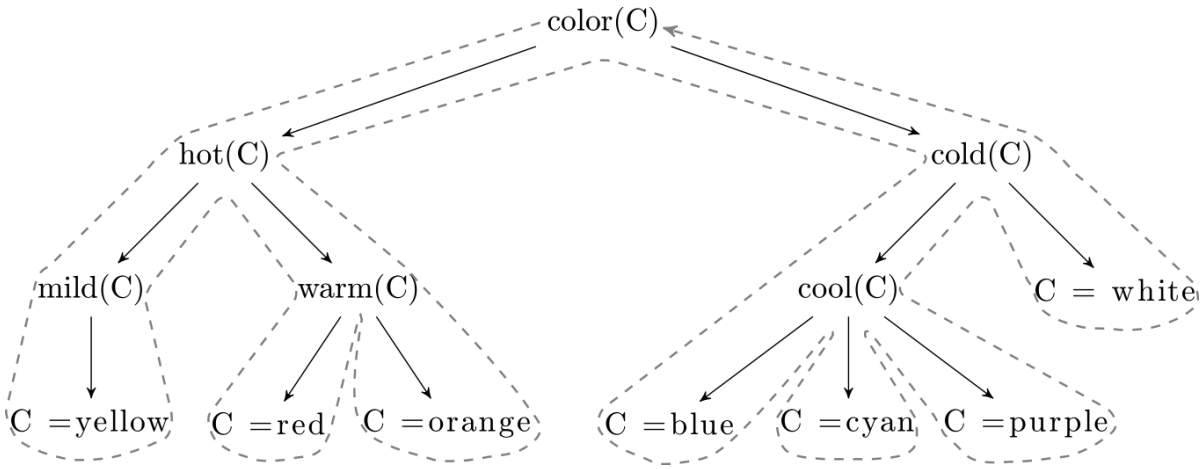

High-Level Constructors for Solution Searching in Or-Parallel Prolog Systems

João André Martins da Silva
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2014

Orientador
Ricardo Jorge Gomes Lopes da Rocha, Professor Auxiliar, Faculdade de Ciências

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

João André Martins da Silva

High-Level Constructors for Solution Searching in Or-Parallel Prolog Systems

Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Mestre
em Engenharia de Redes e Sistemas Informáticos

Advisor: Prof. Ricardo Jorge Gomes Lopes da Rocha

Departamento de Ciência de Computadores
Faculdade de Ciências
Setembro de 2014

Dedicado aos meus pais e irmã.

Abstract

This aim of this work is to design and implement strategies that can improve the performance of logic programs when searching for particular solutions in an Or-Parallel Prolog system. Prolog is a first-order logic predicate language that belongs to the declarative family of programming languages, emphasizing data declaration and use. Or-Parallelism is a form of implicit parallelism that can be applied to Prolog programs, in order to allow the parallel execution of several clauses that match a Prolog goal.

With the availability of specific strategies that can improve the system's performance when searching for solutions in parallel, we expect to: (i) allow for long-running programs, such as those used for simulations, to take less time to execute; (ii) make it possible to execute new programs that deal with larger amounts of data, that would otherwise be too slow to run; (iii) generate more interest in Prolog, which might lead to further research. In particular, our implementation was done on top of the YAP Prolog system, a well-known and established system which makes it possible for users to get all of these benefits.

The strategies proposed in this work have a very important concept at the core: make relatively small changes to the YAP's engine codebase in order to allow them to be easily ported to other implementations of Prolog, and make them available to the user, by using high-level constructors that transparently increase the speedups obtained without forcing the user to make complex source code changes.

Resumo

O objetivo deste trabalho é o desenho e implementação de estratégias que permitam melhorar o desempenho de programas lógicos na procura de soluções particulares num sistema Prolog Ou-Paralelo. O Prolog é uma linguagem de predicados lógicos de primeira ordem que pertence à família das linguagens de programação declarativas, que enfatizam a declaração de dados e o seu uso. O Paralelismo-Ou é uma forma de paralelismo implícito que pode ser aplicada a programas Prolog, de forma a permitir a execução em paralelo de várias cláusulas que correspondem a um golo de Prolog.

Com a disponibilidade de estratégias específicas que permitam melhorar o desempenho do sistema, ao procurar soluções em paralelo, esperamos: (i) permitir que programas de longa duração, como os que são usados em simulações, demorem menos tempo a executar; (ii) tornar possível a execução de novos programas que tratam de maiores quantidades de dados, que seriam de outra forma demasiado lentos a executar; (iii) gerar mais interesse em Prolog, o que leva a mais investigação. Em particular, a nossa implementação foi feita sobre o sistema YAP Prolog, um sistema reconhecido e estabelecido, o que possibilita aos utilizadores obter todos estes benefícios.

As estratégias propostas neste trabalho têm por base um conceito central muito importante: fazer modificações relativamente pequenas ao código fonte do sistema YAP de forma a possibilitar que sejam facilmente convertidas para outras implementações de Prolog, e disponibilizá-las ao utilizador, usando construtores de alto nível que, de forma transparente, aumentam os *speedups* obtidos sem forçar o utilizador a fazer alterações complexas ao seu código.

Acknowledgements

First and foremost I would like to express my gratitude toward my supervisor, Professor Ricardo Rocha for his advice. His suggestions and clear way of explaining details of YAP Prolog system (YAP) proved to be very useful throughout the development of this work.

I acknowledge project LEAP (PTDC/EIA-CCO/112158/2009) and Fundação da Ciência e Tecnologia for the research grant that I received during the development of this work.

I would also like to thank my friends David Aparício, Joana Côrte-Real and Joaquim Silva with whom I spent six interesting weeks in Austin, where they provided me with some ideas and suggestions for this work.

Finally, I want to thank my parents and sister for their unconditional support. Many times, they were the first ones to hear about difficulties I faced developing this work and, simply by listening to me, helped me discover solutions to them.

Contents

List of Tables

List of Figures

List of Acronyms

CLP Constraint Logic Programming.

DCG Definite Clause Grammar.

LP Logic Programming.

SLD resolution Selective Linear Definite clause resolution.

WAM Warren's Abstract Machine.

YAP YAP Prolog system.

1 Introduction

Nowadays, immense amounts of data are generated every day. Even though humans have the ability to detect patterns, we are unable to keep up with this increase of information. We did, however, create machines that are able to process this data and give us the big picture in a way that is simpler to understand. The problem is that only a few people have the knowledge needed to control its execution and develop new programs even if they only perform slightly different tasks on tiny variations of the same problems.

One tool that makes this process easier is *Logic Programming (LP)*. LP belongs to the class of declarative programming languages, which means that programmers expose the problem they are trying to solve in a straightforward manner with the added benefit that the results they get back also use the same language. Arguably, an important language that belongs to this class is Prolog. This language presents features that make complex tasks such as machine learning and language processing easy to program due to its high-level declarative style.

Users do not simply want an expressive language, though. They also want to be able to perform their tasks in a manner that is as fast as possible. For a few years now, being fast implies the use of the multiprocessing features of modern CPUs. Explicit parallelism is, however, a road that is full of challenges that programmers used to writing sequential programs do not want to think about. Once again, Prolog simplifies the task for the programmer: due to its non-determinism, tasks can be run in a partially arbitrary order that can be implicitly parallelized by the runtime system, without altering their semantics.

Another important advantage of using Prolog when compared to other languages is that, having deep roots in mathematical concepts, it is possible to prove the correctness of the programs written in it.

1.1 Motivation

Throughout the last decade multicore computers became standard, with their inclusion in x86, the most widely used PC architecture. For this reason, powerful multicore machines are widely available and are becoming increasingly affordable. Researchers and scientists make use of these machines for studies, experiments and developing models in fields that can go from physics to medicine.

Taking into account the previously mentioned features of Prolog and the availability of powerful multicore computers, it was natural for researchers to make use of a combination of Prolog and multicore computers. In applications with very large data sets, such as those that often arise in the previously mentioned fields, any decrease in the execution times can not only save users time but also allow them to solve bigger problems.

This thesis' objective is to improve performance of the YapOr Or-Parallel Engine, namely by increasing the speedups of parallel constructors that find specific solutions to a query. Our contribution is a set of strategies designed to reduce the time and resources used to solve problems in parallel Prolog and their implementation in the YAP Prolog system [CRD12], a state-of-the-art implementation of Prolog, including several extensions and developed at the University of Porto. We explain the rationale for the changes and how they alter the overall execution of a Prolog program, presenting the modules where these changes take place. These strategies are particularly useful for long-running programs where a reduction of a few percent of the overall time may translate into minutes, hours or even days. The implementation of each of these strategies changes a relatively small portion of the source code, on purpose, in order to make them easy to understand and be re-implemented in other systems and also to simplify the process of combining several of them together.

We hope that this work makes LP a more viable and attractive alternative to other languages for the processing of tasks where large amounts of data must be processed.

1.2 Thesis Outline

This document is structured in 6 chapters and a brief description of each is provided next.

- **Chapter 1: Introduction.** The current chapter.

- **Chapter 2: Background.** This chapter presents relevant background information on logic programming and the Prolog language. Initially, we provide a comparison of LP with other programming paradigms and introduce the reader to the Prolog programming language. The next sections are dedicated to the Warren's Abstract Machine (WAM), a virtual machine used by several modern Prolog implementations, and to the YAAM, YAP's version of the WAM. Finally, we present the YapOr Or-Parallel Engine.

- **Chapter 3: The Implementation of Or-Parallel Solution Searching Constructors.** This chapter describes the methods we used to optimize the YapOr Or-Parallel Engine and describes the several strategies that we have implemented to improve the performance of YapOr.

- **Chapter 4: Experimental Results.** This chapter presents measurements made to assess the system's performance and discusses the results obtained.

- **Chapter 5: Conclusions and Future Work.** This chapter concludes the thesis and presents suggestions for further work.

- **Annex A: Benchmarking Code.** Contains the code used for benchmarking.

2 Background

This chapter introduces the reader to the Logic Programming model of computation and to the Prolog programming language. Next, we explore the inner features of the YAP system. We first take a closer look at the *Warren's Abstract Machine (WAM)* followed by a brief explanation of some optimizations that YAP applies to its own variant of the WAM. Lastly, we describe the Or-Parallel component of YAP, the YapOr engine, upon which this work is based.

2.1 Logic Programming

Logic Programming is considered one of the four main programming paradigms, that were introduced in the following chronological order:

1) **Imperative Programming**, introduced in 1957 by Fortran [Bac78];

2) **Functional Programming**, the basis for 1958's Lisp [McC60];

3) **Object-Oriented Programming**, first used by Simula in 1962 [ND62];

4) **Logic Programming**, introduced by the Absys logic system in 1967 [Elc90].

LP, like functional programming, emphasizes data declaration and the modeling of the relations between data. This paradigm is completely different from that of imperative and object-oriented programming, where the programmer is expected to deal with the low-level details of the underlying representation of data structures and algorithms.

Another similarity to functional programming is that the execution of programs can be modeled as graph reduction. Unlike functional programming, however, LP represents code as a set of *logic clauses* and *facts* (or *axioms*). A logic clause is considered true whenever it only depends on facts or other logic clauses that are themselves true, a recursive process called Selective Linear Definite clause resolution (or SLD resolution).

2.2 Prolog

Prolog (from the french ***pro****grammation en* ***log****ique*) is a programming language designed by Alain Colmerauer as a result of his research related to the development of a *natural language question-answering system* [Kow88]. This system was originally very different from what we call Prolog today - primarily because it did not make use of predicate logic, as modern Prolog does. The idea of using first-order logic to program the system was first used in the Planner programming language. It was the combination of syntax and semantics based on first-order logic and SLD resolution that gave birth to the modern Prolog that we use today.

The first large system to be built with Prolog was the aforementioned natural language system [CR93]. Nowadays, Prolog is still a fairly popular language for natural language processing, which is a good explanation for the availability of syntactic support for parsers in Prolog, that culminated in the *Definite Clause Grammar (DCG)* formalism, in 1980 [PW80].

After Prolog evolved from a system to process natural languages to a programming language, there was still one problem that presented a barrier to its wide adoption: the language implementation was slow. This was mostly due to the fact that the language was interpreted. This changed when, in 1975, David H. D. Warren implemented a compiled version of Prolog for the DEC-10 [War77]. This implementation made use of a virtual machine devised by Warren, which was a precursor of the Warren's Abstract Machine (or WAM) [War83], on which most current implementations of Prolog are still based.

2.2.1 Variants of Prolog

This language served as a basis for several variants and extensions. Some of the most important additions to Prolog were support for *Constraint Logic Programming (CLP)* [WNS97] and for *tabling* [RRS+95] - a mechanism that allows predicates which have a solution, but would never terminate under SLD resolution, to be solved by Prolog through the use of *memoization*.

Additionally Prolog was also used as the basis for other languages, such as Datalog [AHV95], for databases, and Erlang [AVW92], a language designed to build applications where high availability is required, which was originally a Prolog implementation and still uses a virtual machine conceptually close to the WAM.

Some current Prolog variants, besides using virtual machines inspired by the WAM,

have efficient parallel implementations that make use of the implicit parallelism present in LP, making their performance even better and further reducing the gap in execution times, when compared to native binaries in other programming languages.

2.2.2 Prolog by Example

The following example presents the syntax of modern Prolog:

```
1  color(C) :- hot(C).
2  color(C) :- cold(C).
3
4  hot(C) :- mild(C).
5  hot(C) :- warm(C).
6
7  cold(C) :- cool(C).
8  cold(white).
9
10 mild(yellow).
11
12 warm(red).
13 warm(orange).
14
15 cool(blue).
16 cool(cyan).
17 cool(purple).
```

Figure 2.1: A Prolog example where several color-related predicates are defined.

For this code to run, the user needs to present a query. For this example we will use the query `?- color(C)`, which means that we want to determine the names of all the colors. Because of the semantics of Prolog, that make use of SLD resolution, the execution will match each predicate by the order defined in the source code from top to bottom. We use the notation `predicate/arity` when referring a predicate so that, for instance, the `color` predicate of arity 1 is represented as `color/1`. The first clause of `color/1` says that if `hot(C)` is true then `color(C)` is true, meaning that `C` is a color. To know if `hot(C)` is true, we need to know if `mild(C)` or `warm(C)` are true (third and fourth clauses). For example, `mild(C)` is true if `C` is `yellow` (seventh clause). Because `C` is a variable, it unifies with `yellow`, so it is a color.

The execution then goes back to `hot/1` because there are no more alternatives for `mild(C)`. We then choose the next alternative for `hot/1`, `warm/1` and continue this process, generating the search tree presented in Figure 2.2.

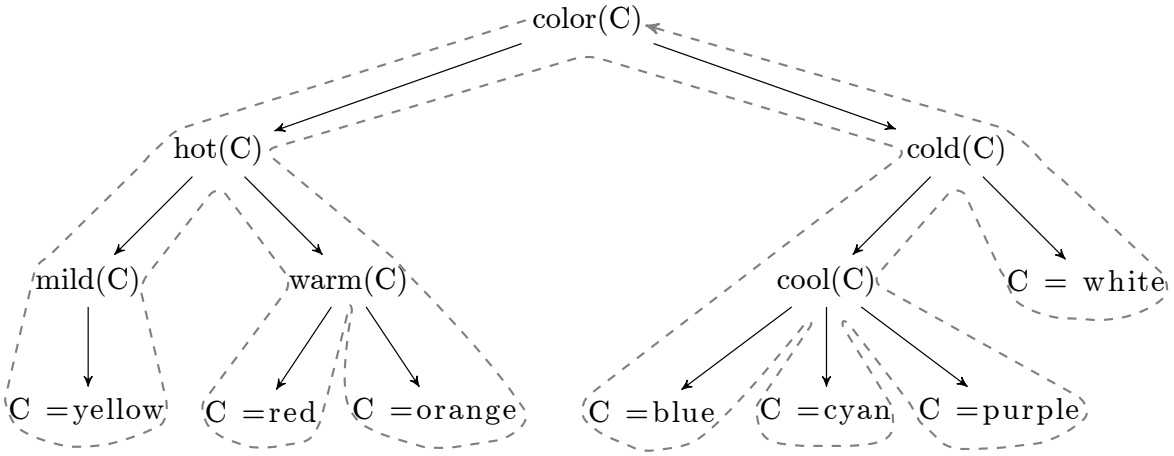

Figure 2.2: Implicit search tree generated for the query `?- color(C).`

2.3 Warren's Abstract Machine

The WAM was introduced by David H. D. Warren in [War83], based on his previous experience with the implementation of the DEC-10 Prolog. This virtual machine is now used, with small changes, by several popular Prolog variants, such as SWI-Prolog [Wie14], GNU Prolog [DAC12] and YAP [CRD12]. The key idea of the WAM is that a Prolog program can be converted to a bytecode representation for a virtual machine, based on creation and lookup of small data structures that represent terms, the most basic units of Prolog programs.

A program that is compiled for the WAM becomes a sequence of instructions that create data structures and match them. Failure to match a data structure is equivalent to a failure to meet one of the conditions of a Prolog predicate.

2.3.1 Memory Areas

The WAM was originally designed to be implemented in hardware and, as such, has an execution model that is reminiscent of a computer. As presented in [AK91], the WAM possesses five main memory areas:

- **Code**, that contains the WAM translation of the Prolog program;
- **Heap**, where compound terms are stored;
- **Trail**, that keeps track of the variable being bound;
- **Stack**, used for environments and choice points;

- **PDL (Push Down List)**, used for the unification process.

The code area is used to store the WAM instructions that are generated when the Prolog program is translated. It is roughly equivalent to the code section of an executable.

The heap is the memory area where the previously mentioned pattern matching occurs and is comparable to the similarly-named heap of programming languages such as Java and C. The WAM has instructions that create and match variables, structures and values in this area.

The trail, coupled with the choice points that are stored in the stack form the mechanism that the WAM uses for backtracking. When a predicate fails, the bindings for the variables found on the trail are undone and the execution continues at the preceding choice, if any, of the current choice point. This mechanism has some similarities to the exception handling facilities of C# and Java.

The stack also contains environments, which are similar to the activation frames of other programming languages, containing the local variables of the program.

The PDL keeps track of the creation and use of variables, for unification, being comparable to the C stack.

2.3.2 Registers

To have efficient access to the various memory areas, the WAM also defines several registers which perform functions similar to those of CPU registers:

- **P**, the current instruction;
- **CP**, the next instruction, if the previous call is successful;
- **H**, the top of the heap;
- **HB**, the top of the heap at the most recent choice point;
- **TR**, the top of the trail;
- **S**, which is used for matching subterms of a compound term;
- **Mode**, containing the current execution mode, *read* or *write*;
- **A**, the top of the stack;

- **E**, a pointer to the current environment;
- **B**, a pointer to the most recent choice point;
- $\mathbf{A_1}$, ..., $\mathbf{A_n}$, the argument registers;
- $\mathbf{X_1}$, ..., $\mathbf{X_n}$, the temporary variable registers.

2.3.3 Instructions

The WAM defines an instructions set that can be divided into four groups:

- **Choice point instructions** which allocate and deallocate memory for choice points, making backtracking possible;
- **Control instructions** that are responsible for managing the calls of subgoals in predicates and manage the environments;
- **Indexing instructions** that, based on the type and value of the first argument of a call, jump to specialized code for that call.
- **Unification instructions** which perform the matching and unification of variables and other types.

The instructions that deal with Prolog data structures operate using the notion of modes, of which there are two:

- **Write mode**, the mode where data structures representing Prolog terms are created;
- **Read mode**, where these data structures are matched.

2.4 YAAM

The YAAM is YAP's optimized version of the WAM. It is based on many of the same ideas of WAM's design, but makes a considerable number of changes, mainly to the memory layout of the WAM.

One of the key insights made by the YAP developers is that memory allocated on the heap is long-lived, which means that a compact representation of terms is ideal. As such, the YAAM uses tag bits to represent the five main types of data presented in

[AK91]: integers, atoms, applications, pairs and references. It additionally supports one more type introduced by YAP: extensions, which allow the use of additional data types not present in strictly-standard Prolog implementations. Examples of additional data types currently available in YAP are multiple-precison integers and arrays of floats and integers.

The WAM does not specify the layout of the memory areas. A naive, but obvious, way would be to allocate memory for each area independently of the others, dynamically adjusting each one as needed. The YAAM, leaves the code area independent but compresses the other memory areas into the following scheme:

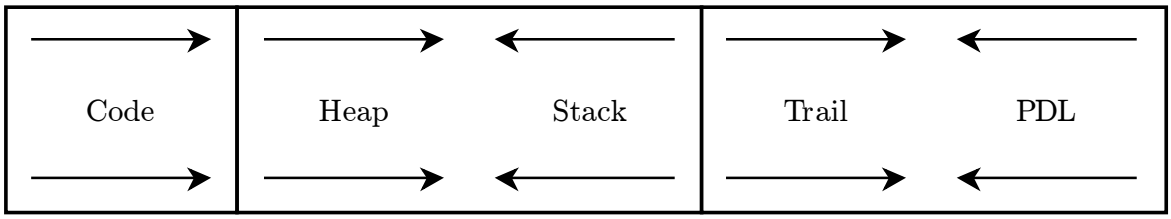

Figure 2.3: YAP's memory area layout.

This scheme presents a some advantages:

- Allocation of a single large memory area, which is faster than the allocation of several small memory areas;
- The layout allows allocation operations to be pointer movements, rather than reallocating and copying memory as in the naive way;
- Detecting overflow of the memory areas can be done by a simple pointer comparison.

A problem with this scheme is that, when the memory needs to be expanded, all of the memory areas must be copied rather than just the area that needs more memory.

2.4.1 Just-in-Time Indexing

Traditionally, Prolog implementations that make use of the WAM index terms by the first argument, making unification faster by only trying alternatives where the type of

the first argument of the query is compatible with the first argument of the matching predicate.

For very large datasets, the developers of YAP tested the difference in performance caused by reordering the arguments and reached the conclusion that it could change performance significantly [CRD12].

The solution developed was a dynamic indexing mechanism, called just-in-time indexing (or JITI). This mechanism adds an instruction to the WAM, `index_pred`, that performs the task of indexing the predicate. Although this mechanism may generate a large amount of indexing code, it generally does not do so [CRD12].

2.5 The YapOr Or-Parallel Engine

With the advent of multicore computers, programs need to be able to divide tasks into smaller subtasks that are executed by each core, in parallel.

There are two main ways in which one can parallelize the execution of a Prolog program implicitly. These are known as *And-Parallelism*, where several subgoals in a clause are executed simultaneously and the failure of any of them results in failure of the goal, and *Or-Parallelism*, where several clauses that match a goal run in parallel in order to find multiple solutions faster.

YapOr is an extension of the YAP system that enables it to make use of the multiple cores that are available on current CPUs. As the name suggests, it exploits the Or-Parallelism inherent to Prolog programs to make solution discovery faster.

Similarly to the Muse system [AK90], the YapOr system uses *environment copying*, where a worker that makes a work request copies the environment of the worker that accepts that request. This is done incrementally, that is, the receiver positions itself above the sender and only copies the portion of memory that is different from its own. Unlike standard Yap, YapOr disallows resizing of the memory areas, which may be a problem for long-running applications that use large amounts of memory. To alleviate this problem, the user can change the size of the areas at compile and/or startup-time.

Since YapOr deals with Or-Parallelism, it needs to track which alternatives of a choice point were given to each worker. To do so, each YapOr worker maintains a linked list of *Or-Frames* to store that information.

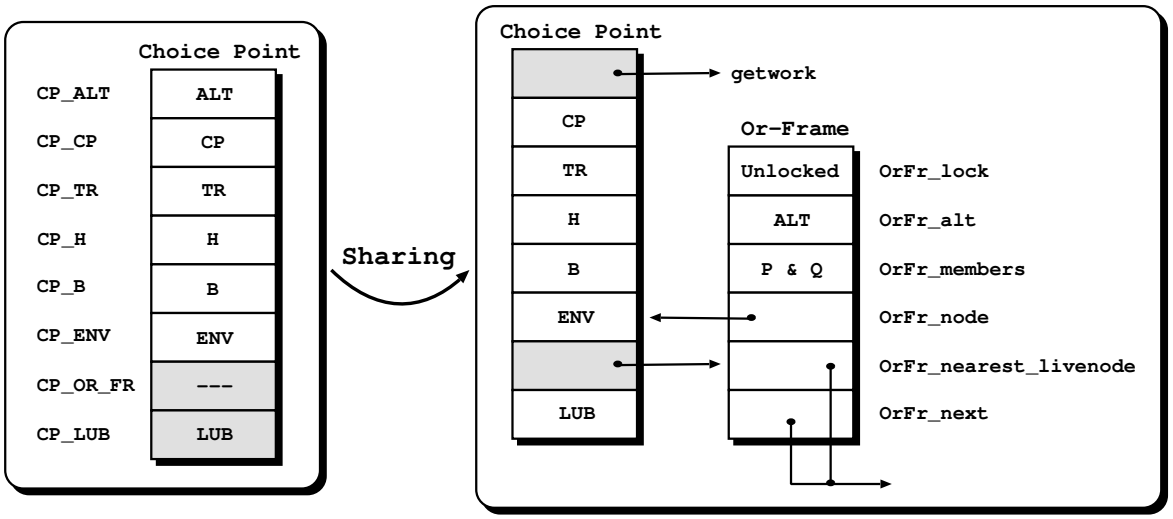

Figure 2.4: The relationship between choice points and Or-Frames (courtesy of Ricardo Rocha, from [Roc01]).

As shown in Figure 2.4, choice points are coupled with Or-Frames; choice points store information relative to the different alternatives that may be taken, mainly in the `CP_ALT`, `CP_CP`, `CP_TR`, `CP_H`, `CP_B` and `CP_ENV`, while Or-Frames maintain information about the different workers that may search for solutions within their associated choice point, mediating access to the choice point with the `OrFr_lock` field and maintaining a bitmap of workers, `OrFr_members`. In this manner, the control of the choice of the next alternative to explorer is moved from the choice point, which is private, to the shared Or-Frames. Or-Frames are only allocated when a worker shares a choice point to which the Or-Frame then becomes associated. Private choice points do not have associated Or-Frames since there is no need to keep information about other workers.

An additional change made by YapOr to the YAP system is the introduction of a new WAM instruction called `get_work` that implements the scheduling strategy and is ultimately responsible for the work sharing process [Roc96]. The strategy can be summarized as follows:

- Ask for work to the worker that simultaneously has the highest load and is nearest.
- When a worker shares work, share all its private nodes.
- When unable to get work, backtrack to a point where it is able to.

More precisely, when a worker backtracks to a shared choice point, it first tries to get the next available alternative. Failing that, i.e. if there is no work left in the shared choice point, a worker will move up in the tree until there are other workers below it in order to ask them for work. If, after this process, no work has been shared, the worker moves up to a better position to be able to get work. The execution ends if all workers reach the top root choice point and are unable to get work from each other.

3 Implementation of Constructors

In this chapter, we present the different strategies we tested to improve the execution of Or-Parallel computations in YapOr. We firstly present the method used to determine what portions of YapOr would benefit the most from optimization. Then, we explain the changes made and the reasoning behind each.

3.1 Methodology

YapOr enables a Prolog programmer to use Or-Parallelism within an application in two main ways:

- `parallel_findall/3`, which runs a query in parallel, retrieving a list of all solutions to that query
- `parallel_findfirst/3`, which executes a query and stops after finding the first, leftmost solution to that query.

Our goal is to increase the speedups we obtain using these parallel primitives, relative to the current version of YapOr. In order to do that, we must first be able to determine where the potential for improvement is. With that in mind, we created a simple synthetic benchmark to test YapOr's `parallel_findall/3`.

The benchmark, presented in Figure 3.1, receives a list of 0s and 1s as input (first argument of `find/4`), and creates all possible lists of 0s and 1s with size smaller or equal to that of the input list until it recreates that list (second argument of `find/4`). This generates a balanced search tree, and takes an exponentially increasing amount of time as the size of the list increases. The choice of a benchmark that takes an exponential amount of time was made in order to be able to very easily test different time scales, from milliseconds to several seconds.

```
1  append([], X, X).
2  append([X|Y], Z, [X|W]) :- append(Y, Z, W).
3
4  options(0).
5  options(1).
6
7  find(In, Out) :- length(In, Len), find(In, [], Len, Out).
8
9  find(ToFind, SoFar, Left, [Item|End]) :-
10     Left > 0, options(Item), append(SoFar, [Item], Next),
11     NextLeft is Left - 1, find(ToFind, Next, NextLeft, End).
12 find(ToFind, ToFind, _, []).
13
14 % A couple of list examples
15 lists(List, Len) :- mkList(Len, 0, List).
16 lists([0,0,0|End], Len) :- mkList(Len - 3, 1, End).
17
18 mkList(0, _, []) :- !.
19 mkList(Len, Num, [Num|End]) :- Left is Len - 1, mkList(Left, Num, End).
20
21 go(Len) :-
22     lists(List, Len),
23     time(parallel_findall(Var, find(List, Var), Solutions)),
24     fail.
```

Figure 3.1: A benchmark that takes exponential time to find a list.

The results obtained with our experiments for this benchmark showed that, on the 24-core machine we used for them, YapOr with 24 workers is able to achieve close to linear speedups, of approximately 21. These results show that the implementation is already very optimized and that in order to get better performance we might need a completely different algorithm. We therefore moved our efforts to the implementation of the `parallel_findfirst/3` constructor.

By making a small change to this benchmark, consisting of changing the use of `parallel_findall/3` to `parallel_findfirst/3`, we tested the implementation of the `parallel_findfirst/3` constructor. For an efficient implementation of this predicate, we expect that the execution takes an increasing amount of time as the position of the first solution moves to the right in the search tree. This time should also be proportional to the number of nodes of the search tree that we must traverse until find the solution. Our experiments for this benchmark, which we performed with varying numbers of workers, showed that the existing implementation of `parallel_findfirst/3` reaches the result in a fast manner - that is, it is faster than `parallel_findall/3`

and the execution time does depend on the number of nodes traversed. The time that `parallel_findfirst/3` took to find a solution at the rightmost path of the tree was also similar to that of `parallel_findall/3`, as expected.

We thus turned our attention to different cases of execution. Because of the SLD resolution, the tree search is performed from left to right. Additionally, as `parallel_findfirst/3` stops after finding the first match, a Prolog programmer that is interested in improving the performance of a program that makes use of `parallel_findfirst/3` will try to make sure that whenever a predicate is executed, its first alternative is the least resource-intensive, then the second and so on, generating a skewered tree that is heavier on the right. This problem has been subject to research and there are algorithms that have good performance in this case, such as [MLAP11] and [PA10] but, because they use work stealing rather than work sharing and we want to make few changes to YAP, we didn't explore these options. To simulate this use case, we changed the benchmark presented above in a way that generates a tree where its right half side is twice as deep as the left half, thus we have an unbalanced tree to search. This is done by adding the following predicate to the benchmark and changing `find/2` as shown next:

```
1 find(In, Out) :- length(In, Len), unbalancedFind(In, [], Len, Out).
2
3 unbalancedFind(ToFind, [], Left, End) :-
4     options(First), NextLeft is Left * (1 + First) - 1,
5     find(ToFind, [First], NextLeft, End).
```

Figure 3.2: A predicate that generates an unbalanced search tree.

Due to the exponential growth of this algorithm, the difference in size between the left and right half sides of the tree increases quickly, with the right side having the vast majority of the search space. This can be easily observed even with a small example such as the one presented in Figure 3.3.

We then performed some tests with this program using YapOr. Once again, searching the leftmost path of the tree, where the lists are composed of zeros only, can not obtain any speedup due to the semantics of Prolog, that require moving left-to-right, top-to-bottom. Queries with a solution on the large, right side of the tree obtained reasonable speedups, considering the large number of nodes that must be searched. Queries on the left side of the tree, however, had small speedups. In particular,

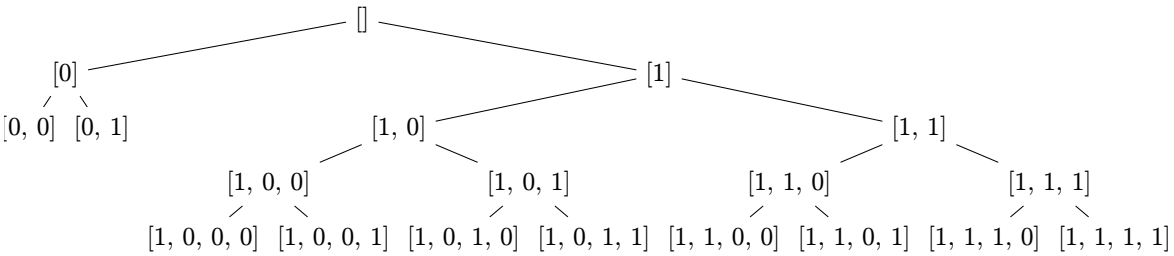

Figure 3.3: Search tree for a call to `find/2` when given a list of length 2.

given a list composed of as many zeros as the number of YapOr workers, followed by ones, produces pathological behavior, in that YapOr achieves speedups that are much smaller than expected. For these cases,the speedup reached a maximum of 2 as we increased the number of workers up to the number of cores of the machine, which is 24 in this case.

In order to make these cases easier to reproduce, we add a new clause to the `lists/1` predicate that generates lists that, given the number of workers and the size of the list, produce the pathological behavior that we expect:

```
1 lists(Pathological, FullLen, Cpus) :-
2     Len is FullLen - Cpus, mkList(Cpus, 0, Left),
3     mkList(Size, 1, Right), append(Left, Right, Pathological).
```

Figure 3.4: A clause used to generate lists that produce pathological behaviour.

3.2 Solution Searching

In order to improve the performance for queries that produce the pathological behavior we have presented, we must first be able to understand why this behavior occurs. When YapOr executes a program in parallel, each worker gets a work slice. The heuristics used by YapOr try to divide the available work in a way that:

- It avoids discarding data that the sender/receiver of work have in common;
- The worker that makes a request receives a large slice of work;
- Each worker does all the work it has before making another request.

Our experiments show that these heuristics work very well for `parallel_findall/3` because they spread the workers throughout the work tree, generating a work distribution such as the one presented in Figure 3.6.

For simplicity, we assume that moving one from a branch to another, adjacent to it, takes one time unit. Initially, only worker 0 can search the tree. After sharing work with worker 1, the appearance of the tree is that of Figure 3.5. In the next time unit, worker 0 shares work with worker 2 and worker 1 shares work with worker 3, resulting in the distribution of Figure 3.6.

This way, if the tree is balanced, each of the workers takes the same time, allowing the Prolog compiler to achieve linear speedups. If the tree is not balanced, the number of times that work has to be shared is relatively small because every time it happens, the amount of work shared is as large as possible.

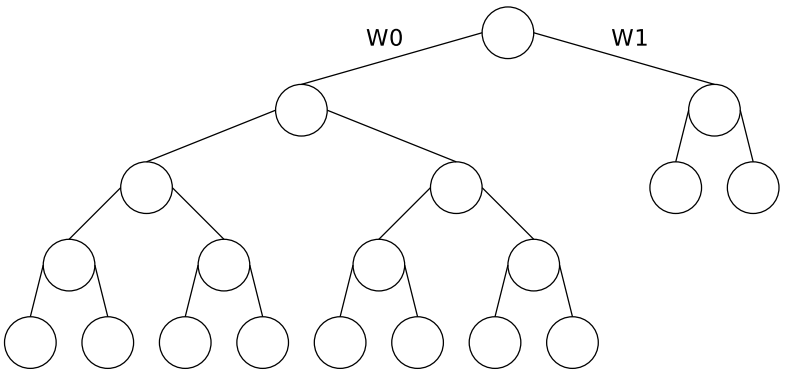

Figure 3.5: YapOr search tree after worker 0 shares work with worker 1.

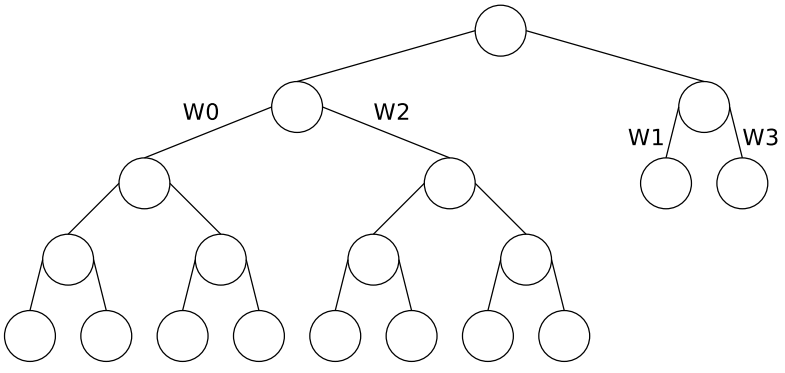

Figure 3.6: Expected YapOr worker distribution, using 4 workers.

For the `parallel_findfirst/3` predicate this strategy can be improved. First of all, because we do not need all the solutions, we should not make the Or-Parallel Engine spread the workers too much. In this case, we should in fact keep them as much to

the left as possible. The workers should also have some space between them to avoid having to much work spent on a very small portion of the search tree. A good strategy should therefore:

- Keep workers on the left side of unexplored portion of the search tree;
- Give workers some distance from each other to avoid trampling; otherwise they might spend to much time moving around the tree.

Before we present the different strategies we first briefly summarize the way the scheduler of the YapOr Engine works, because that is the only component that requires changes for the majority of the strategies presented. One of the main components of the YapOr scheduler is the `get_work` function. The most important tasks carried by this function are, in execution order:

1. Detects whether the current branch has been invalidated by a cut;
2. Picks the next alternative of the current Or-Frame, if any;
3. Gets work from a worker that is below, if any;
4. Move slightly up the tree to get work from a worker that is above;
5. Moves to a better position if necessary;
6. Detects whether the execution of the program has ended.

Notice that tasks 2 to 5 are only executed until one succeeds.

3.2.1 Strategy W - Wait Before Starting

Because we wanted to make changes that were as small as possible to the existing work scheduling and work sharing algorithms, our first attempt was to make each worker, except for worker 0, wait a small amount of time. This small change applies to the `get_work` function which, as previously explained, is the YapOr scheduler. This code is called between step 1 and step 2 of `get_work` as shown previously.

The idea behind this strategy is that we let one worker do a small amount of work, thus moving down the leftmost branches of the tree, with the objective of reducing the size of the subtree that is searched in parallel by all the workers. Notice that we may

use the same wait time for all of the workers or make each worker wait a different, random amount of time. For this strategy to perform well, it is also assumed that after this descent, the subtree that is to be explored has a solution, in which case the spreading pattern of the standard YapOr Engine should produce good results. Using this strategy produces a work distribution that follows the pattern presented in 3.7. This work distribution is very similar to that of YapOr; in fact the only difference is that in the first time unit, worker 0 did not share work and moved down. In the next two time units, after the waiting period of one time unit, the program used YapOr's default algorithm generating the same spread pattern of the default implementation.

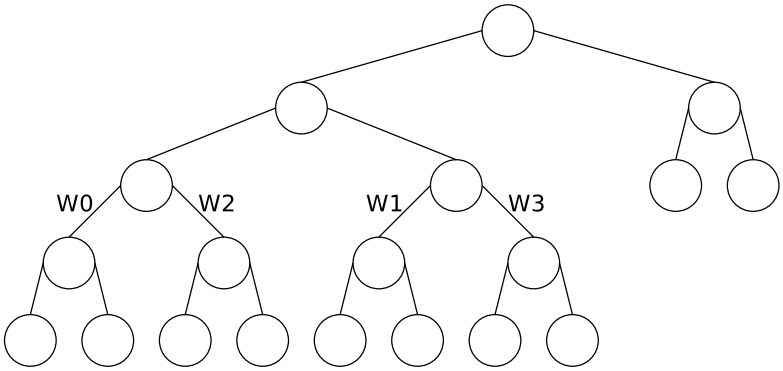

Figure 3.7: Expected worker distribution with strategy W, using 4 workers.

This strategy has an obvious flaw, however, which is that by waiting in the beginning it can increase the execution time when compared to an hypothetical strategy that is capable of descending the search tree similarly without that initial waiting time. This time increase is however very small, for large search spaces.

3.2.2 Strategy B - Get Work Below Before Taking the Next Alternative

In a very similar way to the previous strategy, this strategy requires only a relatively minor change to the `get_work` function. This change, also like in the previous strategy is applied between steps 1 and 2 of `get_work`.

Unlike the previous strategy, this does not rely on waiting time. In this case, what we try to do is to force workers to move downward on the tree, giving priority to that operation over moving to next available alternative. The effect produced by this change is that workers tends to stay on the left side of the particular subtree they are exploring, producing distributions of work similar to that presented in Figure 3.9. Using this strategy, in the first time unit, we get the distribution presented in Figure

3.5. Then worker 2 gets work from worker 0, which is below (as workers that are getting work for the first time are in the root of the tree), resulting in Figure 3.8. After that, worker 3 gets work from worker 0, resulting in Figure 3.9.

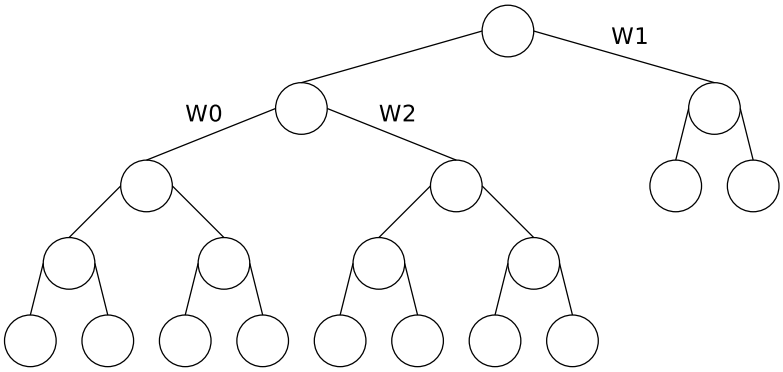

Figure 3.8: Intermediate worker distribution with strategy B, using 4 workers.

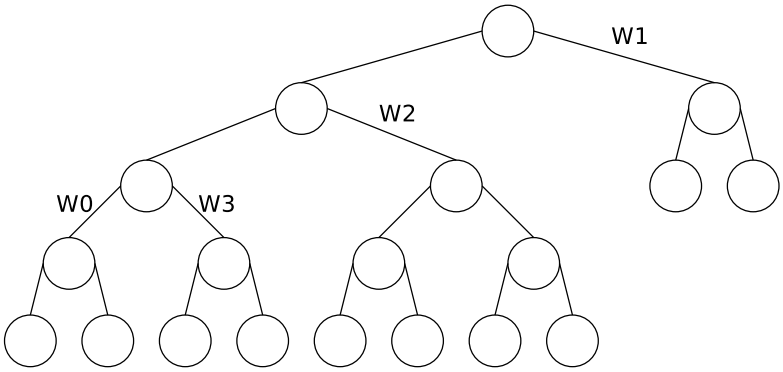

Figure 3.9: Expected worker distribution with strategy B, using 4 workers.

This strategy also has a couple of considerable flaws. Most importantly, it does not avoid the initial spreading of the workers throughout the tree, instead of keeping them close to the left side of the tree. This is problematic because, as we have to explore every node to the left of a solution, we would like to keep workers to the left, unexplored portion of the tree. Additionally, because workers always ask work below, this strategy produces the undesired behavior that each worker gets to do very little work before needing to perform a new work request.

3.2.3 Strategy L1 - Get Work to the Left Before Taking the Next Alternative

This strategy, like the ones before, make a small change to the `get_work` function, between the first and second steps of the enumerated list previously presented.

With this strategy, instead of moving down as before, we try to move workers up and to the left, in an attempt to explore first the unexplored portion of the leftmost side of the tree. This strategy tends to produce a work distribution close to that of Figure 3.10, a characteristic that the next two strategies also exhibit. To understand the reason for this distribution, assume that the execution of the program lead us to the situation presented in Figure 3.9. Worker 3, instead of continuing exploring the current alternative, moves up and asks work to worker 0, effectively moving to the left, yielding Figure 3.10.

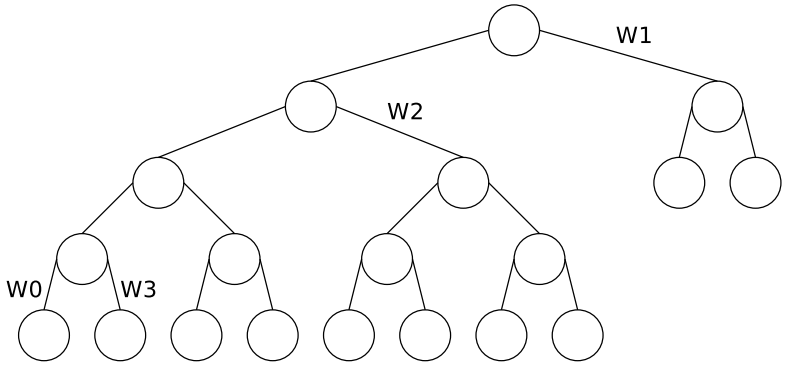

Figure 3.10: Expected worker distribution with strategies L1 to L3, using 4 workers.

We now face a problem we did not have with the previous strategies: for this strategy to work there must be at least one other worker below the requester, before it moves up - otherwise the whole subtree that the requester had control over would be lost, because no other worker would know of its existence.

For this strategy the flaw resides in the fact that the implementation is too simplistic which, coupled with the way YapOr works, can produce bad results. With this strategy, we move up the tree in order to be able to make a work request. We do so because, before making a request, a requester has to be above, in a node common with the worker that receives the request so that incremental copy can be used, as described in Section 2.5. The problem is that the function that actually makes the request and handles the answer, `q_share_work()`, may fail. In such cases, the requester, unable to return to the node it was originally in and that still might contain work, is forced to choose an alternative from the common node, moving to the right - that is, the exact opposite of what we would expect, had `q_share_work()` succeeded.

3.2.4 Strategy L2 - Get Work to the Left, with Fallback

This strategy is an improvement of the previous one. The changes are much more considerable than those of any of the previous strategies. Unlike the previous strategy we do not make the requester immediately move up. Instead, the requester fakes moving up, by keeping a copy of the pointer to the current choice point and moving up the tree without deleting any data. After being above a worker that can receive requests, the requester tries to sequentially ask the available workers for work. This required two major changes to the way the work sharing algorithm works:

- The work sharing must be divided into two procedures - one that can fail and one that always succeeds;
- The ability to cancel a move up operation.

For the first of these changes we created two new functions, `try_q_share_work()` and `q_do_share_work()`, that can and can not fail, respectively. The first of these functions verifies that there are conditions for the work sharing to take place, while the second one performs the actual sharing. The previous `q_share_work` is then replaced by a call to `try_q_share_work` followed by restoring the choice point pointer and actually moving up, cleaning up unnecessary data, concluded with a call to `q_do_share_work`.

The reason for the second change is less obvious, but is easy to understand - like before, we need to make sure that there is a worker below the requester. To do so, the requester locks its initial Or-Frame, moves up until it gets above a worker that receives requests. When moving up, there is however a possibility that the worker that was below actually locks the current Or-Frame before the requester, moves up and unlocks it. In this case the requester can no longer move up and must cancel the operation in order to pick the next available alternative of the current choice point.

By solving the problem of Strategy L1, that the requester might move up and, unable to go back to its original position, be forced right in the tree by choosing the next available alternative; the biggest flaw with this strategy becomes the linear search that has to be performed on the list of workers that are to the left of the requester. In the worst case, the requester may be unable to get any work from any of the potential requestees that it asked, thus forcing the worker to go back to its original position in the search tree, making the time spent executing this strategy useless. There are also cases where, if the requester receives work from the least loaded worker, it receives

considerably less work, or receives work that is farther to the right than the work it would receive from a different worker.

3.2.5 Strategy L3 - Get Work to the Left, in Sorted Order

This strategy focuses on solving one of the previous flaws, namely that the worker may be able to receive work farther to the left than it does when searching the workers sequentially. The changes that we describe next do not affect any additional portions of YapOr relative to the previous strategy, other than the initial memory allocation function.

To mitigate these problems, we implemented a doubly-linked list that is memory-mapped when the program starts and has fixed size during the execution of the program. This linked list keeps the order of the workers in the search tree and has a single lock access mechanism, as lock striping proved difficult to implement correctly. In exchange for the implementation of the list, the workers now know the worker's order, from the leftmost to the rightmost.

When workers move, by calling the `get_work` function, they might change their realtive positions. For example, if they receive work, they are immediately to the right of the worker that shared work, an O(1) operation. If the worker picks an alternative of a choice point, the worker must position itself to the right of the rightmost worker that is at the same subtree, an O(n) operation.

The two remaining problems of the previous strategy - getting less work than the possible maximum and having to go back to the original position in the tree - are kept unsolved and keeping the list updated can be costly in time, which makes this strategy a trade-off of searching linearly to updating linearly.

3.2.6 Strategy F1 - Descend the Tree, Start Sharing After Failing the First Time

This strategy is based on a different idea from the previous ones, although it has some similarities with the first waiting strategy. When the execution starts, only worker 0 is able to do work. This strategy changes the Prolog portion of the implementation of `parallel_findfirst/3` by calling a new predicate `disable_work_sharing/0`, adds a small change to the abstract machine so that after the first call to `fail/0` the work sharing mechanism is enabled and changes the scheduler accordingly, allowing all workers to share work.

The implementation first creates a memory mapped variable `is_work_sharing_enabled` during the initialization of YapOr that is used to keep track of whether or not work sharing is enabled. Then, once again between steps 1 and 2, every worker except for the first waits for the variable to be true and keeps yielding the processor in order to avoid wasting CPU time. The first worker starts execution as normal. When the variable becomes true, after the first `fail/0`, the other workers start making work requests. Using this strategy the worker distribution presents the pattern shown in Figure 3.12. Notice that what always happens is that worker 0 fails at the position shown in Figure 3.11. Afterwards, all the workers can start sharing work, leading to the somewhat random positioning of Figure 3.12, where the workers are all close together.

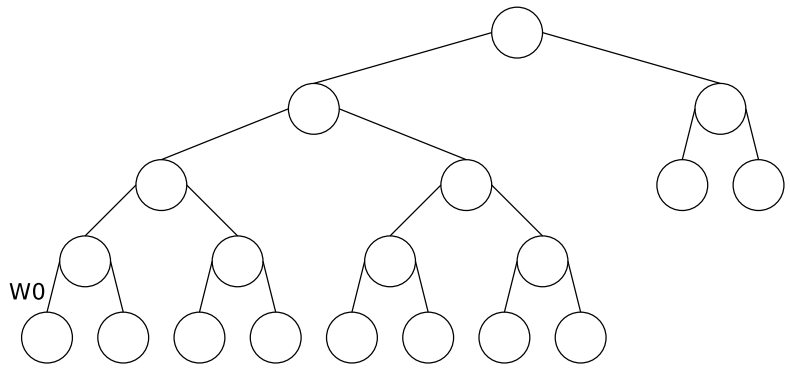

Figure 3.11: Worker 0, immediately before `fail`ing.

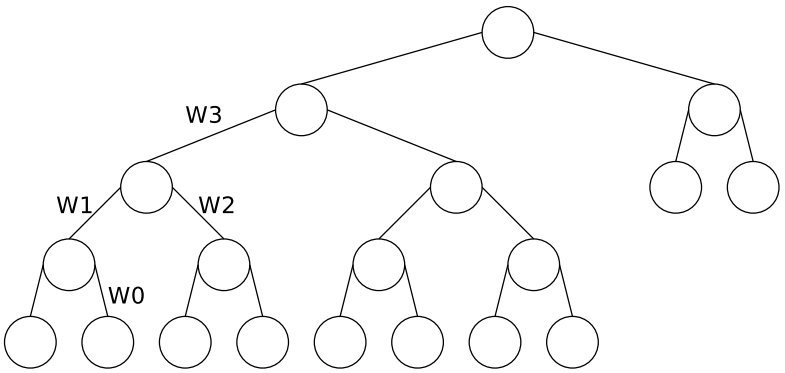

Figure 3.12: Expected worker distribution with strategy F1, using 4 workers.

Unlike the previous strategies, this strategy deals with a major problem detected early on: workers that start by moving to the right, become lost forever because they are never able to complete the search of the big subtree before the other workers complete the small one. This strategy tries to force every worker to start at the bottom left of the search tree and then move up and to the right as needed, minimizing this problem.

This strategy still has a couple of flaws - one of them is that the waiting time before the process of work sharing starts can be a considerable slice of the total runtime; the other is that the first time that work is shared, each worker might copy a huge amount of memory, because it has none in common with the sender, which might be at a very deep level of the tree.

3.2.7 Strategy F2 - Descend the Tree, Share Less After Failing the First Time

This strategy intends to improve upon the previous one by diminishing the amount of work that must be shared overall during the first time each worker receives work. It does not do changes to any additional modules of Yap besides the ones changed by the previous strategy.

We first change the `is_work_sharing_enabled` variable to be an integer instead of a boolean, changing its name in the process to `maximum_sharing_worker`. When `maximum_sharing_worker` is 0, sharing is disabled. After the first `fail/1`, the value is set to 1 and worker 1 gets work from worker 0 and then moves to 2/3 of the depth of worker 0, setting `maximum_sharing_worker` to 2. Then worker 2 does the same, relative to worker 1 and so on until every worker has work. This strategy produces a different worker distribution from the previous strategy; the expected worker distribution is presented in Figure 3.13. The difference is that the state of Figure 3.11 is reached, the work sharing is done sequentially, with each worker moving up one node, forming the distribution of Figure 3.13.

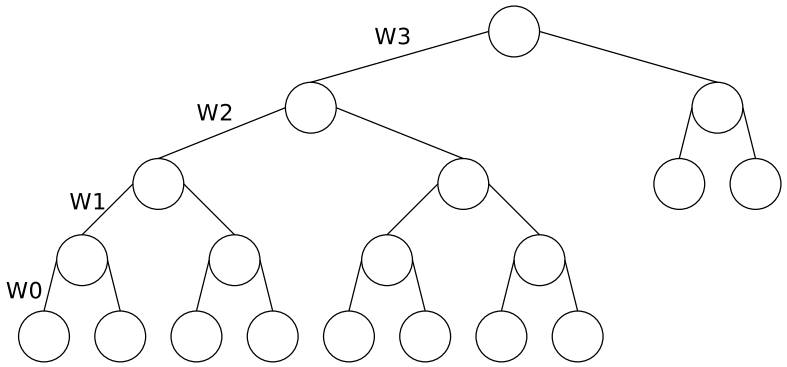

Figure 3.13: Expected worker distribution with strategy F2, using 4 workers.

This strategy would considerably decrease the amount of sharing and spread the workers enough to avoid having too many work requests, except that in practice, we observed that this initial work sharing process fails most of the time, delaying

considerably the initialization of all workers. As discussed in Chapter 4, we can observe that this strategy is unable to improve over the previous one, when implemented in YapOr.

3.2.8 Strategy S - Freeze the Or-Frames, Allowing Voluntary Suspension

This strategy makes use of functions of OPTYAP, the Or-Parallel tabling engine of Yap, that are capable of freezing and unfreezing a complete execution sub-branch of the search tree. It should also be the strategy that causes the biggest variations in terms of execution, producing the most nondeterministic behavior of all the strategies presented. We make changes to the Or-Frame data structure and to the `get_work` function.

The first change we have done was to add a new field to the Or-Frame, `or_fr_continuation` which keeps a pointer to a frozen Or-Frame that is immediately below - or the continuation of - the current Or-Frame. We also slightly altered the `suspend_branch()` and `resume_suspension_frame()` functions to freeze and unfreeze the Or-Frames, respectively. Using the doubly-linked list of strategy 3.2.5, we keep track of the order of workers. If the worker that tries to `get_work` is the rightmost, it suspends all frames that are not common to any other worker, essentially creating a linked list of suspended Or-Frames, and asks all workers from the leftmost for work until it gets work or, failing that, resumes execution of the suspended Or-Frames. Assuming that we have the worker distribution presented in Figure 3.14 and that worker 1 suspends work to move to the left, we arrive at the distribution presented in Figure 3.15.

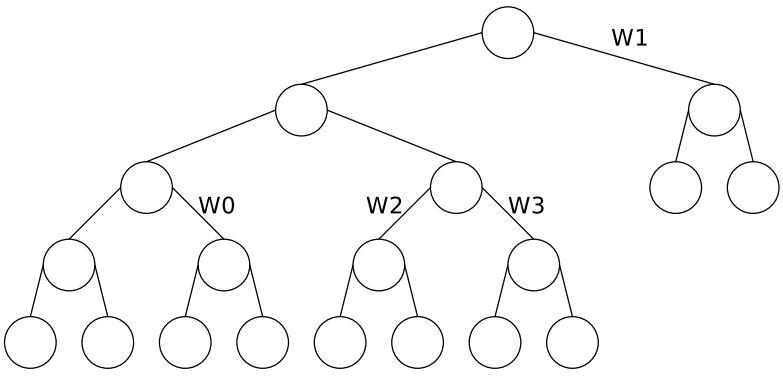

Figure 3.14: Expected worker distribution with strategy S, using 4 workers, before suspending work.

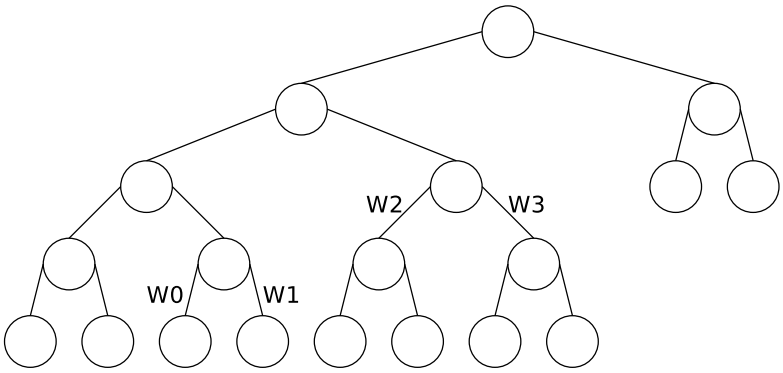

Figure 3.15: Expected worker distribution with strategy S, using 4 workers, after suspending work.

Although preliminary experiments based on voluntary movement that discards Or-Frames presented good performance - when the workers did not discard the Or-Frame with the solution, which could happen because the algorithm was incorrect - we were unable to implement voluntary suspension in practice due to severe runtime errors whenever we attempted to do so.

3.2.9 Summary

We presented different strategies that make different decisions regarding how to schedule work. We saw that we may:

- Wait a certain amount of time;
- Try to always move down the tree;
- Move back to be able to move left;
- Wait until failing the first time;
- Suspend work to move to left.

We also saw that we should avoid sharing too little which leads to trampling, or too much because sharing is a slow operation. Additional possibilities are presented in Chapter 5.

4 Experimental Results

In this chapter, we present experimental results for the strategies proposed in the previous chapter, showing the speedups that each stategy was able to achieve and how they vary based on the size of the problem we want to address. Our results were obtained using two NUMA machines, which were used independently from each other. Both have the same technical specifications: four six-core AMD Opteron 8425 HE processors, totaling 24 cores, each with a clock speed of 2.1 GHz, with 128 GB RAM and using Fedora Core 20 in 64-bit mode.

Since the `parallel_findfirst/3` predicate is not available for Yap's sequential version, all the speedups measured with a certain number of workers are relative to running that same version of YapOr with a single worker.

We ran tests used the different strategies we presented with different problem sizes. As expected of the exponential benchmark we used, we verified that the execution times doubled when the size of the problem increased by 1. Therefore, for simplicity, we always chose a size that took as close to 200 seconds as possible, when using a single worker, for all the results presented in this section.

In the tables we present next, that are accompanied by a graphical representation, there are 4 rows:

1. The execution time using 1 worker;
2. The execution time using the number of workers indicated in the caption;
3. The speedup obtained;
4. The speedup relative to the default YapOr strategy.

To measure the speedups, we ran the benchmark presented in Appendix A 12 times for each pair of strategy and problem size and then we discarded the fastest and the

slowest runs, presenting the average of the remaining 10 runs. We do not present results for `parallel_findall/3` because our tests revealed that these strategies do not improve its execution.

Strategy:	YapOr	W	B	L1	L2	L3	F1	F2
1 worker:	174.142	174.621	174.271	174.923	174.612	175.232	173.965	174.672
4 workers:	96.121	96.138	114.961	96.662	75.284	77.770	92.224	86.589
Speedup:	1.81	1.82	1.52	1.81	2.32	2.25	1.89	2.02
Relative:	1.00	1.01	0.84	1.00	1.28	1.24	1.04	1.12

Table 4.1: Execution times and speedups obtained with 4 workers and problem size 30, for all strategies.

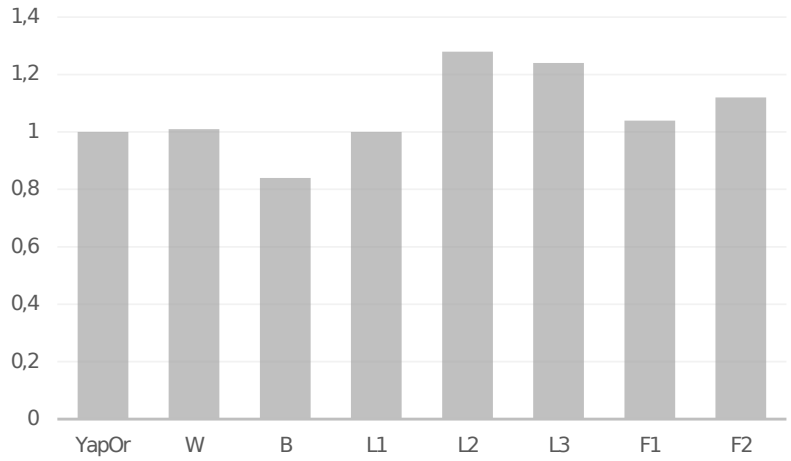

Figure 4.1: Relative speedups obtained with 4 workers.

Looking at table 4.1 we immediately notice that:

- Strategy B significantly decreases the speedup comparatively to standard YapOr;
- Strategies W and L1 do not produce considerable changes;
- Strategies L2 and L3 increase speedups a little bit, where strategy L2 performs better;
- Strategies F1 and F2 increase speedups slightly and, out of the two, strategy F2 is better.

An interesting observation that is not immediately obvious is that strategies F1 and F2 generally have similar runtimes to YapOr's strategy but sporadically produce results

in about a half of YapOr's time, that is, they have speedups close to 4. Out of these strategies, strategy F2 appears to have a higher likelihood of producing the solutions with a relatively good speedup.

In our tests, the results obtained when using 4 workers are very similar to those obtained for other numbers of workers, in terms of the speedup that is achieved. For brevity the remaining tables only show results for standard YapOr, strategy L2 and strategy F2.

Strategy:	YapOr	L2	F2
1 worker:	203.044	204.957	203.002
8 workers:	102.534	88.121	95.424
Speedup:	1.98	2.33	2.12
Relative:	1.00	1.18	1.07

Table 4.2: Execution times and speedups obtained with 8 workers and problem size 34.

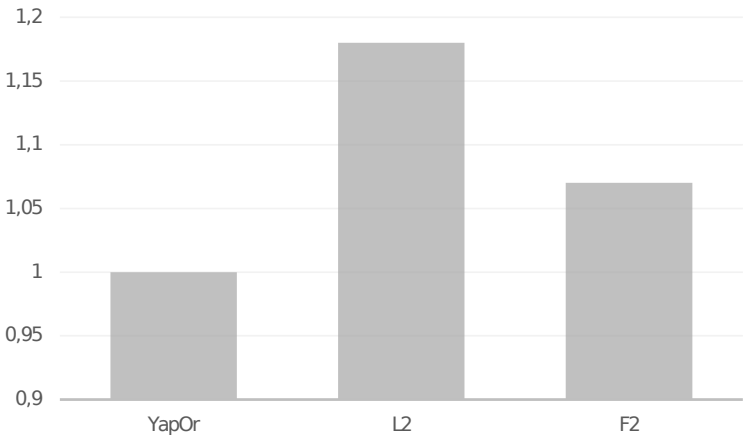

Figure 4.2: Relative speedups obtained with 8 workers.

Strategy:	YapOr	L2	F2
1 worker:	226.521	227.110	227.029
12 workers:	118.145	96.612	112.103
Speedup:	1.92	2.35	2.03
Relative:	1.00	1.22	1.06

Table 4.3: Execution times and speedups obtained with 12 workers and problem size 38.

After analyzing these tables side-by-side we can notice a pattern in the execution of YapOr: in general, when using a number `a + n` of workers and using a list that has `b`

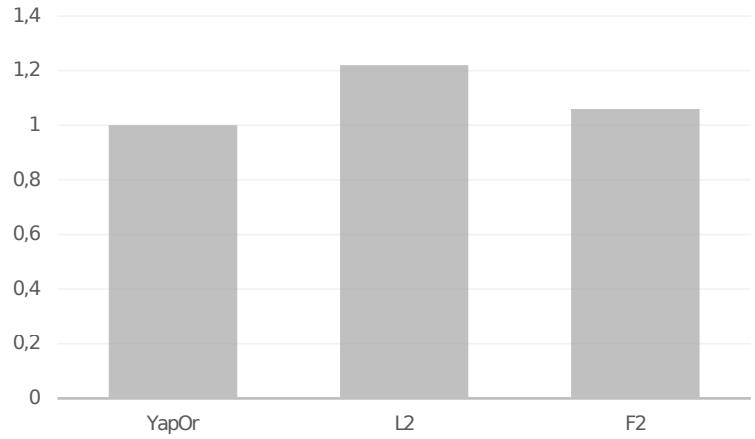

Figure 4.3: Relative speedups obtained with 12 workers.

Strategy:	YapOr	L2	F2
1 worker:	250.255	251.978	252.001
16 workers:	126.344	101.563	118.524
Speedup:	1.98	2.48	2.12
Relative:	1.00	1.25	1.08

Table 4.4: Execution times and speedups obtained with 16 workers and problem size 42.

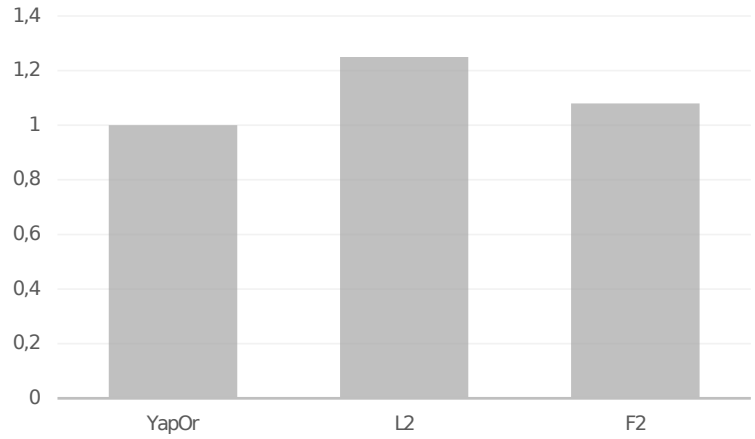

Figure 4.4: Relative speedups obtained with 16 workers.

`+ n` zeros at the beginning and `c` ones following, the execution takes a similar amount of time to that of `a` workers, with a list of `b` zeros and `c` ones. This is to be expected, assuming that all the `n` additional workers take the right side of each of the first n choice points and, as such, reduce the problem to the version without those `n` workers and the smaller list.

Strategy:	YapOr	L2	F2
1 worker:	142.813	143.122	144.672
20 workers:	71.324	58.556	66.890
Speedup:	2.00	2.44	2.16
Relative:	1.00	1.22	1.08

Table 4.5: Execution times and speedups obtained with 20 workers and problem size 45.

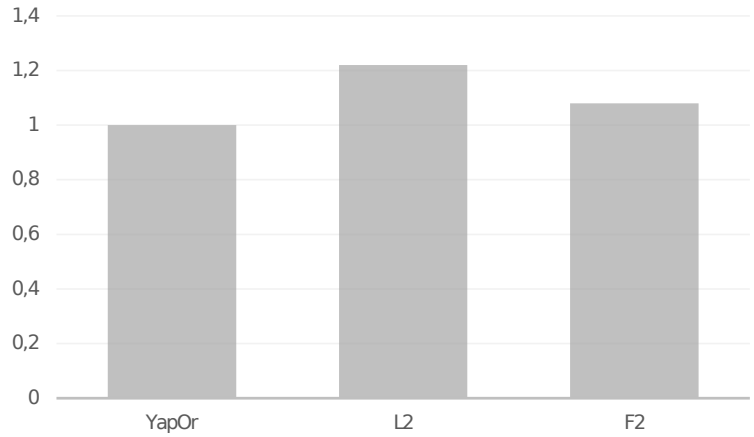

Figure 4.5: Relative speedups obtained with 20 workers.

Strategy:	YapOr	L2	F2
1 worker:	148.312	149.216	150.275
24 workers:	74.142	60.922	70.443
Speedup:	2.00	2.45	2.13
Relative:	1.00	1.23	1.07

Table 4.6: Execution times and speedups obtained with 24 workers and problem size 49.

We can also see that, despite our efforts, the speedups do not increase in any considerable manner with the addition of workers, although the speedups tend to stabilize at a speedup of 2, with the standard YapOr strategy.

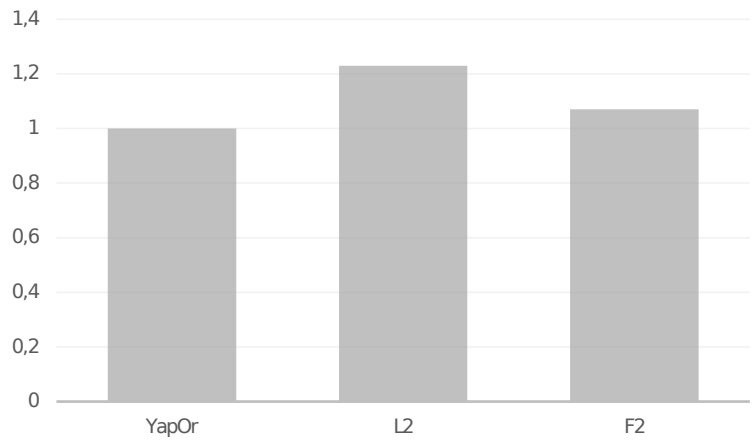

Figure 4.6: Relative speedups obtained with 24 workers.

5 Conclusions and Future Work

This chapter summarizes the main contributions of the work and presents paths for future work, that we believe to be potentially worthwhile to study.

5.1 Main Contributions

Different strategies of work sharing were designed and implemented in the YapOr Or-Parallel Engine. Some of these strategies are capable of producing solutions faster than the original implementation, allowing for more work to be done in a certain amount of time. These implementations were tested with varying numbers of workers, to show that the improvements obtained are kept as the size of the problem increases.

The work presented in this document therefore consists of the design and implementation of different strategies designed to improve the speedup of the Or-Parallel execution of the `parallel_findfirst/3` predicate.

5.2 Further Work

It is our hope that the work presented in this thesis allows for new research opportunities and that it can be used as the basis for following improvements to the parallel execution of Prolog. We now present a few suggestions of related work that may be done in the future:

- Conclude the implementation of the voluntary suspension strategy (strategy S) which, based on our preliminary experiments appears to have potential to be the fastest of the strategies we have proposed.
- Evaluate the efficiency of hybrid approaches, mixing some of the proposed strategies; for instance, using `get_work_below` when on the root choice point, `get_work_above` when possible and suspension otherwise.

- Use advanced work stealing techniques such as those presented in [MLAP11], which also deal with unbalanced tree search and avoid the trampling problem.
- Implement new constructors for Or-Parallelism, such as `find_any/3` which would find one solution but not necessarily the first. For the implementation of this constructor, it may be useful to keep track of whether a branch may or not have a cut, in order to choose the least speculative branches first.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[AK90] K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.

[AVW92] J. Armstrong, S. Virding, and M. Williams. Use of Prolog for developing a new programming language. 277, 1992.

[Bac78] J. Backus. The history of Fortran I, II, and III. In *History of programming languages I*, pages 25–74. ACM, 1978.

[CR93] A. Colmerauer and P. Roussel. The Birth of Prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[CRD12] V. S. Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

[DAC12] D. Diaz, S. Abreu, and P. Codognet. On the implementation of GNU Prolog. volume 12, pages 253–282. Cambridge University Press, 2012.

[Elc90] E. W. Elcock. ABSYS: The First Logic Programming Language – a Retrospective and Commentary. *Journal of Logic Programming*, 9(1):1–17, July 1990.

[Kow88] R. A. Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, 31(1):38–43, January 1988.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[MLAP11] R. Machado, C. Lojewski, S. Abreu, and F. J. Pfreundt. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science - R&D*, 26(3-4):229–236, 2011.

[ND62] K. Nygaard and O. Dahl. SIMULA: An Extension of ALGOL to the Description of Discrete-Event Networks. In *Second International Conference on Information Processing*, 1962.

[PA10] V. Pedro and S. Abreu. Distributed Work Stealing for Constraint Solving. In *Proceedings of InForum 2010*. Universidade do Minho, September 2010.

[PW80] F. C. N. Pereira and D. H. D. Warren. Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[Roc96] R. Rocha. Um Sistema Baseado na Cópia de Ambientes para a Execução de Prolog em Paralelo. MSc Thesis, University of Minho, Portugal, July 1996. In Portuguese.

[Roc01] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Department of Computer Science, University of Porto, 2001.

[RRS+95] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *International Conference on Logic Programming*, pages 687–711. The MIT Press, 1995.

[War77] D. H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[Wie14] J. Wielemaker. *SWI-Prolog Reference Manual*, 2014.

[WNS97] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, 1997.

Benchmarking Code

A

```
1  append([Head|Left], Item, [Head|Tail]) :- append(Left, Item, Tail).
2  append([], List, List).
3
4  options(0).
5  options(1).
6
7  unbalancedFind(ToFind, [], Left, End) :-
8      options(First), NextLeft is Left * (1 + First) - 1,
9      find(ToFind, [First], NextLeft, End).
10
11 find(ToFind, SoFar,  Left, [Item|End]) :-
12     Left > 0, options(Item), append(SoFar, [Item], Next),
13     NextLeft is Left - 1, find(ToFind, Next, NextLeft, End).
14 find(ToFind, ToFind, _,    []).
15
16 find(In, Out) :- length(In, Len), unbalancedFind(In, [], Len, Out).
17
18 mkList(0,   _,   [])        :- !.
19 mkList(Len, Num, [Num|End]) :- Left is Len - 1, mkList(Left, Num, End).
20
21 lists(Pathological, FullLen, Cpus) :-
22     Len is FullLen - Cpus, mkList(Cpus, 0, Left),
23     mkList(Len, 1, Right), append(Left, Right, Pathological).
24
25 go(Len, Cpus) :-
26     lists(List, Len, Cpus),
27     time(parallel_findall(Var, find(List, Var), Solutions)),
28     fail.
```

Figure A.1: A test designed to stress the `parallel_findfirst/3` implementation.

```
#!/bin/bash
pushd $(dirname $0) 2>&1 >/dev/null
DIR=$(pwd)

for VERSION in "yap1" "yap2"; do
    cd $DIR/../$VERSION 2>&1 >/dev/null
    echo "Size = $2"

    TIME1=$(echo "go($1, $2)." |
            ./yap -l $DIR/bench.pro 2>&1 >/dev/null |
            head -n3 | tail -n1)

    TIMEN=$(echo "go($1, $2)." |
            ./yap -l $DIR/bench.pro -w $CPUS 2>&1 >/dev/null |
            head -n3 | tail -n1)

    echo $TIME1
    echo $TIMEN

    if [ "$TIMEN" == "0.000" ]; then
        TIMEN="0.001"
    fi

    TIME1=$(echo "$TIME1" | awk '{print $6}')
    TIMEN=$(echo "$TIMEN" | awk '{print $6}')

    echo | awk '{res='"$TIME1 / $TIMEN"'; printf "Speedup = %.2f\n", res}'
done;

popd 2>&1 > /dev/null
```

Figure A.2: Calculates the speedup of the parallel execution of A.1.

```
#!/bin/bash
LEFT=$#
CPUS=$(lscpu | grep "CPU(s):" | awk '{print $2;}' | head -n1)
echo $(hostname) "(1 worker VS" $CPUS "workers):"

while [ $LEFT -gt 0 ]; do
    echo
    bash runbench.sh $CPUS $1
    shift
    LEFT=$(echo "$LEFT - 1" | bc)
done
```

Figure A.3: A wrapper that calls A.2 for each problem size passed as an argument.