeat, ..., fail` loops can also be achieved in STEAM-IL by adding,

[3]Although the code is compiled, we still need support for dynamic predicates, which will generally be pre-compiled to bytecode.

for instance, an instruction that generates a special or-box that has unlimited alternatives. Adding support for this type of predicates to STEAM and STEAM-IL, as well as calls to foreign predicates are features we have plans to implement in future work.

Anther current limitation of the `wam2steam` compiler relates to the WAM code produced by `pl2wam` being very specific to the GNU Prolog compiler and its internals (by being dependent of many specific built-in, non-standard, predicates and automatically applying WAM-specific optimizations), making it difficult to translate some Prolog programs. Its a matter of future study if we will modify the behavior of `pl2wam` or rely on a specifically built Prolog-to-WAM compiler.

4.4 Preparing for execution

The main purpose of translating a Prolog program to STEAM-IL is to enable the execution of that program in a EAM-based runtime. This can be achieved in various ways:

- By compiling STEAM-IL directly to a low level language: like Andre's `wam2eam` [AA], one possibility is to compile STEAM-IL directly to C code
- By compiling STEAM-IL to native code, like Diaz's GNU Prolog [DC00a], using an intermediate, platform-independent assembly language which will later be compiled to native assembly, while relying on a runtime which is compiled separately and linked with the generated native code.
- By compiling STEAM-IL to bytecode, which will be executed by an interpreter. This alternative is in part similar to the previous one, as the runtime is separated from the STEAM-IL program.

In our view, the approach of compiling STEAM-IL to executable code would result in a loss of flexibility, as we would be hard-coding the runtime into the generated program. Also, as we are aiming to a pluggable system, it seems more logical to have a runtime that is logically detached from the program we're going to compile. In the next chapters, we will propose a model that copes with of the two latter alternatives, either by the means of an STEAM-IL interpreter or a compiler.

One interesting alternative we have considered was the use of the LLVM framework [LA04], as compilation from STEAM-IL to the LLVM IR[4] would provide portability to all the LLVM supported architectures. However, the LLVM framework doesn't (yet) support parallel programming models natively, so we will revisit this alternative in the future.

4.5 Closing remarks

This chapter presented the STEAM Intermediate Language, as well as the `wam2steam` compiler and its internal implementation details, focusing in the abstract analysis of the WAM code, in order to obtain the information needed to correctly execute Prolog programs.

By using an intermediate language that reflects the EAM, we can previously perform some optimizations to the source code, while also providing useful predicate annotations that can be posteriorly exploited by the runtime, either to make decisions in regards to the execution order, either to handle special predicates with special care.

As we are aiming at a pluggable system, the following step is to perform the effective execution of the STEAM-IL code, by means of an EAM runtime. In the next chapters we will propose a base model for the execution of STEAM-IL programs, which we'll call *STEAM*.

[4] Intermediate Representation

5

STEAM - Scalable, Transparent EAM

Nowadays it's hard to find a computer with less than two processing cores. The demand for more and more processing power has led to a situation where it's difficult to improve single-processor performance, which made the hardware industry focus in parallel, multi-processor (or multi-core) machines.

In what concerns Prolog, with the advent of stagnancy of single-core performance, the need for parallel compilers has increased, as the only way to exploit the full speed of multi-processor machines is by implementing parallel strategies for executing Prolog programs. The EAM provides a strong base for running Prolog programs in parallel hardware, as it allows to better express parallelism than the widely used WAM approach.

This chapter introduces STEAM, a model that leverages the base constructs of the Extended Andorra Model to achieve implicit, parallel execution of Prolog programs.

5.1 Introduction

STEAM aims to provide a base model for implementation of Prolog compilers, using the EAM as reference and relying on two base requirements:

1. **Scalability** - STEAM must be scalable in distributed environments.
2. **Transparency** - As a Prolog engine, it must be transparent to the user, e.g. its percepted behaviour shall be the same as a traditional, sequential Prolog engine's.

STEAM relies on the STEAM-IL presented in the previous chapter, in order to provide a flexible, pluggable system for executing Prolog programs in parallel, while keeping the traditional Prolog syntax and semantics.

While STEAM is based on the Extended Andorra Model, it contains some new concepts, in order to provide adaptability to contemporary parallel architectures. The following sections describe the STEAM model in detail, whilst referring the similarities and differences from the EAM.

5.2 Definitions

The main component of the STEAM is the *And-Or Tree*, henceforth *Tree*, which is comprised by *and-boxes* (figure 5.1), that represent a logical conjunction of Prolog goals, and *or-boxes* (figure 5.2), that represent alternative clauses for a Prolog rule.

We use the term *configuration* to describe a state of a computation, by means of a Tree. The *initial configuration* comprises a single and-box that contains one or more and-nodes, corresponding to the initial conjunction of goals that form the initial query.

A *computation* is the process by which the initial configuration passes, by means of successive application of the rewrite rules, until it becomes a *final configuration*, which

can consist either in an and-box with an *answer* (or set of answers) that comprises the successful binding(s) of the variables in the query, or in a *failure* to get a valid solution.

A computation can be applied to a branch of the Tree. When a computation ends, the final and-box will have either succeeded or failed. We call these and-boxes *true-boxes* or *fail-boxes*.

5.3 STEAM base constructs

Following Warren's EAM specification [War90], STEAM uses a Tree of and-boxes (figure 5.1) and or-boxes (figure 5.2), that will be subject to transformations, according to logically correct rewriting rules. Further, in STEAM, and-box's children have to be or-boxes and or-boxes can only have and-boxes as children.

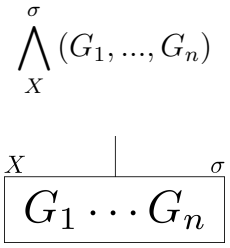

Figure 5.1: Graphical representation of a STEAM and-box

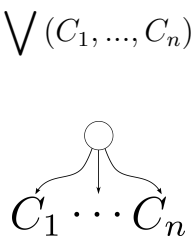

Figure 5.2: Graphical representation of a STEAM or-box

5.4 Rewriting rules

Let us describe the STEAM's rules formally. We'll represent an and-box with local variables $X = \{X_1, ..., X_n\}$ and constraints σ as $\bigwedge_X^\sigma$, an or-box as $\bigvee$. The rewrites are shown in the form:

$$\text{(Previous configuration)} \xrightarrow[\text{(rule)}]{\text{(vars or boxes)}} \text{(Rewritten configuration)}$$

- **Expansion** (figure 5.3) consists in expanding a goal G in an and-box into an or-box corresponding to all the alternative clauses that make the definition of the G predicate. Let A, B and G be atomic goals and $C_1, ..., C_n$ and-boxes corresponding to each alternative clause in G's definition.

$$\bigwedge_X^\sigma (A, G, B) \xrightarrow[expansion]{G} \bigwedge_X^\sigma \left(A, \bigvee (C_1, \ldots, C_n), B \right) \tag{5.1}$$

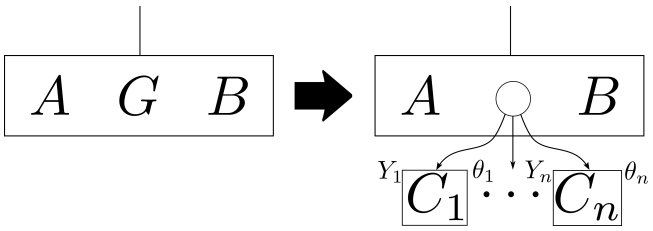

Figure 5.3: STEAM expansion rule.

The expansion rule mimics the EAM local forking rule.

- **Determinate promotion** (figure 5.4): when an or-box has only one child and-box (i.e., has a single alternative), this and-box can be merged with the (grand) parent and-box.

$$\bigwedge_X^\sigma \left(A, \bigvee \left(\bigwedge_W^\theta (G) \right), B \right) \xrightarrow[det\_prom]{G} \bigwedge_{X,W}^{\sigma\theta} (A, G, B) \tag{5.2}$$

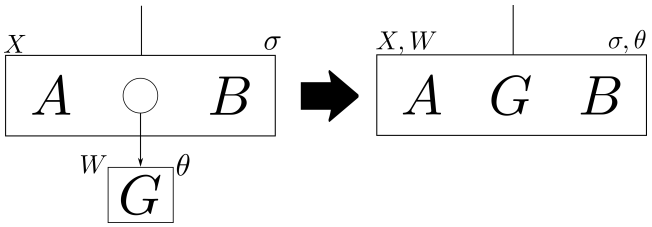

Figure 5.4: STEAM determinate promotion rule.

This rule is equivalent to the EAM determinate promotion rule, usually meaning the end of a computation. Further, this rule allows us to recover memory when a computation finalizes. There is, however, a case in which merging the child and the parent and-boxes may result in an invalid behavior: if the child and-box contains pruning operators (e.g., cut), the merging must be avoided. With this situation in mind, we have a specific promotion rule which can be applied only if the child and-box doesn't include pruning operators.

- **Determinate careful promotion** (figure 5.5): when standard determinate promotion can result in incorrect behavior, the child box is not merged into the parent, instead its constraints and bindings are merged with the ones in the parent. An example when this rule must be applied is the case where the box to be promoted contains pruning operators (e.g., cut), which can't be promoted to the parent box, as it would alter the scope of the pruning operator.

$$\bigwedge_{X}^{\sigma}\left(A,\bigvee\left(\bigwedge_{W}^{\theta}(G)\right),B\right)\xrightarrow[det\_cprom]{G}\bigwedge_{X,W}^{\sigma\theta}\left(A,\bigvee\left(\bigwedge(G)\right),B\right) \tag{5.3}$$

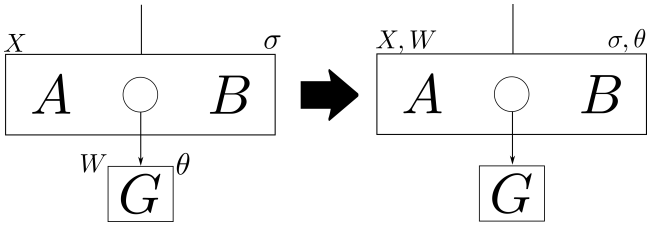

Figure 5.5: STEAM determinate careful promotion rule.

- **Splitting or non-determinate promotion** (figure 5.6): occurs when no determinate or-box exists that can be promoted. Consists in placing two copies of the

parent and-box under a new or-box. One of the copies has the promoted and-box and the other has the remaining children of the original or-box.

$$\frac{\overset{\sigma}{\underset{X}{\bigwedge}}\left(A, \bigvee\left(\bigwedge\left(C_1\right), ..., \overset{\theta}{\underset{W}{\bigwedge}}\left(C_i\right), ..., \bigwedge\left(C_n\right)\right), B\right) \xrightarrow[split]{C_i}}{\bigvee\left(\overset{\sigma\theta}{\underset{X,W}{\bigwedge}}\left(A, C_i, B\right), \overset{\sigma}{\underset{X}{\bigwedge}}\left(A, \bigvee\left(\bigwedge\left(C_1\right), ..., \bigwedge\left(C_n\right)\right), B\right)\right)} \tag{5.4}$$

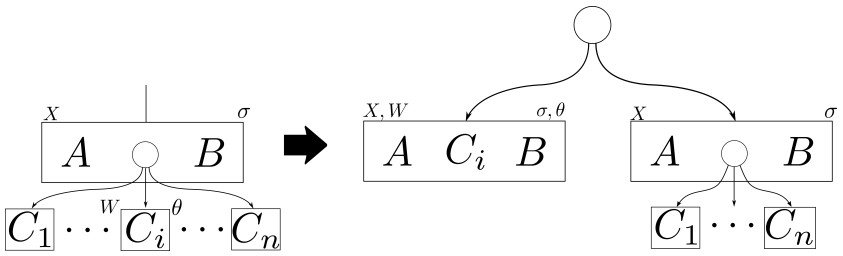

Figure 5.6: STEAM splitting rule.

This rule is the same as the EAM non-determinate promotion, and should be used only when there's no other applicable rule, as it's the most (computationally) expensive rule, and generates duplicate work, as all the goals that are to be solved after the splitting point must be recomputed for each of the new branches generated by the split.

- **In-loco expansion** (figure 5.7): if a goal to be expanded is deterministic, the application of the expansion rule can be readily followed by the application of the promotion rule. Lopes proposed a combined rule called deterministic-reduce-and-promote in the BEAM [LCC99], which we will also adopt.

  Let $C_1, ..., C_n$ be the calls in the unfolded clause G, with local variables $Y = \{Y_1, ..., Y_m\}$, imposing constraints θ.

$$\overset{\sigma}{\underset{X}{\bigwedge}}\left(A, G, B\right) \xrightarrow[in-loco]{G} \overset{\sigma\theta}{\underset{X,Y}{\bigwedge}}\left(A, C_1, ..., C_n, B\right) \tag{5.5}$$

- **In-loco careful expansion** (figure 5.8): this rule is analogous to the previous

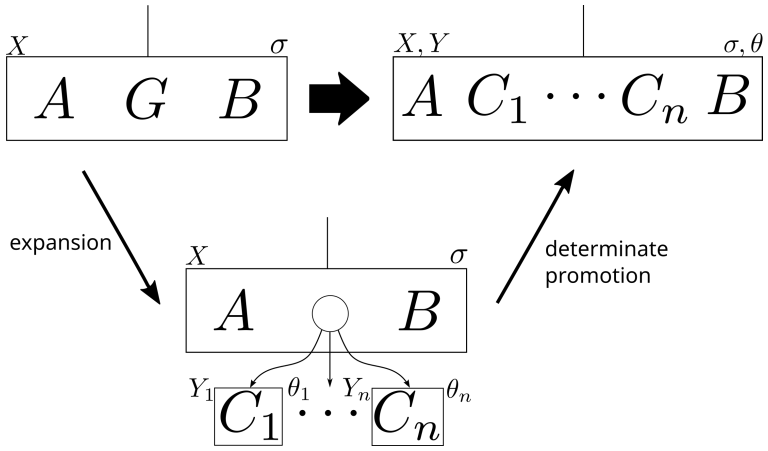

Figure 5.7: STEAM in-loco expansion rule.

one, but it uses the determinate careful promotion instead, when the expanded (and-) box to be promoted includes pruning operators. In this case, only the bindings and constraints are promoted.

$$\bigwedge_{X}^{\sigma}(A,G,B) \xrightarrow[c\_in-loco]{G} \bigwedge_{X,Y}^{\sigma\theta}\left(A,\bigvee(C_1,...,C_m),B\right) \tag{5.6}$$

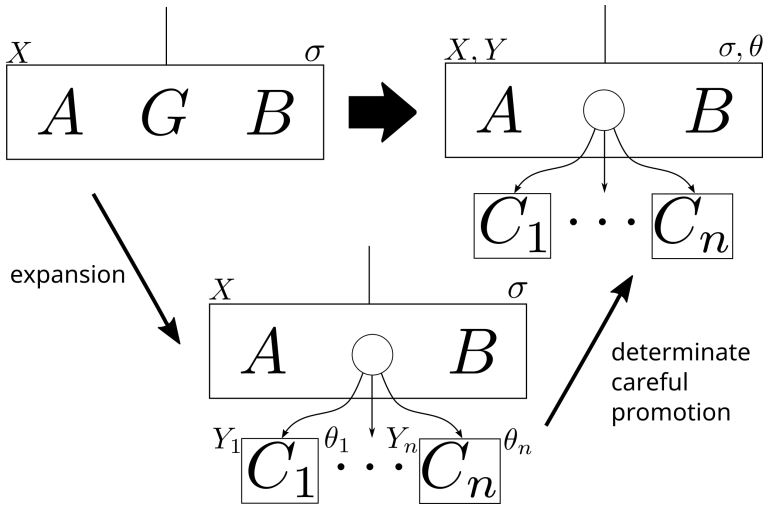

Figure 5.8: STEAM in-loco careful expansion rule.

5.5 Reducing the search space

Reducing the size of the Tree provides two outcomes: on the one hand we can release the memory allocated to the data structures that contain the removed branch; on the other hand, we also reduce the search space for the computation being performed. With this in mind, it makes sense that we implement rewriting rules that will allow early reducing of the size of the Tree. This pruning of the Tree can occur implicitly (simplification) or explicitly (activated by pruning operators).

5.5.1 Simplification

stEAM uses two rewriting rules that represent the logical properties of conjunctions and disjunctions, thus achieving implicit pruning and subsequent reduction of the search space:

- **or-identity** (figure 5.9): if an and-box fails under an or-box, we can simply remove it from the Tree (and, consequently, its whole branch).

$$\bigvee(C_1, ..., fail, ..., C_n) \xrightarrow[or-id]{} \bigvee(C_1, ..., C_n) \tag{5.7}$$

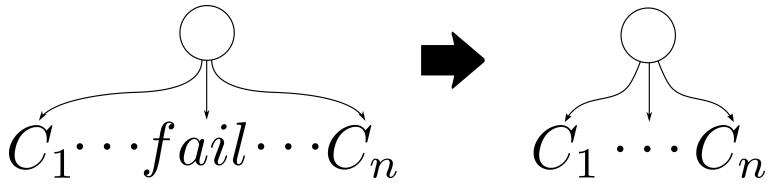

Figure 5.9: stEAM or-identity

- **and-annihilator** (figure 5.10): when a goal inside an and-box fails, the whole and-box (and its children or-boxes) also fails.

$$\bigwedge(G_1, ..., fail, ..., G_n) \xrightarrow[and-an]{} fail \tag{5.8}$$

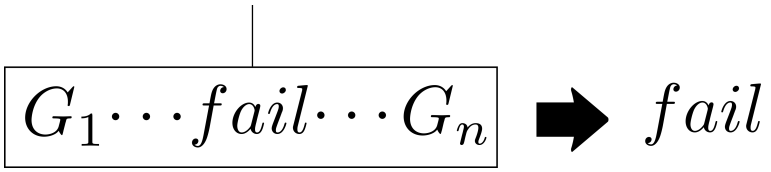

Figure 5.10: STEAM and-annihilator

5.5.2 Pruning operators

Standard Prolog has only one pruning operator, **!** (cut), which essentially removes all untested alternatives for a clause from the search space, as soon as one clause's bindings succeed. In the WAM, this corresponds to deleting the choice points stored in the stack, in order to disable backtracking. In STEAM we adopt the analogous behaviour, by removing untested alternatives under an or box that are right-siblings of the alternative containing the cut operator, as soon as the goals in the and-box containing the cut operator (the ones at the left of the operator) succeed.

Assuming we have an or-box with one of its children and-boxes having a cut operator between goals C_1 and C_2, we can discard all the right siblings of that and-box as soon as C_1 succeeds. Figure 5.11 shows a graphical representation of the STEAM cut operation.

$$\bigwedge_X^{\sigma}\left(A, \bigvee\left(L, \bigwedge_W^{\theta}\left(C_1, !, C_2\right), R\right), B\right) \xrightarrow[cut]{C_1} \bigwedge_X^{\sigma}\left(A, \bigvee\left(L, \bigwedge_W^{\theta}\left(C_1, C_2\right)\right), B\right) \quad (5.9)$$

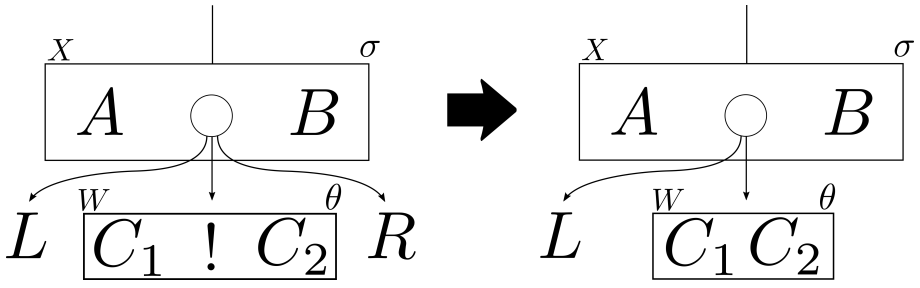

Figure 5.11: STEAM implicit pruning (cut)

A special case of the cut operator, which is also subject to optimizations in WAM implementations, is the *neck cut*. A neck cut is a cut operator that is the first goal of a clause, thus removing future alternatives as soon as the head of the clause unifies whith the query. The neck cut can be applied as soon as the head of the alternative

containing it matches the query (or the goal that expands to this alternative).

$$\bigwedge_X^{\sigma}\left(A, \bigvee\left(P, \bigwedge_W^{\theta}(!, C), Q\right), B\right) \xrightarrow[neck\_cut]{C} \bigwedge_X^{\sigma}\left(A, \bigvee\left(P, \bigwedge_W^{\theta}(C)\right), B\right) \tag{5.10}$$

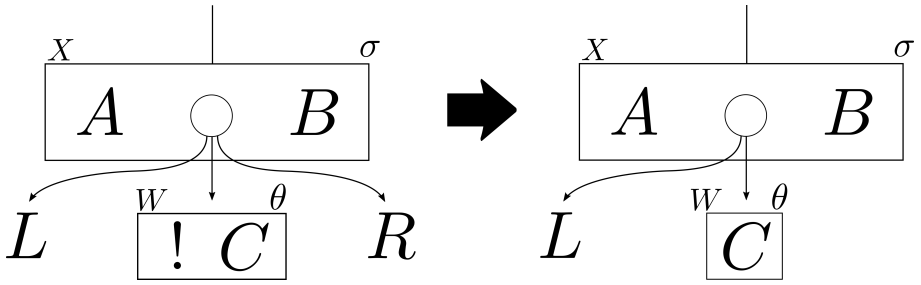

Figure 5.12: STEAM implicit pruning (neck cut)

5.6 Termination

In [Lop01], Lopes pointed out that the default EAM strategy of suspending every computation that tries to perform external bindings can lead to non-termination, especially in the presence of recursive predicates. Gupta, in his prototype EAM interpreter [GW91], proposed an optimization to tackle this problem: by using *eager nondeterminate promotion*, one can split the Tree in order to get the (split) suspended boxes to continue their computation. One drawback of Gupta's approach is that variables must be classified as *guessable* or *non-guessable*, either by abstract analysis of the source code or by annotations provided by the programmer. This classification of variables can be very hard to perform automatically at compile-time in some programs, so this means we would have to rely on programmer annotations, which breaks our purpose to achieve implicit parallelism in already existing Prolog programs.

In the BEAM, Lopes adopted a similar but simpler strategy: by using AKL's *stability* concept [Jan94a], *stable* and-boxes can be split eagerly. By definition, an and-box A as stable relatively to it's (grand) parent and-box if two conditions are met:

1. There's no determinate rule to be applied to A or its children;

2. None of the future rewrites that can be applied to other parts of the Tree would cause condition 1 to no longer be valid.

This definition of stability, although effective, can't be implemented efficiently, as we must know what goals share variables with the goals in A, which is a NP-Complete problem [DK89].

AKL marks a box as *unstable* when it suspends on external variables. An unstable box will never become stable in AKL. The BEAM tries to overcome this by using special markers in and-boxes, in order to detect stability. The and-boxes marked as stable can then be split eagerly, in order to defect the aforementioned non-termination problem.

In order to exemplify the non-termination risk in the EAM, let's consider the example program '`graph.pl`' in figure 5.13 and the initial query

```
?- path(X,Y).
```

```
1 edge(1,2). edge(1,3).
2
3 path(X,Y,[X,Y]) :- edge(X,Y).
4 path(X,Y,[X|T]) :- edge(X,Z), path(Z,Y,T).
```

Figure 5.13: Example Prolog program (`'graph.pl'`).

We can observe the execution of the query in an EAM environment in figure 5.14. In step *(1:)*, an and-box is created with the initial query `path(X,Y)`. This and-box has two internal variables, `X` and `Y`, and will be subject to the *local forking* rule, which unfolds the predicate `path/2` into it's both alternative clauses (step *(2:)*. These alternatives are then subject to the same local forking rule (step (3:)), where different situations occur: in the leftmost boxes, respective to the unfolding of the call `edge(X,Y)`, the unfolding leads to the instantiation of the external variables `X` and `Y`, which, in the EAM, cause the suspension of the boxes (as we can't bind external variables). The other alternative follows the same process, leading to the suspension of the children boxes of the `edge(X,Z)` goal and the local forking of the `path(Z,Y)` goal. This process will continue indefinitely, as the variables to be bound will always be external to the boxes, leading to infinite suspensions and, consequently, non-termination.

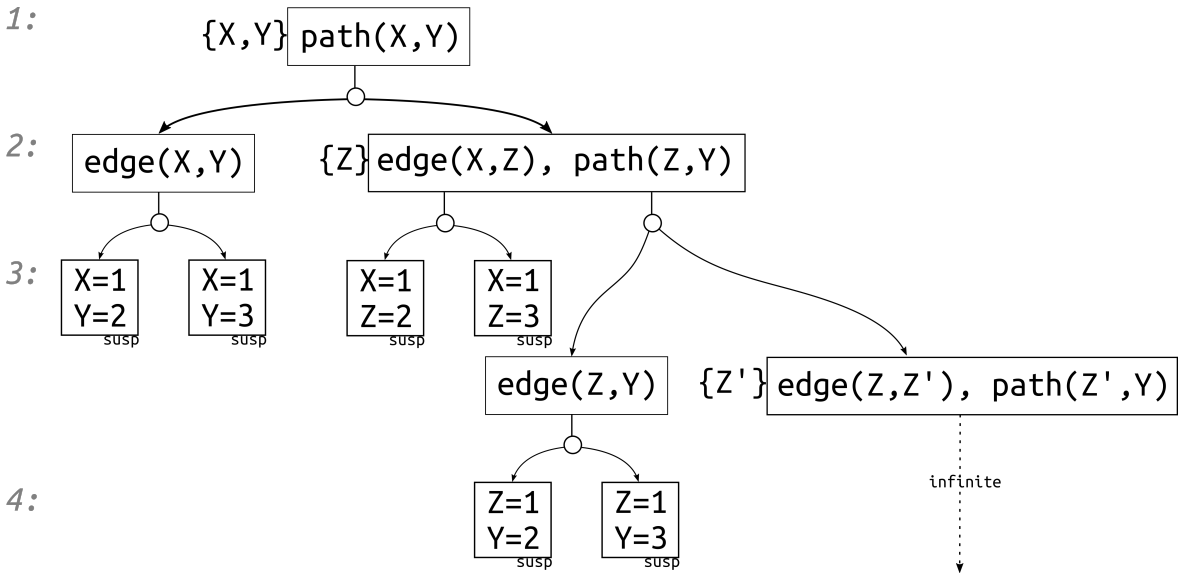

Figure 5.14: EAM non-termination example.

In a WAM context, the first bindings would be allowed, with success. In case of failure (or search for alternative results), backtracking would revert the variables to the unbound state, allowing to search for alternative bindings.

In stEAM, we want to achieve the same behavior as in the WAM, but taking advantage of parallelism. In the case discussed above, we could have found several solutions for the query, in advance, if it weren't for the suspension of boxes.

5.7 Suspension

Although suspension is an essential mechanism for the EAM, as discussed above, suspension of computations can lead to non-termination and, in consequence, no advantage in exploiting parallelism.

stEAM approaches suspension in a different form from the EAM specification, by "violating" the main Warren's principle in the design of the EAM: to minimize inference and maximize parallelism.

The main purpose of stEAM is to allow the exploitation of implicit parallelism in Prolog programs in parallel hardware, while keeping the sequential semantics intact. In order to do that, work that would normally be suspended (in order to favor determin-

istic computations) should be allowed to proceed in parallel with other, non suspended work, which, in turn, would lead to more inference. This gives rise to the first base principle in STEAM:

Favor maximum parallelism over minimum inference.

By favoring maximum parallelism, STEAM can take advantage of, for instance, SPMD[1] programming models, that allow to perform the same computation over different parts of a large amount of data simultaneously.

Suspension is used in STEAM sparingly, by not forcibly suspend every goal that tries to bind an external variable, suspending only boxes that are not essential to the first solutions of the problem, and then only when there aren't hardware resources to compute them (delaying non-essential computations).

In fact, STEAM doesn't suspend in the above example, by allowing each and-box to have it's private copy of the variables, thus making them internal to the and-box. When successful bindings occur, STEAM tries to merge them with the constraints of the parent box, allowing one of the two following situations:

- Compatible bindings mean that the computation has succeeded, and one (part of the) solution was found;
- Incompatible bindings mean that the computation has failed in the current branch, which can be discarded from the tree.

In the following sections, we'll further discuss how to enable STEAM to achieve maximum parallelism.

5.8 STEAM under the hood

This section describes how STEAM's base constructs fit in memory, as well as the extra information needed to allow the correct[2], parallel execution of Prolog programs.

[1] Single Program Multiple Data

[2] By correct, we mean equivalent to that of sequential Prolog

The first directive we'll use is to reuse WAM instructions as much as possible, which means we have to replicate (at least some of) the WAM memory areas (e.g., the **Heap**), while others will make no sense in a Tree-based system (e.g., the **Stack**), as we no longer rely on backtracking to explore alternative solutions, by exploring them in parallel. As we are aiming at parallel resolution, some of these memory areas must be private to and- and or-boxes.

5.8.1 Memory model

STEAM divides the memory in two separate areas:

- the **Queue** or **Tree memory** - This is the area where the And-Or Tree nodes and their associated data structures are created. The name Queue is used because this area also serves as a queue from where work is fetched.
- the **Code Area** - This area is where the STEAM code is located. Implementation-wise, it can have the abstract STEAM-IL code to be interpreted, or point directly to the executable code segment (if we chose to compile the STEAM-IL to native code).

The Queue naming is merely an abstraction, as we may look at the Tree as if it was a queue of goals to be solved. Using this perspective, we can maintain Prolog's left-to-right evaluation semantics, by assuring that goals at the front of the Queue are preferred over goals at the end of the Queue. This also works for solution ordering, as the first solutions (by the Prolog standard evaluation order) will be located closer to the front of the Queue and later solutions will be closer to the end of the Queue.

Figures 5.15 to 5.17 show how STEAM's Queue abstraction works.

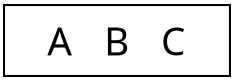

Figure 5.15: STEAM Queue, before expansion.

Parallelism implies some independence between memory areas and data structures that we want to process in parallel. With that in mind, STEAM allows nodes to have a

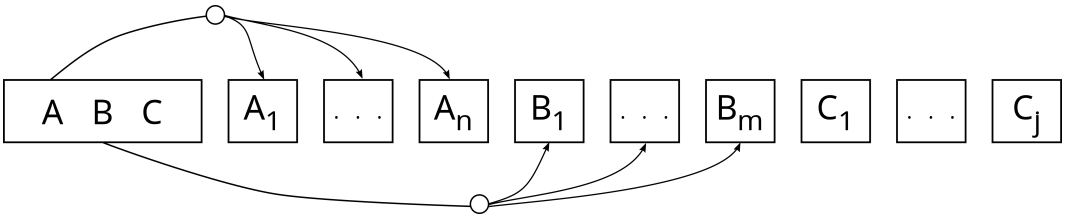

Figure 5.16: STEAM Queue, after expansion of A, B and C.

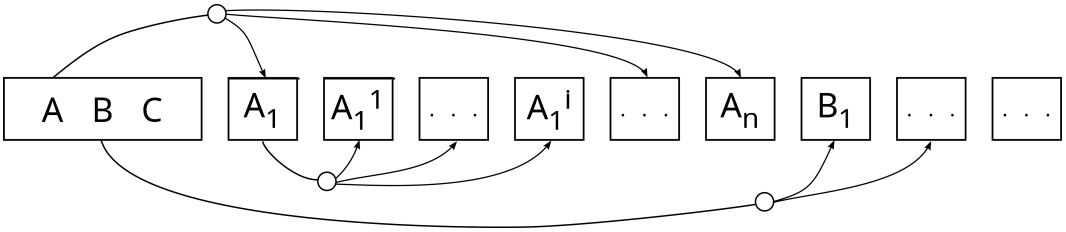

Figure 5.17: STEAM Queue, after expansion of A_1.

private Heap, where compound terms will be built. Like WAM-based implementations that use the Stack to perform the Push-Down-List (PDL) role[3], STEAM lets the private Heap play the PDL role in unification operations.

Other memory needs come directly from the STEAM-IL code generated by `wam2steam`.

An and-box is the place where variable bindings are stored. Variables are seen by STEAM as in the WAM, as a set of machine registers that will have memory addresses of:

- Other variables (by unification); or
- Addresses on the Heap.

This means that and-boxes must have a Heap and a set of registers that mimic the WAM registers (X-registers and Y-registers, as well as other Heap-registers. Stack-registers aren't needed in STEAM). Pointers to it's parent and children or-boxes are also needed, in order to allow access to external data structures.

We can see a detailed graphical representation of an STEAM and-box in figure 5.18

Furthermore, Y-registers are shared between every goal inside an and-box, while X-registers are to be private to each called goal.

[3]This was previously discussed in chapter 3

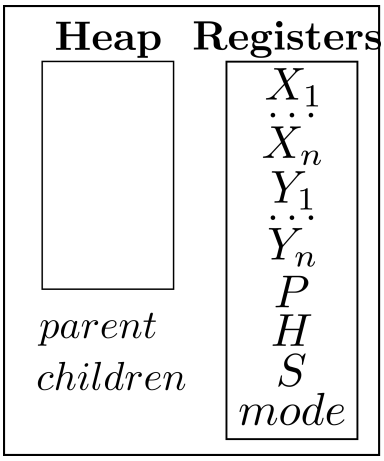

Figure 5.18: STEAM and-box internal layout

In the EAM specification, Warren distinguishes between local and external variables: variables are *local* to an and-box if they are defined in that and-box, and are *external* otherwise. This distinction is necessary in order to prevent conflicting bindings on the same variable.

STEAM takes a different approach: in STEAM, all variables are local and can therefore be bound at any node of the Tree. This means we must have a method to detect conflicting bindings *a posteriori*, between parent and children and-boxes, in order to allow us to prune invalid solutions from the Tree. This method is presented in detail in section 5.8.2.

Or-boxes in the EAM serve only as placeholders for and-boxes that represent alternative clauses to a predicate. STEAM looks at or-boxes as the source for the alternative bindings that each alternative clause can produce. In order to have those alternative bindings, STEAM enhances or-boxes with ***Binding Vectors***. A Binding Vector is similar to a Binding Array [War87b] in the sense that it keeps alternate bindings for variables.

In STEAM, each or-box has one Binding Vector with as many slots as alternatives under that or-box. When an alternative and-box finishes its computation, the bindings produced are stored in the Binding Vector at the slot corresponding to that alternative.

As each alternative can produce more than one binding for a variable, each slot in a Binding Vector can be a pointer to the or-box's grandchildren Binding Vectors. Further, each alternative may produce bindings for more than one variable, which makes the

Binding Vector more like a multi-dimensional array of Binding Vectors, but we'll keep the nomenclature for brevity purposes. Also, as an or-box is always linked to a specific goal inside an and-box, for context purposes sometimes we may refer to a Binding Vector as the Binding Vector for the goal, without losing the initial meaning.

Figure 5.19 shows the internal layout of a STEAM or-box.

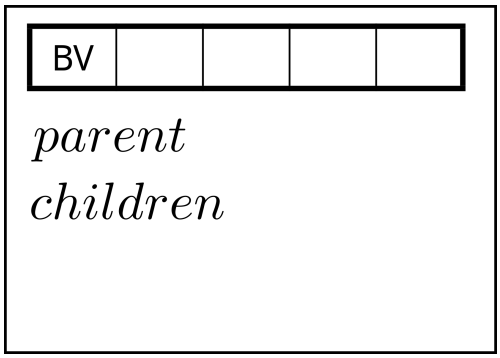

Figure 5.19: STEAM or-box internal layout

In the next section, we'll discuss how unification between conjunctive goals is performed, by using the Binding Vector.

5.8.2 Binding and unification

The main part of a computation is dealing with variable binding and unification. In contrast with the EAM proposal, the STEAM allows all variables to be bound, by allowing each (and-) box to have it's own set of variables.

In a standard, sequential Prolog engine, solutions are found by performing a left-to-right evaluation of the goals in a query. This left-to-right evaluation consists mostly in finding bindings for the goal at the left, then finding compatible bindings for the second goal, an so forth. In STEAM, we try to leverage parallelism by allowing this search for compatible bindings to occur in parallel. We call this method ***Parallel Unification***.

Parallel Unification between two goals A and B, appearing in sequence inside an and-box, occurs when *sufficient*[4] bindings are available on the Binding Vector for goal A (represented as BV_A) and at least one binding is available on the Binding Vector for

[4]The notion of sufficient bindings can be set by a *threshold* on the number of bindings or on time spent searching for them

goal B (represented as BV_B), consisting in unifying all the bindings in BV_A with all the bindings in BV_B. In particular, if all the possible bindings for A and B are available, we can keep the unified Binding Vectors $BV_A \cdot BV_B$ in the parent and-box, augmenting the formerly existing constraints σ. Formally,

$$\bigwedge_X^{\sigma} \left(\bigvee^{BV_A} (...), \bigvee^{BV_B} (...), G_1, ..., G_n \right) \xrightarrow[par\_unify]{} \bigwedge_X^{\sigma(BV_A \cdot BV_B)} (G_1, ..., G_n) \tag{5.11}$$

An immediate drawback of using Binding Vectors that can grow in dimensions is the exponential growth that it can achieve when we don't proceed with caution. However, STEAM has to store only the possible bindings that don't fail and, even then, sometimes only the first alternative. For instance, in the query `p(X), q(X), !, ...` as soon as one binding of X succeeds for both `p/1` and `q/1`, the whole Binding Vector can be discarded. The annotations provided by the STEAM-IL make possible to know in advance that only one alternative is to be found. In the case of search for all solutions, the size of the vector can be constrained, either by looking for a subset of solutions at a time, either by performing the splitting rule. A successful implementation of parallel unification has already been achieved in Smith and Hickey's Multi-SLD [SH94].

5.8.3 Dealing with extra-logical predicates

One of the difficulties in parallelizing Prolog execution has to do with extra-logical predicates (see section 2.3), which impose order to the execution of the computation. Many existing parallel implementations only consider Prolog programs without these kind of predicates, as they generally mean loss of parallelism, imposing a sequential evaluation of the goal that contains the extra-logical predicate and, therefore, the goals for which this goal is descendent in the Tree.

In STEAM we have the extra-logical predicates previously annotated in the STEAM-IL, as well as the goals that call these predicates. This allows to proceed carefully with the branch that contains the extra-logical predicate, while allowing sibling branches to continue executing in parallel.

The extra-logical predicate will be executed as soon as one solution is found for the goals at its left. The goals at the right can be processed in two different ways, depending of the nature of the extra-logical predicate:

1. If the predicate has side-effects (`assert` or `retract`), all the goals at the right must wait for the side-effects to execute if they depend of the side effect;
2. If the predicate doesn't have side-effects, the goals at the right side may execute in parallel, if there are sufficient resources to process them.

In the first case, we may not be able to detect if the predicate's side-effects affect the goals, and we must chose one of two options:

1. Let the goals execute in parallel normally, possibly having to re-execute them later, if the side-effects would alter the results in some form;
2. Suspend all the goals at the right, until the side-effects are performed.

In [GHC93], Gupta discusses some advantages of recomputation in the presence of extra-logical predicates. STEAM leaves this choice open for the implementation.

5.9 Resolution strategy

Although STEAM is a model intended for parallel execution, at some point choices have to be made in order to keep resource usage (i.e., memory and cpu(s)) below the reasonable limits. If resources were unlimited, one would desire to expand the whole tree and try to execute all the branches simultaneously, in parallel. As limits always exist, in order to solve an arbitrary Prolog query, when a decision has to be made in order to chose which goal to execute, STEAM uses the following strategy:

1. **Solve determinate goals first** - If there are determinate goals in the query, solve them first. Determinate goals can be immediately promoted, regardless of the order of execution. Moreover, these goals' search spaces are frequently smaller in nature. By solving determinate goals first, STEAM reduces the search space, while possibly constraining the other goals in the query.

2. **Solve facts before rules** - The search space for facts is strictly "horizontal" in terms of the EAM: a fact is always unfolded (by the expansion rule) into an or-box with one and-box for each alternative clause, and all of those and-boxes will be leafs in the Tree, as none of them can be further expanded. Also, facts are frequently helpful in constraining the search space.

3. **Process goals inside an and-box left-to-right** - We want to preserve the sequential Prolog semantics, which means giving preference to goals on the left over goals on the right. As in a sequential Prolog engine, the goals on the left will constrain the search space for the following goals.

STEAM leaves up to the runtime to follow this exact order of preference, or change it as needed, perhaps relying upon annotations from the STEAM-IL code. Nevertheless, when dealing with parallel execution, we should always try to find sets of solutions instead of trying to find all solutions at a time. This order of resolution is meant to preserve resources when we have, for instance, a limited number of parallel workers, by choosing which branches of the Tree to send the workers to.

5.10 Concluding remarks

This chapter presented the details about STEAM, a model that extends the EAM to allow parallel execution of Prolog programs while keeping the standard semantics. STEAM allows the exploitation of both and-parallelism and or-parallelism, while taking care of the special cases brought by extra-logical predicates.

The Binding Vector and Parallel Unification are the basis for STEAM to find solutions to a goal, by performing a join between different goals' bindings when they are available. While comparing all possible solutions in a multi-dimensional matrix would seem impracticable when single-processor machines were the default, nowadays we have at our disposal hardware whose main purpose is doing this kind of work in parallel. However, STEAM also allows the splitting of the Tree, in order to allow resolution in hardware that has not mass-parallelism built-in. The model can always be improved, in order to provide increased performance or decreased memory usage. Also, STEAM is wittingly open, in order to accommodate specific adaptations to different parallel

programming models.

In the next chapter we'll discuss those adaptations to a specific parallel programming model, the PGAS (Partitioned Global Address Space) model.

6

Design for STEAM on a PGAS model

As the single-processor performance evolves, it becomes more and more difficult to increase the processing power of a single chip, either by physical reasons (e.g. heat vs size) or economic reasons (the hardware becomes too expensive to produce in a profitable way), so it becomes likely that single-processor performance will reach its limit someday.

The DSM (Distributed Shared Memory) programming model establishes a layer of abstraction where local and remote memory addresses are seen as living in the same address space. As a further extension tho the standard DSM model, the PGAS model defines partitions over that shared memory, allowing to use the notion of *affinity*, which

enables the distributed algorithm to perform operations over the data that is local, thus running the algorithm where the data is located, preventing the overhead of moving the data to the physical machine where the code is running.

In a PGAS programming model, we can generally use a SPMD approach, which let us run the same algorithm over the same set of data, which is exactly what we need in order to perform Parallel Unification.

As previously discussed, STEAM allows the exploitation of both and- and or-parallelism, relying on Parallel Unification of bindings provided by conjunctive goals in the same and-box. In the next section we'll discuss how Parallel Unification can be performed in a PGAS programming model.

6.1 The PGAS programming model

In what concerns distributed programming models, the PGAS model provides a higher level abstraction over the benefits of other well-known programming models. By partitioning the memory space, we have access to exploiting locality as in the Message Passing model, but without the need to do explicit message passing. By allowing threads to access non-local data transparently, we have the benefits of the Shared Memory model, while still having the ability to prefer local addresses. One can argue that this transparency comes with a price to pay in performance, but the frameworks that support the PGAS model are being improved at a fast rate. As the PGAS frameworks get more and more mature, they present as a way of enhancing programmers productivity when dealing with High Performance Computing (HPC) projects.

UPC and X10 are two languages/frameworks that are actively being the target of research and improvements, and were the ones we chose to test our ideas. We tried to keep close to the capabilities of both UPC and X10 while developing the STEAM over PGAS model.

UPC is a syntactic extension to the C programming language, allowing the programmer to use the PGAS principles (e.g., shared memory, locality) easily, without having to worry about low-level communication or message passing. X10 is a higher-level language, based on Scala, that extends the PGAS model with asynchronous task-based

parallelism (which makes X10 an APGAS[1] language).

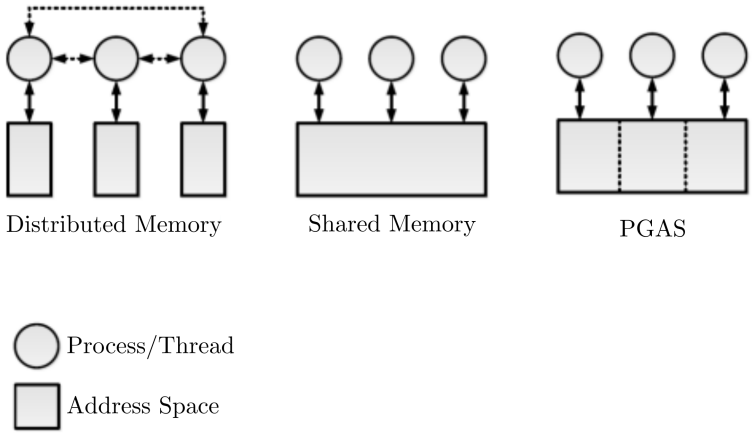

Figure 6.1: Parallel programming models.

6.2 Partitioned STEAM

As previously stated, the PGAS model divides the memory in partitions spread among different threads of execution. Generally, each thread executes the same algorithm, having affinity with their own memory partition. However, it can access the memory partitions of the other threads transparently, even if the other threads are running in a different machine. The ability to exploit data locality is one of the advantages of the PGAS model, but having transparent access to remote data is also very useful.

In a PGAS context, we can define a worker as the algorithm that each thread executes, which means that all the workers must perform exactly the same function (although not necessarily at the same time). In STEAM there are various tasks to be performed that can be split into equal parts, and those are the ones that we will focus into:

- Perform rewrite rules on the Tree;
- Execute calls;
- Unify arguments.

[1] Asynchronous Partitioned Global Address Space

Thus, we define a worker as a thread that performs three different operations: applying rewriting rules, executing calls and unifying. This implies that each worker has to have specific storage to deal with each of those operations.

In what concerns and- and or-parallel work, STEAM-PGAS workers are agnostic, which means that any worker can do either and- or or-parallel work.

In a PGAS system, each worker or thread has it's own memory slot, corresponding to a part of the global memory. This means that the sum of all workers' memory slots represents the whole memory.

STEAM-PGAS uses this partitioning to give each worker an and- or or-box, which we'll further refer as the worker's local box. Nevertheless, each worker has also access to all of the remaining boxes, albeit not local. This allows a worker to send or receive bindings from parent (or grandparent) boxes.

Figure 6.2 shows a graphical representation of the building blocks that give form to a worker.

Each worker will have a private Heap, used to build compound terms and help with unification tasks (doubling as a PDL[2]). Also, there must be space for the current and- or or-box that is being processed.

6.3 Global memory model

As previously discussed, the PGAS memory model partitions the memory in equal parts and "gives" one partition to each thread or worker, which can then take advantage of address locality. In STEAM-PGAS, besides the components described above, each worker has also locality over a part of the global box memory, as well as over a part of the global heap.

The global memory model of STEAM-PGAS consists of the following components:

- The ***global heap*** is where compound terms created in succeeding and-boxes are copied, in order to be available to the higher levels of the Tree. These terms can

[2]The Push Down List, used in the WAM, was introduced in chapter 3

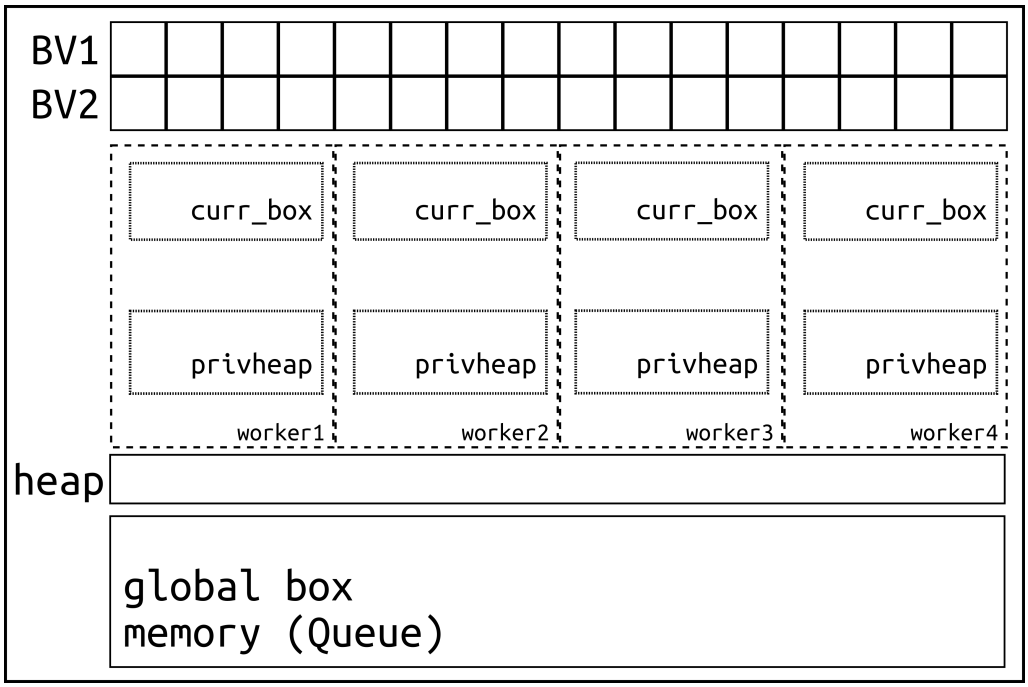

Figure 6.2: Example of a PGAS system with 4 STEAM workers

either be part of the solution or part of the object of parallel unification.

- The ***global binding vectors*** - there are two global binding vectors, where possible bindings for variables are put, in order to allow parallel unification.

- The ***global box memory*** or ***Queue*** is the area where the and- and or-boxes are created at runtime, by the execution of the STEAM-IL code.

6.4 Executing Prolog with STEAM

In this section, we'll describe the process of executing a Prolog program in the STEAM-PGAS model.

The initial configuration consists in a single and-box, corresponding to the initial query. This and-box is created in the global box memory, which, because we are using a PGAS model, will be local to one of the workers. This worker takes the and-box and performs a rewriting rule on the tree, generating (possibly) new boxes, which, in turn, will be local to other (or the same) workers. This process is straightforward, until one of these situations occur:

1. There are no more boxes to execute;
2. All the workers are busy.

In the first scenario, the work is probably done, and a solution (or no solution) was found. In the second scenario, a decision must be made: as there are more boxes than workers, what box will be executed next? The first worker to become available will have to choose a box to work on, according to the following criteria:

1. Prefer boxes that are placed on the left side of the Tree, in order to preserve Prolog left-to-right evaluation semantics;
2. Follow the resolution strategy defined in section 5.9;
3. If there are suspended boxes (waiting for a semaphore), check if the semaphore has reached zero, if so, remove the box from the waiting queue;

If there are available boxes, but none of them resides in the local memory of the worker, the worker can choose a remote box, lock it and copy it to its local memory, in order to work in that box.

As a working example, let's revisit the graph program from the previous chapter, '`graph.pl`' (figure 6.3), representing graphs: edges and paths. The corresponding STEAM-IL code can be seen in figure 6.8. Consider the initial query:

```
?- path(X,Y).
```

```
1 edge(1,2). edge(1,3).
2
3 path(X,Y,[X,Y]) :- edge(X,Y).
4 path(X,Y,[X|T]) :- edge(X,Z), path(Z,Y,T).
```

Figure 6.3: Revisiting the Prolog program (`'graph.pl'`).

the computation starts by creating an and-box with the initial goal `path(X,Y)` (figure 6.4). This and-box will be located in the local area of a specific worker, W_x, which locks the box and performs the expansion rule on it, resulting into one new or-box with two children and-boxes (figure 6.5).

Figure 6.4: Initial and-box for the query `path(X,Y)`.

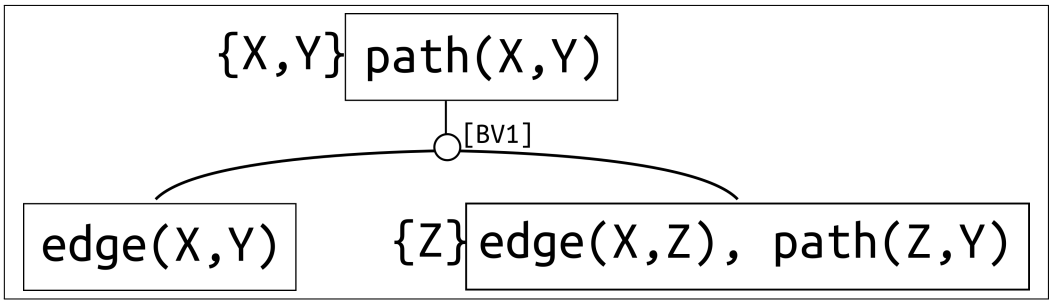

Figure 6.5: Expansion rule applied on the and-box.

The newly created or-box will contain a binding vector `BV1` for the variables `X` and `Y` with two cells, one for each alternative binding produced for each alternative unfolded clause. This vector won't be used in parallel unification, as it's parent goal doesn't have any conjunctive siblings. However, as it represents the leftmost alternative node for the initial and-box, it already contains two solutions to the query.

Suppose we want to keep searching for alternative results.

In figure 6.6, we can see a part of the next step: each of three workers grabs one of the new boxes and processes it. The worker that gets part *a)* binds the variables `X` and `Y`, placing this (possible) bindings in the binding vector `BV2`; the same goes for the worker that gets part *b)*, placing the bindings into `BV3`, however, this binding vector will be used for parallel unification with `BV4`, as they're siblings; The worker with the part *c)* will further expand the Tree, which can be seen in figure **??**.

As soon as both `BV3` and `BV4` have results (either full or hitting the threshold), a worker locks the or-boxes containing those binding vectors and copies the bindings to the global binding vectors. Then, the worker sends an *interrupt*, which makes the other workers (and itself) change their operation mode to parallel unification mode, which will match the bindings for variable `Z` in both vectors (as `Z` is the only common variable to both binding vectors).

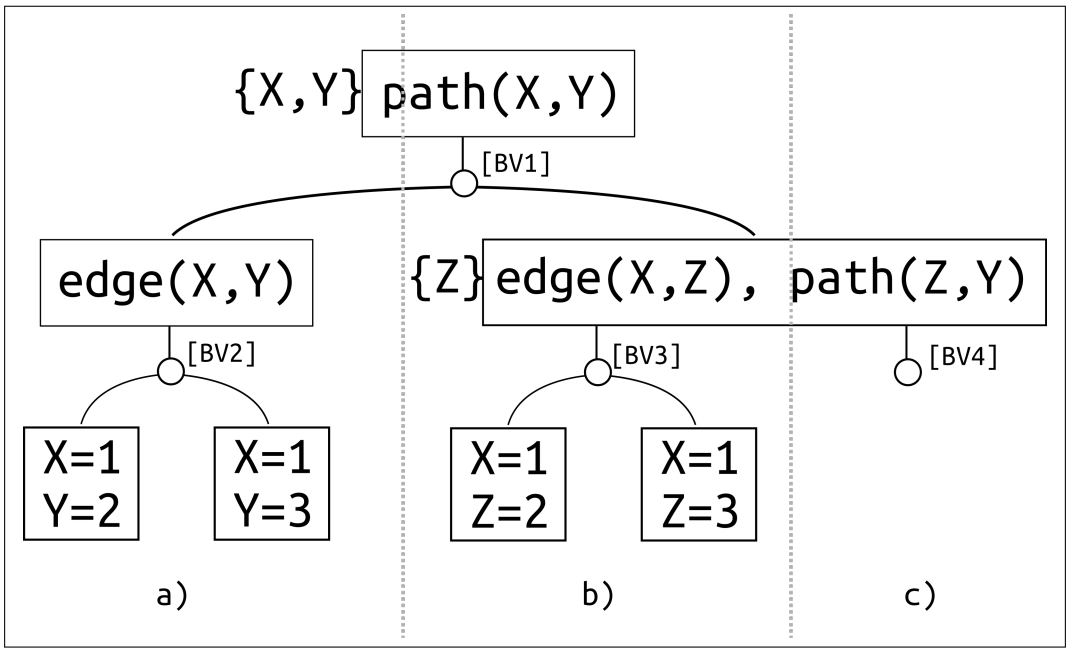

Figure 6.6: Further expansions on the Tree (left part).

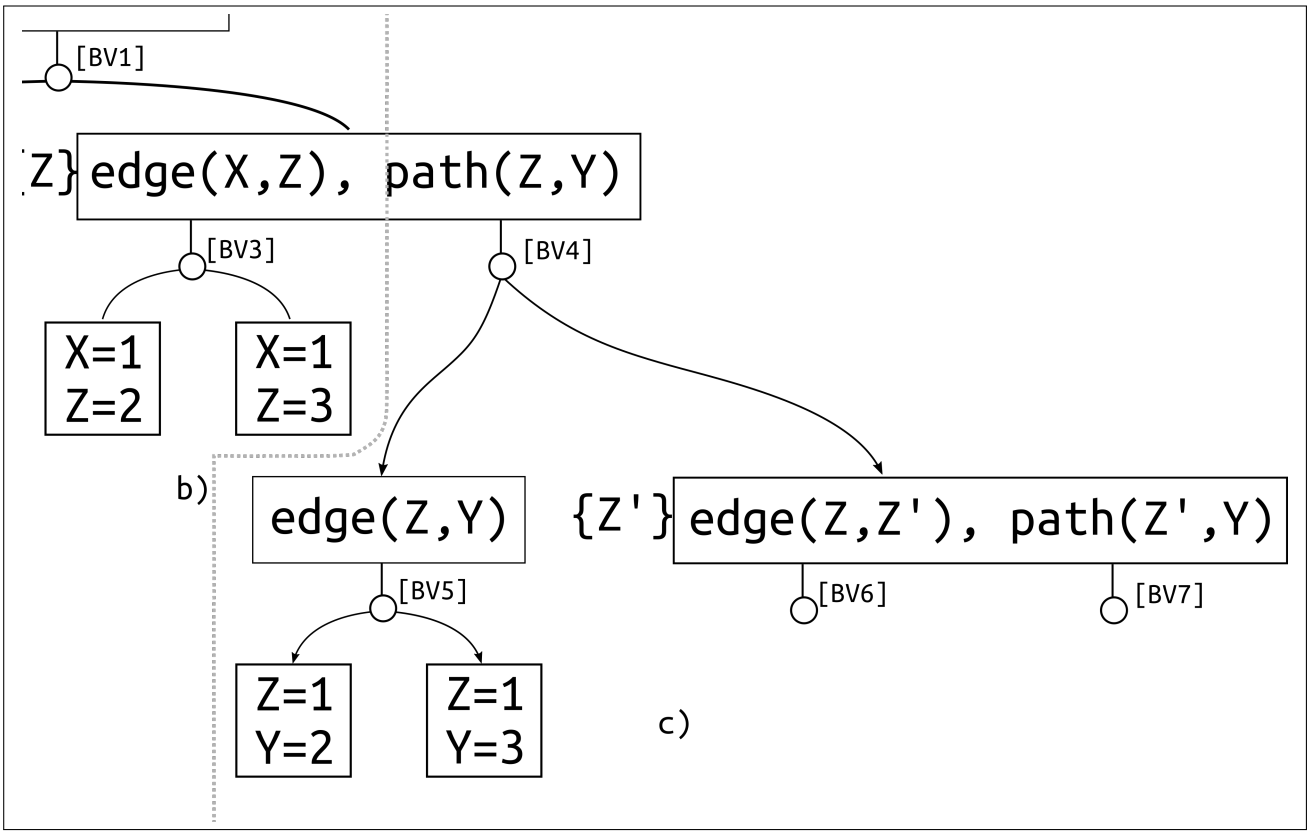

Figure 6.7: Further expansions on the Tree (continued).

```
predicate edge/2:
  switch_on_term L000003, FAIL, L000001, FAIL, FAIL
L000001:
  switch_on_integer 1 L000002
L000002:
  allocate_or (2) L000004 L000006
L000003:
  allocate_or (2) L000004 L000006
L000004:
  get_integer 1, 0
  get_integer 2, 1
  proceed
L000006:
  get_integer 1, 0
  get_integer 3, 1
  proceed

predicate path/3:
  allocate_or (2) L000007 L000009
L000007:
  get_list 2
  unify_local_value x0
  unify_list
  unify_local_value x1
  unify_nil
  call edge/2
  proceed
L000009:
  allocate_and LA1 LA2
LA1:
  get_variable y0, 1
  get_list 2
  unify_local_value x0
  unify_variable y1
  put_variable y2, 1
  call edge/2
  proceed
LA2:
  put_unsafe_value y2, 0
  put_value y0, 1
  put_value y1, 2
  call path/3
  proceed
```

Figure 6.8: STEAM-IL for '`graph.pl`'.

6.5 Parallel Unification

Parallel unification is the task that better fits a PGAS programming model. To exemplify, suppose we have a Prolog program with a large number of clauses for facts `p/1` and `q/1`. Say we have n alternative clauses for `p/1` and m alternative clauses for

`q/1`. For the query `?- p(X), q(X)`, we have to find an `X` that satisfies both `p(X)` and `q(X)`. In a standard, sequential Prolog engine, the system would instantiate `X` with each clause of `p/1` and then try to unify with each clause of `q/1`, resulting in a worst case of $n \times m$ sequential unifications.

With STEAM, parallel unification occurs when the binding vectors for `p/1` and `q/1` have bindings for X, and the workers are interrupted to start testing the compatibility between all of the bindings of the two conjunctive goals.

As soon as we have sufficient solutions for both or-boxes, either by having all the solutions available, either by hitting a predefined threshold on the number of solutions available, a worker detects it and copies both of the binding vectors' bindings to the global binding vectors and then sends an interrupt signal to all the workers, which then switch to parallel unification mode.

The `allocate_or` L_1 `...` L_n instruction will create an or-box with a binding vector with n slots for possible bindings. As soon as the threshold is hit, we can start to perform parallel unification between the binding vector and it's sibling binding vector.

Supposing we have to perform parallel unification between two conjunctive sets of possible bindings:

$$S_1 = \{(X = 1, Y = 2), (X = 1, Y = 3), (X = 3, Y = 4)\}$$

and

$$S_2 = \{(A = 2, X = 1), (A = 0, X = 2), (A = 1, X = 3)\}$$

Figure 6.9 shows how the binding vectors would look like before the workers start performing the parallel unification. As each worker has locality to a part of the global binding vectors, the sets can be distributed homogeneously among the global binding vectors, allowing each worker to process an equal amount of data.

The results of the parallel unification can be stored in a double-bit array, where each comparison sets the first bit to 1 and if there is a match the second bit is also set to 1. In the example provided, we would have a set S_{sol} of valid bindings:

$$S_{sol} = \{(A = 2, X = 1, Y = 2), (A = 1, X = 3, Y = 4)\}$$

Where the solutions would be found by workers *1* and *3*, as worker *2* couldn't match any of the bindings.

Generalizing and comparing this with the sequential alternative, if we have a set of n alternative bindings to unify with another set of m alternative bindings, we can say that we have to perform the same $(n \times m)$ unifications, but the time spent is in the order of $1/w$ of the sequential version, where w is the number of available workers.

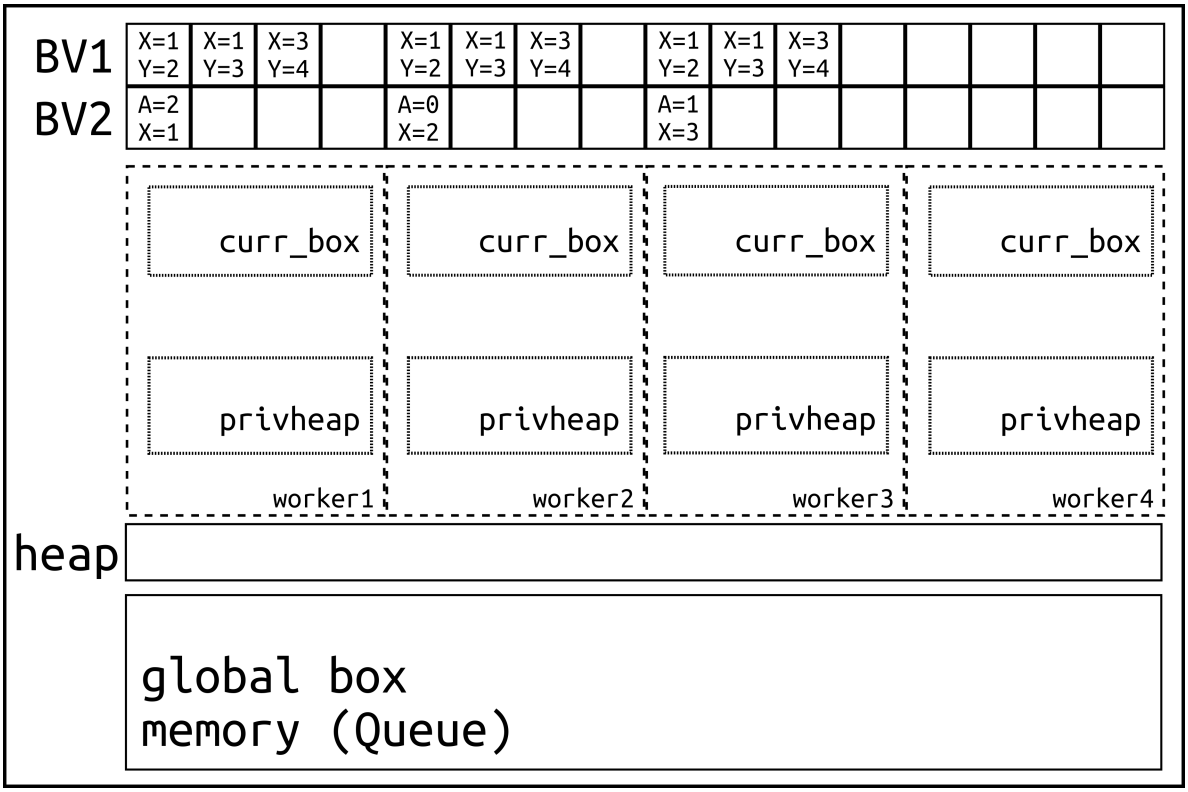

Figure 6.9: Binding vectors for S_1 and S_2.

6.6 Results propagation

As results become available, they have to be propagated up in the Tree. Two different situations may arise:

1. The parent box and the branch with available solutions belong in the same

worker's local memory space;

2. The parent box is in a different worker's memory space from the branch with available solutions.

In the first case, we may just extend the parent's binding vector with the new bindings, as a worker is guaranteed to have privileged access to it's local memory.

In the second case, the worker with the new data must get a lock on the parent box, extend it's binding vector and then release the lock.

6.7 Concluding remarks

This chapter presents a design for implementing STEAM in a PGAS programming model. A prototype implementation is being developed, in order to allow us to compare the results of this design with other parallel and sequential implementations of Prolog.

We strongly believe that the EAM, with the STEAM model in particular, is well suited for taking advantage of the novel distributed programming models, with special emphasis in the PGAS model.

7

Conclusions

This thesis presents the design of STEAM, a parallel system for executing Prolog programs efficiently, by resorting to David H.D. Warren's Extended Andorra Model (EAM) with implicit control and it's further exploration by Ricardo Lopes on the BEAM.

Although recent research efforts into declarative parallelism has been gearing towards CLP, we feel that there's still a space for research into the exploitation of implicit parallelism in Prolog programs. There are still legacy systems written in pure Prolog which would benefit from the possibility of being run in recent parallel hardware.

Albeit some focus have been given to the EAM, we feel that the model hasn't been sufficiently explored and, with the emergence of newer distributed programming models, namely the Partitioned Global Address Space (PGAS), the EAM provides a solid

base for parallel execution of Prolog, and this work shows that exploiting both and- and or-parallelism is feasible through the use of the Extended Andorra Model.

Our main goal was to research into the implicit parallel execution of Prolog programs, without interfering with the programming task. By taking advantage of the implicit parallelism that exists in Prolog, existing, unmodified Prolog programs can be executed in parallel without any effort from the programmer. There are other parallel models which require the modification of the original source code, in order to take advantage of parallelism, and others that fully explore the implicit parallelism of Prolog, by chosing only one of the forms (either and- or or-parallelism, but not both).

STEAM presents a set of novelties over the EAM, namely:

- The compilation of WAM-code to STEAM-code, allowing the runtime to execute the code in an EAM-based context;
- Avoiding suspension of boxes, in order to maximize parallelism;
- Using binding vectors to allow parallel unification.

These features allow the implementation of a Prolog engine over distributed programming paradigms, taking advantage of the new parallel hardware platforms (e.g., GPGPUs, Xeon-Phi, etc.).

By using an approach of compiling WAM code directly to EAM code, STEAM provides a pluggable system to existing Prolog compilers, by offering an alternative compilation scheme, targeted at parallel environments.

7.1 Future work

There are still many improvements to be made over this work. Some we wish to accomplish in the near future are:

- Finalize the implementation of the STEAM-PGAS prototype, in order to allow us to perform benchmarks in multi-core and hybrid machines, as well as to compare results with other parallel and sequential Prolog engines;

- Augment the abstract analysis core of `wam2steam`, in order to obtain more helpful information about the Prolog predicates, to improve execution time and reduce the search space;
- As we are using the `pl2wam` tool from GNU Prolog, we intend to enable the use of GNU Prolog built-ins, albeit some have to be modified to accomodate the new structures.

As always, in this kind of work, the ideas presented here can (and certainly will) be improved during the evolution of the research, either by observing the results found in the implementations, either by contributions from other researchers.

Bibliography

[AA] Paulo André and Salvador Abreu. Producing EAM code from the WAM.

[AA10] Paulo André and Salvador Abreu. Casting of the WAM as an EAM. *arXiv preprint arXiv:1009.3806*, 2010.

[Abr94] Salvador Pinto Abreu. *Improving the Parallel Execution of Logic Programs*. PhD thesis, PhD thesis, Universidade Nova de Lisboa, 1994.

[Abr00] Salvador Abreu. Towards the oar language and computational model. *Implementation Technologies for (Constraint) Logic Programming Languages*, page 89, 2000.

[ACHS88] Karen Appleby, M Carllson, Seif Haridi, and D Sawhlin. Garbarge collection for prolog based on wam. *Communications of the ACM*, 31(6):719–741, 1988.

[AK90] Khayri AM Ali and Roland Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[AK94] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to Or-Parallel Prolog. *International Journal of Parallel Programming*, 19, 1994.

[AkF99] Hassan Aït-kaci and Forêt Des Flambertins. *Warren's Abstract Machine ATUTORIAL RECONSTRUCTION*. 1999.

[AMTW12] Krzysztof Apt, Victor W. Marek, Mirek Truszczynski, and David S. Warren. The Logic Programming Paradigm: A 25-Year Perspective. 2012.

[AP] Salvador Abreu and Luis Moniz Pereira. Towards akl with intelligent pruning.

[AP93] Salvador Abreu and Luís Moniz Pereira. Design for akl with intelligent pruning. In *Extensions of Logic Programming*, pages 3–10. Springer, 1993.

[APC92] Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving backward execution in the andorra family of languages. In *JICSLP*, pages 384–398. Citeseer, 1992.

[AR97] Lourdes Araujo and Jose J. Ruz. A parallel Prolog system for distributed memory. *The Journal of Logic Programming*, 33(1):49–79, 1997.

[Ara97] Lourdes Araujo. Correctness proof of a distributed implementation of Prolog by means of Abstract State Machines. *Journal of Universal Computer Science*, 3(5):568–602, 1997.

[BBP+81] David L Bowen, Lawrence Byrd, Luis M Pereira, Fernando CN Pereira, and David HD Warren. Prolog on the decsystem-10 user's manual. In *Technical Report*. Department of Artificial Intelligence, University of Edinburgh, 1981.

[BCHP96] Francisco Bueno, D. Cabeza, M. Hermenegildo, and German Puebla. Global analysis of standard Prolog programs. *Programming Languages and Systems—ESOP'96*, pages 108–124, 1996.

[BdCD00] Ricardo Bianchini and Inês de Castro Dutra. Parallel logic programming systems on scalable architectures. *Journal of Parallel and Distributed Computing*, 60(7):835–852, 2000.

[BdKH+88] Uri Baron, Jacques Chassin de Kergommeaux, Max Hailperin, Michael Ratcliffe, Philippe Robert, Jean-Claude Syre, and Harald Westphal. The parallel ecrc prolog system pepsys: An overview and evaluation results. In *FGCS*, volume 88, pages 841–850, 1988.

[BDL+88] Ralph Butler, Terry Disz, Ewing L Lusk, Robert Olson, Ross A Overbeek, and Rick L Stevens. Scheduling or-parallelism: An argonne perspective. In *ICLP/SLP*, pages 1590–1605, 1988.

[BG89] Reem Bahgat and Steve Gregory. Pandora: Non-deterministic parallel logic programming. In *ICLP*, pages 471–486, 1989.

[BH92] Francisco Bueno and Manuel V Hermenegildo. An automatic translation scheme from prolog to the andorra kernel language. In *FGCS*, pages 759–769, 1992.

[BJ] Dan Bonachea and Jaein Jeong. Gasnet: A portable high-performance communication layer for global address-space languages.

[BJK+95] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[BLM93] Johan Bevemyr, Thomas Lindgren, and Håkan Millroth. Exploiting recursion-parallelism in prolog. In *PARLE'93 Parallel Architectures and Languages Europe*, pages 279–290. Springer, 1993.

[BLOO86] Ralph Butler, EL Lusk, Robert Olson, and RA Overbeek. Anlwam: A parallel implementation of the warren abstract machine. *Mathematics and Computer Science Division, Argonne National Lab*, 1986.

[BRSW91] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David HD Warren. Flexible scheduling of or-parallelism in aurora: The bristol scheduler. In *Parle'91 Parallel Architectures and Languages Europe*, pages 825–842. Springer, 1991.

[CCF86] Christian Codognet, Philippe Codognet, and Gilberto Filé. A very intelligent backtracking method for Logic Programs. In *ESOP 86*, pages 315–326. Springer, 1986.

[CCS96] V Santos Costa, Manuel Eduardo Correia, and Fernando Silva. Performance of sparse binding arrays for or-parallelism. In *Proceedings of the VIII Brazilian Symposium on Computer Architecture and High Performance Processing–SBAC-PAD*, 1996.

[CD95] Philippe Codognet and Daniel Diaz. WAMCC: Compiling Prolog to C. In *ICLP*, volume 95, pages 317–331, 1995.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp (FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

[CDC+99a] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[CDC+99b] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[CFS94] Takashi Chikayama, Tetsuro Fujise, and Daigo Sekita. A portable and efficient implementation of KL1. In *Programming Language Implementation and Logic Programming*, pages 25–39. Springer Berlin Heidelberg, 1994.

[CG83] Keith L Clark and Steve Gregory. *Parlog: A parallel logic programming language*. Imperial College of Science and Technology Department of Computing, 1983.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[Cha13] Brad Chamberlain. Chapel: A next-generation pgas language. In *UW Applied Math 483/583 Lecture*, 2013.

[Cia92] Paolo Ciancarini. Parallel programming with logic languages: A survey. *Computer Languages*, 17(4):213–239, 1992.

[CKPR73] A Colmeraner, Henri Kanoui, Robert Pasero, and Philippe Roussel. Un systeme de communication homme-machine en francais. Luminy, 1973.

[CLT+10] Li Chen, Lei Liu, Shenglin Tang, Lei Huang, Zheng Jing, Shixiong Xu, Dingfei Zhang, and Baojiang Shou. Unified parallel c for gpu clusters: Language extensions and compiler implementation. In *Languages and Compilers for Parallel Computing*, pages 151–165. Springer, 2010.

[CM12] Mats Carlsson and Per Mildner. Sicstus Prolog-the First 25 Years. *Theory Pract. Log. Program.*, 12(1-2):35–66, January 2012.

[Co05] U. P. C. Consortium and others. UPC language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.

[Con12] John S. Conery. Parallel execution of logic programs. 2012.

[Cos] Vıtor Santos Costa. Parallelism and Implementation Technology for Logic Programming Languages.

[Cos99] Vítor Santos Costa. Optimising bytecode emulation for Prolog. *Principles and Practice of Declarative Programming*, pages 261–277, 1999.

[CR96] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

[CRD12] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[CRS00] Vỳtor Santos Costa, Ricardo Rocha, and Fernando Silva. Novel models for or-parallel logic programs: A performance analysis. In *Euro-Par 2000 Parallel Processing*, pages 744–753. Springer Berlin Heidelberg, 2000.

[CSP88] Chien Chen, Ashok Singhal, and Yale N Patt. PUP: An Architecture to Exploit Parallel Unification in Prolog. Technical report, DTIC Document, 1988.

[CSW95] W Chen, T Swift, and DS Warren. Efficient implementation of general logical queries. *J. Logic Prog*, 1995.

[CWY91a] Vitor Santos Costa, David HD Warren, and Rong Yang. Andorra I: a parallel Prolog system that transparently exploits both And-and or-parallelism. *ACM SIGPLAN Notices*, 26(7):83–93, 1991.

[CWY91b] Vitor Santos Costa, David HD Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. 1991.

[CWY91c] Vitor Santos Costa, David HD Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *ICLP*, volume 91, pages 443–456, 1991.

[CZSS11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[DAC12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[DC93] Daniel Diaz and Philippe Codognet. A minimal extension of the wam for clp (fd). In *ICLP*, pages 774–790, 1993.

[DC00a] Daniel Diaz and Philippe Codognet. GNU Prolog: beyond compiling Prolog to C. *Practical Aspects of Declarative Languages*, pages 81–92, 2000.

[DC00b] Daniel Diaz and Philippe Codognet. The GNU prolog system and its implementation. In *Proceedings of the 2000 ACM symposium on Applied computing-Volume 2*, pages 728–732. ACM, 2000.

[DC01] Daniel Diaz and Philippe Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001:2001, 2001.

[dCD94] Inês de Castro Dutra. Strategies for scheduling and-and or-work in parallel logic programming systems. *Department of Computer Science, University of Bristol*, 1994.

[dCD95] Inês de Castro Dutra. *Distributing And-and Or-Work in the Andorra-I Parallel Logic Programming System*. PhD thesis, Citeseer, 1995.

[DE98] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[Deb94] Saumya K Debray. Implementing logic programming systems: The quiche-eating approach. In *Implementations of Logic Programming Systems*, pages 65–75. Springer, 1994.

[DET96a] Bart Demoen, Geert Engels, and Paul Tarau. Segment order preserving copying garbage collection for wam based prolog. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 380–386. ACM, 1996.

[DET96b] Bart Demoen, Geert Engels, and Paul Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 380–386. ACM, 1996.

[DHS00] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[DK89] Arthur Delcher and Simon Kasif. Some results on the complexity of exploiting data dependency in parallel logic programs. *The Journal of Logic Programming*, 6(3):229–241, 1989.

[DKC94] Jacques Chassin De Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys (CSUR)*, 26(3):295–336, 1994.

[DN00] Bart Demoen and Phuong-Lan Nguyen. On the impact of argument passing on the performance of the WAM and B-Prolog. 2000.

[DN08] Bart Demoen and Phuong-Lan Nguyen. Environment Reuse in the WAM. *Logic Programming*, pages 698–702, 2008.

[DP85] Alvin M. Despain and Yale N. Patt. Aquarius-A High Performance Computing System for Symbolic/Numeric Applications. In *COMPCON*, pages 376–382, 1985.

[EC98] Jesper Eskilson and Mats Carlsson. SICStus MT—a multithreaded execution environment for SICStus Prolog. In *Principles of Declarative Programming*, pages 36–53. Springer, 1998.

[Gal85] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.

[GC94] Gopal Gupta and Vítor Santos Costa. Optimal implementation of and-or parallel prolog. *Future Generation Computer Systems*, 10(1):71–92, 1994.

[GC96] Gopal Gupta and Vitor Santos Costa. Cuts and side-effects in and-or parallel prolog. *The Journal of logic programming*, 27(1):45–71, 1996.

[GCP94] Gopal Gupta, Vítor Santos Costa, and Enrico Pontelli. Shared Paged Binding Array: A Universal Datastructure for Parallel Logic Programming. *UNIVERSITY OF OREGON*, 1994.

[GG01] Hai-Feng Guo and Gopal Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Logic Programming*, pages 181–196. Springer, 2001.

[GH91] Gopal Gupta and Manuel Hermenegildo. Ace: and/or-parallel copying-based execution of logic programs. In *Parallel Execution of Logic Programs*, pages 146–158. Springer, 1991.

[GHC93] Gopal Gupta, Manuel V Hermenegildo, and Vítor Santos Costa. And-or parallel prolog: A recomputation based approach. *New Generation Computing*, 11(3-4):297–321, 1993.

[GLLO85] John Gabriel, Tim Lindholm, EL Lusk, and Ross A Overbeek. *A tutorial on the warren abstract machine for computational logic*. Argonne National Laboratory, 1985.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[GW91] Gopal Gupta and David HD Warren. An interpreter for the extended andorra model. *Preliminary report, Department of Computer Science, University of Bristol*, 1991.

[Hay89] Ralph Haygood. A prolog benchmark suite for aquarius. 1989.

[Hay94] Ralph Clarke Haygood. Native Code Compilation in SICStus Prolog. In *ICLP*, pages 190–204, 1994.

[HBC+12] Manuel V Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F Morales, and German Puebla. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[Hem94] Rolf Hempel. The mpi standard for message passing. In *High-Performance Computing and Networking*, pages 247–252. Springer, 1994.

[Her86a] Manuel V. Hermenegildo. Abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel. 1986.

[Her86b] Manuel V. Hermenegildo. An abstract machine for restricted AND-parallel execution of logic programs. In *Third International Conference on Logic Programming*, pages 25–39. Springer Berlin Heidelberg, 1986.

[HG91] Manuel V. Hermenegildo and KJ Greene. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9(3-4):233–256, 1991.

[HH89] Zhiyi Hwang and Shouren Hu. A compiling approach for exploiting And-parallelism in parallel logic programming systems. *PARLE'89 Parallel Architectures and Languages Europe*, pages 335–345, 1989.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computational model. *SICS Research Report*, 1990.

[HJP92] Seif Haridi, Sverker Janson, and Catuscia Palamidessi. Structural operational semantics for AKL. *Future Generation Computer Systems*, 8(4):409–421, 1992.

[Jan94a] Sverker Janson. AKL-a multiparadigm programming language. *Uppsala University, SICS*, 1994.

[Jan94b] Sverker Janson. *AKL, a multiparadigm programming language: based on a concurrent constraint framework*. Number 19 in Uppsala theses in computing science. Computing Science Dept., Uppsala University ; Swedish Institute of Computer Science, Uppsala : Kista, Sweden, 1994.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. *SICS Research Report*, 1991.

[JM92] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of the andorra kernel language. In *SICS research report*. Swedish Institute of Computer Science, 1992.

[Kan86] Paris C. Kanellakis. Logic programming and parallel complexity. *ICDT'86*, pages 1–30, 1986.

[KB95] Andreas Krall and Thomas Berger. Incremental Global Compilation of Prolog with the Vienna Abstract Machine. In *ICLP*, pages 333–347, 1995.

[Kog90] Peter M Kogge. *The architecture of symbolic computers*. McGraw-Hill, Inc., 1990.

[Kra94] Andreas Krall. Implementation techniques for Prolog. In *WLP*, pages 1–15, 1994.

[Kra96] Andreas Krall. The vienna abstract machine. *The Journal of logic programming*, 29(1):85–106, 1996.

[KWm12] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LBD+90] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David HD Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, and others. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2-3):243–271, 1990.

[LC00] Ricardo Lopes and V Santos Costa. Memory Management for the BEAM. In *CL2000 First Workshop on Memory Management in Logic Programs*, 2000.

[LCC99] Ricardo Lopes, Vítor Santos Costa, and V'itor Santos Costa. The beam: A first eam implementation. In *In 1999 Joint Conference on Declarative Programming (APPIA-GULP-PRODE) proceedings, L'Aquila*. Citeseer, 1999.

[LCS04] Ricardo Lopes, Vítor Santos Costa, and Fernando Silva. Pruning in the extended Andorra model. In *Practical Aspects of Declarative Languages*, pages 120–134. Springer, 2004.

[LH93] M. R. Levy and R. N. Horspool. Translating Prolog to C: a WAM-based approach. In *Proceedings of the Compulog Network Area Meeting on Programming Languages*, 1993.

[Lin94] Thomas Lindgren. A Continuation-Passing Style for Prolog. In *ILPS*, volume 94, pages 603–617, 1994.

[Llo12] John W. Lloyd. Foundations of logic programming. 2012.

[LLV+13] Miao Luo, Mingzhe Li, Akshay Venkatesh, Xiaoyi Lu, and Dhabaleswar K DK Panda. Upc on mic: early experiences with native and symmetric modes. In *7th International Conference on PGAS Programming Models*, page 198, 2013.

[LO89] Ewing L. Lusk and Ross A. Overbeek, editors. *Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio, USA, October 16-20, 1989. 2 Volumes*. MIT Press, 1989.

[Lop96] Ricardo Nuno Lopes. Execução de Prolog com Alto Desempenho. *Master's thesis, Universidade do Minho-Departamento de Inform atica*, 1996.

[Lop01] Ricardo Lopes. *An Implementation of the Extended Andorra Model*. PhD thesis, PhD thesis, Universidade do Porto, 2001.

[LSCA00] Ricardo Lopes, Fernando Silva, Vıtor Santos Costa, and Salvador Abreu. The RAINBOW: Towards a Parallel Beam. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages, CL*, pages 38–54. Citeseer, 2000.

[MA95] Johan Montelius and Khayri AM Ali. An and/or-parallel implementation of akl. *New Generation Computing*, 14(1):31–52, 1995.

[MA96] Johan Montelius and Khayri AM Ali. An And/Or-parallel implementation of AKL. *New Generation Computing*, 14(1):31–52, 1996.

[MAD13] Rui Machado, Salvador Abreu, and Daniel Diaz. Parallel Performance of Declarative Programming Using a PGAS Model. In *Practical Aspects of Declarative Languages*, pages 244–260. Springer, 2013.

[MCH04] J. Morales, Manuel Carro, and Manuel Hermenegildo. Improved compilation of Prolog to C using moded types and determinism information. *Practical Aspects of Declarative Languages*, pages 86–103, 2004.

[MD89] André Mariën and Bart Demoen. On the management of choicepoint and environment frames in the WAM. In Lusk and Overbeek [LO89], pages 1030–1047.

[MD93] Remco Moolenaar and Bart Demoen. A parallel implementation for akl. In *Progamming Language Implementation and Logic Programming*, pages 246–261. Springer, 1993.

[Mil93] Johan Bevemyr Thomas Lindgren Hakan Millroth. Reform prolog: the language and its implementation. In *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, page 283. MIT Press, 1993.

[Mon94] Johan Montelius. Penny, a Parallel Implementation of AKL. In *Workshop on Design and Impl. of Parallel Logic Programming Systems*, pages 4–8, 1994.

[Mon97] Johan Montelius. The Computation Model. 1997.

[MZ88] Tadao Murata and Du Zhang. A predicate-transition net model for parallel interpretation of logic programs. *Software Engineering, IEEE Transactions on*, 14(4):481–497, 1988.

[NCS01] Henrik Nässén, Mats Carlsson, and Konstantinos Sagonas. Instruction Merging and Specialization in the SICStus Prolog Virtual Machine. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 49–60, New York, NY, USA, 2001. ACM.

[Neu95] Ulrich Neumerkel. Continuation Prolog: A new intermediary language for WAM and BinWAM code generation. In *Post-ILPS'95 Workshop on Implementation of Logic Programming Languages. F16G*, 1995.

[New87] Michael O Newton. A High Performance Implementation of Prolog. 1987.

[NWSDSM13] Sebastian Nanz, Sam West, Kaue Soares Da Silveira, and Bertrand Meyer. Benchmarking usability and performance of multicore languages. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 183–192. IEEE, 2013.

[Nä01] Henrik Nässén. Optimizing the SICStus Prolog virtual machine instruction set. *SICS Research Report*, 2001.

[Pac97] Peter S. Pacheco. Parallel programming with MPI. 1997.

[PGH95] Enrico Pontelli, Gopal Gupta, and Manuel Hermenegildo. &ACE: a high-performance parallel Prolog system. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 564–571. IEEE, 1995.

[PGT+96] Enrico Pontelli, Gopal Gupta, Dongxing Tang, Manuel Carro, and Manuel V. Hermenegildo. Improving the efficiency of nondeterministic independent and-parallel systems. *Computer Languages*, 22(2):115–142, 1996.

[Phe08] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[PN91] Doug Palmer and Lee Naish. Nua-prolog: An extension to the wam for parallel andorra. In *ICLP*, pages 429–442, 1991.

[RD92] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *Computer*, 25(1):54–68, 1992.

[Riv97] CarIos B. Rivera. *DWAM: An Or-parallel Wam Implementation of Prolog*. PhD thesis, University of Regina, 1997.

[RSB+89] Kotagiri Ramamohanarao, John Shepherd, Isaac Balbin, Graeme Port, Lee Naish, James Thom, Justin Zobel, and Philip Dart. The nu-prolog deductive database system. In *Prolog and databases: implementations and new directions*, pages 212–250. Halsted Press, 1989.

[RSC99] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. Yapor: an or-parallel prolog system based on environment copying. *Progress in Artificial Intelligence*, pages 178–192, 1999.

[RSC00] Ricardo Rocha, Fernando Silva, and Vıtor Santos Costa. A tabling engine for the Yap Prolog system. In *Proceedings of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'00), La Habana, Cuba*, 2000.

[RSC05] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1-2):161–205, 2005.

[RSSC00] Ricardo Rocha, Fernando Silva, and V. Santos Costa. YapTab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, volume 7787, 2000.

[SB99] Luis Moura Silva and Rajkumar Buyya. Parallel programming models and paradigms. *High Performance Cluster Computing: Architectures and Systems*, 2:4–27, 1999.

[SBP+11] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification, 2011.

[SD08] Tom Schrijvers and Bart Demoen. Uniting the prolog community. In *Logic Programming*, pages 7–8. Springer, 2008.

[SH94] Donald A Smith and Timothy J Hickey. Multi-sld resolution. In *Logic Programming and Automated Reasoning*, pages 260–274. Springer, 1994.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, 1989.

[SHC95] Zoltan Somogyi, Fergus J. Henderson, and Thomas Charles Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996.

[She96] Kish Shen. Overview of DASWAM: exploitation of dependent AND-parallelism. *The Journal of logic programming*, 29(1):245–293, 1996.

[Smi94] Donald A Smith. Why multi-sld beats sld (even on a uniprocessor). In *Programming Language Implementation and Logic Programming*, pages 40–56. Springer, 1994.

[Smi96] Donald A Smith. Multilog and data or-parallelism. *The Journal of logic programming*, 29(1):195–244, 1996.

[Smo95] Gert Smolka. The Oz programming model. *Computer science today*, pages 324–343, 1995.

[SS93] Dan Sahlin and Thomas Sjöland. Demonstration: static analysis of AKL. *Static Analysis*, pages 282–283, 1993.

[SSW93] Konstantinos Sagonas, Terrance Swift, and David Scott Warren. Xsb: An overview of its use and implementation. *SUNY at Stony Brook*, 1993.

[SSW+03] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, G. de Castro, and Rui F. Marques. The XSB System. *Web page: http://xsb. sourceforge. net*, 2003.

[SW94] Terrance Swift and David S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. In *In Proceedings of the Symposium on Logic Programming*, pages 633–654, 1994.

[SW10] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *CoRR*, abs/1012.5123, 2010.

[SW12] Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[Tar] Paul Tarau. Architecture and Implementation Aspects of the Lean Prolog System.

[Tar92] Paul Tarau. BinProlog: a continuation passing style Prolog engine. In *Programming Language Implementation and Logic Programming*, pages 479–480. Springer Berlin Heidelberg, 1992.

[Tar06] Paul Tarau. The Jinni 2004 Prolog Compiler: a High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. *BinNet Corp*, 2006.

[Tar11] Paul Tarau. Coordination and concurrency in multi-engine prolog. In *Coordination Models and Languages*, pages 157–171. Springer Berlin Heidelberg, 2011.

[TF86] Akikazu Takeuchi and Koichi Furukawa. Parallel logic programming languages. In *Third International Conference on Logic Programming*, pages 242–254. Springer Berlin Heidelberg, 1986.

[Tic89a] Evan Tick. Comparing two parallel logic-programming architectures. *Software, IEEE*, 6(4):71–80, 1989.

[Tic89b] Evan Tick. A Performance Comparison of AND-and OR-Parallel Logic Programming Architectures. In *ICLP*, pages 452–467, 1989.

[Tic90] Evan Tick. Compile-time granularity analysis for parallel logic programming languages. *New Generation Computing*, 7(2-3):325–337, 1990.

[Tin88] Peter A. Tinker. Performance of an OR-parallel logic programming system. *International Journal of Parallel Programming*, 17(1):59–92, 1988.

[UC90] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[Ued86] Kazunori Ueda. *Guarded horn clauses*. Springer, 1986.

[Ued89] Kazunori Ueda. Parallelism in logic programming. In *In Information Processing 89, Proc. IFIP 11th World Computer Congress, North-Holland/IFIP*, 1989.

[Var94] Konstantinos Varsamos. Automatic Transformation of Deterministic Prolog Programs to KL1. 1994.

[VR84] Peter Lodewijk Van Roy. *Can logic programming execute as fast as imperative programming?* PhD thesis, University of California at Berkeley, 1984.

[vR92] P. van Roy. Aquarius Prolog. *IEEE Computer*, 1992.

[VR94] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *The Journal of Logic Programming*, 19:385–441, 1994.

[VXDRS91] André Véron, Jiyang Xu, S. A. Delgado-Rannauro, and K. Schuerman. Virtual memory support for OR-parallel logic programming systems. In *Parle'91 Parallel Architectures and Languages Europe*, pages 843–860. Springer Berlin Heidelberg, 1991.

[W+81] David HD Warren et al. *Higher-order extensions to Prolog-are they needed*. Department of Artificial Intelligence, University of Edinburgh, 1981.

[War78] David HD Warren. Applied logic: its use and implementation as a programming tool. 1978.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, October 1983.

[War87a] David HD Warren. Or-parallel execution models of prolog. In *TAPSOFT'87*, pages 243–259. Springer, 1987.

[War87b] David HD Warren. The sri model for or-parallel execution of prolog: Abstract design and implementation issues. In *SLP*, volume 87, pages 92–102, 1987.

[War88] David HD Warren. The andorra model. In *Gigalips Project workshop. U. of Manchester*, volume 1124, 1988.

[War89] David HD Warren. Extended andorra model. In *PEPMA Project workshop, University of Bristol*, 1989.

[War90] David HD Warren. The extended andorra model with implicit control. In *Parallel Logic Programming Workshop, Box*, volume 1263, 1990.

[War95] David S Warren. Programming in tabled prolog. 1995.

[War99] David S. Warren. *Programming in Tabled Prolog*. 1999.

[Wie03] Jan Wielemaker. Native preemptive threads in SWI-Prolog. In *Logic Programming*, pages 331–345. Springer, 2003.

[Wie14] Jan Wielemaker. SWI-Prolog version 7 extensions. In *Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014*, page 109, 2014.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[Zho00] Neng-Fa Zhou. Garbage collection in B-Prolog. In *In Proceedings of the First Workshop on Memory Management in Logic Programming Implementations*, 2000.

[Zho07] Neng-Fa Zhou. A register-free abstract prolog machine with jumbo instructions. In *ICLP*, volume 4670, pages 455–457, 2007.

[Zho12] Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.

[ZSYY00] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. In *Practical Aspects of Declarative Languages*, pages 109–123. Springer, 2000.