On Applying Or-Parallelism and Tabling to Logic Programs

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

2001
Ricardo Jorge Gomes Lopes da Rocha

On Applying Or-Parallelism and Tabling to Logic Programs

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2001
To Rute
I Lobe You
Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisors, Fernando Silva and Vítor Santos Costa, for their constant support, encouragement and guidance. Both were always there, ready to discuss about any question and helped me to fix the problems and find the answers to carry on this work. To both I also would like to thank the revisions, comments and suggestions regarding the improvement of this thesis.

I am grateful to C. R. Ramakrishnan for his contribution in building runnable versions of the XMC benchmark programs out of the XMC environment, and to Ashwin Srinivasan for his availability in providing access to his parallel machine that allowed us to evaluate our system.

I am also grateful to the Junta Nacional de Investigação Científica e Tecnológica (now Fundação para a Ciência e a Tecnologia) for their support under the research grant PRAXIS XXI/BD/9276/96 during the first 18 months of this work.

To Ricardo Lopes, Michel Ferreira, Álvaro Figueira, Luís Lopes, Eduardo Correia, Luís Antunes and the guys from CDUP’s football, I would like to testify their friendship and the enjoyable moments spent together through these years.

To my family, I would like to say that they are part of this work. To my parents, José and Noémia, for all the love you always gave me. To my sister, Silvia, for being such a nice person. To my wife, Rute, for all the affection, support and understanding. I love you.

I would like to express a final acknowledgment to all of those that throughout my life, at different levels and by different means, had contributed to shape my personality and help me to be the person I am. Thank you all!

Ricardo Rocha
September 2001
Abstract

Logic programming languages, such as Prolog, provide a high-level, declarative approach to programming. They offer a great potential for implicit parallelism and thus allow parallel systems to automatically reduce a program's execution time without any programmer intervention. For complex applications that take several hours, if not days, to return an answer, even modest parallel execution speedups can be directly translated to very significant productivity gains.

Despite the power, flexibility and good performance that Prolog has achieved, the past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. Unfortunately, some deficiencies in Prolog's evaluation strategy – SLD resolution – limit the potential of the logic programming paradigm. Tabling has proved to be a viable technique to efficiently overcome SLD's susceptibility to infinite loops and redundant subcomputations.

With this research we aim at demonstrating that implicit or-parallelism is a natural fit for logic programs with tabling. To substantiate this belief, we propose novel computational models that integrate tabling with or-parallelism, we design and implement an or-parallel tabling engine – OPTYap – and we use a shared memory parallel machine to evaluate its performance. To the best of our knowledge, OPTYap is the first implementation of a parallel tabling engine for logic programming systems. OPTYap builds on Yap's efficient sequential Prolog engine. Its execution model is based on the SLG-WAM for tabling, and on the environment copying for or-parallelism.

The results in this thesis make it clear that the mechanisms proposed to parallelize search in the context of SLD resolution can indeed be effectively and naturally generalized to parallelize tabled computations, and that the resulting systems can achieve good performance on shared memory parallel machines. More importantly, it emphasizes our belief that through applying or-parallelism and tabling to logic programs we can contribute to increase the range of applications for Logic Programming.
**Resumo**

Uma das vantagens do Prolog, como linguagem de Programação Lógica, é possuir uma semântica que possibilita a exploração de paralelismo implícito. Esta característica permite reduzir o tempo de execução de um programa, sem que para isso sejam necessárias anotações adicionais do programador. Para aplicações complexas, que demoram várias horas, senão dias, a calcular uma solução, mesmo ganhos de velocidade modestos em execução paralela, podem traduzir-se em significantes ganhos de produtividade.

Apesar do poder, da flexibilidade e dos bons resultados que o Prolog tem demonstrado desde o aparecimento da WAM, um amplo esforço tem vindo a ser desenvolvido para aumentar o seu poder declarativo e expressivo. A estratégia de resolução SLD, na qual o Prolog se baseia, é limitadora do potencial inerente ao paradigma da Programação Lógica. Uma das mais bem sucedidas técnicas para solucionar a incapacidade da resolução SLD no que respeita a ciclos infinitos e computações redundantes é a Tabulação.

Com este trabalho pretende-se demonstrar que a exploração implícita de paralelismo-Ou em programas lógicos com tabulação pode ser tão eficaz como o é em programas lógicos comuns. Para tal, propõem-se novos modelos para integrar paralelismo-Ou com tabulação, desenvolve-se um novo sistema, o OPTYap, e avalia-se o seu desempenho. Tanto quanto é do nosso conhecimento, o OPTYap é o primeiro sistema a explorar paralelismo em programas lógicos com tabulação. O OPTYap foi desenvolvido tendo por base o sistema Yap, um dos mais rápidos sistemas de execução sequencial de Prolog. O seu modelo de execução é baseado na SLG-WAM, para tabulação, e em cópia de ambientes, para paralelismo-Ou.

Os resultados mostram que os mecanismos para execução paralela de programas lógicos podem generalizar-se para computações que usam tabulação, e que os sistemas daí resultantes obtém igualmente bons desempenhos em máquinas paralelas de memória partilhada. Este trabalho reforça a convicção de que paralelismo e tabulação podem contribuir para expandir o leque de aplicações alvo da Programação Lógica.
Résumé

Les langages de Programmation Logique, tels que le Prolog, fournissent une approche déclaratif de niveau élevé à la programmation. Ils offrent un grand potentiel pour l'exploration implicite du parallélisme et permettent ainsi, aux systèmes parallèles, de réduire automatiquement le temps d'exécution d'un programme, sans interposition du programmeur. Pour des applications complexes qui prennent plusieurs heures, sinon des jours, pour renvoyer une réponse, même modestes gains de vitesse peuvent être directement traduits aux gains très significatifs de productivité.

En dépit de la puissance, de la flexibilité et des bons résultats que le Prolog a réalisé, les dernières années ont vu un large effort pour augmenter son pouvoir déclaratif et expressif. Malheureusement, quelques insuffisances dans la stratégie d'évaluation du Prolog, résolution SLD, limitent le potentiel du paradigme de Programmation Logique. Tabulation a montré être une technique viable pour surmonter efficacement la susceptibilité de SLD aux calculs infinis et aux computations redondantes.

Avec cette recherche nous visons démontrer que l'exploration implicite du parallélisme-Ou est un ajustement naturel pour des programmes logiques avec la tabulation. Pour justifier cette croyance, nous proposons des nouveaux modèles pour intégrer parallélisme-Ou avec tabulation, nous concevons une nouvelle système, l'OPTYap, et nous évaluons son exécution. Au meilleur de notre connaissance, OPTYap est la première mise en place d'un système parallèle de tabulation. OPTYap a été développé ayant pour base le système Yap, l'un des plus rapides systèmes d'exécution séquentielle de Prolog. Son modèle d'exécution est basé sur la SLG-WAM, pour tabulation, et sur copie d'ambients, pour parallélisme-Ou.

Les résultats indiquent que les mécanismes proposés pour exécution parallèle, dans le contexte de la résolution SLD, peuvent être généralisés pour la tabulation, et que les systèmes résultants peuvent obtenir aussi des bons résultats aux systèmes parallèles de mémoire partagée. Cette thèse souligne notre croyance qu’en appliquant parallélisme-
Ou et tabulation aux programmes de logique nous pouvons contribuer à l’augmentation de l’ensemble des applications pour la Programmation Logique.
4.1 The SLG-WAM Abstract Machine ........................................ 77
  4.1.1 Basic Tabling Definitions ......................................... 78
  4.1.2 SLG-WAM Overview .................................................. 79
  4.1.3 Batched Scheduling .................................................. 79
  4.1.4 Fixpoint Check Procedure .......................................... 80
  4.1.5 Incremental Completion .............................................. 81
  4.1.6 Instruction Set for Tabling ......................................... 82
4.2 Extending Yap to Support Tabling ...................................... 83
  4.2.1 Overview ..................................................................... 84
  4.2.2 Table Space .............................................................. 85
  4.2.3 Generator and Consumer Nodes ..................................... 88
  4.2.4 Subgoal and Dependency Frames ..................................... 90
  4.2.5 Freeze Registers ........................................................ 92
  4.2.6 Forward Trail ............................................................ 93
  4.2.7 Completion and Leader Nodes ....................................... 96
  4.2.8 Answer Resolution ...................................................... 99
  4.2.9 A Comparison with the SLG-WAM .................................. 102
4.3 Local Scheduling ............................................................. 104
4.4 Chapter Summary ........................................................... 108

5 Parallel Tabling ................................................................. 109
  5.1 Related Work ................................................................... 109
  5.2 Novel Models for Parallel Tabling ..................................... 112
    5.2.1 Or-Parallelism within Tabling (OPT) ............................. 113
5.2.2 Tabling within Or-Parallelism (TOP) .......................... 114
5.2.3 Comparing the Models .......................................... 116
5.2.4 Framework Motivation for the OPT Model .................. 117
5.3 Chapter Summary .................................................. 120

6 OPTYap: The Or-Parallel Tabling Engine .......................... 123

   6.1 Implementation Overview ...................................... 123
   6.2 The Parallel Data Area ......................................... 124
       6.2.1 Memory Organization .................................. 125
       6.2.2 Page Management ....................................... 127
       6.2.3 Improving Page Management for Answer Trie Nodes .... 129
   6.3 Concurrent Table Access ....................................... 130
       6.3.1 Trie Structures ......................................... 130
       6.3.2 Table Locking Schemes ................................ 134
   6.4 Data Frames Extensions ....................................... 138
       6.4.1 Or-Frames ................................................ 138
       6.4.2 Subgoal and Dependency Frames ......................... 140
   6.5 Leader Nodes .................................................. 141
   6.6 The Flow of Control .......................................... 146
       6.6.1 Public Completion ...................................... 146
       6.6.2 Answer Resolution ...................................... 151
       6.6.3 Getwork ................................................ 154
   6.7 SCC Suspension ................................................ 156
   6.8 Scheduling Work .............................................. 161

16
6.9 Local Scheduling .............................................. 165
6.10 Chapter Summary ............................................ 169

7 Speculative Work ............................................. 171

7.1 Cut Semantics ................................................. 172

7.2 Cut within the Or-Parallel Environment ...................... 173
  7.2.1 Our Cut Scheme ........................................... 173
  7.2.2 Tree Representation ..................................... 175
  7.2.3 Leftmostness .............................................. 177
  7.2.4 Pending Answers .......................................... 177
  7.2.5 Scheduling Speculative Work ............................. 178

7.3 Cut within the Or-Parallel Tabling Environment .............. 179
  7.3.1 Inner and Outer Cut Operations ......................... 180
  7.3.2 Detecting Speculative Tabled Answers .................. 181
  7.3.3 Pending Tabled Answers .................................. 183

7.4 Chapter Summary ............................................. 185

8 Performance Analysis ......................................... 187

8.1 Performance on Non-Tabled Programs ......................... 188
  8.1.1 Non-Tabled Benchmark Programs ......................... 189
  8.1.2 Overheads over Standard Yap ............................ 190
  8.1.3 Speedups for Parallel Execution ......................... 191

8.2 Performance on Tabled Programs ............................. 192
  8.2.1 Tabled Benchmark Programs .............................. 193
  8.2.2 Timings for Sequential Execution ....................... 194
8.2.3 Characteristics of the Benchmark Programs .......................... 196
8.2.4 Parallel Execution Study ............................................... 198
8.2.5 Locking the Table Space .............................................. 206
8.3 Chapter Summary ......................................................... 208

9 Concluding Remarks ....................................................... 211
9.1 Main Contributions ..................................................... 211
9.2 Further Work .............................................................. 214
9.3 Final Remark .............................................................. 216

A Benchmark Programs ..................................................... 219
A.1 Non-Tabled Benchmark Programs ..................................... 219
A.2 Tabled Benchmark Programs .......................................... 222

References ................................................................. 224
List of Tables

8.1 Yap, YapOr, YapTab, OPTYap and XSB execution time on non-tabled programs. .......................... 191
8.2 Speedups for YapOr and OPTYap on non-tabled programs. .............. 192
8.3 YapTab, OPTYap and XSB execution time on tabled programs. ....... 195
8.4 Characteristics of the tabled programs. ................................. 197
8.5 Speedups for OPTYap using batched scheduling on tabled programs. . 198
8.6 Statistics of OPTYap using batched scheduling for the group of programs with parallelism. ................. 201
8.7 Statistics of OPTYap using batched scheduling for the group of programs without parallelism. .................. 202
8.8 Speedups for OPTYap using local scheduling on tabled programs. .... 204
8.9 Statistics of OPTYap using local scheduling for the group of programs showing worst speedups than for batched scheduling. ................. 205
8.10 OPTYap execution time with different locking schemes for the group of programs with parallelism. ............... 207
# List of Figures

2.1 WAM memory layout, frames and registers. ........................................... 37  
2.2 An infinite SLD evaluation. ................................................................. 48  
2.3 A finite tabled evaluation. ................................................................. 49  
2.4 Fibonacci complexity for SLD and tabled evaluation. ......................... 51  
3.1 Backtracking to the bottom common node. ........................................... 61  
3.2 Incremental Copying. ........................................................................... 62  
3.3 Requesting work from workers within the current subtree. ..................... 65  
3.4 Requesting work from workers outside the current subtree. ................... 66  
3.5 Scheduling strategies to move idle workers to better positions. ............... 67  
3.6 Memory organization in YapOr. ............................................................ 69  
3.7 Remapping the local spaces. ............................................................... 70  
3.8 Sharing a choice point. ....................................................................... 71  
3.9 The sharing work process. ................................................................... 74  
4.1 Compiled code for a tabled predicate. ................................................... 84  
4.2 Using tries to organize the table space. ................................................ 86  
4.3 Detailed tries structure relationships. .................................................... 87  
4.4 Structure of interior, generator and consumer choice points. ................. 88
4.5 The nodes and their interaction with the table and dependency spaces. 89
4.6 Structure of subgoal and dependency frames. 90
4.7 Dependencies between choice points, subgoal and dependency frames. 91
4.8 Pseudo-code for \texttt{adjust_freeze_registers()}. 93
4.9 The forward trail implementation. 94
4.10 Pseudo-code for \texttt{restore_bindings()}. 95
4.11 Spotting the current leader node. 97
4.12 Pseudo-code for \texttt{compute_leader_node()}. 97
4.13 Pseudo-code for \texttt{completion()}. 98
4.14 Pseudo-code for \texttt{answer_resolution()}. 99
4.15 Scheduling for a backtracking node. 101
4.16 Batched versus local scheduling: an example. 104
4.17 Handling generator nodes for supporting batched or local scheduling. 106
4.18 Pseudo-code for \texttt{completion()} with a local scheduling strategy. 107

5.1 Exploiting parallelism in the OPT model. 114
5.2 Exploiting parallelism in the TOP model. 115
5.3 Which is the leader node? 119
5.4 CAT's incremental completion for parallel tabling. 120

6.1 Using memory pages as the basis for the parallel data area. 125
6.2 Inside the parallel data area pages. 126
6.3 Pseudo-code for \texttt{alloc_struct()}. 127
6.4 Pseudo-code for \texttt{free_struct()}. 128
6.5 Pseudo-code for \texttt{get_struct()}. 129
6.6 Detailing the trie structure organization. .................................................. 131
6.7 trie_node_check_insert() call sequence for the new answer operation. 132
6.8 Pseudo-code for trie_node_check_insert() .............................................. 133
6.9 Pseudo-code for trie_node_check_insert() with a TLNL scheme. .......... 135
6.10 Pseudo-code for trie_node_check_insert() with a TLWL scheme. ....... 136
6.11 Pseudo-code for trie_node_check_insert() with a TLWL-ABC scheme.137
6.12 New data fields for the or-frame data structure. ................................. 139
6.13 Spotting the generator dependency node. ....................................... 142
6.14 Modified pseudo-code for compute_leader_node(). ......................... 143
6.15 The generator dependency node inconsistency. ................................... 144
6.16 Dependency frames in the parallel environment. ............................. 145
6.17 The flow of control in a parallel tabled evaluation. ....................... 147
6.18 Pseudo-code for public_completion() ............................................. 149
6.19 Pseudo-code for answer_resolution() ............................................. 152
6.20 Scheduling for a backtracking node in the parallel environment. ....... 153
6.21 Pseudo-code for getwork(). ......................................................... 155
6.22 From getwork to public completion. .............................................. 156
6.23 Suspending a SCC. ........................................................................ 157
6.24 Using or-frames to link suspended SCCs. ......................................... 159
6.25 Resuming a SCC. ........................................................................ 160
6.26 Scheduling for the nearest node with unexploited alternatives. ....... 162
6.27 Pseudo-code for move_up_one_node() ............................................. 164
6.28 Local scheduling situation requiring special implementation support. .. 166
6.29 Pseudo-code for getwork() with a local scheduling strategy. .......... 167
6.30 Pseudo-code for answer_resolution() with a local scheduling strategy. 167

6.31 Pseudo-code for public_completion() with a local scheduling strategy. 168

7.1 Cut semantics overview. ................................................. 172

7.2 Pruning in the parallel environment. ..................................... 174

7.3 Pruning useless work as early as possible. ........................... 176

7.4 Search tree representation. ................................................. 177

7.5 Dealing with pending answers. .......................................... 178

7.6 The two types of cut operations in a tabling environment. .......... 180

7.7 Pseudo-code for clause_with_cuts(). .................................... 182

7.8 Pseudo-code for speculative_tabled_answer(). ...................... 183

7.9 Dealing with pending tabled answers. .................................. 184
Chapter 1

Introduction

Logic programming provides a high-level, declarative approach to programming. Arguably, Prolog is the most popular and powerful logic programming language. Throughout its history, Prolog has demonstrated the potential of logic programming in application areas such as Artificial Intelligence, Natural Language Processing, Knowledge Based Systems, Machine Learning, Database Management, or Expert Systems. Prolog's popularity was sparked by the success of the sequential execution model presented in 1983 by David D. H. Warren, the *Warren Abstract Machine (WAM)* [109]. The WAM compilation technology proved to be highly efficient and Prolog systems have been shown to run logic programs nearly as fast as equivalents C programs [85].

Prolog programs are written in a subset of First-Order Logic, Horn clauses, that has an intuitive interpretation as positive facts and rules. Programs use the logic to express the problem, whilst questions are answered by a resolution procedure with the aid of user annotations. The combination was summarized by Kowalski's motto [59]:

\[
\text{algorithm} = \text{logic} + \text{control}
\]

Ideally, one would want Prolog programs to be written as logical statements first, and for control to be tackled as a separate issue. In practice, the operational semantics of Prolog is given by SLD resolution [61], a refutation strategy particularly simple that matches current stack based machines particularly well. Unfortunately, the limitations of SLD resolution mean that Prolog programmers must be concerned with SLD semantics throughout program development. For instance, it is in fact quite possible that logically correct programs will enter infinite loops.
Several proposals have been put forth to overcome some of the SLD limitations and therefore improve the declarativeness and expressiveness of Prolog. One such proposal that has been gaining in popularity is the use of tabling (or tabulation or memoing [65]). In a nutshell, tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. It can be shown that tabling based models are able to reduce the search space, avoid looping, and have better termination properties than SLD based models. In fact, it has been proven that termination can be guaranteed for all programs with the bounded term-size property [20].

Work on SLG resolution [20], as implemented in the XSB logic programming system [44], proved the viability of tabling technology for applications such as Natural Language Processing, Knowledge Based Systems and Data Cleaning, Model Checking, and Program Analysis. SLG resolution also includes several extensions to Prolog, namely support for negation [10], hence allowing for novel applications in the areas of Non-Monotonic Reasoning and Deductive Databases.

One of the major advantages of logic programming is that it is well suited for parallel execution. The interest in the parallel execution of logic programs mainly arose from the fact that parallelism can be exploited implicitly from logic programs. This means that parallel execution can occur automatically, that is, without input from the programmer to express or manage parallelism, hence making parallel logic programming as easy as logic programming.

Logic programming offers two major forms of implicit parallelism, Or-Parallelism and And-Parallelism. Or-parallelism results from the parallel execution of alternative clauses for a given predicate goal, while and-parallelism stems from the parallel evaluation of subgoals in an alternative clause. Arguably, or-parallel systems, such as Aurora [62] and Muse [6], have been the most successful parallel logic programming systems so far. Experience has shown that or-parallel systems can obtain very good speedups for applications that require search. Examples can be found in application areas such Parsing, Optimization, Structured Database Querying, Expert Systems and Knowledge Discovery applications. Parallel search can be also useful in Constraint Logic Programming.

Tabling works for both deterministic and non-deterministic applications, but it has frequently been used to reduce the search space. This rises the question of whether
further efficiency improvements may be achievable through parallelism. Freire and colleagues were the first to propose that tabled goals could indeed be a source of implicit parallelism [39]. In their model, each tabled subgoal is computed independently in a separate computational thread, a generator thread. Each generator thread is the sole responsible for fully exploiting its subgoal and obtain the complete set of answers. We argue that this model is limitative in that it restricts parallelism to concurrent execution of generator threads. Parallelism arising from non-tabled subgoals or from alternative clauses to tabled subgoals should also be exploited.

1.1 Thesis Purpose

Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for tabling and parallel systems. An interesting observation is that tabling is still about exploiting alternatives to find answers for goals. Our suggestion is that we should aim at using the same technique to exploit parallelism from both tabled and non-tabled subgoals. By doing so we can both extract more parallelism, and reuse the mature technology for tabling and parallelism. Towards this goal, we designed two new computational models [79], the Or-Parallelism within Tabling (OPT) and Tabling within Or-Parallelism (TOP) models. The models combine tabling with or-parallelism by considering all open alternatives to subgoals as being amenable to parallel exploitation, be they from tabled or non-tabled subgoals.

This thesis addresses the design, implementation and evaluation of OPTYap [84]. OPTYap is an or-parallel tabling system based on the OPT model [80] that, to the best of our knowledge, is the first available system that can exploit parallelism from tabling applications. The OPT model considers tabling as the base component of the parallel system. Each worker\(^1\) behaves like a sequential tabling engine that fully implements all the tabling operations. The or-parallel component of the system is triggered to allow synchronized access to common parts of the search space or to schedule workers running out of alternatives to exploit. We take advantage of the hierarchy of or-parallelism within tabling to structure OPTYap’s design and thus simplify its implementation.

\(^1\)The term worker is widely used in the literature to designate each computational unit or agent involved in the parallel environment. A worker is the abstract notion that represents, at the machine level, a system processor or process.
We validate our design through a performance study of the OPTYap system builds on the YapOr [78, 81] and YapTab [83, 82] engines. YapOr is an or-parallel engine that extends Yap’s efficient sequential engine [29] to exploit implicit or-parallelism in Prolog programs. It is based on the environment copy model, as first implemented in the Muse system [5]. YapTab is a sequential tabling engine that extends Yap’s execution model to support tabled evaluation for definite programs, that is, for programs not including negation. YapTab’s implementation is largely based on the ground-breaking work for the XSB system [88, 76], and specifically on the SLG-WAM [86, 89, 87]. YapTab has been designed from scratch and its development was done taking into account the major purpose of further integrate it to achieve an efficient parallel tabling computational model, whilst comparing favorably with current state of the art technology. In other words, we aim at developing an or-parallel tabling system that, when executed with a single worker, runs as fast or faster than the current available sequential tabling systems. Otherwise, the parallel performance results would not be significant and fair, and thus it would be hard to evaluate the efficiency of the parallel implementation.

We intend with our work to study and understand the implications of combining tabling with or-parallelism and thereby develop an efficient execution framework to exploit maximum parallelism and obtain good performance results. Accordingly, the thesis presents novel data structures, algorithms and implementation techniques that efficiently solve some difficult problems arising with the integration of both paradigms. Our major contributions include the dependency frame data structure; the generator dependency node concept and a novel algorithm to compute and detect leader nodes; a novel termination detection scheme to allow public completion; support for suspension of strongly connected components; improvements to scheduling technology; implementation techniques to deal with concurrent table access; and support for speculative tabled answers.

In order to substantiate our claims we studied in detail the performance of our or-parallel tabling engine, OPTYap, up to 32 workers. First, we evaluate the sequential and parallel behavior of OPTYap for non-tabled programs and compare it with that of Yap and YapOr. We then evaluate OPTYap with tabled programs and study its performance for sequential and parallel execution. The gathered results show that OPTYap introduces low overheads for sequential execution and that it compares favorably
with current versions of XSB. Furthermore, the results show that OPTYap maintains YapOr's speedups for parallel execution of non-tabled programs, and that there are tabled applications that can achieve very high performance through parallelism. In our study we gathered detailed statistics on the execution of each benchmark program to help us in understanding and explaining some of the parallel execution results.

Ultimately, this thesis aims at substantiating our belief that tabling and parallelism can together contribute to increasing the range of applications for Logic Programming.

1.2 Thesis Outline

The thesis is structured in nine major chapters that, in some way, reflect the different phases of the work. We provide a brief description of each chapter next.

Chapter 1: Introduction. Is this chapter.

Chapter 2: Logic Programming, Parallelism and Tabling. Provides a brief introduction to the concepts of logic programming, parallel logic programming, and tabling, focusing on Prolog, or-parallelism, SLG resolution, and abstract machines for standard Prolog and tabling, namely the WAM and the SLG-WAM.

Chapter 3: YapOr: The Or-Parallel Engine. Presents the design and implementation of the YapOr Prolog system. It introduces the general concepts of the environment copying model, and then describes the major implementation issues to extend the Yap Prolog system to support the model. Most of YapOr's development was prior to the present work.

Chapter 4: YapTab: The Sequential Tabling Engine. First, it briefly describes the fundamental aspects of the SLG-WAM abstract machine, and then details YapTab's implementation. This includes discussing the motivation and major contributions of the YapTab design, and presenting the main data areas, data structures and algorithms to extend the Yap Prolog system to support sequential tabling. YapTab has been designed and implemented from scratch and its development was the first step towards the current or-parallel tabling system.

Chapter 5: Parallel Tabling. In this chapter we propose two new computational models, OPT and TOP, to efficiently implement the parallel evaluation of tabled
logic programs. Initially, we describe related work to get an overall view of alternative approaches to parallel tabling. Next, we introduce and detail the fundamental aspects underlying the new computational models, and then we discuss their advantages and disadvantages. At last, we focus on the OPT computational model in order to discuss its implementation framework.

Chapter 6: OPTYap: The Or-Parallel Tabling Engine. Presents the implementation details for the OPTYap engine. We start by presenting an overall view of the main issues involved in the implementation of the or-parallel tabling engine and then we introduce and detail the new data areas, data structures and algorithms used to implement it.

Chapter 7: Speculative Work. Discusses the problems arising with speculative computations and introduces the mechanisms used in YapOr and OPTYap to deal with them. Initially, we introduce the cut semantics and its particular behavior within or-parallel systems. Next, we present the cut scheme implemented in YapOr and then discuss speculative tabling execution and present the support currently implemented in OPTYap.

Chapter 8: Performance Analysis. In this chapter we assess the efficiency of the or-parallel tabling implementation by presenting a detailed performance analysis. We start by reporting an overall view of the overheads of supporting the several Yap extensions: YapOr, YapTab and OPTYap. Next we compare OPTYap’s performance with that of YapOr on a similar set of non-tabled programs. Then we use a set of tabled programs to measure the sequential behavior of YapTab, OPTYap and XSB, and to assess OPTYap’s performance when running the tabled programs in parallel. At last, we study the impact of using different locking schemes to deal with concurrent accesses to the table space data structures.

Chapter 9: Concluding Remarks. Discusses the research, summarizes the contributions and suggests directions for further work.

Chapters 4, 6 and 7 include pseudo-code for some important procedures. In order to allow an easier understanding of the algorithms being presented in such procedures, the code corresponding to potential optimizations or synchronizations is never included, unless its inclusion was essential for the description. The Prolog code for the set of benchmarks used in Chapter 8 is included, at the end of the thesis, as Appendix A.
Chapter 2

Logic Programming, Parallelism and Tabling

The aim of this chapter is to provide a brief overview of the research areas embraced by this thesis, highlighting the main ideas behind the key aspects of each area. We discuss logic programming, parallel logic programming and tabling. Throughout, we focus on Prolog, or-parallelism, SLG resolution, and abstract machines for standard Prolog and tabling, namely in the WAM and the SLG-WAM.

2.1 Logic Programming

Logic programming languages, together with functional programming languages, form a major class of languages called declarative languages. A common characteristic of both groups of languages is that they have a strong mathematical basis. Logic programming languages are based on the predicate calculus, while functional programming languages are based on the lambda calculus.

Declarative languages are considered to be very high-level languages when compared with conventional imperative languages because, generally, they allow the programmer to concentrate more on what the problem is, leaving much of the details of how to solve the problem to the computer. The mathematical basis of such languages makes programming an easier task. The programmer can specify the problem at a more application-oriented level and thus simplify the formal reasoning about it.
Logic programming [61] is a programming paradigm based on Horn Clause Logic, a subset of First Order Logic. Logic programming is a simple theorem prover that given a theory (or program) and a query, uses the theory to search for alternative ways to satisfy the query. Logic programming is often mentioned to include the following major features [56]:

- Variables are *logical* variables which can be instantiated *only once*.
- Variables are *untyped* until instantiated.
- Variables are instantiated via *unification*, a pattern matching operation finding the most general common instance of two data objects.
- At unification failure the execution *backtracks* and tries to find another way to satisfy the original query.

Carlsson [17] claims that Logic programming languages, such as Prolog, are cited to include the following advantages:

**Simple declarative semantics.** A logic program is simply a collection of predicate logic clauses.

**Simple procedural semantics.** A logic program can be read as a collection of recursive procedures. In Prolog, for instance, clauses are tried in the order they are written and goals within a clause are executed from left to right.

**High expressive power.** Logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.

**Inherent non-determinism.** Since in general several clauses can match a goal, problems involving search are easily programmed in these kind of languages.

These advantages lead to compact code that is easy to understand, program and transform. Furthermore, they make logic programming languages very attractive for the exploitation of implicit parallelism.
2.1. LOGIC PROGRAMMING

2.1.1 Logic Programs

A logic program consists of a collection of Horn clauses. Using Prolog's notation, each clause may be a rule of the form

\[ A : \neg B_1, ..., \neg B_n. \]

where \( A \) is the head of the rule and the \( B_1, ..., B_n \) are the body subgoals, or it may be a fact and simply written as

\[ A. \]

Rules represent the logical implication

\[ \forall (B_1 \land ... \land B_n \rightarrow A) \]

while facts assert \( A \) as true. A separate type of clauses are those where the head goal is false. These type of clauses are called queries and, in Prolog, they are written as

\[ : \neg B_1, ..., \neg B_n. \]

A goal is a predicate applied to a number of terms (or arguments) of the form

\[ p(t_1, ..., t_n) \]

where \( p \) is the predicate name, and the \( t_1, ..., t_n \) are the terms used as arguments. Each term can be either a variable, an atom, or a compound term of the form \( f(u_1, ..., u_m) \) where \( f \) is a functor and the \( u_1, ..., u_m \) are themselves terms. Terms in a program represent world objects while predicates represent relationships among those objects. Variables represent unspecified terms while atoms represent symbolic constants.

Information from a logic program is retrieved through query execution. The execution of a query \( Q \) against a logic program \( P \), leads to consecutive assignments of terms to the variables of \( Q \) till a substitution \( \theta \) satisfied by \( P \) is found. A substitution is a function that given a variable of \( Q \) returns a term assignment. Answers (or solutions) for \( Q \) are retrieved by reporting for each variable \( X \) in \( Q \) the corresponding assignment \( \theta(X) \). When a variable \( X \) is assigned a term \( T \), then \( X \) is said to be bound and \( T \) is called the binding of \( X \). A variable can be bound to another different variable or to a non-variable term.

Execution of a query \( Q \) with respect to a program \( P \) proceeds by reducing the initial conjunction of subgoals of \( Q \) to another conjunctions of subgoals according to a
resolution rule. The resolution rule of Prolog, Selective Linear Definite resolution (SLD resolution) [61], is a simplified version of the general inference rule that result from the pioneering work on resolution by Robinson [77]. SLD resolution proceeds as follows:

- Let us assume that

\[ : \neg G_1, \ldots, \neg G_n. \]

is the current conjunction of subgoals. Initially and according to a predefined select literal rule, a subgoal (or literal) \( G_i \) is selected.

- Assuming that \( G_i \) is the selected subgoal, then the program is searched for a clause whose head goal unifies with \( G_i \). If there are such clauses then, according to a predefined select clause rule, one is selected.

- Consider that

\[ A : \neg B_1, \ldots, \neg B_m. \]

is the selected clause that unifies with \( G_i \). The unification process has determined a substitution \( \theta \) to the variables of \( A \) and \( G_i \) such that \( A\theta = G_i\theta \). Execution may proceed by replacing \( G_i \) with the body subgoals of the selected clause and by applying \( \theta \) to the variables of the resulting conjunction of subgoals

\[ : \neg (G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta. \]

Notice that in the case the selected clause be a fact, \( G_i \) is simply removed from the conjunction of subgoals

\[ : \neg (G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n)\theta. \]

- When there are no clauses unifying with the selected subgoal, then a failure occurs. Failures are resolved through applying a backtracking mechanism. Backtracking exploits alternative execution paths by (i) undoing all the bindings made since the preceding selected subgoal \( G_p \), and by (ii) reducing \( G_p \) with the next available clause unifying with it.

- The computation stops either when all alternatives have been exploited or when an answer is found. An answer is found whenever the conjunction of subgoals is reduced to the true subgoal, which therefore corresponds to the determination of a query substitution satisfied by the program.
In [59], Kowalski stated that logic programming is about expressing problems as logic and using a resolution procedure to obtain answers from the logic. Therefore, in a computer implementation, the select literal and select clause rules, that is the resolution procedure, must be specified. Different specifications lead to different algorithms and different languages (or semantics) can thus be obtained. Next, we introduce the logic programming language Prolog.

2.1.2 The Prolog Language

Prolog is the most popular logic programming language. The name Prolog was invented by Colmerauer as an abbreviation for PROgramation en LOGic to refer to a software tool designed to implement a man machine communication system in natural language [24].

The pioneering work on resolution by Robinson, culminated in 1965 with the publication of his historical paper [77] describing the now well known general inference rule, Resolution with Unification. Starting from Robinson’s work, it was Kowalski [58] and Colmerauer and colleagues [24] who first recognized the procedural semantics of Horn clauses and provided some theoretical background showing that Prolog can be read both procedurally and logically.

In 1977, David H. D. Warren made Prolog a viable language by developing the first compiler for Prolog [108]. This helped to attract a wider following to Prolog and made the syntax used in this implementation the de facto Prolog standard. In 1983, Warren proposed a new abstract machine for executing compiled Prolog code [109] that has come to be known as the Warren Abstract Machine, or simply WAM. The WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM’s technology.

The interest in logic programming has increased considerably when the Japanese announced their Fifth Generation project. As a result, since then, many different sequential and parallel models were proposed and implemented. The advances made in the compilation technology of sequential implementations of Prolog proved to be highly efficient which has enabled Prolog compilers to execute programs nearly as fast as the conventional programming languages like C [85].
The operational semantics of Prolog is given by SLD resolution, as it matches current computer architectures particularly well. Prolog applies SLD resolution by specifying that the \textit{select literal} rule chooses the leftmost subgoal in a query, and that the \textit{select clause} rule follows the textual order of the clauses in the program. To make Prolog a useful programming language for real world problems, some additional features not found within first order logic were introduced. These features include:

\textbf{Meta-logical predicates.} These predicates inquire the state of the computation and manipulate terms.

\textbf{Cut predicate.} This predicate adds a limited form of control to the execution. It prunes unexploited alternatives from the computation.

\textbf{Extra-logical predicates.} These are predicates which have no logical meaning at all. They perform input/output operations and manipulate the Prolog database, by adding or removing clauses from the program being executed.

\textbf{Other predicates.} These include \textit{arithmetic} predicates to perform arithmetic operations, \textit{term comparison} predicates to compare terms, \textit{extra control} predicates to perform simple control operations, and \textit{set} predicates that give the complete set of answers for a query.

An important aspect of many of these predicates is that their behavior is \textit{order-sensitive}. This means that they can potentially produce different outcomes if different \textit{select literal} or \textit{select clause} rules are specified. Moreover, the use of some of these predicates relies on a deep knowledge of Prolog execution. For readers not familiar with Prolog, a more detailed presentation of these topics can be found in some of the standard textbooks on Prolog, such as [23, 61, 95].

\section*{2.1.3 The Warren Abstract Machine}

Prolog became the most popular logic programming language largely due to the success of its efficient implementations based on the Warren Abstract Machine (WAM) [109]. Currently, most of the \textit{state of the art} systems for logic programming languages still rely on WAM's technology.
The WAM is a stack-based architecture with simple data structures and a low-level instruction set. At any time, the state of a computation is obtained from the contents of the WAM data areas, data structures and registers. See Figure 2.1 for a detailed illustration of the WAM's organization.

![WAM Memory Layout Diagram](image)

Figure 2.1: WAM memory layout, frames and registers.

The WAM defines the following execution stacks:

**PDL:** The PDL is a push down list used by the unification process.

**Trail:** Organized as an array of addresses, it stores the addresses of the (stack or heap) variables which must be reset upon backtracking. The TR register always points to the top of this stack.

**Stack:** Also mentioned as *local stack*, it stores the environment and choice point frames:

- Environments track the flow of control in a program. An environment is pushed onto the stack whenever a clause containing several body subgoals is picked for execution, and it is popped off before the last body subgoal
gets executed. An environment frame consists of the stack address of the previous environment, to reinstate if popped off; the code address of the next instruction, to execute upon successful return from the invoked clause; and a sequence of cells, as many as the number of permanent variables\(^1\) in the body of the invoked clause. The E register points to the current active environment.

- Choice points store open alternatives. A choice point contains all the necessary data to restore the state of the computation back to when the clause was entered; plus a pointer to the next clause to try, in case the current one fails. A choice point frame is pushed onto the stack when a goal is called for execution and has more than one candidate clause. It is popped off when the last alternative clause is taken for execution. The B register points to the current active choice point, which is always the last.

Note that some WAM implementations, like XSB [44] and SICStus Prolog [18], use separate execution stacks to store environments and choice points.

**Heap:** Sometimes also referred as *global stack*, it is an array of data cells used to store variables and compound terms that cannot be stored in the stack. The H register points to the top of this stack.

**Code Area:** Contains the WAM instructions comprising the compiled form of the loaded programs.

Figure 2.1 mentions other important WAM registers: the S register that is used during unification of compound terms; the HB register that is used to determine if a binding is conditional or not\(^2\); the P register that points to the current WAM instruction being executed; and the CP register points to where to return to after successful execution of the current invoked call.

Four main groups of instructions can be enumerated in the WAM instruction set:

**Choice point instructions:** As the name indicates these instructions manipulate choice points. They allow to allocate/remove choice points and to recover the state of a computation through the data stored in choice points.

---

\(^1\)A permanent variable is a variable which occurs in more than one body subgoal [1].

\(^2\)Conditional bindings are discussed next in subsection 2.2.1.
2.2. PARALLELISM IN LOGIC PROGRAMS

Control instructions: allocate/remove environments and manage the call/return sequence of subgoals.

Unification instructions: These instructions implement specialized versions of the unification algorithm according to the position and type of the arguments. There are proper unification instruction to perform head unification, to perform subargument unification, and to prepare arguments for subgoals. These three major classes are further subdivided in specialized versions to treat first occurrence of variables in a clause, non-first occurrences, constants in the clause, lists, and other compound terms.

Indexing instructions. These type of instructions accelerate the process of determining which clauses unify with a given subgoal call. Depending on the first argument of the call, they jump to specialized code that can directly index the unifying clauses.

The apparent simplicity of WAM hides several intricate implementation issues. Complete books, such as Aît-Kaci’s tutorial on the WAM [1], discuss these topics.

2.2 Parallelism in Logic Programs

Traditional implementations of Prolog were designed for common, general-purpose sequential computers. In fact, WAM based Prolog compilers proved to be highly efficient for standard sequential architectures and have helped to make Prolog a popular programming language. The efficiency of sequential Prolog implementations and the declarativeness of the language have kindled interest on implementation for parallel architectures. In these systems, several processors work together to speedup the execution of a program. Parallel implementations of Prolog should obtain better performance for current programs, whilst expanding the range of applications we can solve with this language.

The following main forms of implicit parallelism can be identified in logic programs:

Or-parallelism: Appears from the non-determinism of the select clause rule, when a subgoal call unifies with more than one of the clauses defining the subgoal
predicate. It corresponds to the parallel execution of the bodies of the alternative matching clauses. Or-parallelism is thus an efficient way of searching for alternative answers to the query.

**And-parallelism:** Appears from the non-determinism of the \( \text{select}_{\text{literal}} \) rule, when more than one subgoal is present in the query or in the body of a clause. It corresponds to the parallel execution of such subgoals. Two main forms of and-parallelism are known [49]:

- **Independent and-parallelism:** Occurs when the subgoals, in the query or in the body of a clause, do not share unbound variables. This guarantees that potential bindings for the variables in each subgoal are compatible with the outcome bindings from the other subgoals.

- **Dependent and-parallelism.** Occurs when the subgoals, in the query or in the body of a clause, have common unbound variables. Notice that the parallel execution of such subgoals can lead to incompatible bindings to the common variables. Two major approaches arise: (i) the dependent subgoals only execute simultaneously until one of them binds a common variable. As an alternative, it is possible to continue executing the subgoals even after a common variable has been bound, but in such case, the bindings produced have to be checked for compatibility at the end; or (ii) the dependent subgoals are executed independently and once a common variable is bound by a subgoal, called the *producer*, the other subgoals, called the *consumers*, read the binding as an input argument for the variable. Parallelism can be further exploited by having the producer computing alternative bindings for the common variable and the consumers computing with a particular binding.

**Unification parallelism.** Appears during the process of unifying the arguments of a subgoal with those of a head clause for the predicate with the same name and arity. The different argument terms can be unified in parallel as can the different sub-terms in a term [11]. Unification parallelism is very fine grained and has not been the major focus of research in parallel logic programming.

Original research on the area resulted in several different proposals that successfully supported these forms of parallelism. Arguably, some of the most well-known systems
are: Aurora [62] and Muse [6] for or-parallelism; &-Prolog [54] and &ACE [68, 69] for independent and-parallelism; DASWAM [91, 90] and ACE [67] for dependent and-parallelism; and Andorra-I [33, 113] for or-parallelism together with dependent and-parallelism. A complete and detailed presentation of such systems and the challenges and problems in their implementation can be found in [49].

Intuitively, as each form of parallelism explores different points of non-determinism in the operational semantics of the language, it should be possible to exploit all of them simultaneously. The overall principle in the design of a parallel system that exploits several forms of parallelism simultaneously is orthogonality [27]. In an orthogonal design, each form of parallelism should be exploited without affecting the exploitation of the other. However, no efficient parallel system has been built yet that achieves this, because practical experience has shown that this orthogonality is not so easily translatable to the implementation level. A system extracting maximum parallelism from logic programs while achieving the best possible performance is the ultimate goal of researchers in parallel logic programming.

### 2.2.1 Or-Parallelism

Of the forms of parallelism available in logic programs, or-parallelism is arguably one of the most successful. Intuitively, in a first step, or-parallelism seems easier and more productive to exploit implicitly than and-parallelism. As referred by Lusk *et al.* [62] the main advantages of exploiting or-parallelism are:

**Generality.** It is relatively straightforward to exploit or-parallelism without restricting the power of the logic programming language. In particular, or-parallelism can profit from Prolog's adequacy to generate all answers to a query.

**Simplicity.** Or-parallelism can be exploited without requiring any extra programmer annotation or any complex compile-time analysis.

**Closeness to Prolog.** Implementation technology for Prolog sequential execution can be easily extended to cope with or-parallelism. This means that one can easily preserve the language semantics and take full advantage of existing implementation technology to achieve high performance for a single worker.
Granularity. Or-parallelism offers good potential to be exploited in Prolog programs. For a large class of Prolog programs, the grain size of an or-parallel computation, that is, the potential amount of or-parallel work that can be performed without interaction with other pieces of work proceeding in parallel, is coarse grain [100, 56].

Applications. Or-parallelism arises in a wide range of applications, namely for applications in the general area of Artificial Intelligence involving detection of all answers or large searches, whether it be exercising the rules of an expert system, proving a theorem, parsing a natural language sentence, or answering a database query.

These are arguably the main reasons why most of the research towards implicit parallel Prolog systems starts from or-parallelism. The issues raised in attempting to exploit several forms of parallelism are sufficiently complex that most research efforts are focusing primarily on one single form. The least complexity of or-parallelism makes its implementation more attractive as a first step.

Intuitively, or-parallelism seems easy to implement as the various alternative branches of the search tree are independent of each other, therefore requiring minimum synchronization between them. However, practice has shown that implementation of or-parallelism is not an easy task. Two major problems must be addressed when exploiting or-parallelism: (i) multiple binding representation and (ii) work scheduling.

Multiple Binding Representation

The multiple binding representation is a crucial problem for the efficiency of an or-parallel system. The concurrent execution of alternative branches of the search tree can result in several conflicting bindings for shared variables. The environments of alternative branches have to be organized in such a way that conflicting bindings can easily be discernible. A binding of a variable is said to be conditional if the variable was created before the last choice point, otherwise it is said unconditional. The main problem in the management of multiple environments is that of efficiently representing and accessing conditional bindings, since unconditional bindings can be treated as in normal sequential execution.
2.2. PARALLELISM IN LOGIC PROGRAMS

Essentially, the problem of multiple environment management is solved by devising a mechanism where each branch has some private area where it stores its conditional bindings. A number of approaches have been proposed to tackle this problem [110, 50]. However, each approach has associated costs that are incurred at the time of node\(^3\) creation, at the time of variable access, at the time of variable binding, or at the time of environment switching to start executing a new branch. Gupta and Jayaraman [50] claim that, for an ideal or-parallel system, the cost of all these operations should be \textit{constant time}. The term constant time is used to mean that the time for these operations is independent of the number of nodes in the or-parallel search tree, as well as the number of goals and the size of terms that appear in goals [48]. However, Gupta and Jayaraman [50] conjectured that it is impossible to executed all operations in constant time because the cost of at least one of this operations will increase by a non-constant overhead. More recently, this intuitive result has been formally proved to hold by Ranjan \textit{et al.} [75].

\textbf{Work Scheduling}

Even though the cost of managing multiple environments cannot be completely avoided, it may be minimized by the or-parallel system if it is able to divide efficiently the available work during execution. The system component responsible for finding and distributing parallel work to available workers is known as the \textit{scheduler}. Work scheduling is a complex problem because of the dynamic nature of work in or-parallel systems, as in fact, unexploited branches arise irregularly. To efficiently deal with this irregularity, careful scheduling strategies are required. Several different strategies have been proposed to tackle this problem [16, 5, 12, 94, 93].

Two major policies are known to dispatch work for or-parallel execution: (\textbf{\textit{i}}) topmost and (\textbf{\textit{ii}}) bottommost. In the topmost policy, when an idle worker asks for work, only a restricted number of nodes with available work is made \textit{public}\(^4\). The nodes are made public in sequence, starting from the root node, and the number of nodes is selected according to the scheduler's strategy. The topmost policy leads to bigger private regions of the search tree, which intuitively one would expect to correspond to an

\(^3\)A node is the abstract notion that is implemented at the engine level as a choice point.

\(^4\)A node is called \textit{public} when it is shared by several workers. Otherwise, when belonging to an unique worker, it is called \textit{private}.
increase in the granularity of an or-parallel computation. In contrast, the bottommost policy turns public the whole private region of a worker when it shares work. This maximizes the amount of shared work and possibly avoids that the requesting worker runs out of work too early and therefore invokes the scheduler too often. Practice showed that, for shared memory parallel systems, bottommost is the best policy to dispatch work for or-parallel execution and thus achieve higher granularity of or-computations.

A major problem for scheduling is the presence of pruning operators like the cut predicate. When a cut predicate is executed, all alternatives to the right of the cut are pruned, therefore never being executed in a sequential system. However, in a parallel system, the work corresponding to these alternatives can be early picked for parallel execution, therefore resulting in wasted computational effort when pruning takes place. This form of work is known as speculative work. Giving higher scheduling priority to work on the left part of the search tree is a way of reducing the probability of further pruning. An advanced scheduler must be able to reduce to a minimum the speculative computations and at the same time maintain the granularity of the work scheduled for execution [8, 13, 94]. The speculative work problem is discussed in detail in Chapter 7.

2.2.2 Or-Parallel Execution Models

A number of execution models have been proposed in the literature towards exploiting or-parallelism (a detailed analysis of about 20 models can be found in [50]). These models mainly differ in the mechanism employed for solving the problem of environment representation. Arguably, the two most successful ones are environment copying [6, 5], as implemented in the Muse system, and binding arrays [112, 111], as implemented in the Aurora system.

In the environment copying model each worker maintains its own copy of the environment in which it can write without causing binding conflicts. In this model even unconditional bindings are not shared. When a variable is bound, the binding is stored in the private environment of the worker doing the binding.

When an idle worker picks work from another worker, it copies all the stacks from the sharing worker. Copying of stacks is made efficient through the technique of
incremental copying. The idea of incremental copying is based on the fact that the idle worker could have already traversed a part of the search tree that is common to the sharing worker, and thus it does not need to copy this part of stacks. Furthermore, copying of stacks is done from the virtual memory addresses of the sharing worker to exactly the same virtual memory addresses of the idle worker, which therefore avoids potential reallocation of address values.

As a result of copying, each worker can carry out execution exactly like a sequential system, requiring very little synchronization with other workers. Synchronization between workers is achieved through a single auxiliary data structure associated with the choice points. In section 3.1 we analyze in detail the environment copying approach to or-parallelism, including its implementation.

On the other hand, in the binding arrays model each worker maintains a private array data structure, called the binding array, where it stores its conditional bindings. Each variable along a branch is assigned to an unique number that identifies its offset entry in the binding array. The numbering of variables is done so that it forms a strict increasing sequence. This is achieved by maintaining a private counter. New variables are always marked with the current value of the counter. Next, the counter is incremented. The counter is saved at every choice point so that whenever a worker gets an alternative branch it can get a copy of the counter and continue its own numbering.

Conditional bindings are stored in the private binding array of the worker doing the binding, at the offset location given by the offset value of that conditional variable. In addition, the conditional binding together with the address of the conditional variable are stored in a global binding tree, that is, the WAM’s trail stack. This global binding tree is then used to ensure consistency when a worker switches from one branch to another, as in such cases, the switching worker has to update its binding array to reflect the bindings of the new branch.

Both models allow constant-time cost for node creation, variable access and for variable binding, but induce non-constant time cost for environment switching.

The success of these models is partly due to the fact that their corresponding systems do support sequential Prolog semantics. A parallel system is said to support sequential
46  CHAPTER 2. LOGIC PROGRAMMING, PARALLELISM AND TABLING

Prolog semantics when it achieves the same effect of sequential execution and supports all the additional features not found within pure Horn Clause Logic [21] (refer to the meta-logical, extra-logical and other predicates of subsection 2.1.2). The advantage of such an approach is that all existing Prolog programs can be taken and executed in parallel without any modifications.

Arguably, copying is the most efficient way to maintain or-parallel environments. Most modern parallel logic programming systems, including SICStus Prolog [18], ECLiPSe [107], and YAP [31] use copying as a solution to the multiple bindings problem. Copying was made popular by the Muse or-parallel system, a system derived from an early release of SICStus Prolog. Muse showed excellent performance results [8, 9, 7, 30, 26, 32] and in contrast to other approaches, it also showed low overhead over the corresponding sequential system. On the other hand, copying has a few drawbacks. First, it can be expensive to exploit more than just or-parallelism with copying, as the efficiency of copying largely depends on copying large contiguous blocks of memory, which is difficult to guarantee in the presence of and-parallelism [51]. A second issue is that copying makes it more expensive to suspend branches during execution, which can be a problem when implementing cuts and side-effects, although Ali and Karlsson [8] proposed solutions to efficiently solve this problem.

2.3 Tabling for Logic Programs

Prolog execution is based on SLD resolution for Horn clauses. This strategy allows efficient implementation, but suffers from fundamental limitations, such as in dealing with infinite loops and redundant subcomputations. These limitations make Prolog unsuitable to important applications such as, for example, Deductive Databases. The limitations of SLD resolution are well known, and extensive efforts have been made to remedy them. One approach is to use resolution strategies similar to SLD, but that can avoid redundant computations by remembering subcomputations and reusing their results in order to respond to later requests. This process of remembering and reuse has been widely called tabling, tabulation or memoing [65]. Tabling methods have been proposed from a number of different starting points and given a number of different names: OLDT [103], SLD-AL [105], Extension Tables [37], and Backchain Iteration [106] are the better known. The tabling concept also forms the basis of
a transformation used with bottom-up evaluation to compute answers for deductive database queries, that is known by the generic name of magic [14, 73].

Another major research direction followed during the past years in order to increase the expressiveness of logic programming, was concerned with the introduction of negation on the body subgoals of clauses. Although the inclusion of negation seems simple, the definition of the declarative semantics of logic programs including negative subgoals is a major problem [10]. One of the most popular resolution methods that includes negation is SLDNF [22], an extension to SLD resolution that supports negation as finite failure. However, this method has not proved to be sufficient for important areas of application, such as Deductive Databases, Non-Monotonic Reasoning and models for executable specifications, such as Model Checking.

A strong drawback of SLDNF results from its inadequacy in handling positive and negative loops. Because tabling methods already address the handling of positive loops, it is natural, then, to extend them to handle negative loops and thereby support frameworks such as the well-founded semantics [43]. The well-founded semantics provides a natural and robust declarative meaning to all logic programs with negation. However, practical use of the well-founded semantics depends upon the implementation of an effective and efficient evaluation procedure.

Although various procedural semantics have been proposed for the well-founded semantics, one such proposal that has been gaining in popularity is Linear resolution with Selection function for General logic programs (SLG resolution) [20]. SLG resolution is a tabling based method of resolution that has polynomial time data complexity and is sound and search space complete for all non-floundering queries under the well-founded semantics. SLG resolution can thus reduce the search space for logic programs and in fact it has been proven that it can avoid looping and thus terminate for all programs with the bounded-term-size property [20]. SLG’s popularity is largely due to the work done on the XSB system [88, 76], and namely on the SLG-WAM [86, 89, 87], the original engine of the XSB system.

Next, we further motivate the need for tabling (or memoing) in a logic programming framework, and then we briefly review the underlying features of SLG resolution and SLG-WAM. At the end, we overview other related implementations of tabling.
2.3.1 Examples of Tabled Evaluation

The basic idea behind tabling is straightforward: programs are evaluated by storing newly found answers of current subgoals in a proper data space, called the table space. The method then uses this table to verify for repeated calls to subgoals. Whenever such a repeated subgoal is found, the subgoal’s answers are recalled from the table instead of being re-evaluated against the program clauses. In the following, we illustrate the advantage of tabling through an example.

Consider the Prolog program of Figure 2.2 that defines a small directed graph (represented by the arc/2 predicate) with a relation of reachability (given by the path/2 predicate), and the query goal \(-\text{path}(a,Z)\).

\[
\begin{align*}
\text{path}(X,Z) & :\text{- path}(X,Y), \text{path}(Y,Z) \\
\text{path}(X,Z) & :\text{- arc}(X,Z) \\
\text{arc}(a,b) & \\
\text{arc}(b,c) & \\
\text{- path}(a,Z) & 
\end{align*}
\]

Figure 2.2: An infinite SLD evaluation.

Applying SLD evaluation to solve the given query goal will lead to an infinite SLD tree due to the existence of positive loops. Regard, for example, what happens when the leftmost branch of the corresponding search tree is exploited. In contrast, if tabling is applied then the search tree is finite, hence termination is ensured. Figure 2.3 illustrates the evaluation sequence for the same program and query goal using tabling.

The figure depicts the evaluation sequence for the given query goal. At the top,
the figure illustrates the program code and the appearance of the table space at the end of the evaluation. Declaration :- table path/2 in the program code indicates that predicate path/2 should be tabled, therefore tabling will be applied to solve its subgoals calls. The bottom block shows the resulting forest of trees for the three tabled subgoal calls. The numbering of nodes denote the evaluation sequence.

Whenever a tabled subgoal is first called, a new tree is added to the forest of trees and
a new entry is added to the table space. On the other hand, *variant calls*\(^5\) to tabled subgoals start, instead, by consuming the answers already stored in the table space for the corresponding subgoal. When all currently available answers are consumed, the execution of the variant subgoal is *suspended* until new answers arise. In Figure 2.3, the former situation is depicted by white oval boxes surrounding the subgoal calls, while the latter is depicted by gray oval boxes.

Let us examine the evaluation in more detail. The evaluation begins by creating a new tree rooted by \(\text{path}(a,Z)\) and by inserting a new entry in the table space for it. Next, \(\text{path}(a,Z)\) is first resolved against the first clause for \(\text{path}/2\), creating node 1. Execution proceeds with \(\text{path}(a,Y)\) and in principle the same procedure should be applied again. However, since \(\text{path}(a,Y)\) is a variant of the initially encountered subgoal \(\text{path}(a,Z)\), no new tree is created, and instead, the execution of node 1 suspends. This happens because currently the subgoal has no answers stored in the table space. Therefore, the only resolution applicable is to node 0 where the second \(\text{path}/2\) clause is tried, thus leading to a first answer for \(\text{path}(a,Z)\). After exhausting all alternatives in node 2, the computation is resumed at node 1 with the newly found answer, which in turn leads to a first call to subgoal \(\text{path}(b,Z)\). The evaluation creates a new tree rooted by \(\text{path}(b,Z)\), inserts a new entry in the table space for it, and proceeds as for the latter case. The process continues, giving rise to one more tree, for subgoal \(\text{path}(c,Z)\), and to more answers, one for \(\text{path}(a,Z)\) and the other for \(\text{path}(b,Z)\).

By avoiding the recomputation of \(p(a,Y)\), \(p(b,Y)\) and \(p(c,Y)\) in nodes 1, 5 and 11, respectively, tabling ensures the termination of the given query. Besides avoiding infinite loops, tabling also reduces the number of steps we need to perform and may reduce the complexity of a program. This later property is better clarified next. Consider, for example, the well-known Fibonacci program as defined in Figure 2.4.

The figure presents, by using a tree structure, the number of calls to the \(\text{fib}/2\) predicate given the query \(?- \text{fib}(5,Z)\), for the cases where SLD or tabled evaluation are applied. As predicate \(\text{fib}/2\) is declared as tabled, each different subgoal call is only computed once, as for repeated calls, the corresponding answer is already stored in the table space. To compute \(\text{fib}(n)\) for some integer \(n\), SLD will search a tree whose size is exponential in \(n\). Because tabling remembers subcomputations, the number of

\(^5\) A call is a variant of another call if the two calls are the same up to variable renaming.
2.3. TABLING FOR LOGIC PROGRAMS

resolution steps for this example is linear in \( n \).

2.3.2 SLG Resolution for Definite Programs

Restricted to the class of definite programs, that is, to the class of programs not including negation, SLG resolution reduces to SLD with tabling, and does not significantly differ from the other tabling evaluation methods previously referred. Remember that the aim of this thesis is to address the problem of or-parallel tabling, focusing on traditional tabling, that is, tabling not extended to include negation.

In the following, we offer a brief review of SLG resolution for definite programs using the simplified definitions from Sagonas and Swift [87]. For a more detailed discussion, the reader is referred to Chen and Warren [20].

Definition 2.1 (SLG System) An SLG system is a forest of SLG trees, along with an associated table. Root nodes of SLG trees are subgoals of tabled predicates. Non-root nodes either have the form \texttt{fail} or

\[ \text{Answer_Template} : \text{Goal_List} \]
The Answer Template is a positive literal, and Goal List is a possibly empty sequence of subgoals. The table is a set of ordered triples of the form

\[ \langle \text{Subgoal}, \text{Answer Set}, \text{State} \rangle \]

where the first element is a subgoal, the second a set of positive literals, and the third either the constant complete or incomplete. □

**Definition 2.2 (SLG Evaluation)** Given a tabled program \( P \), an SLG evaluation \( \theta \) for a subgoal \( G \) of a tabled predicate is a sequence of systems \( S_0, S_1, \ldots, S_n \) such that:

- \( S_0 \) is the forest consisting of a single SLG tree rooted by \( G \) and the table \( \{ \langle G, \emptyset, \text{incomplete} \rangle \} \);
- for each finite ordinal \( k \), \( S_{k+1} \) is obtained from \( S_k \) by an application of one of the operations in definition 2.3.

If no operation is applicable to \( S_n \), \( S_n \) is called a final system of \( \theta \). □

In a SLG system, the nodes of an SLG tree are often described by its status. The root node of an SLG tree has status generator. Non-root nodes may have status answer, if its Goal List is empty; interior, if its selected subgoal is non-tabled; or consumer if its selected subgoal is tabled. Using this terminology, the SLG operations for definite programs are defined as follows.

**Definition 2.3 (SLG Operations for Definite Programs)** Given a tabled program \( P \) and a system \( S_k \) of an SLG evaluation \( \theta \), \( S_{k+1} \) may be produced by one of the following operations.

- **New Tabled Subgoal Call.** Given a consumer node \( N \) with selected subgoal \( S \), where \( S \) is not present in the table of \( S_k \), create a new SLG tree with root \( S \) and add the entry \( \langle S, \emptyset, \text{incomplete} \rangle \) to the table.

- **Program Clause Resolution.** Let \( N \) be a node in \( S_k \) that is either a generator node \( S \) or an interior node of the form

  \[ \text{Answer Template} : - S, \text{Goals}. \]
Let
\[ C = \text{Head} : - \text{Body} \]
be a program clause such that Head unifies with S with substitution \( \theta \) and assume that \( C \) has not been used for resolution at node \( \mathcal{N} \). Then

- if \( \mathcal{N} \) is a generator node, produce a child of \( \mathcal{N} \)
  \[ (S : - \text{Body})\theta. \]
- if \( \mathcal{N} \) is an interior node, produce a child of \( \mathcal{N} \)
  \[ (\text{Answer Template} : - \text{Body, Goals})\theta. \]

**Answer Resolution.** Let \( \mathcal{N} \) be a consumer node

\[ \text{Answer Template} : - S, \text{Goals}. \]

Let \( A \) be an answer for \( S \) in \( \mathcal{S}_k \) and assume that \( A \) has not been used for resolution against \( \mathcal{N} \). Then produce a child of \( \mathcal{N} \)

\[ (\text{Answer Template} : - \text{Goals})\theta. \]

where \( \theta \) is the substitution unifying \( S \) and \( A \).

**New Answer.** Let

\[ A : - \]

be a node in a tree rooted by a subgoal \( S \), such that \( A \) is not an answer in the table entry for \( S \) in \( \mathcal{S}_k \). Then add \( A \) to the set of answers for \( S \) in the table.

**Completion.** If \( \mathcal{C} \) is a set of subgoals that is completely evaluated (according to definition 2.5), remove all trees whose root is a subgoal in \( \mathcal{C} \), and change the state of all table entries for subgoals in \( \mathcal{C} \) from *incomplete* to *complete*. □

**Definition 2.4 (Subgoal Dependency Graph)** Let \( \mathcal{S}_k \) be an SLG system and \( \mathcal{F} \) its SLG forest. We say that a tabled subgoal \( S \) directly depends on a tabled subgoal \( S' \) if and only if the tree rooted by \( S \) contains a consumer node whose selected subgoal is \( S' \).
The Subgoal Dependency Graph of $\mathcal{S}_k$

$$SDG(\mathcal{S}_k) = (V, E)$$

is a directed graph in which $V$ is the set of root goals for trees in $\mathcal{F}$ and $(S, S')$ belongs to $E$ if and only if subgoal $S$ directly depends on subgoal $S'$. □

Since the subgoal dependency graph of a given system is a directed graph, it can be partitioned into Strongly Connected Components, or SCCs. As an artifact of the SLG-WAM, it can happen that the stack segments for a SCC $S$ remain within the stack segments for another SCC $S'$. In such cases, $S$ cannot be recovered in advance when completed, and thus, its recovering is delayed until $S'$ also completes. To approximate SCCs in SLG-WAM's stack-based implementation of SLG resolution, Sagonas [86] denotes a set of SCCs that can be recovered together as an Approximate SCC or ASCC. An ASCC is termed independent if it depends on no other ASCC which it does not contain. This terminology leads to the following operational definition of when a set of subgoals has been completely evaluated.

**Definition 2.5 (Completely Evaluated Set of Subgoals)** Given an SLG system $\mathcal{S}_k$, a set $C$ of subgoals is completely evaluated if and only if either of the following conditions is satisfied:

1. $C$ is an independent ASCC of $SDG(\mathcal{S}_k)$ and for each subgoal $S$ in $C$ all applicable SLG operations other than Completion have been performed for nodes in the tree rooted by $S$ according to definition 2.3.

2. $C = \{S\}$ and $S$ contains an answer identical to itself in the table entry for $S$.

We say that a subgoal $S$ is completely evaluated if and only if $C$ is a completely evaluated set of subgoals and $S$ belongs to $C$. □

For simplicity, throughout the thesis we will not distinguish between SCCs and ASCCs and we will use the SCC notation to refer the approximation resulting from the stack organization. The correct distinction between both notations is necessary when determining negative loops among subgoals in programs with negation, which is not our case.
2.3.3 SLG-WAM: an Abstract Machine for SLG Resolution

SLG resolution has been firstly implemented in the XSB system by extending the WAM into the SLG-WAM, an abstract machine designed to fully integrate Prolog SLD code and tabling SLG code with minimal overhead. The SLG-WAM extends the WAM layout both to include a representation of tables, and to operate over a forest of SLG trees rather than over a single SLD tree. Performance evaluation of the SLG-WAM, as reported in [88, 99], showed that it can compute in-memory recursive queries an order of magnitude faster than current deductive databases systems.

The data structures, data areas, instructions set, and algorithms used by the SLG-WAM for definite programs are described in [98], while extensions to handle normal logic programs according to the well-founded semantics are discussed in [86, 89, 87]. Here, we briefly summarize the main extensions made by the SLG-WAM to the WAM in order to support SLG resolution for definite programs.

1. The SLG-WAM includes a proper space for tables, and the table access methods are tightly integrated with WAM data structures.

2. The SLG-WAM is able to suspend computations when it encounters consumer subgoals and to resume them at a later point to consume newly found answers. The need for suspending and resuming requires efficient mechanisms to restore an environment to the same computational state as it was before being suspended.

3. Since a computation can be resumed in suspended consumer nodes, space for these nodes cannot be reclaimed upon backtracking, but only when the SCC to which they belong is completed. A mechanism was developed to detect completion of SCCs in order to allow early space reclamation.

4. The decision of applying a certain SLG operation to continue an evaluation gives rise to possible alternative scheduling strategies. Such alternatives can influence differently the architecture and performance of the abstract machine. Originally, SLG-WAM had a simple scheduling mechanism, named single stack scheduling, which formed the basis of the XSB system as described in [98]. A detailed description of the operational semantics of single stack scheduling can be found in [97]. Meanwhile, practice showed that single stack scheduling was expensive in terms of trailing and choice point creation, and thus, Freire and colleagues...
proposed two more sophisticated scheduling strategies, named \textit{batched scheduling} and \textit{local scheduling} \cite{40}, to overcome these problems. Since version 1.5, XSB had used batched scheduling as the default strategy, although the last version of XSB (version 2.4 released in July 2001) has adopted local scheduling as the default. Another scheduling strategy that has been evaluated in SLG-WAM was \textit{breadth-first scheduling} \cite{41}. Breadth-first scheduling was proposed by Freire \cite{38} to address the inability of resolution-based systems to deal with applications that require massive amounts of data residing in external databases.

5. The preceding features are implemented by using WAM-like instructions.

We return to these topics in section 4.1 where a complete description of the fundamental aspects underlying the SLG-WAM abstract machine is given.

\section{Other Related Implementations}

More recently, other related mechanisms for tabling have been implemented. Ramesh and Chen \cite{74} implemented a technique based on program transformation to incorporate tabled evaluation into existing Prolog systems. Their approach uses the C language interface, available in most Prolog systems, to implement external tabling primitives that provide direct control over the search strategies for a transformed program. A tabled logic program is transformed to include the tabling primitives through source level transformations, and only the resulting transformed program is compiled. The mechanism is independent from the Prolog's engine which makes it easily portable to any Prolog system with a C language interface.

Demoen and Sagonas proposed a copying approach to deal with tabled evaluations and implemented two different models, the CAT \cite{35} and the CHAT \cite{36}. The main idea of the CAT implementation is that it replaces SLG-WAM's freezing of the stacks by copying the state of suspended computations to a proper separate stack area. The CHAT implementation improves the CAT design by combining ideas from the SLG-WAM with those from the CAT. It avoids copying all the execution stacks that represent the state of a suspended computation by introducing a technique for freezing stacks without using freeze registers. We discuss CAT and CHAT in more detail in subsection 5.2.4.
2.4. **CHAPTER SUMMARY**

Zhou *et al.* [114] and Guo and Gupta [47] implemented tabling mechanisms that work on a single SLD tree without requiring suspensions/resumptions of computations and mechanisms to preserve the state of suspended computations. Zhou *et al.* implements a linear tabling mechanism whose main idea is to let variant calls execute from the remaining clauses of the former first call. The main idea is as follows: when there are answers available in the table, the call consumes the answers; otherwise, it uses the predicate clauses to produce answers. Meanwhile, if a call that is a variant of some former call occurs, it takes the remaining clauses from the former call and tries to produce new answers by using them. The variant call is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a fixpoint is reached. The Guo and Gupta approach [47] is similar. It is based on dynamic reordering of alternatives with variant calls and it uses the alternatives leading to variant calls to repeatedly recompute them until a fixpoint is reached. This approach is discussed in more detail in subsection 5.1.

None of these approaches showed to outperform SLG-WAM performance. The only candidate that actually competes against SLG-WAM is CHAT [36]. It showed comparable (and for some programs better) execution time performance to those of SLG-WAM and lower and more controlled memory consumption than SLG-WAM. However, as will be discussed in subsection 5.2.4 we believe that, considering the further integration with or-parallelism, SLG-WAM is still the better choice for sequential tabling.

### 2.4 Chapter Summary

In order to make this thesis as self-contained as possible, we presented in this chapter a short survey on logic programming, parallel logic programming and tabling.

We gave a brief description of logic programs, the Prolog language and its implementation in the WAM. We discussed parallelism and focused on or-parallelism. We gave emphasis to the problems that must be addressed when exploiting or-parallelism and introduced the environment copying and binding arrays proposals to solve those problems. We motivated for the advantages of tabling in a logic programming framework, and briefly reviewed the underlying features of SLG resolution and SLG-WAM. At the end, we presented other related implementations of tabling.
Chapter 3

YapOr: The Or-Parallel Engine

This chapter describes the design and implementation of the YapOr engine. YapOr is an or-parallel Prolog system that extends the Yap Prolog system to support implicit or-parallelism in Prolog programs. YapOr is based on the environment copying model, as first implemented in the Muse system [6]. The YapOr engine is the basis for the or-parallel component of our combined or-parallel tabling engine.

We start by introducing in further detail the general concepts of the environment copying model, and then we describe the major implementation issues that we addressed in order to extend the Yap Prolog system to support the model.

3.1 The Environment Copying Model

The environment copying model is based on the multi-sequential approach [2, 110]. In this approach, a set of workers are expected to spend most of their computational time performing reductions as sequential engines. When a worker fully exploits its set of available alternatives, it starts looking for unexploited work from fellow workers. Which workers it asks for work and which work it receives is up to the scheduler to decide.
3.1.1 Basic Execution Model

In more detail, a set of workers perform parallel execution of a program. Initially, all but one workers are idle, that is, looking for their first work assignment. A single worker, say $P$, starts executing the initial query as a normal Prolog engine. Whenever $P$ executes a goal that matches several clauses, it creates a private choice point in its local stack to save the state of the computation at predicate entry. This choice point marks the presence of potential work to be performed in parallel.

As soon as an idle worker finds that there is available work in the system, it will request that work directly from the worker owning it. Consider, for example, that worker $Q$ requests work from worker $P$. If $P$ has unexploited work, it will share its private choice points with $Q$. To do so, worker $P$ must turn the choice points public first. In Muse this operation is implemented by allocating special data structures, named or-frames, in a shared space to permit synchronized access to the newly shared choice points. After concluding this operation, worker $P$ will handle $Q$ a pointer to the bottom shared choice point.

The next step is taken by worker $Q$. In order for $Q$ to take a new task, it must copy the computation state from worker $P$ up to the bottom shared choice point. After copying, worker $Q$ must synchronize its status with the newly copied computation state. This is done first by simulating a failure to the bottom choice point, and then by backtracking to the next available alternative within that branch. Worker $Q$ will then start its execution as a normal sequential Prolog engine would.

At some point, a worker will fully explore its subtree and will become idle again. At this point, it will return into the scheduler loop and start looking for busy workers with available work in order to request unexploited work from them. It thus enters the behavior just described for $Q$. Eventually, the execution tree will be fully explored and execution will terminate with all workers idle.

---

1A worker is said to be busy when it is exploiting alternatives. A busy worker is a potential source of unexploited alternatives.
3.1.2 Incremental Copying

The sharing work operation poses a major overhead to the system as it involves copying the full execution stacks between workers. Hence, an incremental copying strategy \[6\] has been devised to minimize this source of overhead.

The main goal of sharing work is to position the workers involved in the operation at the same node of the search tree, leaving them with the same computational state. Incremental copying achieves this goal by allowing the receiving worker to keep the part of its state that is consistent with that of the giving worker. Only the differences between both are copied.

This strategy can be better understood through Figures 3.1 and 3.2. Suppose that worker \( Q \) does not find any available work in its branch\(^2\) (nodes \( N_1, N_2 \) and \( N_5 \)), and that there is a worker \( P \) with unexploited alternatives (in nodes \( N_3 \) and \( N_4 \)). \( Q \) asks \( P \) for sharing and backtracks up to the lowest node (\( N_2 \)) that is common to \( P \), therefore becoming partially consistent with part of \( P \).

![Diagram](image.png)

Figure 3.1: Backtracking to the bottom common node.

If worker \( P \) decides to share its private nodes (\( N_3 \) and \( N_4 \)) with \( Q \), then worker \( Q \) only has to copy the stacks differences between both. These differences are calculated through the register information stored in the common choice point found by \( Q \) and

\(^2\)When we say a worker branch, we mean the current set of nodes of that worker.
through the top registers of the local, heap and trail stacks of $\mathcal{P}$. In Figure 3.2 the stack segments representing the differences to be copied are colored gray. Note that to fully synchronize the computational state between the two workers, worker $\mathcal{Q}$ further needs to install from $\mathcal{P}$ the conditional bindings made to variables belonging to the maintained segments (this is the case of binding VAL 1 made to variable VAR 1). The references to such variables are obtained through consulting the trail entries of the copied trail segment.

![Figure 3.2: Incremental Copying.](image)

### 3.2 The Muse Approach for Scheduling Work

We can divide the execution time of a worker in two modes: scheduling mode and engine mode. A worker enters in scheduling mode whenever it runs out of work and starts searching for available work. As soon as it gets a new piece of work, it enters in engine mode. In this mode, a worker runs like a standard Prolog engine.

The scheduler is the system component that is responsible for distributing the available
work between the various workers. The scheduler must arrange the workers in the search tree in such a way that the total parallel execution time will be the least possible. The scheduler must also maintain the correctness of sequential Prolog semantics. To obtain best performance the scheduler must minimize the scheduling overheads present in operations such as sharing nodes, copying segments of the stacks, backtracking, restoring and undoing previous variable bindings.

3.2.1 Scheduler Strategies

Ali and Karlsson [5] proposed the following scheduler strategies for the Muse implementation:

- When a busy worker shares work, it must share all the private nodes it has at that moment. This will maximize the amount of shared work and possibly avoid that the requesting worker runs out of work too early.

- The scheduler should select the busy workers that are nearest to the idle worker, and from these select the one that holds the highest work load. Being near corresponds to the closest position in the search tree. The work load is a measure of the amount of unexplored private alternatives. This strategy minimizes the stacks parts to be copied and maximizes the amount of shared work.

- To guarantee the correctness of a sharing operation, it is necessary that the idle worker is positioned at a node that belongs to the branch on which the busy worker is working. To minimize overheads, the idle worker backtracks to the bottommost common node before requesting work. This reduces the time spent by the busy worker in the sharing operation.

- If at a certain point in time the scheduler does not find any available work in the system, it moves the idle worker to a better position in the search tree, if some. A better position corresponds to a position where the overheads of a future sharing operation should be lower.

We can resume the scheduler algorithm as follows: when a worker runs out of work it searches for the nearest unexploited alternative in its branch. If there is no such alternative, it selects a busy worker with excess of work load to share work with,
according to the strategies above. If there is no such a worker, the idle worker tries to move to a better position in the search tree.

### 3.2.2 Searching for Busy Workers

There are two alternative searches for busy workers in the execution tree: search within the current subtree or search outside the current subtree\(^3\). Idle workers always start to search within the current subtree, and only if they do not find any busy worker there, will they search outside. The advantages of selecting a busy worker within the current subtree instead of outside are mainly two. One is that the idle worker can immediately make the sharing request, as its current node is already common to the busy worker. This avoids backtracking in the tree and undoing variable bindings. The second advantage is that the idle worker will maintain its relative position in the search tree. This maximizes the portion of the stacks that are common to both workers, as the current stacks of the idle worker are fully common with those of the busy worker, which should minimize the stack segments to be copied. Regard that this scheduling strategy corresponds to the bottommost policy of dispatching work for or-parallel execution.

Figures 3.3 and 3.4 present different situations in order to better illustrate the scheduler strategies to select busy workers positioned respectively within and outside the idle worker current subtree. In these figures, \(Q\) represents the idle worker, \(P\) the busy worker, and the different \(Q_j\)'s other idle workers.

The algorithm to select a busy worker within the current subtree of an idle worker \(Q\) can be resumed as follows. Initially, the scheduler determines the set \(SB\) of busy workers within the current subtree of \(Q\). Then, it removes from \(SB\) each worker \(P\) whose nearest idle worker in \(P\)'s branch is not \(Q\). This guarantees that \(SB\) remains only with busy workers whose nearest idle worker is \(Q\). Finally, from the remaining workers in \(SB\), the scheduler selects the one with the highest work load.

Applying this algorithm to the four situations presented in Figure 3.3, \(Q\) can request work from \(P\) in all situations except (b). In situation (a) despite \(Q_1\) being in a deeper position than \(Q\), \(Q_1\) is not in \(P\)'s branch, and thus \(Q\) is the nearest idle worker to

\(^3\)When we say a worker subtree, we mean the subtree rooted by the current node of that worker.
3.2. THE MUSE APPROACH FOR SCHEDULING WORK

Figure 3.3: Requesting work from workers within the current subtree.

\( \mathcal{P} \). However, this will stop being the case if \( \mathcal{Q}_1 \) backtracks to the previous node. We allow \( \mathcal{Q} \) to request work from \( \mathcal{P} \) because in a more complex case it will be difficult to predict what will be the behavior of the potentially idle workers in the same situation as \( \mathcal{Q}_1 \). In situation (b), \( \mathcal{Q}_2 \) is closer to the busy worker \( \mathcal{P} \) than \( \mathcal{Q} \), and thus, \( \mathcal{Q}_2 \) is the one that should request work from \( \mathcal{P} \). In situation (c), \( \mathcal{Q} \) is the nearest worker to \( \mathcal{P} \) and in situation (d), \( \mathcal{Q} \) and \( \mathcal{Q}_4 \) are equally distant from \( \mathcal{P} \). Thus, both workers can request work from \( \mathcal{P} \). As a worker can only answer a request at a time, the first one making the request is the one that is served.

As mentioned before, an idle worker \( \mathcal{Q} \) searches outside its current subtree only if it cannot request work within. To select busy workers outside, the scheduler verifies first if there are other idle workers positioned above\(^4\) in \( \mathcal{Q} \)'s branch. If this is the case, it immediately aborts the search. This condition benefits the idle workers in upper nodes, because they are closer to the busy workers positioned outside the current subtree of

---

\(^4\)Throughout the thesis, it is assumed that root nodes are always the topmost nodes of the search tree and that leaf nodes are always the bottommost. Therefore, and considering a node \( \mathcal{N} \) and a node \( \mathcal{M} \) positioned between \( \mathcal{N} \) and the root node of the search tree, the following terminology can be correctly used: \( \mathcal{M} \) is above \( \mathcal{N} \) (\( \mathcal{N} \) is below \( \mathcal{M} \)); \( \mathcal{M} \) is in a upper position than \( \mathcal{N} \) (\( \mathcal{N} \) is in a lower position than \( \mathcal{M} \)); or \( \mathcal{M} \) is older than \( \mathcal{N} \) (\( \mathcal{N} \) is younger than \( \mathcal{M} \)).
Q, and hence they should be the ones requesting work.

When the previous condition fails, then the scheduler determines the set $SB$ of busy workers outside the current subtree of $Q$. Next, from $SB$ it removes each worker $P$ that has idle workers in its branch. From the remaining workers in $SB$, the scheduler selects the one with the highest work load.

![Diagram](image)

Figure 3.4: Requesting work from workers outside the current subtree.

Figure 3.4 presents three different situations when worker $Q$ is searching for busy workers outside its current subtree. It can request work from $P$ only in situation (c). In situation (a), $Q$ has $Q_1$ above in its branch and in situation (b), $Q_2$ is in $P$’s branch. In situation (c) none of the previous circumstances hold, and both idle workers, $Q$ and $Q_3$, can request work from $P$.

### 3.2.3 Distributing Idle Workers

The third step of the main scheduler algorithm says that when the scheduler neither finds unexplored alternatives nor busy workers, it tries to move the idle worker to a better position in the search tree. This scheduling strategy aims to distribute the idle workers in such a way as that the probability of finding, as soon as possible,
busy workers with excess of work within the corresponding idle workers' subtrees is substantially increased.

An idle worker $Q$ moves to a better position in one of the following two cases: (i) there are busy workers outside $Q$'s subtree and $Q$ is not within the subtree of any other idle worker; or (ii) all workers within $Q$'s subtree are idle. In the first situation, $Q$ backtracks until it reaches the first node that is above all the busy workers that are not within the subtree of any other idle worker. In the second one, $Q$ backtracks until it reaches a node where there is at least one busy worker. Figure 3.5 shows an example that illustrates this scheme. The left figure shows the initial workers' positions and the right figure shows their positions after moving. From the two situations that induce an idle worker to move to a better position, worker $Q_1$ fits the first one, $Q_2$ the second, and $Q_3$ none. $Q_1$ moves up because it is a topmost idle worker that has a busy worker, $P_1$, outside its subtree. $Q_2$ moves up because there is no available work within its subtree. On the other hand, $Q_3$ maintains its relative position because it has $Q_1$ idle above and $P_3$ busy below.

![Figure 3.5: Scheduling strategies to move idle workers to better positions.](image)

Our goal is for idle workers to move closest to busy workers so that sharing overheads decrease. The first situation moves idle workers to cover all possible sources of work. This is done by only moving up topmost idle workers, therefore preventing other idle
workers of losing their positions. The second situation releases idle workers from closed work positions and moves them to parts of the tree that still have work.

3.3 Extending Yap to Support Or-Parallelism

The extensions required to implement support for or-parallelism in Yap can be divided in three major areas: (i) those related to the environment copying model; (ii) those related to the scheduling policy; and (iii) those related to scheduling support.

To implement environment copying major changes were required on the memory organization and its management. Other extensions were also introduced to support the process of sharing work. Regarding the scheduling policy, YapOr implements the Muse approach described in the previous section. To support this approach, YapOr introduces changes in choice point manipulation and in code compilation. Further, it introduces new mechanisms to synchronize workers when exploiting shared branches and to compute work load. We next describe those extensions.

3.3.1 Memory Organization

Following the original WAM definition [109], the Yap Prolog system includes four main memory areas: code area, heap, local stack and trail. The local stack contains both environment frames and choice points. Yap also includes an auxiliary area used to support some internal operations.

The YapOr memory is divided into two major addressing spaces: the global space and a collection of local spaces, as illustrated in Figure 3.6. The global space is further divided in two major areas. One contains the code area inherited from Yap and the other includes all the data structures necessary to support parallelism. Each local space represents one system worker and it contains the four WAM execution stacks inherited from Yap: heap, local, trail, and auxiliary stack. The relative position of the memory areas presented in the figure does not necessary imply an identical memory mapping implementation.

In order to efficiently meet the requirements of incremental copy, we follow the principles used in Muse to map the set of memory local spaces. The starting worker, that
3.3. EXTENDING YAP TO SUPPORT OR-PARALLELISM

is \textit{worker}_0, asks for shared memory in the system’s initialization phase. Afterwards, the remaining workers are created, through the use of the \texttt{fork} function \cite{96}, and inherit the previously mapped addressing space. Then, each new worker rotates the local spaces, in such a way that all workers will see their own spaces at the same virtual memory addresses. Figure 3.7 helps to understand this remapping scheme. It considers 3 workers and it illustrates the resulting mapping address view of each worker after rotating the inherited local spaces. It can be seen that each worker accesses its own local space starting from the \texttt{Addr}_0 virtual memory address.

This mapping scheme allows for efficient memory copying operations during incremental copying. To copy a stack segment between two workers, we simply copy directly from one worker space to the relative virtual memory address in the other worker’s space. Suppose, for instance, that \textit{worker}_2 wants to copy to \textit{worker}_1 stacks a segment of its stacks that starts at address \texttt{Addr}_X (from \textit{worker}_2’s view). Using the mappings from Figure 3.7 the target memory address for this copying operation is \texttt{Addr}_X + (\texttt{Addr}_2 - \texttt{Addr}_0) (from \textit{worker}_2’s view). The major advantage of this scheme is that no reallocation of address values in the copied segments is necessary.

In YapOr, this memory scheme is implemented through two different and alternative UNIX shared memory management functionalities, the \texttt{mmap} and \texttt{shmget} functions \cite{96}. These functions let us map shared memory segments at given addresses,

\footnote{The \texttt{fork} function belongs to the UNIX libraries and it allows to create child processes equal to the caller parent.}
and unmap and remap them later at new addresses.

### 3.3.2 Choice Points and Or-Frames

The bottommost policy of dispatching work for parallel execution requires that a busy worker releases all of its current private choice points when sharing work. This maximizes the amount of shared work with the requesting worker and induces coarse grain tasks which has proven to be very successful within environment copying.

In order to correctly exploit a shared branch, we need to synchronize workers in such a way that we avoid executing twice the same alternative, as different workers referencing a choice point might pick the same alternative for work. To do so, the worker making a choice point public adds an or-frame data structure to the shared space per public choice point. The or-frames form a tree that represents the public search tree. Figure 3.8 illustrates how a private choice point is made public.
3.3. EXTENDING YAP TO SUPPORT OR-PARALLELISM

From the figure we can see the extended structure of a choice point. The first six fields are inherited from Yap, while the last two were introduced in YapOr. The inherited CP_ALT, CP_CP, CP_TR, CP_H, CP_B and CP_ENV choice point fields store, respectively, the next unexploited alternative; success continuation program counter; top of trail at choice point creation; top of global stack at choice point creation; failure continuation choice point; and current environment [1]. The CP.OR.FR field stores the pointer to the correspondent or-frame when the choice point is shared. Otherwise, it not used. The CP.LUB field stores the local untried branches and reflects the number of private unexplored alternatives above. It is used for computing worker load (see subsection 3.3.3).

As an optimization, we can reduce the two introduced new fields to just one. While the choice point is private, the field should act like the CP.LUB one, storing the local untried branches. When it is made public it can act like the CP.OR.FR field because the information in CP.LUB becomes unnecessary, as all unexplored alternatives in upper choice points have been made public.

Figure 3.8 presents the choice point data structure before and after a sharing operation. Sharing a choice point involves updating the CP_ALT field to point at the getwork pseudo-instruction (see subsection 3.3.5) and storing the pointer to a newly allocated or-frame in the CP.OR.FR field.

We next briefly introduce the functionality of each or-frame data field and describe...
how they are initialized. The OrFr.lock field supports a busy-wait locking mutex mechanism that guarantees atomic updates to the or-frame data. It is initially set to unlocked. The OrFr.alt field stores the pointer to the next available alternative as previously stored in the CP_ALT choice point field. OrFr.members is a bitmap that stores the set of workers for which their current branch contains the choice point. OrFr.node is a back pointer to the correspondent choice point. OrFr.nearest_livenode is a pointer to the or-frame that corresponds to the nearest choice point above with unexploited alternatives. Hence, if a worker reaches a public choice point with a NULL pointer in the OrFr.nearest_livenode field, it knows it is out of work. Last, the OrFr.next field is a pointer to the parent or-frame on the current branch.

To delimit the private from the public region, each worker holds a TOP.OR.FR register that points to the or-frame corresponding to the bottom shared choice point on the current branch.

### 3.3.3 Worker Load

Each worker maintains a local register, LOCAL.load, that estimates the number of private unexploited alternatives. The LOCAL.load register helps the scheduler when searching for a busy worker to request work from. There is a compromise thus between its correct value and the efficiency of the parallel process. Our implementation updates the LOCAL.load register only when creating a new choice point. With this scheme it is possible to maintain a very good approximation of its correct value avoiding regular actualizations, as we show next.

In subsection 3.3.2 we said that the CP.LUB choice point field is used to compute the worker's load. We also said that the CP.LUB field stores the number of local untried branches in the choice points above. This number does not include branches starting at the current choice point in order to avoid regular actualizations when backtracking occurs. Computing LOCAL.load is thus achieved by adding the value of CP.LUB with the number of the alternative branches in the newly created choice point.

The number of unexploited alternatives in a choice point is found by consulting the or.arg argument of the next available alternative. The or.arg argument holds the number of available alternatives starting from the current alternative, and it is generated by the compiler. Consider, for instance, a predicate with three alternative
3.3. EXTENDING YAP TO SUPPORT OR-PARALLELISM

clauses. To represent the three alternatives, the first clause is compiled with a value of three in the or_arg argument. The second clause is compiled with a value of two to represent the two remaining clauses, and the last clause is always compiled with a value of one.

A worker has shareable work if the value in the LOCAL_load register is positive. Nevertheless, a great number of Prolog programs contain predicates that generate relatively small tasks. To attain good performance it is fundamental to avoid sharing such fine grained work. In YapOr, the scheduler only considers that a worker has shareable work when its load register is greater than a certain threshold value (the threshold value is dynamically configurable in the system’s initialization phase). This introduces some delay in propagating work, avoiding eager sharing, therefore allowing a worker to build up a reserve of local work which may increase task granularity.

3.3.4 Sharing Work Process

The process of sharing work makes parallel execution of goals possible. This process takes place when an idle worker Q makes a sharing request to a busy worker P and receives a positive answer. P can refuse a sharing request if (i) Q is not above P, or (ii) if P has a load value above the threshold value and the OrFr_nearest_livenode of its current top shared or-frame is NULL. The latter case happens when P does not have any unexploited alternatives except the one it is executing. When Q receives a negative answer it returns to scheduler mode.

Sharing is implemented by two model dependent functions: p_share_work(), for the busy worker, and q_share_work(), for the idle one. In copying, the sharing process can be divided in four main steps. The initial step is where the auxiliary variables are initialized and the limits of stack segments to be copied are computed. The sharing step is where the private choice points are turned into public ones. The copy step is where the computed segments are copied from the busy worker stacks to the idle worker ones. Finally, the installation step is where the bindings trailed in the copied trail segment that refer to conditional variables stored in the maintained segments are copied to the idle worker stacks. To minimize overheads, both workers cooperate in the execution of the four steps. The sharing work algorithm is detailed in Figure 3.9.

Initially, the idle worker Q waits for a sharing signal while the busy worker P computes
### CHAPTER 3. YAPOR: THE OR-PARALLEL ENGINE

**p_share_work()**

<table>
<thead>
<tr>
<th>Compute stacks to copy</th>
<th>Signals</th>
<th><strong>q_share_work()</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;-----sharing-----&quot;</td>
<td>Wait sharing signal</td>
</tr>
<tr>
<td>Share private nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;-----nodes_shared--&quot;</td>
<td></td>
</tr>
<tr>
<td>Help Q in copy ?</td>
<td>&quot;&lt;----copy_done-----&quot;</td>
<td>Copy local stack ?</td>
</tr>
<tr>
<td></td>
<td>&quot;----copy_done-----&quot;</td>
<td></td>
</tr>
<tr>
<td>Wait copy_done signal</td>
<td>&quot;&lt;-----ready------&quot;</td>
<td>Install bindings</td>
</tr>
<tr>
<td>Back to Prolog execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release shared node ?</td>
<td></td>
<td>Fail to top shared node</td>
</tr>
<tr>
<td>Wait ready signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.9:** The sharing work process.

the stacks to copy. After that, \( P \) prepares its private nodes for sharing whilst \( Q \) performs incremental copying. \( Q \) copies the stacks from \( P \) in the following order: trail, heap and local stack. The local stack can only be copied after \( P \) finishes the sharing step. \( P \) may help in the copying process to speed it up. It copies the stacks to \( Q \) but in a reverse order. This scheme has proved to be efficient as it avoids some useless variables checks and locks. The two workers then synchronize to determine the end of copying. At last, \( P \) goes back to Prolog execution and \( Q \) installs the bindings referring variables in the maintained part of the stacks and restarts a new task from the recently installed work. To avoid possible undoing of bindings, \( P \) cannot release a shared node from its stacks until \( Q \) does not complete the installation step.

### 3.3.5 New Pseudo-Instructions

YapOr introduces four new instructions over Yap, namely, `getwork_first_time`, `getwork`, `getwork_sequential`, and `synch`. These instructions are never generated by the compiler. The former three are introduced according to the progress of parallel execution, while the latter is called before a side effect instruction gets executed. Next, we briefly describe how each instruction fits the YapOr execution model.

Whenever the search tree for the top level goal is fully exploited, all workers, except \( \text{worker}_0 \), execute the `getwork_first_time` instruction. This instruction blocks the
workers. They will wait for a signal from worker_0, that indicates the beginning of a new query goal. worker_0 is further responsible to present the answers encountered for the last exploited query and to control the interface with the user until he asks for a new query goal.

As mentioned in the previous subsection, the CP_ALT choice point fields are updated to point to the getwork instruction when they are being shared. This sharing procedure forces the future execution of the getwork instruction every time a worker backtracks to a shared choice point. The execution of this instruction allows the workers sharing the correspondent or-frame synchronized access to unexploited alternatives, guaranteeing that every alternative is exploited only once.

Sometimes it may be advantageous to declare a predicate as sequential [57] to force the scheduler to traverse its alternatives in a left to right fashion. A :- sequential pred/n declaration can be useful when the programmer wants to guarantee that only after an alternative is fully exploited the next one should be taken. Sequential predicates are implemented in YapOr by using the getwork_sequential instruction instead of a getwork instruction when sharing a choice point for a predicate declared as sequential. This variant of the getwork instruction ensures that the alternatives are taken one at a time according to its left to right order. Note that the subtree corresponding to each alternative can still be exploited in parallel.

A major problem when implementing parallel Prolog systems is the support for cuts and side effects. For cuts, YapOr currently implements a scheme based on the strategies described in [8] that prunes useless work as early as possible. A complete description of this scheme can be found is section 7.2. For side effects, the current implementation of YapOr is very simple. As soon as a worker reaches the execution of a side effect, it enters the synth instruction. The synth instruction implements a delaying procedure that waits until the worker's current branch becomes the leftmost one in the search tree. Only when the worker becomes leftmost synth returns and the side effect execution proceeds. If the worker, while waiting, is pruned by a left branch then the side effect is never executed. This ensures that side effects are executed in the same way and in the same order as in sequential execution.
3.4 Chapter Summary

This chapter introduced the YapOr or-parallel engine. YapOr extends the Yap Prolog system to support implicit or-parallelism in Prolog programs. YapOr’s implementation is largely based on the Muse approach for or-parallelism. We presented the environment copying model, as first implemented in the Muse system, and described the Muse strategies to scheduling work for or-parallel execution.

Next, we described the main issues in extending the Yap Prolog system to support or-parallelism. These included the extensions related with the environment copying model, such as, memory organization and work sharing; those related with the scheduling policy and the scheduling strategies; and those related with scheduling support, such as, code compilation, choice point manipulation, and work load.
Chapter 4

YapTab: The Sequential Tabling Engine

YapTab is a sequential tabling engine that extends the Yap Prolog system to support tabling. YapTab is based on the SLG-WAM engine [86, 89, 87] as first implemented in the XSB Prolog system. YapTab is also the base tabling engine for the combined or-parallel tabling engine that we address later.

First, we briefly describe the fundamental aspects of the SLG-WAM abstract machine, and then we detail the YapTab implementation. This includes discussing the motivation and major contributions of the YapTab design, and presenting the main data areas, data structures and algorithms to extend the Yap Prolog system to support tabling.

4.1 The SLG-WAM Abstract Machine

Remember that the scope of this thesis is to address the problem of combining or-parallelism and tabling for logic programs not including negation. Hence, we will only consider those aspects of the SLG-WAM abstract machine that are relevant for the support of variant-based tabling of definite programs.
4.1.1 Basic Tabling Definitions

Tabling is about storing and reusing intermediate answers for goals. Whenever a tabled subgoal $S$ is called for the first time, an entry for $S$ is allocated in the table space. This entry will collect all the answers generated for $S$. Repeated calls to variants of $S$ are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated for $S$, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as follows:

**Generator nodes:** nodes corresponding to first calls to tabled subgoals. They use program clause resolution to produce answers.

**Consumer nodes:** nodes corresponding to variant calls to tabled subgoals. They consume answers from the table space.

**Interior nodes:** nodes corresponding to non-tabled predicates. These nodes are evaluated by standard SLD resolution.

For definite programs, tabling based evaluation has four main types of operations:

**Tabled Subgoal Call:** looks up if the subgoal is in the table and if not, inserts it and allocates a new generator node. Otherwise, allocates a consumer node and starts consuming the available answers.

**New Answer:** verifies whether a newly generated answer is already in the table, and if not, inserts it.

**Answer Resolution:** consumes the next found answer, if any.

**Completion:** determines whether a SCC is completely evaluated, and if not, schedules a possible resolution to continue the execution.

Space for a subgoal can be reclaimed when the subgoal has been *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when all available alternatives have been exploited and the variant subgoals have consumed all the available answers. Remember that a number
of subgoals may be mutually dependent, forming a strongly connected component (or SCC), and therefore can only be completed together. The completion operation is then performed at the leader of the SCC, that is, at the oldest subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC \[87\].

4.1.2 SLG-WAM Overview

The SLG-WAM extends the WAM to fully integrate Prolog and tabling. In short, the SLG-WAM introduces a new set of instructions to deal with the tabling operations, a special mechanism to allow suspension and resumption of computations, and two new memory areas: a table space, used to save the answers for tabled subgoals; and a completion stack, used to detect when a set of subgoals is completely evaluated.

Further, whenever a consumer node gets to a point in which it has consumed all available answers, but the correspondent tabled subgoal has not yet completed and new answers may still be generated, the current computation must be suspended. The SLG-WAM implements the suspension mechanism through a new set of registers, the freeze registers, which protect the WAM stacks at the suspension point so that all data belonging to the suspended branch cannot be erased. To later resume a suspended branch, the bindings belonging to the branch must be restored. SLG-WAM achieves this by using an extension of the standard trail, the forward trail, to keep track of the bindings values.

4.1.3 Batched Scheduling

Usually it is possible to apply more than one strategy to continue after suspending a computation. For instance, there may be alternative clauses to resolve with generator or interior nodes, answers to be returned to consumer nodes, or completion operations to be performed. The decision of which operation to perform is determined by the scheduling strategy. The SLG-WAM default scheduling strategy (for versions 1.5 and higher of XSB) is called Batched Scheduling \[40\].

Batched scheduling takes its name because it tries to minimize the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation
CHAPTER 4. YAP'TABLE: THE SEQUENTIAL TABLELING ENGINE

continues until it resolves all program clauses for the subgoal in hand. Only then the newly found answers will be returned to consumer nodes.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. Calls to non-tabled subgoals allocate interior nodes. First calls to tabled subgoals allocate generator nodes and variant calls allocate consumer nodes. However, if we call a variant tabled subgoal, and the correspondent subgoal is already completed, we can avoid consumer node allocation and instead perform what is called a completed table optimization [87]. This optimization allocates a node, similar to an interior node, that will consume the set of found answers executing compiled code directly from the trie data structure associated with the completed subgoal. In [71, 72], I. V. Ramakrishnan et al. shows that the built-in set of SLG-WAM instructions introduced to execute the compiled answer tries can outperform standard WAM compiled code.

When backtracking we may encounter three situations: (i) if backtracking to a generator or interior node with available alternatives, the next program clause is taken; (ii) if backtracking to a consumer node, we take the next unconsumed answer from the table space; (iii) if there are no available alternatives or no unconsumed answers, we simply backtrack to the previous node on the current branch. Note however that, in case (iii), if the node without alternatives is a leader generator node, then we must check for completion.

4.1.4 Fixpoint Check Procedure

In order to perform completion, the scheduler must ensure that all answers have been returned to all consumer subgoals in the SCC. The process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

At engine level, the fixpoint check procedure is controlled by the leader of the SCC. The procedure traverses the consumer nodes in the SCC in a bottom-up manner to determine whether the subgoals in a SCC have been completely evaluated or whether further answers need to be returned to consumer nodes. Initially, it searches for the bottom consumer node with unresolved answers. If there is such a node, it is resumed and as long as there are newly found answers, it will consume them. After consuming
the available set of answers, the consumer suspends and fails into the next consumer
node with unresolved answers. This process repeats until it reaches the last consumer
node, in which case it fails into the leader node in order to allow the re-execution of
the fixpoint check procedure. When a fixpoint is reached, all subgoals in the SCC are
marked completed, the stack segments belonging to the completed subtree are released
and the freeze registers are updated.

Please refer to subsection 2.3.1 and to Figure 2.3 for an example of a tabling evaluation
sequence.

\[ 4.1. \text{ THE SLG-WAM ABSTRACT MACHINE} \]
4.1.6 Instruction Set for Tabling

The SLG-WAM provides a new set of instructions in order to implement the four main tabling operations. Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me`, and `table_trust_me` SLG-WAM instructions, in a similar manner to the WAM’s `try_me`, `retry_me`, and `trust_me` sequence.

The `table_try_me` instruction extends the WAM’s `try_me` instruction to support the tabled subgoal call operation. When `table_try_me` corresponds to a first call to a tabled subgoal, it inserts the subgoal at hand into the table space, by allocating the necessary data structures; pushes a new generator choice point and a new environment onto the local stack; pushes a completion frame onto the completion stack; and initializes all cells in these structures.

On the other hand, if the call is a variant call, then the subgoal is already in the table space, and two different situations may occur, depending on whether the subgoal is completed or not. If the subgoal is completed, the `table_try_me` instruction implements the completed table optimization. Otherwise, a consumer choice point is allocated, the freeze registers are updated to the current top stack pointers, and the available answers start being consumed. The answer resolution operation is supported through setting the `CP_ALT` consumer choice to point to the SLG-WAM `answer_resolution` instruction. This instruction is responsible for guaranteeing that all answers are given once and just once to each variant subgoal. The `answer_resolution` instruction gets executed through backtracking or through direct failure to a consumer node in the fixpoint check procedure.

The `table_retry_me` and `table_trust_me` differ from the `retry_me` WAM instruction in that they always restore a generator choice point, rather than an interior (WAM-style) choice point. The only difference between both instructions is in the way they update the `CP_ALT` generator choice point field. In the `table_retry_me` implementation, the `CP_ALT` field is made to point to the compiled code for the next clause, while in the `table_trust_me` it is updated to the completion instruction. The completion instruction implements the completion operation in order to ensure the complete and correct evaluation of the subgoal search space. It gets executed through backtracking or through direct failure from the last node on the chain of consumer nodes as described in the fixpoint check procedure.
Tabled predicates defined by a single clause are compiled using the SLG-WAM table_try_me_single instruction. This instruction optimizes the table_try_me instruction for the case when the tabled predicate is defined by a single clause. Similarly to the table_trust_me instruction, the CP_ALT generator choice point field is made to point to the completion instruction.

The SLG-WAM introduces a new_answer instruction to implement the new answer operation. This instruction is produced by the compiler when compiling a clause for a tabled predicate. It is the final instruction of the clause’s compiled code and it includes the functionalities of the deallocate and proceed WAM instructions. As the new_answer instruction is the final instruction of a compiled tabled clause, the arguments from the body of the clause have been resolved when the instruction is reached. Thus, by dereferencing them we obtain the binding substitution which identifies the answer for the subgoal.

To give a flavor of what to expect from the compiled code of a tabled predicate, consider the following path/2 definition:

```prolog
:- table path/2.
path(X,Z) :- path(X,Y), arc(Y,Z).
pred(X,Z) :- arc(X,Z).
```

Figure 4.1 shows the resulting compiled code for the two clauses of the tabled predicate path/2, using the just described instruction set. As path/2 is defined by several clauses, a table_try_me instruction begins the code for its first clause, with the label pointing at the start of the second clause as an argument. On the other hand, as the second clause is the last clause for path/2, its code begins with a table_trust_me instruction. The code for both clauses follows the usual WAM code for the head and body subgoals of the clauses. The exception is that at the end a new_answer instruction closes each block.

### 4.2 Extending Yap to Support Tabling

YapTab has been designed to achieve an efficient tabling computational model that can be integrated with an or-parallel component. To achieve high performance, we are very interested in developing a sequential tabling implementation that compares
favorably with the current state of the art technology. In other words, we want the parallel tabling system, when executed with a single worker, to run as fast or faster than the current available sequential systems. Otherwise, the parallel performance results would not be significant and fair, and thus it would be hard to evaluate the efficiency of the parallel implementation.

### 4.2.1 Overview

The YapTab design is WAM based, as is the SLG-WAM. It implements two tabling scheduling strategies, batched and local [40]. Our initial design only considers positive programs. As in the original SLG-WAM, it extends the WAM with a new data area, the table space; a new set of registers, the freeze registers; an extension of the standard trail, the forward trail; and support for the four main tabling operations: tabled subgoal call, new answer, answer resolution and completion.

The major differences between both designs, and corresponding implementations, reside in the issues that can be a potential source of overheads when the tabling engine is extended to a parallel model. In a parallel environment, duplication of items is a major source of overhead. It requires synchronization mechanisms when updating common items and when replicating the new values. To efficiently integrate tabling with parallelism we should minimize this duplication.

To address this need, YapTab introduces a new data structure, the dependency frame, that resides in a single shared space that we name the dependency space. The de-
pendency frame data structures maintain in a single space all data concerning tabling suspensions. By data related with tabling suspension we mean the data involved in the fixpoint check procedure and in the resumption of suspended nodes. The introduction of this new data structure allows us to reduce the number of extra fields in tabled choice points and eliminates the need for a separate completion stack, avoiding potential synchronization points, and thus simplifying the complexity in managing shared tabling suspensions.

To benefit from the philosophy behind the dependency frame data structure, all the algorithms related with suspension, resumption and completion were redesigned. We next present the main data areas, data structures and algorithms implemented to extend the Yap system to support tabling. The algorithms described assume a batched scheduling strategy implementation, we discuss local scheduling later.

### 4.2.2 Table Space

The table space can be accessed in different ways: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is already in the table, and if not insert it; to pick up answers to consumer nodes; and to mark subgoals as completed. Hence, a correct table design with efficient algorithms to access and manipulate the table data is a critical issue to obtain a valid tabling system implementation.

Our implementation uses tries as the basis for tables, as proposed by I. V. Ramakrishnan et al. [71, 72]. Tries provide complete discrimination for terms and permit lookup and possibly insertion to be performed in a single pass through a term. In later chapters, we shall discuss the performance of tries on the parallel environment.

Figure 4.2 shows the general tries structure for a tabled predicate. At the entry point we have the *table entry* data structure. This structure is allocated when a predicate declared as tabled is being compiled, so that a pointer to the table entry can be included in the compiled code. This guarantees that further calls to the predicate will access the table starting from the same point.

Below the table entry, we have the *subgoal trie structure*. Each different tabled subgoal call to the predicate in hand corresponds to an unique path through the subgoal trie structure, always starting from the table entry, passing by several subgoal trie data
units, the subgoal trie nodes, and reaching a leaf data structure, the subgoal frame. The subgoal frame acts like an entry point to the answer trie structure and stores additional information about the subgoal. Each unique path through the answer trie data units, the answer trie nodes, corresponds to a different answer to the entry subgoal.

Figure 4.3 details the tries structure by presenting an example for a concrete predicate $t/2$ after the execution of several table$_\text{try me single}$ and new$_\text{answer}$ instructions. Each invocation of table$_\text{try me single}$ leads to either finding a path through the subgoal trie nodes until a matching subgoal frame is reached, or creating a new path when one does not exist. This happens, respectively, when we are in the presence of either variant or first subgoal calls. In a similar fashion, each invocation of the new$_\text{answer}$ instruction corresponds to finding or creating a new path through the answer trie nodes, starting from the corresponding subgoal frame.

Searching through a chain of sibling nodes that represent alternative paths is done sequentially. However, if the chain becomes larger than a threshold value, we dynamically index the nodes through a hash table to provide direct node access and therefore optimize the search.

Analyzing the figure, it can be observed that the answer trie for call $t(X, w)$ stores only the binding $a$ to the unbound variable, and avoids storing the complete answer $(a, w)$. This optimization is called substitution factoring [71, 72]. The core idea behind this
4.2. EXTENDING YAP TO SUPPORT TABLING

Optimization is to only store in the answer trie substitutions for the unbound variables in the subgoal call.

Each subgoal frame includes two pointers to provide access to the answers already stored in table. The SgFr_first_answer pointer provides access to the first found answer, while the SgFr_last_answer pointer provides access to the last. Furthermore, the leaves’ answer nodes are chained together in insertion time order, in such a way that, starting from the SgFr_first_answer pointer, and following the chain of leaf nodes, we reach the node pointed by the SgFr_last_answer pointer once and only once.

Using this chain, a consumer node can ensure that no answer is skipped or consumed twice. This is done by holding a private pointer to the leaf node of its last consumed answer and following the chain of leaves to consume new answers. To load an answer, the trie nodes for the answer in hand are traversed in bottom-up order, starting from the pointer to the leaf node and following the parent pointer to the preceding node on the path until reaching the subgoal frame.
The answer trie structure is not traversed in a top-down manner because the insertion and consumption of answers is an asynchronous process. Since new trie nodes may be inserted at anytime and anywhere in the answer trie structure, this induces complex dependencies that may limit the efficiency of possible top-down control schemes. Remark that the completed table optimization allows us to efficiently traverse the answer trie structure in a top-down way. However, it is only performed when the subgoal is completed, which ensures that no more nodes are added. A field of the subgoal frames marks subgoals as completed.

### 4.2.3 Generator and Consumer Nodes

Generator and consumer nodes correspond, respectively, to first and variant calls to tabled subgoals, while interior nodes correspond to normal, not tabled, subgoals. The abstract notion of a node is implemented at the engine level as a choice point. Figure 4.4 details YapTab’s choice point structure for these nodes.

![Interior CP, Generator CP, Consumer CP](image)

Figure 4.4: Structure of interior, generator and consumer choice points.

Remember that interior nodes are implemented as normal WAM choice points and that the CP_ALT, CP_CP, CP_TR, CP_H, CP_B and CP_ENV choice point fields store respectively, the next unexploited alternative; success continuation program counter; top of trail; top of global stack; failure continuation choice point; and current environment. Generator and consumer nodes are also implemented as WAM choice points, but extended with an extra field, respectively, the CP_SG.FR and CP_DEP.FR fields.

The SLG-WAM implements the generator nodes as WAM choice points extended with several extra fields. One of those fields stores the pointer to the correspondent
subgoal frame, the others hold the top freeze registers at choice point creation. Our implementation only requires the subgoal frame pointer because we adjust the freeze registers by using the top of stack values kept in the consumer choice points (see subsection 4.2.5 for details).

Regarding consumer nodes, SLG-WAM also implements them as WAM choice points with several extra fields. In YapTab, we move consumer information to a dependency frame and leave the pointer to this frame in the CP\_DEP\_FR field. Figure 4.5 illustrates the relationships between the novel choice points fields and the table and dependency spaces.

![Diagram of nodes and their interaction with the table and dependency spaces.](image)

Figure 4.5: The nodes and their interaction with the table and dependency spaces.

The dependency frames are linked together to form a dependency graph between consumer nodes. Additionally, they store information to efficiently check for completion points, and to efficiently move across the dependency graph. As we shall see, this functionality replaces the need for a completion stack.

To take advantage of substitution factoring, we create in the local stack a substitution factor where we store references to the set of unbound variables in the subgoal call.
The substitution factor is created when traversing the subgoal trie structure to check for/insert the subgoal call, the dereferenced pointers to unbound variables from the subgoal are pushed onto the local stack. The substitution factor thus points to variables on the local or heap stack. A generator choice point executing a new answer operation determines the answer substitution simply through dereferencing the substitution factor. A consumer choice point can correctly load an answer from the table space by unifying the substitution factor pointers with the meanwhile copied answer substitution.

4.2.4 Subgoal and Dependency Frames

The subgoal and dependency frames are the main data structures required to control the flow of a tabled computation. As mention before, the subgoal frames provide access to the answer trie structure and to check for and mark the completion of a subgoal. The dependency frames synchronize suspension, resumption and completion of subcomputations. Figure 4.6 details the subgoal and dependency frame structures.

![Subgoal Frame and Dependency Frame Diagram](image)

A subgoal frame includes six fields. The `SgFr_gen_cp` is a back pointer to the corresponding generator choice point; the `SgFr_answer_trie` points to the top answer trie node, and is mainly used to access the answer trie structure to check for/insert new answers; the `SgFr_first_answer` points to the leaf answer trie node of the first available answer; the `SgFr_last_answer` points to the leaf answer trie node of the last available answer; the `SgFr_completed` is a flag that indicates if the subgoal is completed or not; and the `SgFr_next` points to the next subgoal frame, that is, to the subgoal frame for the youngest generator older than the current choice point. It is
used to traverse subgoal frames when performing completion. To access the subgoal frames chain, we use a \texttt{TOP.SG.FR} register that points to the youngest subgoal frame.

Each dependency frame is also a six field data structure. The \texttt{DepFr.back.cp} points to the generator choice point involved in the last unsuccessful completion operation, and is used by the fixpoint check procedure to schedule for a backtracking node (see 4.2.8 for details); the \texttt{DepFr.leader.cp} points to the leader choice point and it is used to check for completion points; the \texttt{DepFr.cons.cp} is a back pointer to the consumer choice point; the \texttt{DepFr.sg.fr} and the \texttt{DepFr.last.ans} point to the correspondent subgoal frame and to the last consumed answer, respectively, and they provide access to the table space in order to search for and to pick up new answers; and the \texttt{DepFr.next} is a pointer to the next dependency frame, that is, to the dependency frame for the youngest consumer older than the current choice point. It is used to form a dependency graph between consumer nodes to efficiently check for leader nodes and to efficiently implement the completion and fixpoint check procedures. To access the dependency graph, we use a \texttt{TOP.DEP.FR} register that points to the youngest dependency frame.

Figure 4.7 shows an example of how the data structures presented are used in a particular evaluation. The leftmost sub-figure presents the execution tree dependencies between the predicates involved in the example.

![Diagram of dependencies between choice points, subgoal and dependency frames.](image)

Figure 4.7: Dependencies between choice points, subgoal and dependency frames.

The first instance of \texttt{table.try.me} searches the table space for the corresponding
CHAPTER 4. YAPTab: THE SEQUENTIAL TABLELING ENGINE

subgoal t(X). As this is the first call to the subgoal, it must allocate a subgoal frame and store a generator choice point. Assuming that t/1 is a three clause predicate, the CP_ALT field of the generator choice point will point to the table_retry_me instruction that starts the compiled code of the second clause. Assuming that v/1 is a two clause predicate, an analogous situation occurs with the first call to subgoal v(Y). The only difference is that the CP_ALT field will now point to a table_trust_me instruction (note that this initialization is not illustrated in the figure).

Following the example, the second call to v(Y) searches the table space and finds that it is a variant call of the subgoal v(var 0). Thus, it allocates a dependency frame and stores a consumer choice point. A consumer choice point is initialized with its CP_ALT field pointing to the answer_resolution pseudo instruction. Assuming that no answers were found for subgoal v(var 0), the computation will backtrack to the previous choice point CP2. The table_trust_me instruction gets executed, and the CP_ALT field is update to the completion pseudo instruction. The second call to t(X) is similar to the second call to v(Y).

The dependency frame fields DepFr_back_cp and DepFr_leader_cp and the pseudo instructions answer_resolution and completion are detailed next.

4.2.5 Freeze Registers

A tabled evaluation can be seen as a sequence of subcomputations that suspend and later resume. The SLG-WAM preserves the environment of a suspended computation by freezing stacks. A set of freeze registers, one per stack, says where stacks are frozen. Freeze registers protect therefore the space belonging to the suspended branch until the completion of the appropriate subgoal call takes place. It is only upon completion that we can release the space previously frozen and adjust the freeze registers.

The SLG-WAM extends the generator choice points to store the freeze registers at choice point creation, so that they can be adjusted if completion takes place. In YapTab, we adjust the freeze registers by using the top stack values kept in the youngest consumer choice point, after completion. We access that choice point through the top dependency frame as given by the TOP_DEP.FR register. Figure 4.8 shows the pseudo-code that adjusts the freeze registers.
adjust_freeze_registers() {
    B_FZ = DepFr_cons_cp(TOP_DEP_FR)  // B_FZ is the stack freeze register
    H_FZ = CP_H(B_FZ)  // H_FZ is the heap freeze register
    TR_FZ = CP_TR(B_FZ)  // TR_FZ is the trail freeze register
}

Figure 4.8: Pseudo-code for adjust_freeze_registers().

The introduction of freeze registers creates situations where the current stack registers can point to older positions than those given by the freeze registers. To guarantee that frozen segments are safe from being overwritten, we need to guarantee that new data always is placed at the younger position of both registers. Several schemes may be followed to ensure that: (i) we always compare the top stack register with the freeze register and determine the youngest; (ii) we have an additional register that always holds the youngest; or (iii) we ensure that, when writing, the top stack register is always younger than the freeze register and thus proceed as usual. Scheme (iii) is the one which introduces the least overheads for the execution. However, it cannot be applied to the local stack because tabled evaluation leads to situations where B is necessarily older than B_FZ.

By default, YapTab implements scheme (i) to deal with the local stack and scheme (iii) to deal with the heap and trail stacks. As a configuration option, it is possible to execute YapTab using scheme (ii) for the local stack. The following subsection details the implementation of scheme (iii) for the trail stack.

4.2.6 Forward Trail

To resume the computation to a suspended consumer node, we have to restore all the variable bindings to their state at the time the node was suspended. The forward trail is a data structure that extends the standard WAM trail entries to record variable bindings. In the SLG-WAM, each forward trail frame has three fields: the address of the trailed variable, as in the WAM; the value to which the variable was bound, so that it can be restored later; and a pointer to the parent trail frame, used to correctly move across the variables in a branch, hence avoiding variables in frozen segments [87].

In YapTab, the forward trail is implemented without parent trail frame pointers. Yap already uses the trail to store information beyond the normal variable trailing, say to control dynamic predicates and to implement multi-assignment variables. We
extend this information to also control the chain between frozen segments. In terms of computation complexity the two approaches are equivalent. The main advantage of our scheme is that we only need two fields.

Figure 4.9 illustrates our implementation scheme. Consider that the execution has reached the consumer node marked as (a) and that the computation is suspended as there are no available answers to be consumed. At this point, the trail freeze register TR.FZ is set to the trail register TR.

Now if backtracking takes place up to the node marked as (b), the bindings belonging to the backtracked segment are untrailed and TR is made to point to the next untrailed frame. At this point, TR points to a position above the one pointed by TR.FZ. To ensure that the trail segment corresponding to the frozen branch is not erased, and is not used by untrailing operations corresponding to different branches, we use a special trail frame to mark the existence of a frozen segment just above it (see illustration (c)). This frame records the continuation trail frame that allows for the frozen segment to be ignored in a future untrailing operation. The trail register TR is updated to point to this new trail frame.

Suppose that the execution has evolved to situation (d), in which the trail shows a more complex chaining of segments, and assume that the computation is being resumed to the first suspended node. To accomplish the correct restoration of the variable environment, the bindings belonging to the current branch need to be unbound and the bindings belonging to the branch being resumed need to be restored. Similarly to other
strategies presented previously, we can minimize the overhead of these operations by only unbinding/rebinding up to the youngest frame common to both branches, \( X = a \) in this case. By following TR, and visiting \( Z = f \) and \( Y = d \), we unbind variables \( Z \) and \( Y \), and by following TR1, and visiting \( Z = c \) and \( Y = b \), we bind \( Z \) and \( Y \) to \( c \) and \( b \), respectively.

Figure 4.10 shows the pseudo-code for restoring a variable environment given the top trail frame for the current branch (argument `unbind_fr`) and the top trail frame for the branch being resumed (argument `rebind_fr`).

```plaintext
restore_bindings(trail frame unbind_fr, trail frame rebind_fr) {
    common_fr = rebind_fr
    while (unbind_fr != common_fr) {
        while (unbind_fr > common_fr) { // rewind loop
            ref = Trail_Addr(--unbind_fr)
            if (ref is a variable)
                unbind_variable(ref)
            else if (ref is a frozen segment pointer)
                unbind_fr = ref
        }
        while (unbind_fr < common_fr) { // search a common frame
            ref = Trail_Addr(--common_fr)
            if (ref is a frozen segment pointer)
                common_fr = ref
        }
    }
    while (rebind_fr != common_fr) { // rebind loop
        ref = Trail_Addr(--rebind_fr)
        if (ref is a variable)
            bind_variable(ref, Trail_Value(rebind_fr))
        else if (ref is a frozen segment pointer)
            rebind_fr = ref
    }
}
```

Figure 4.10: Pseudo-code for `restore_bindings()`.

The procedure starts with both `unbind_fr` and `common_fr` following their chains until a common frame is reached, with `unbind_fr` unbinding variables as it goes. Then, `rebind_fr` follows its chain till the just found common frame, restoring the variables on the way. Note that the frames traversed by `common_fr` and `rebind_fr` are the same. However, variables are not restored when first searching for the common frame because they can be later unbound in the rewind loop. Note also that the rebind loop applies the bindings in the opposite order in which they were trailed. This is safe since no branch can have more than one trail entry for the same variable.
4.2.7 Completion and Leader Nodes

The completion operation takes place when a generator node exhausts all alternatives and finds itself as a leader node. We designed novel algorithms to quickly determine whether a generator node is a leader node. The key idea is that each dependency frame holds a pointer to the presumed leader node of its SCC. Using the leader node from the dependency frames, a generator node can quickly determine whether it is a leader node. A generator finds itself as a leader node when there are no younger dependencies, that is, no younger consumer nodes, or when it is the leader node referred in the top dependency frame.

The algorithm requires computing leader node information when allocating a dependency frame for a new consumer node \( C \). To do so, we first hypothesize that the leader node is the generator node for the variant subgoal call relative to \( C \), say \( \mathcal{G} \). Next, for all consumer nodes between \( C \) and \( \mathcal{G} \), we check whether they depend on an older generator node. Consider that the oldest dependency is for the generator node \( \mathcal{G}' \). If this is the case, then \( \mathcal{G}' \) is the leader node, otherwise our hypothesis was correct and the leader is indeed the initially found generator node \( \mathcal{G} \).

Figure 4.11 presents a small example that illustrates how the current leader node changes during evaluation. By current leader node we mean the leader of the current SCC. In situation (a), the generator node \( \mathcal{N}_3 \) is the current leader node because there are no younger consumer nodes. Moving to situation (b), a new consumer node is created and a new dependency frame is allocated. Because \( \mathcal{N}_4 \) is a variant subgoal a for the generator node \( \mathcal{N}_1 \) and there are no other consumer nodes in between, \( \mathcal{N}_1 \) is the leader node for \( \mathcal{N}_4 \)’s dependency frame. As a result, the current leader node for the new set of nodes including \( \mathcal{N}_4 \) becomes \( \mathcal{N}_1 \). Situation (c) is similar to (a), and \( \mathcal{N}_5 \) becomes the new current leader node. The consumer node \( \mathcal{N}_6 \), from situation (d), is a variant subgoal c for generator node \( \mathcal{N}_3 \). Since consumer node \( \mathcal{N}_4 \) is between nodes \( \mathcal{N}_6 \) and \( \mathcal{N}_3 \) and depends on an older generator node, \( \mathcal{N}_1 \), the leader node information for \( \mathcal{N}_6 \)’s dependency frame is also \( \mathcal{N}_1 \). This turns again \( \mathcal{N}_1 \) as the current leader node.

Figure 4.12 shows the procedure that computes the leader node information for the current consumer node. The procedure traverses the dependency frames for the consumer nodes between the current consumer and its generator in order to check for older dependencies. As an optimization it only searches until it finds the first
4.2. EXTENDING YAP TO SUPPORT TABLING

dependency frame holding an older reference (the DepFr_leader_cp field). The nature of the procedure ensures that the remaining dependency frames cannot hold older references.

```plaintext
compute_leader_node(dependency frame dep_fr) {
    leader_cp = SgFr_gen_cp(DepFr_sg_fr(dep_fr))
    df = TOP_DEP_FR
    while (DepFr_cons_cp(df) is younger than leader_cp) {
        // searching for an older dependency
        if (leader_cp is equal or younger than DepFr_leader_cp(df)) {
            leader_cp = DepFr_leader_cp(df)
            break
        }
        df = DepFr_next(df)
    }
    DepFr_leader_cp(dep_fr) = leader_cp
}
```

Figure 4.11: Spotting the current leader node.

Figure 4.12: Pseudo-code for compute_leader_node().

We next give an argument on the correctness of the algorithm. Consider a consumer node with generator node $G$ and assume that its leader node $D$ is found in the dependency frame for consumer node $C$. Now hypothesize that there is a consumer node $N$ younger than $G$ with a reference $D'$ older than $D$. Therefore, when previously
computing the leader node for $C$ one of the following situations occurred: (i) $D$ is the
generator node for $C$ or (ii) $D$ was found in a dependency frame for a consumer node
$C'$. Situation (i) is not possible because $N$ is younger than $D$ and it holds a reference
older than $D$. Regarding situation (ii), $C'$ is necessarily younger than $N$ as otherwise
the reference found for $C$ had been $D'$. By recursively applying the previous argument
to the computation of the leader node for $C'$ we conclude that our initial hypothesis
cannot hold because the number of nodes between $C$ and $N$ is finite.

Figure 4.13 presents the pseudo-code that implements the completion() procedure.
It gets executed when the computation fails to a generator choice point with no
alternatives left.

```java
completion(generator node G) {
  if (G is the current leader node) {
    df = TOP_DEP_FR
    while (DepFr_cons_cp(df) is younger than G)) {
      if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        // dependency frame with unconsumed answers
        DepFr_back_cp(df) = G
        C = DepFr_cons_cp(df)
        restore_bindings(CP_TR(G), CP_TR(C))
        goto answer_resolution(C)
      }
      df = DepFr_next(df)
    }
    perform_completion()
    adjust_freeze_registers()
  }
  backtrack_to(CP_B(G))
}
```

Figure 4.13: Pseudo-code for completion().

Whenever a generator node finds out that it is the current leader node, it checks
whether there are younger consumer nodes with unconsumed answers. This can be
implemented by going through the chain of dependency frames looking for a frame
with unconsumed answers. If there is such a frame, it resumes the computation to
the corresponding consumer node. However, before resuming it must update the
DepFr_back_cp dependency frame field (more details in 4.2.8) and use the forward
trail to restore bindings.

Otherwise, it can perform completion. This includes marking as completed all the
subgoals in the SCC, using the TOP_SG_FR to go through the subgoals frames, and
deallocating all the younger dependency frames, using the TOP_DEP_FR register to go
through the dependency frames. At last, the algorithm must adjust the freeze registers and backtrack to the previous node to continue the execution.

In order to make the pseudo-code for procedures more intuitive and less verbose, throughout the thesis, we will frequently use goto statements like the one on Figure 4.13. With a goto statement we intend to denote that the flow of execution continues within the called procedure and that there is no return to the caller.

### 4.2.8 Answer Resolution

When a consumer choice point is allocated, its CP_ALT field is made to point to the answer_resolution instruction. This instruction is responsible for resuming the computation and guaranteeing that every answer is consumed once and just once. Figure 4.14 shows the procedure that implements the answer_resolution instruction. The procedure gets executed either when the computation fails or is resumed to a consumer choice point.

```plaintext
answer_resolution(consumer node C) {
    dep_fr = CP_DEP_FR(C)
    if (DepFr_last_ans(dep_fr) != SgFr_last_answer(DepFr_sg_fr(dep_fr))) {
        // unconsumed answers in current dependency frame
        load_next_answer_from_subgoal(DepFr_sg_fr(dep_fr))
        proceed
    }
    back_cp = DepFr_back_cp(dep_fr)
    if (back_cp == NULL)
        backtrack_to(CP_B(C))
    df = DepFr_next(dep_fr)
    while (DepFr_cons_cp(df) is younger than back_cp)) {
        if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
            // dependency frame with unconsumed answers
            DepFr_back_cp(df) = back_cp
            back_cp = DepFr_cons_cp(df)
            restore_bindings(CP_TR(C), CP_TR(back_cp))
            answer_resolution(back_cp)
        }
        df = DepFr_next(df)
    }
    restore_bindings(CP_TR(C), CP_TR(back_cp))
    goto completion(back_cp)
}
```

Figure 4.14: Pseudo-code for answer_resolution().

The answer_resolution() procedure first checks the table space for unconsumed
answers for the subgoal in hand. If there are new answers, it loads the next available answer and proceeds the execution. Otherwise, it schedules for a backtracking node.

If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. This is the case when the DepFr_back_cp dependency frame field is NULL. Otherwise, we know that DepFr_back_cp points to the generator node $G$ from where the computation has been resumed during the last unsuccessful completion operation. Therefore, backtracking must retry the next consumer node that has unconsumed answers and that is younger than $G$. If there is no such a consumer node then backtracking must be done to the generator node $G$.

Figure 4.15 presents two different situations that illustrate the functionality of the DepFr_back_cp field in the process of scheduling for a backtracking node. In both situations, the illustration sequence starts with the computation in a leader node position and assuming that all younger consumer nodes have unconsumed answers. A $W$ is used to mark the node where the computation is positioned at each illustration. The vertical dashed line in between the nodes denotes the possible existence of other nodes not related to execution of tabled predicates.

In situation (a), the execution of the completion() procedure in the leader node $L$ leads the computation to be resumed to the younger consumer node $C_2$. Before resuming, the DepFr_back_cp field of the dependency frame relative to $C_2$ is updated to $L$. Then, after all available unconsumed answers for $C_2$ have been consumed, answer_resolution() schedules for a backtracking node. As there is a consumer node $C_1$ younger than the generator given by the DepFr_back_cp field of $C_2$, then backtracking is done to $C_1$ and the DepFr_back_cp field of the dependency frame relative to $C_1$ is updated to $L$. As there is no consumer nodes between $L$ and $C_1$, $L$ is scheduled for backtracking when all available unconsumed answers for $C_1$ have been consumed.

Situation (a) corresponds to a complete loop step for the fixpoint check procedure. Starting from a leader node, it goes through the younger consumer nodes and ends eventually returning to the leader node. Situation (b) presents a slightly different sequence. It also starts from a leader node position, $L_2$, and resumes the computation to a consumer node $C_2$. However, when exploiting an unconsumed answer for $C_2$, a new consumer node is allocated and in consequence the current leader node changes and becomes $L_1$. Despite this leader modification, the backtracking sequence is similar to
the one of situation (a). After consuming all the available unconsumed answers for \( C_2, C_1 \) is scheduled for backtracking, and after consuming all the available unconsumed answers for \( C_1, L_2 \) is scheduled for backtracking.

At that point, we may question why waste time backtracking to the previous leader node \( L_2 \) if there is a new leader node \( L_1 \). Note that completion only resumes the computation to younger consumer nodes because all younger generator and interior nodes are necessarily exploited, that is, without alternatives. As \( \text{DepFr\_back\_cp} \) points to \( L_2 \), this allows us to conclude that all younger generator and interior nodes than \( L_2 \) are exploited. However, nothing can be said about the generator and interior nodes older than \( L_2 \). Hence, despite \( L_1 \) becoming the current leader node in the sequence of situation (b), between \( L_1 \) and \( L_2 \) may exist other nodes not exploited, and therefore
we still have to backtrack to $L_2$.

### 4.2.9 A Comparison with the SLG-WAM

The major difference between YapTab and the original SLG-WAM design resides in the way YapTab handles suspensions. The SLG-WAM considers that the control of leader detection and scheduling of unconsumed answers should be done at the level of the data structures corresponding to first calls to tabled subgoals, and it does so through associating completion frames to generator nodes. On the other hand, YapTab considers that such control should be performed through the data structures corresponding to variant calls to tabled subgoals, and thus it associates dependency frames to consumer nodes. We argue that managing dependencies at the level of the consumer nodes is a more intuitive approach that we can take advantage of.

The SLG-WAM's design presents therefore some differences when compared with YapTab. First, the SLG-WAM uses an auxiliary data space, the completion stack, in order to determine when a generator node is a leader node. Each completion frame corresponds to a different subgoal call and a new completion frame is allocated whenever a new generator node is created. The dependencies introduced by variant subgoal calls update the top completion frame in the completion stack according to a proper rule (for details about the completion stack please consult [86, 87]). The process of determining if a generator node is a leader node requires, in the worst case, consulting the completion frames of all younger subgoal calls.

YapTab uses dependency frames to determine when a node is a leader node. In order to motivate for the implementation required for the or-parallel tabling engine, we also assumed an auxiliary data space, the dependency space, where dependency frames are stored. However, for a strictly sequential engine, we can simplify the implementation by moving the data from the dependency space to the local stack and by storing dependency frames as extensions of consumer nodes. A dependency frame is allocated for each new consumer node and the leader node for the resulting SCC is computed in advance and stored in the dependency frame. By consulting the leader data stored in the youngest dependency frame, a generator node can thus determine in constant-time if it is the current leader node.

Another relevant difference is how consumer nodes with unconsumed answers are
scheduled for execution. Consider a leader node with several different groups of consumer nodes within its SCC, with each group corresponding to a common variant subgoal call. The SLG-WAM proceeds as follows. The groups are scheduled one at a time, starting from the group corresponding to the oldest subgoal call until reaching the group corresponding to the youngest subgoal call. If there are unconsumed answers for a particular group, the process is aborted by causing the evaluation to be resumed at the nodes with unconsumed answers. After such a batch of answers has been consumed, the evaluation returns to the leader node. When returning to the leader node, the process repeats until no unconsumed answers are found in a single pass through the whole set of groups. In this case, a fixpoint is reached and the SCC is completely evaluated.

YapTab simplifies the process by considering the whole set of consumer nodes within a SCC as a single group, independently of the subgoal call associated with each one. By following the chain of dependency frames, YapTab traverses in a single pass the whole set of consumer nodes which we argue may therefore reduce the overheads of scheduling consumer nodes with unconsumed answers in controlling the loop procedure.

In short, the YapTab's resolution scheme attained with the previously presented compute_leader_node, completion and answer_resolution procedures, improves SLG-WAM's scheme in that it: (i) replaces the need for a completion stack; (ii) quickly determines when a generator node is a leader node; and (iii) automatically schedules the set of consumer nodes with unconsumed answers within a SCC.

Furthermore, in practice, we found that this solution simplifies the implementation of fundamental aspects that may influence the parallel system's efficiency. Sharing tabling suspensions is straightforward, as the worker requesting work only needs to update its private top dependency frame pointer to the one of the sharing worker. Concurrent accesses or updates to the shared suspension data can be synchronized through the use of a locking mechanism at the dependency frame level. The completion algorithm for shared branches can take advantage of the dependency frame data structure to avoid explicit communication and synchronization between workers.
4.3 Local Scheduling

The algorithms described in the previous subsections assume a batched scheduling strategy. We are interested in alternative tabling scheduling strategies in order to study its impact when combining tabling with parallelism. Local scheduling is an alternative tabling scheduling strategy that tries to evaluate subgoals as independently as possible [40]. Evaluation is done one SCC at a time, and answers are returned outside of a SCC only after that SCC is completely evaluated. In other words, with local scheduling answers will only be returned to the leader’s calling environment when its SCC is completely evaluated. Because local scheduling completes subgoals sooner, we can expect less complex dependencies when running in parallel. Figure 4.16 clarifies the differences between batched and local scheduling evaluation.

Figure 4.16: Batched versus local scheduling: an example.
At the top, Figure 4.16 illustrates the program code and query goal used for both evaluations. Below, the figure depicts the evaluation sequence for each scheduling strategy, which includes the resulting table space and the resulting forest of trees. The numbering of nodes denote the evaluation sequence.

The most interesting aspect that results from the figure, is how both strategies handle the evaluation of the tabled subgoal call \( b(X) \). The first answer for \( b(X) \) binds \( X \) to 1. Batched scheduling then proceeds executing as in standard Prolog with the continuation call \( b(Y) \), while local scheduling fails to find the complete set of answers for \( b(X) \) and therefore completes the SCC before returning answers to the calling environment.

For local scheduling, the variant subgoal calls to \( b(x) \) at steps 5 and 8 are resolved by executing compiled code directly from the trie structure associated with the completed subgoal \( b(X) \). For batched scheduling, the same variant subgoal calls lead to suspension points that are resolved by consuming answers as they are being found.

The clear advantage of local scheduling shown in the example of Figure 4.16 does not always hold. In batched scheduling when a new answer is found, variable bindings are automatically propagated to the calling environment. Since local scheduling delays answers, it does not benefit from this propagation, and instead, when explicitly returning the delayed answers, it incurs an extra overhead for copying them out of the table. Local scheduling does perform arbitrarily better than batched scheduling for applications that benefit from answer subsumption, that is, where we delete non-minimal answers every time a new answer is added to the table. On the other hand, Freire et al. [40] showed that on average local scheduling is 15% slower than batched scheduling.

We next present how local scheduling is implemented on top of batched scheduling. As the reader will see, it is straightforward to extend the engine to perform local scheduling.

To prevent answers from being returned to the calling environment of a generator node, after a new answer is found for a particular tabled subgoal, local scheduling fails and backtracks in order to search for the complete set of answers. Therefore, when backtracking to a generator node, we must also act like a consumer node to consume the answers that could not be returned to their environment. In our
approach, we implement a generator choice point also as a consumer choice point. Figure 4.17 illustrates how generators are differently handled if supporting batched or local scheduling.

For local scheduling, when we store a generator node we also allocate a dependency frame. The dependency frame is initialized similarly as for the consumer nodes. As an optimization we can avoid calling `compute_leader_node()` procedure to initialize the `DepFr.leader.cp` field, as it will always compute the new generator node as the leader node. To access subgoal frames, in batched scheduling we use the `CP.SG.FR` generator choice point field. In local scheduling we must use the `CP.DEF.FR` generator choice point field and follow the `DepFr.sg.fr` field of the dependency frame. Further, to fully implement local scheduling, we need to slightly change the `completion()` procedure. Figure 4.18 shows the modified pseudo-code.

There is a major change to the completion algorithm for local scheduling. As newly found answers cannot be immediately returned, we need to consume them at a later point. If we perform completion with success, we start consuming the set of answers that have been found by executing compiled code directly from the trie data structure associated with the completed subgoal. Otherwise, we must act like a consumer node and start consuming answers.
4.3. LOCAL SCHEDULING

completion(generator node G) {
  if (G is the current leader node) {
    df = TOP_DEP_FR
    while (DepFr_cons_cp(df) is younger than G) {
      if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        DepFr_back_cp(df) = G
        C = DepFr_cons_cp(df)
        restore_bindings(CP_TR(G), CP_TR(C))
        goto answer_resolution(C)
      }
      df = DepFr_next(df)
    }
    perform_completion()
    adjust_freeze_registers()
    goto completed_table_optimization(DepFr_sg_fr(CP_DEP_FR(G))) // new
  }
  CP_ALT(G) = answer_resolution // new
  load_first_answer_from_subgoal(DepFr_sg_fr(CP_DEP_FR(G))) // new
  proceed // new
}

Figure 4.18: Pseudo-code for completion() with a local scheduling strategy.

Empirical work from Freire et al. [40, 41] showed that, regarding the requirements of an application, the choice of the scheduling strategy can differently affect the memory usage, execution time and disk access patterns. Freire argues [38] that there is no single best scheduling strategy, and whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. As a means of achieving the best possible performance, Freire and Warren [42] proposed the ability of using multiple strategies within the same evaluation, by supporting mixed-strategy evaluation at the predicate level.

We believe that YapTab is more suitable than the SLG-WAM to be extended to support a mixed-strategy evaluation. In result of its clear design based on the dependency frame data structure, extending YapTab to use multiple strategies at the predicate level seems straightforward. Only two features have to be addressed: (i) support strategy-specific Prolog declarations like ':- batched path/2.' in order to allow the user to define the strategy to be used to resolve the subgoals of a given predicate; (ii) at compile time generate appropriate tabling instructions, such as batched_new_answer or local_completion, accordingly to the declared strategy for the predicate. With these two simple compiler extensions we are able to use all the algorithms described and already implemented for batched and for local scheduling without any further modification. Although in this work we concentrated on the
issues concerning the exploitation of parallel implementation, we expect to exploit a mixed-strategy evaluation in the future.

4.4 Chapter Summary

In this chapter we introduced the YapTab engine. YapTab extends the Yap Prolog system to support sequential tabling in Prolog programs. YapTab’s implementation is largely based on the SLG-WAM approach to tabling.

We started by presenting the SLG-WAM abstract machine, as first implemented in the XSB system, and then we focused on its key aspects, namely, the batched scheduling strategy, the incremental completion optimization and its instruction set for tabling.

Next, we discussed the motivation for the YapTab design and described the main issues in extending the Yap Prolog system to support sequential tabling. We introduced a novel data structure, the dependency frame, and a new completion detection algorithm not based on the intrinsically sequential completion stack. YapTab innovates by considering that the control of leader detection and scheduling of unconsumed answers should be done at the level of the data structures corresponding to variant calls to tabled subgoals.

To further study the impact of alternative scheduling strategies when combining tabling with parallelism, we implemented an alternative strategy, local scheduling, and described how it was implemented on top of batched scheduling.
Chapter 5
Parallel Tabling

In this chapter we propose two new computational models to efficiently implement the parallel evaluation of tabled logic programs. We start by describing related work to get an overall view of alternative approaches to parallel tabling. Next, we introduce and detail the fundamental aspects underlying the new computational models, and then we discuss their advantages and disadvantages. Last, we focus on the elected computational model to discuss its implementation framework.

5.1 Related Work

One important advantage of logic programming is that it allows the implicit exploitation of parallelism. This is true for SLD based systems, and should also apply for SLG based systems. A first proposal on how to exploit implicit parallelism in tabling systems was Freire’s Table-parallelism [39]. Table-parallelism resembles the Linda’s tuple-space model, in that it views the table space as a shared data structure through which cooperating agents may synchronize and communicate.

In the Table-parallelism model, each tabled subgoal is computed independently in a single computational thread, a generator thread. Each generator thread is associated with an unique tabled subgoal and it is responsible for fully exploiting its search tree in order to obtain the complete set of answers. As new answers are being produced, they are inserted in the table space. A generator thread dependent on other tabled subgoals will asynchronously consume answers as the correspondent generator threads
CHAPTER 5. PARALLEL TABLING

will make them available.

Within this model, parallelism results from having several generator threads running concurrently. Parallelism arising from non-tabled subgoals or from execution alternatives to tabled subgoals is not exploited. Moreover, in order to fully implement the model a deep redesign of the base tabling engine is required, including new scheduling strategies and a new completion algorithm. Load balancing for this model can also be a difficult task. When the number of tabled subgoals is large, the dependencies between them can be quite intricate. Even when the number of tabled subgoals is small, some subgoals may have much larger search spaces than others. We expect that the scheduling problem of selecting which subgoals to allocate to which processors would be even harder than for traditional parallel systems.

More recent work [46], proposes a different approach to the problem of exploiting implicit parallelism in tabled logic programs. Curiously, this new approach was also named as Table-parallelism. The approach is a consequence of a new sequential tabling scheme whose design simplifies the exploitation of parallelism. The new sequential tabling scheme is based on dynamic reordering of alternatives with variant calls, and it works in a single SLD tree without requiring suspension of goals and freezing of stacks. The alternatives leading to variant calls are denominated as looping alternatives. This dynamic alternative reordering strategy not only tables the answers to tabled subgoals, but also the looping alternatives. A tabled subgoal will repeatedly recompute its looping alternatives until a fixpoint is reached.

If we find a variant call to a tabled predicate when exploiting a subgoal $S$, the current alternative clause $A$ is tabled as a looping alternative and it is reordered and placed at the end of the alternative list for the call. Moreover, the variant call is not expanded immediately, given it can lead to an infinite loop. Instead, a failure is simulated in order for $A$ to be backtracked over. After exploiting all matching clauses, the subgoal $S$ enters a looping state, where the looping alternatives, if they exist, start being tried repeatedly. If no new answer for $S$ is added to the table in a complete cycle over the looping alternatives, then we can say that subgoal $S$ has reached its fixpoint. Within this model, parallelism arises if we schedule the multiple looping alternatives to different workers. Communication among the different workers can be done through the table space.

An important characteristic of tabling is that it avoids recomputation of tabled sub-
goals. An interesting point of the dynamic reordering strategy is that it avoids recomputation through performing recomputation. The process of retrying alternatives may cause redundant recomputations of the non-tabled subgoals that appear in the body of a looping alternative. It may also cause redundant consumption of answers if the body of a looping alternative contains more than one variant subgoal call. Furthermore, to really judge the potential of the model as proclaimed by the authors [4], a more detailed performance evaluation is needed.

We believe that parallelism may cause even more drawbacks in this model. A major problem in parallel execution with this model is the way alternatives may be scheduled to be recomputed. Assume, for instance, two workers, $W_1$ and $W_2$, recomputing two different looping alternatives for the same subgoal. Consider that within its alternative, $W_1$ consumes the available answers for a given subgoal $S$ and then backtracks to continue exploitation. Suppose that in the meantime $W_2$ finds a new answer for $S$. When $W_1$ exhausts its looping alternative, it has to recompute it from the beginning in order to consume the newly found answer. However, a similar situation may occur and $W_2$ may find another answer for $S$ that may lead to a new recomputation of the alternative owned by $W_1$. Therefore, parallelism may not come so naturally as for SLD evaluations and parallel execution may lead to doing more work.

There have been other proposals for concurrent tabling but in a distributed memory context. Hu [55] was the first to formulate a method for distributed tabled evaluation termed Multi-Processor SLG (SLGMP). This method matches subgoals with processors in a similar way to Freire’s approach [39]. Each processor gets a single subgoal and it is responsible for fully exploiting its search tree and obtain the complete set of answers. One of the main contributions of SLGMP is its controlled scheme of propagation of subgoal dependencies in order to safely perform distributed completion. An implementation prototype of SLGMP was developed, but as far as we know no results have been reported.

A different approach for distributed tabling was proposed by Damásio in [34]. The architecture for this proposal relies on four types of components: a goal manager that interfaces with the outside world; a table manager that selects the clients for storing tables; table storage clients that keep the consumers and answers of tables; and prover clients that perform evaluation. An interesting aspect of this proposal is the completion detection algorithm. It is based on a classical credit recovery
algorithm [64] for distributed termination detection. Dependencies among subgoals are not propagated and, instead, a controller client, associated with each SCC, controls the credits for its SCC and detects completion if the credits reach the zero value. An implementation prototype has also been developed, but further analysis is required.

Marques et al. [63] have proposed an initial design for an architecture for a multi-threaded tabling engine. Their first aim is to implement an engine capable of processing multiple query requests concurrently. The main idea behind this proposal seems very interesting, however the work is still in an initial stage.

5.2 Novel Models for Parallel Tabling

Our work is based on the observation that tabling is still about exploiting alternatives to finding answers for goals, and that or-parallel systems have precisely been designed to achieve this goal efficiently. Our suggestion is that all alternatives to subgoals should be amenable to parallel exploitation, be they from tabled or non-tabled subgoals, and that or-parallel frameworks can be used as the basis to do so. This gives an unified approach with two major advantages. First, it does not restrict parallelism to tabled subgoals, and, second, it can draw from the large experience in implementing or-parallel systems. We believe that this approach can be an efficient model for the exploitation of parallelism in tabling-based systems.

One of the important characteristics of tabling-based systems is that some subgoals need to suspend on other subgoals to obtain the full set of answers. Or-parallel systems also need to suspend, either while waiting for leftmostness in the case of side-effects, or to avoid speculative execution. The need for suspending introduces an interesting similarity between tabling and or-parallelism that influenced our work. We therefore propose two new computational models, the OPT and TOP models.

To develop an efficient parallel tabling system we believe that it should exploit maximum parallelism and take maximum advantage of current parallel and tabling technology. A key idea in our proposals is that we want to explore in parallel all the available alternatives, be they from generator, consumer or interior nodes. For efficiency reasons we are also most interested in multi-sequential systems [110], that is, in systems where workers compute independently in the search tree, and mainly communicate with each
other to fetch work.

5.2.1 Or-Parallelism within Tabling (OPT)

In this first approach, that we name *Or-Parallelism within Tabling (OPT)*, parallel evaluation is done by a set of independent tabling engines that may share different common branches of the search tree during execution. Each worker can be considered a sequential tabling engine that fully implements the tabling operations: access the table space to insert new subgoals or answers; allocate data structures for the different types of nodes; suspend tabled subgoals; resume subcomputations to consume newly found answers; and complete private (not shared) subgoals. As most of the computation time is spent in exploiting the search tree involved in a tabled evaluation, we can say that tabling is the base component of the system.

The or-parallel component of the system is triggered to allow synchronized access to the shared parts of the execution tree, in order to get new work when a worker runs out of alternatives to exploit, and to perform completion of shared subgoals. Unexploited alternatives should be made available for parallel execution, regardless of whether they originate from generator, consumer or interior nodes. From the viewpoint of SLG resolution, the OPT computational model generalizes the Warren’s multi-sequential engine framework for the exploitation of or-parallelism. Or-parallelism stems from having several engines that implement SLG resolution, instead of implementing Prolog’s SLD resolution.

Figure 5.1 illustrates how parallelism can be exploited in the OPT model. It assumes two workers, \( W_1 \) and \( W_2 \), and it represents a possible evaluation for the following program code with \(?- a(X)\) as the query goal.

\[
\begin{align*}
&:- \text{table a/1.} \\
&a(X) :- a(X). \quad \quad \text{a(X) :- b(X).} \\
&b(1). \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{b(X) :- ...} \\
&b(X) :- ... \\
&?- a(X). 
\end{align*}
\]

Consider that worker \( W_1 \) executes the query goal. It first inserts an entry for the
tabled subgoal \(a(X)\) into the table space and creates a generator node for it. The execution of the first alternative leads to a recursive call for \(a(X)\), \(W_1\) hence creates a consumer node for \(a(X)\) and, because there are no available answers, it backtracks. The next alternative finds a non-tabled subgoal \(b(X)\) for which an interior node is created. The first alternative for \(b(X)\) succeeds and an answer for \(a(X)\) is therefore found: \(a(1)\). The worker inserts the newly found answer in the table and then starts exploiting the next alternative for \(b(X)\).

![Diagram](image)

Figure 5.1: Exploiting parallelism in the OPT model.

At this point, worker \(W_2\) moves in to share work. Consider that worker \(W_1\) decides to share all of its private nodes. The two workers will share three nodes: the generator node for \(a(X)\), the consumer node for \(a(X)\), and the interior node for \(b(X)\). Worker \(W_2\) takes the next unexploited alternative of \(b(X)\) and from now on, both workers can find further answers for \(a(X)\) and any of them can restart the shared consumer node.

### 5.2.2 Tabling within Or-Parallelism (TOP)

The second approach, that we name *Tabling within Or-Parallelism (TOP)*, considers that a parallel evaluation is performed by a set of independent WAM engines, each managing an unique branch of the search tree at a time. These base engines are extended to include direct support to the basic table access operations, that allow the insertion of new subgoals and answers.

We have seen that subgoals in tabling based systems need to suspend on other subgoals to obtain the full set of answers. Or-parallel systems also need to suspend, either while...
waiting for leftmostness in the case of side-effects, or to avoid speculative execution. The need for suspending introduces an important similarity between tabling and or-parallelism. The TOP approach therefore unifies or-parallel suspensions and suspensions due to tabling. When exploiting parallelism, some branches may be suspended, say, because they are speculative or not leftmost, or because they include consumer nodes waiting for more answers, while others are available for parallel execution. In TOP, the or-parallel suspension mechanism is extended to also manage the suspensions related to the tabling evaluation. Consequently, a suspended branch can wake up for reasons such as, new answers have been found for the consumer node on that branch, the branch becoming leftmost, or just for lack of non-speculative work in the search tree. The TOP name arises from the fact that tabled evaluation is attained by embracing the tabling suspension mechanism within the or-parallel component.

Figure 5.2 illustrates how parallelism is exploited under this approach for the same previous program. We can observe from the left figure that as soon as \( W_1 \) suspends on consumer node for \( a(X) \), it makes the whole branch public and only after it backtracks to the upper node. The suspended branch thus stops being the responsibility of \( W_1 \) and becomes, instead, shared work that anyone can wake up when new answers to \( a(X) \) are found.

![Figure 5.2: Exploiting parallelism in the TOP model.](image)

Continuing the execution, \( W_1 \) finds an answer for subgoal \( a(X) \) in the first alternative for subgoal \( b(X) \). So, when worker \( W_2 \) starts looking for work, it can choose whether to resume the consumer node with the newly found answer or to ask worker \( W_1 \) to share his private nodes. The right figure assumes that the first option was chosen.
5.2.3 Comparing the Models

The TOP model is a very attractive model, as it provides a clean and systematic unification of tabling and or-parallel suspensions. Workers have a clearly defined position, because a worker always occupies the tip of a single branch in the search tree. Everything else is shared work. It also has practical advantages, such as the fact that in this approach we can guarantee that a suspended branch will only appear once, instead of possibly several times for several workers. On the other hand, as suspended nodes are always shared in or-parallel systems, the unified suspension may result in having a larger public part of the tree, which may increase overheads. Besides, in order to support all forms of suspension with minimal overhead, the unified suspension mechanism must be implemented efficiently.

In TOP, we have a standard Prolog system extended with an or-parallel/tabling component. If adopting SLG-WAM for tabling, this means that TOP is most adequate for, say, binding arrays models [112, 111] for the or-parallel component, as a result of the similar cactus stack organization that both approaches use. An alternative for tabling is Demoen and Sagonas’s CAT [35] model. CAT seems to fulfill best the requirements of the TOP approach, since it assumes a linear stack for the current branch and uses an auxiliary area to save the suspended nodes. If implementing TOP based on CAT, then we should adopt for the or-parallel component an environment copying model [6, 5] as it fits best with the kind of operations that CAT introduces.

On the other hand, the OPT approach offers interesting advantages. First, it reduces to a minimum the overlap between or-parallelism and tabling. In OPT we have a tabling system extended with an or-parallel component. Moreover, it enables different combinations for or-parallelism and tabling, giving implementors the highest degree of freedom. For instance, one can use the SLG-WAM for tabling, and environment copying or binding arrays for or-parallelism.

Taking into account the advantages and disadvantages presented, we decided to focus our work on the design and implementation of the OPT model. Our choice seems the most natural as we believe that the OPT approach gives the highest degree of orthogonality between or-parallelism and tabling. The hierarchy of or-parallelism within tabling results in a property that one can take advantage of to structure the design and thus simplify the implementation.
5.2. NOVEL MODELS FOR PARALLEL TABLING

5.2.4 Framework Motivation for the OPT Model

We adopted a framework based on the YapOr and YapTab engines in order to implement the OPT model. We choose to use environment copying for or-parallelism and SLG-WAM for tabling based on the fact that these are, respectively, two of the most successful or-parallel and tabling engines. In our case, we already had the experience of implementing environment copying in the Yap Prolog, the YapOr system, with excellent performance results when compared with the Muse system [78, 81]. Adopting YapOr for the or-parallel component of the combined system was therefore our first choice.

On the other hand, YapTab was initially developed based on the SLG-WAM because, at the time, SLG-WAM was the most, and perhaps unique, successful tabling engine. The later appearance of the CAT [35] and CHAT [36] approaches to tabling, opened new paths and raised questions about the direction our work should follow. Instead of freezing computations, CAT uses an external data area to where it copies suspended computations. It turns out that CAT may have arbitrarily worst behavior than the SLG-WAM for some programs, and thus, a variation of the CAT approach, the CHAT, was later proposed to overcome some limitations of the CAT design. CHAT is a hybrid approach that combines certain features of the SLG-WAM with others of CAT. It innovates by introducing a technique for freezing stacks without using neither freeze registers nor stack copying. CHAT still copies the choice point and the trail stacks but not the environment and heap stacks. Instead, the latter are protected by manipulating pointers in the choice points.

We considered these new models as alternatives to the SLG-WAM, but after studying and considering their integration with an or-parallel component, we decided not to change the course of our work because CAT and CHAT have major problems for support parallelism over YapOr. First, to take best advantage of CAT or CHAT we need to have separate environment and choice point stacks, but Yap has an integrated local stack. Second, and more importantly, we believe that CHAT is not appropriate for parallel execution and that CAT is less suitable than the SLG-WAM to an efficient extension to or-parallelism.

Regarding CHAT, we argue that it is its choice point manipulation technique that makes it inappropriate as a base model to support parallel execution. Consider, for
example, two different workers, \( \mathcal{W}_1 \) and \( \mathcal{W}_2 \), exploiting alternative branches from a public choice point \( \mathcal{N} \) and \( \mathcal{W}_1 \) suspending a computation that requires manipulating pointers in \( \mathcal{N} \). Obviously, parallelism is not compatible with this kind of choice point manipulation. If \( \mathcal{W}_2 \) backtracks to \( \mathcal{N} \) then we can expect arbitrary behavior when \( \mathcal{W}_2 \) restores \( \mathcal{N} \)'s pointers.

As the SLG-WAM, CAT assumes an incremental completion technique in order to be more efficient in terms of memory consumption and to minimize the size of stacks to be copied. It was precisely this incremental completion principle that we believe it is less suitable to an efficient extension of the model to or-parallelism. CAT implements incremental completion through an incremental copying mechanism that saves intermediate states of the execution stacks. The mechanism works as follows: when suspending a consumer node, the state of the computation is saved to a proper CAT area up to the nearest generator node \( \mathcal{G} \) on the current branch, in such a way that if execution fails back to \( \mathcal{G} \), all younger consumer nodes have saved all information needed for their restoration. If \( \mathcal{G} \) is a leader node, on reaching fixpoint, completion can occur and the space for the CAT area can be freed. Otherwise, to allow for the younger consumers to be further restored, since backtracking over \( \mathcal{G} \) will occur we need to perform an incremental state saving. Incremental saving is always done up to the next nearest generator node and linked to the CAT areas previously saved up to \( \mathcal{G} \). This incremental saving of computational states maximizes sharing between common state segments and therefore, avoids double copying of the same segments.

In sequential tabling, the notion of leader node only makes sense if that node is a generator node. However, if we want to preserve incremental completion efficiency in a parallel tabling environment, we need to enlarge the concept behind the notion of leader node. Consider, for example, the situation from Figure 5.3. Starting from a common public node, worker \( \mathcal{W}_1 \) takes the leftmost alternative while worker \( \mathcal{W}_2 \) takes the rightmost. While exploiting their alternatives, \( \mathcal{W}_1 \) calls a tabled subgoal \( a \) and \( \mathcal{W}_2 \) calls a tabled subgoal \( b \). As this is the first call to both subgoals, a generator node is stored for each one. Next, each worker calls the tabled subgoal firstly called by the other, and consumer nodes are therefore allocated. At that point, we may question at which node we should check for completion? Intuitively, we might choose a node that is common to both branches and the youngest common node seems the better choice. As an alternative, we might store a dummy generator node at the beginning of the stacks
in order to guarantee that there is always an older generator node where we will check for completion. Obviously, if adopting this latter approach, incremental completion is not practicable and the efficiency of the model in terms of memory consumption and the size of stacks to be saved and later reinstalled is the worst possible.

As motivated by the example, if one adopts the youngest common node approach, then in a parallel tabled evaluation any kind of node (generator, consumer or interior) may be a leader node. Moreover, situations where a worker has several consumer nodes but not a single generator node are common. The efficiency of the CAT's incremental completion technique is based in the fact that the next place where completion may take place is in the upper generator node and that between two generator nodes there cannot exist another completion point. Parallel tabling does not preserve these properties. As an example, consider the situation from Figure 5.4.

The figure shows three workers, $W_1$, $W_2$ and $W_3$ executing a tabled evaluation in parallel. The left sub-figure shows a situation where $W_2$ and $W_3$ are about to suspend, respectively, the consumer nodes for the tabled subgoals $a$ and $b$. The sub-figure on the right shows the resulting state if the youngest common node approach is adopted for suspension. Note that nodes $N_1$ and $N_3$ are, respectively, the youngest common nodes to the branches of the generator and consumer nodes for $a$ and $b$. Therefore, consumer node for $a$ is suspended at $N_1$ and consumer node for $b$ is suspended at $N_3$. 

![Figure 5.3: Which is the leader node?](image)
Assume now that no more suspensions occur until $W_2$ and $W_3$ both backtrack from $N_3$. Such a situation leads to a major problem. How should the last worker leaving $N_2$ handle the suspension for $b$?

To solve this problem we need a very flexible mechanism that can decide when a suspension depends on upper suspensions. Besides, even if such mechanism is efficiently implemented, introducing parallelism over CAT would activate incremental saving whenever backtracking from public nodes. Moreover, incremental saving should be performed up to the parent node, as potentially it can hold other suspensions or be the next completion point. Obviously, this node-to-node segmentation of the incremental saving technique will degrade the efficiency of any parallel system. The problems behind the management of incremental completion in parallel tabling were the major reason why we were unwilling to change our initial framework choice.

5.3 Chapter Summary

In this chapter we proposed two novel computational models for parallel tabled evaluation, OPT and TOP models, and we discussed their fundamental aspects, advantages and drawbacks. We also discussed two related approaches to exploit parallelism from
tabled logic programs, the Table-parallelism approach from Freire et al. [39] and the Table-parallelism approach from Guo and Gupta [46].

We then motivated for a framework based on the YapOr and YapTab engines to implement the OPT model and stated the reasons for our choice. In the next two chapters we present the details for the implementation.

YapOr's engine was recently extended [32] to support two newer or-parallel binding approaches based on the Sparse Binding Array [27, 25] and on the Copy-On-Write [28] models. Therefore, we aim at integrating these binding models with YapTab in order to enlarge the combinations for the or-parallel tabling engine.
Chapter 6

OPTYap: The Or-Parallel Tabling Engine

This chapter presents the implementation details for the OPTYap engine. OPTYap is an or-parallel tabling system that implements the OPT computational model. As introduced in previous chapters, the OPT model is based on environment copying for the or-parallel component, and on the SLG-WAM for the tabling component. Our initial design only supports parallel tabled evaluation for definite clauses.

We start by presenting an overall view of the main issues involved in the implementation of the or-parallel tabling engine and then we introduce and detail the new data areas, data structures and algorithms required to implement it.

6.1 Implementation Overview

In our model, a set of independent workers will execute a tabled program by traversing a search tree where each node is a candidate entry point for parallelism. Each worker physically owns its environment, that is, a set of stacks, and shares the data structures that support tabling and scheduling. During execution, the search tree is implicitly divided into a public and private regions. Workers in their private region execute nearly as in sequential tabling. Workers exploiting the public region of the search tree must be able to synchronize in order to ensure the correctness of the tabling operations.
Parallel execution requires novel algorithms in a number of different situations. In some cases, parallel execution is straightforward, such as when backtracking to a public generator or to an interior node in order to take the next available alternative; when backtracking to a public consumer node to take the next unconsumed answer; or when inserting new answers into the table space. However, parallel execution can be quite complex in other situations. Therefore, it is a crucial implementation issue to achieve efficiency within the parallel tabling system. Complex cases include completion, resumption of computations, and the fixpoint check procedure, when operating over the public part of the execution tree. In a parallel tabling system, the relative positions of generator and consumer nodes are not as clear as for sequential systems, hence we need more complex algorithms to determine whether a node can be a leader node and to determine whether a SCC can be completed. As we shall see, the condition of being a leader node is not, by itself, sufficient to perform completion.

We follow a multi-sequential design. Therefore, a worker running out of alternatives to exploit enters in scheduling mode and uses the YapOr scheduler to search for busy workers with unexploited work. Alternatives are made available for parallel execution, regardless of whether they originate from generator, consumer or interior nodes. A worker is said to have shareable work if it contains private nodes with unexploited alternatives or with unconsumed answers. When a worker shares work with another worker, incremental copying is used to set the environment for the requesting worker.

### 6.2 The Parallel Data Area

A crucial part for the efficiency of a parallel system is how concurrent handling of shared data is achieved and synchronized. In this section we present the data-area design that allows for an efficient management of data structures in OPTYap. Memory allocation in OPTYap follows the same organization as in YapOr (please refer to Figure 3.6). Memory is divided into a global addressing space and a collection of local spaces, each one supporting one system worker. The global space includes the code area and a parallel data area that consists of all the data structures required to support concurrent execution. OPTYap extends the parallel data area to include the table and dependency spaces inherited from YapTab. A new data space preserves the stacks of suspended branches with dependencies in other branches (further details are given in
6.2. THE PARALLEL DATA AREA

section 6.7).

6.2.1 Memory Organization

The parallel data area stores data structures that may be accessed and updated concurrently. A major source of overhead regarding data access or update in parallel systems are memory cache misses and page faults. To deal with these, we need to achieve good locality for these data structures.

An important characteristic of almost all parallel data structures in the parallel data area is that elements of the same type are linked together to improve the efficiency of the common procedures that search through a chain until a certain condition is met. Hence, a good heuristic for increasing locality is to organize memory in such a way that data structures that are near at the abstract chain level, are also near at the memory level.

Modern computer architectures use pages to handle memory. Pages are fixed size blocks of contiguous memory cells. If we guarantee that most consecutive memory references are also physically consecutive, we may obtain access to the whole set of references when loading a memory page. Based on this characteristic, we adopt a page organization scheme in order to split memory among different data structures resident in the parallel data area. Figure 6.1 gives an overview of the parallel data area memory organization.

![Figure 6.1: Using memory pages as the basis for the parallel data area.](image)

Figure 6.1 shows that each memory page only contains data structures of the same type. Whenever a new request for a data structure of type $T$ appears, the next available structure on one of the $T$ pages is returned. If there are no available structures in any $T$ page, then a new $T$ page must be requested. If there are pages already marked as free, as in the figure, then one of them is made to be of type
Otherwise, a new page can be released from a pool of unreleased pages. This is achieved by making the page given by the Unreleased_Pages pointer to be of type $\mathcal{T}$, and by updating the pointer to the next unreleased page. A page is freed when all its data structures are released. A free page can be immediately reassigned to a different structure type. Figure 6.2 details the parallel data area pages organization.

Access to pages of a given data type is synchronized by the page entry data structure. In Figure 6.2, PAGES_T is the page entry that allows access to the data structures of type $\mathcal{T}$. A page entry structure includes two data fields. The Pg_lock field implements a lock mechanism to synchronize access to available data structures, in such a way that only a worker at a time may be updating the chain of available pages or the set of available data structures. The Pg_free_pg field is a pointer to the first page with available data structures of the given type.
6.2.2 Page Management

The management of pages and data structures within pages is achieved by allocating a special page header structure at the beginning of each page and by uniformly splitting the remaining of each page in data structures of the type being handled. A page header consists of four fields. The PgHd_str_in_use field stores the number of structures in use within the page. When it goes to zero the page can be freed. The PgHd_free_str field points to the first available data structure within the page. The PgHd_next and PgHd_previous fields point, respectively, to the next and previous pages with available structures. Within a page, available data structures are linked through their next fields. Access to free pages is also synchronized by a proper page entry data structure, named PAGES_void. The management of theses pages is simple because the PgHd_next page header field is sufficient to maintain the chain of free pages.

Figures 6.3 and 6.4 present, respectively, the pseudo-code for allocating and freeing a data structure of a given page entry type.

```
alloc_struct(page entry pg_entry) {
    lock(Pg_lock(pg_entry))
    if (Pg_free_pg(pg_entry) == NULL) // if no available pages then ...
        Pg_free_pg(pg_entry) = alloc_page() // ... request a new page
    PgHd_str_in_use(header)++
    str = PgHd_free_str(header)
    PgHd_free_str(header) = struct_next(str)
    if (PgHd_free_str(header) == NULL) { // if no available structures then...
        Pg_free_pg(pg_entry) = PgHd_next(header) // ... move to next page
        if (PgHd_next(header) != NULL)
            PgHd_previous(PgHd_next(header)) = NULL
    }
    unlock(Pg_lock(pg_entry))
    return str
}
```

Figure 6.3: Pseudo-code for alloc_struct().

The alloc_struct() procedure initially checks for available pages. If there are no pages a new one is requested through a call to alloc_page(). Next, we get the first available structure from the page we obtained and update the page header to point to the next available structure. If no more structures are available then the page is fully used. Hence, we update the page entry at hand to point to the next page with available structures.
free_struct(page entry pg_entry, data structure str) {
    header = page_header(str)    // header of the page that includes str
    lock(Pg_lock(pg_entry))
    if (--PgHd_str_in_use(header) == 0) { // if no structures in use then ...
        // ... put page free
        if (PgHd_previous(header)) {
            PgHd_next(PgHd_previous(header)) = PgHd_next(header)
            if (PgHd_next(header) != NULL)
                PgHd_previous(PgHd_next(header)) = PgHd_previous(header)
        } else {
            Pg_free_pg(pg_entry) = PgHd_next(header)
            if (PgHd_next(header) != NULL)
                PgHd_previous(PgHd_next(header)) = NULL
        }
        free_page(header)
    } else {
        struct_next(str) = PgHd_free_str(header)
        PgHd_free_str(header) = str
        if (struct_next(str) == NULL) { // if first available structure then ...
            // ... put page available
            PgHd_previous(header) = NULL
            PgHd_next(header) = Pg_free_pg(pg_entry)
            if (PgHd_next(header) != NULL)
                PgHd_previous(PgHd_next(header)) = header
            Pg_free_pg(pg_entry) = header
        }
    }
    unlock(Pg_lock(pg_entry))
}

Figure 6.4: Pseudo-code for free_struct().

The free_struct() procedure starts by determining if the page that includes the structure being released is fully available, that is, without any other structure being used. If this is the case the page stops being of the current type and instead it is made free. Otherwise, the structure is chained in the available structures within the page, and if it is the first structure made available then the page is also chained in the available pages for that type.

The management scheme attained with the alloc_struct() and free_struct() procedures enables local references for data structures of the same type. Subsequent allocate requests for data structures of the same type are serviced from the same memory page, and data structures being freed are chained within their own pages in order to keep locality of reference in further requests. Moreover, reclaiming unused pages is trivial as a simple reference count is sufficient to detect unused pages; allocating and freeing data structures are fast, constant-time operations, all we have to do is to move a structure to or from a list of free structures; and memory fragmentation
is minimal, the only wasted space is the unused portion at the end of a page when it cannot accommodate any more data structures.

To the best of our knowledge, the idea of page-based allocation of shared memory was first proposed by Bonwick for his Solaris Slab memory allocator [15]. Bonwick also proposes several alignment mechanisms in order to reduce cache misses. Our performance evaluation has not shown the need for such sophisticated mechanisms in OPTYap.

### 6.2.3 Improving Page Management for Answer Trie Nodes

During parallel evaluation, some data structures may induce high lock contention in the page entry access, because of higher rates of concurrent allocating and release requests. Through experimentation, we observed that this problem mainly occurs with answer trie nodes. In order to attenuate these overheads, we introduced a different mechanism to specifically deal with answer trie nodes. The idea is that: each worker maintains a private pre-allocated set of available answer trie nodes. When a worker runs out of pre-allocated answer trie nodes, it asks for an available answer trie node page and pre-allocates all the structures in it. To implement that mechanism, a new local register is necessary and a different procedure to request for available data structures is used. We next present the pseudo-code for that procedure.

```c
get_struct(page entry pg_entry, data structure local_str) {
    str = local_str
    if (str == NULL) { // if no available pre-allocated structures then ...
        // ... get an available page and pre-allocate all the structures in it
        lock(Pg_lock(pg_entry))
        if (Pg_free_pg(pg_entry) == NULL)
            Pg_free_pg(pg_entry) = alloc_page()
        header = Pg_free_pg(pg_entry)
        PgHd_str_in_use(header) = structs_per_page(pg_entry)
        str = PgHd_free_str(header)
        PgHd_free_str(header) = NULL
        Pg_free_pg(pg_entry) = PgHd_next(header)
        unlock(Pg_lock(pg_entry))
    }
    local_str = struct_next(str)
    return str
}
```

Figure 6.5: Pseudo-code for get_struct().

The get_struct() procedure includes support for the pre-allocation mechanism and
it replaces the alloc_struct() procedure when dealing with requests for answer trie nodes. The second argument is the local register that points to the next available pre-allocated data structure.

The procedure starts by checking if pre-allocated data structures are available. If this is the case, it gets the first available structure and updates the local register to point to the next pre-allocated structure. Otherwise, a new page is requested and the local register is made to point to the first available structure within that page. Moreover, the page is marked as fully used and the page entry is updated to the next page with available structures.

6.3 Concurrent Table Access

The table space is the major data area open to concurrent access operations in a parallel tabling environment. To maximize parallelism, whilst minimizing overheads, accessing and updating the table space must be carefully controlled. Reader/writer locks are the ideal implementation scheme for this purpose. However, several different approaches may be taken. One is to have an unique lock for the table, thus enabling a single writer for the whole table space; or one can have one lock per table entry, allowing one writer per predicate; or one lock per path, allowing one writer per subgoal call; or one lock per trie node, to attain least contention on locks; or hybrid locking schemes combining the above.

6.3.1 Trie Structures

The table data structures, and mainly the subgoal trie and answer trie structures, should be protected from races when operations that can change their structure are being executed. The tabling operations that change the subgoal trie and answer trie structures are the tabled subgoal call operation and the new answer operation.

Three different situations may occur when executing a tabled subgoal call operation. If the subgoal in hand is the first call to a tabled predicate, then a complete path of subgoal trie nodes is inserted into the subgoal trie structure. The opposite is when the subgoal is a variant of a subgoal in the table space, then no subgoal trie nodes are
inserted or updated, and thus, the subgoal trie structure remains unaltered. Last, if the subgoal is partially common to other tabled subgoals, only the divergent subgoal trie path is inserted into the subgoal trie structure. A similar set of situations may occur for the new answer operation. The difference is that the new answer operation works over the answer trie structure instead of the subgoal trie structure.

A table locking scheme must consider the situations described above. To better understand the peculiarities behind alternative locking schemes, we next give a more detailed description about the organization and handling of trie structures. Figure 6.6 illustrates the trie structure organization by focusing in more detail on one of the answer trie structures previously presented in Figure 4.3, including the complete set of the trie nodes contents and dependencies.

Figure 6.6: Detailing the trie structure organization.
A trie node is a data structure with four main data fields. The TrNode_entry stores the term that represents the node; the TrNode_next is a pointer to the sibling node that represents an alternative path; the TrNode_parent is a back pointer to the preceding node on path; and the TrNode_child is a pointer to the next node on path.

The figure presents the organization for the answer trie structures. The subgoal trie structures are organized similarly. The difference resides in how the TrNode_child field of the leaves trie nodes are processed. In an answer trie structure, the TrNode_child field of the leaves answer trie nodes forms a chain through the answers already stored in the table. In the subgoal trie structure, the TrNode_child field of the subgoal trie leaves gives access to the correspondent subgoal frame (please refer to subsection 4.2.2).

The completed table optimization allows compiled code execution from a trie. The optimization requires that answer trie nodes include two extra fields. One field, the TrNode_instr, stores the compiled instruction that implements unification for the term stored in the node. The other, the TrNode_or_arg field, stores the number of sibling nodes and supports the worker load computation scheme (see subsection 3.3.3). For simplicity, these fields were not included in Figure 6.6.

Besides the nodes needed to represent the several alternative paths, a root node marks the beginning of a trie structure. In Figure 6.6, the root node is the one represented with a ‘-’ in the TrNode_entry field. This root node synchronizes access to the first level of sibling nodes (nodes with terms e and b in the figure). Its usefulness can be better understood through Figure 6.7. It illustrates a trie_node_check_insert() call sequence in the context of a new answer operation. For a tabled subgoal call operation a similar sequence will be used.

```c
// SG_FR is the subgoal frame for the subgoal in hand
// (T1, ..., Tn) are the substitution factors for the new answer

current_node = SgFr_answer_trie(SG_FR) // start from the root node
current_node = trie_node_check_insert(T1, current_node)
... current_node = trie_node_check_insert(Tn, current_node)
```

Figure 6.7: trie_node_check_insert() call sequence for the new answer operation.

The trie_node_check_insert() is called by tabled subgoal call and new answer operations to traverse the subgoal and answer trie structures. It is called for each
term that represents the path being checked or inserted. Given a term \( T \) and a trie node \( P \), the `trie_node_check_insert()` procedure returns the child trie node of \( P \) that represents the given term \( T \). If such node was not already inserted by a previous operation then a new trie node to represent \( T \) is allocated and inserted as a child of \( P \).

Figure 6.8 introduces the algorithm that implements the `trie_node_check_insert()` procedure. Initially the algorithm traverses the chain of sibling nodes that represent alternative paths from the given parent node and checks for one representing the given term. If such a node is found then execution is stopped and the node returned. Otherwise, in order to represent the given term a new trie node is allocated and inserted in the beginning of the chain. We should stress that trie nodes corresponding to new paths are inserted in the trie structure through invocation of the `new_trie_node()` procedure. This procedure allocates new trie nodes, and it initializes the fields of the newly allocated node. The `TrNode_entry`, `TrNode_next`, `TrNode_parent` and `TrNode_child` fields are respectively initialized with the first, second, third and forth argument.

```
trie_node_check_insert(term t, trie node parent) {
    // check if the node representing t is already inserted
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_entry(child) == t)
            // node representing t found
            return child
        child = TrNode_next(child)
    }
    // insert a new node to represent t
    child = new_trie_node(t, TrNode_child(parent), parent, NULL)
    TrNode_child(parent) = child
    return child
}
```

Figure 6.8: Pseudo-code for `trie_node_check_insert()`.

We should mention that at this point we are still not considering any locking scheme to synchronize access to the trie structures. Furthermore, currently we do not support dynamic tries, that is, using tries to represent clauses for dynamic predicates. The locking schemes that we present next assume therefore that, whilst evaluating a subgoal, we cannot remove trie nodes from the tables.
6.3.2 Table Locking Schemes

We are now ready to discuss the different locking schemes. In a nutshell, we can say that there are two critical issues that determines the efficiency of a table locking scheme. One is the lock duration, that is, the amount of time a data structure is locked. The other is the lock grain, that is, the amount of data structures that are protected through a single lock request. It is the balance between lock duration and lock grain that compromises the efficiency of different table locking approaches. For instance, if the lock scheme is short duration or fine grained, then inserting many trie nodes in sequence, corresponding a long trie path, may result in a large number of lock requests. On the other hand, if the lock scheme is long duration or coarse grain, then going through a trie path without extending or updating its trie structure, may unnecessarily lock data and prevent possible concurrent access by others.

OPTYap implements four alternative locking schemes to deal with concurrent accesses to the table space data structures, the Table Lock at Entry Level scheme, the Table Lock at Node Level scheme, the Table Lock at Write Level scheme, and the Table Lock at Write Level - Allocate Before Check scheme.

The Table Lock at Entry Level (TLEL) scheme was the first table locking scheme implemented in OPTYap. The TLEL scheme allows a single writer per subgoal trie structure and a single writer per answer trie structure. To do so, it uses the table entries and the subgoal frames to lock, respectively, the subgoal trie and answer trie structures. Within this scheme, a single lock request is sufficient to protect the trie structure subject to concurrent access (coarse grain lock scheme). However, the trie structure is only unlocked when the path for the subgoal/answer in hand was completely traversed (long duration lock scheme).

The main drawback of TLEL is the contention resulting from its lock duration scheme. We then implemented a new lock scheme, the Table Lock at Node Level (TLNL). The TLNL only enables a single writer per chain of sibling nodes that represent alternative paths from a common parent node. Its implementation leads to extending the trie node data structure with a new TrNode_lock field, used to lock access to the node’s children. This scheme has the advantage that in order to traverse a trie structure each node on path only needs to be locked once. Within this scheme, the number of lock requests is proportional to the length of the path, and the period of time a node is
locked is proportional to the average time needed to traverse the node (mean duration lock scheme). Remark however, that a lock on a node synchronizes access to the chain of children nodes (fine grain lock scheme) and not to the node itself.

To fully implement this node level lock scheme, it is also necessary to adapt the procedure responsible for traversing trie structures. Figure 6.9 shows the pseudo-code that implements the trie_node_check_insert() procedure to support the TLNL scheme. The main difference from the original trie_node_check_insert() procedure is that here we lock the parent node while accessing its children nodes.

```c
trie_node_check_insert(term t, trie node parent) {
    lock(TrNode_lock(parent)) // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_entry(child) == t) {
            unlock(TrNode_lock(parent)) // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(t, TrNode_child(parent), parent, NULL)
    TrNode_child(parent) = child
    unlock(TrNode_lock(parent)) // unlocking before return
    return child
}
```

Figure 6.9: Pseudo-code for trie_node_check_insert() with a TLNL scheme.

An important drawback of the TLNL scheme is that the amount of memory in the parallel data area can increase substantially. During larger tabled evaluations, the trie nodes, and mainly the answer trie nodes, are the major data types responsible for the high percentage of memory pages being used in the parallel data area. Including an extra field in the subgoal and answer trie node data structure leads, respectively, to a 25% and 16% size growth. Due to the high number of trie nodes pages, this ratio can proportionally reflect the parallel data area memory usage.

We next developed a new scheme, the Table Lock at Write Level (TLWL) scheme, in order to avoid the TLNL drawbacks without losing its benefits. In fact, the TLWL scheme improves over TLNL by reducing memory usage, whilst also reducing lock duration. Like TLNL, the TLWL scheme only enables a single writer per chain of sibling nodes that represent alternative paths to a common parent node. However, in TLWL, the common parent node is only locked when writing to the table is likely.
Figure 6.10 presents the pseudo-code that implements the TLWL scheme. Initially, the chain of sibling nodes that succeed the given parent node is traversed without locking. Only when the given term is not found is that we lock the parent node. This avoids locking when the term already exists in the chain. Moreover, it delays locking while insertion of a new node to represent the term is not likely. Notice that we need to check if, during our attempt to lock, other worker expanded the chain to include the given term.

```c
trie_node_check_insert(term t, trie node parent) {
    child = TrNode_child(parent) // keep the initial child node
    initial_child = child
    while (child) { // traverse the initial chain of sibling nodes ...
        if (TrNode_entry(child) == t) // ... searching for t
            return child
        child = TrNode_next(child)
    }
    lock(GLOBAL_locks[hash_node(parent)]) // locking the common parent node
    // traverse the nodes inserted in the meantime by other workers before ...
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_entry(child) == t) {
            unlock(GLOBAL_locks[hash_node(parent)]) // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    // ... insert a new node to represent t
    child = new_trie_node(t, TrNode_child(parent), parent, NULL)
    TrNode_child(parent) = child
    unlock(GLOBAL_locks[hash_node(parent)]) // unlocking before return
    return child
}
```

Figure 6.10: Pseudo-code for `trie_node_check_insert()` with a TLWL scheme.

It can be observed that TLWL maintains the lock granularity of TLNL (fine grain lock scheme), but reduces the lock duration (short duration lock scheme). On average, the number of lock requests in the TLWL scheme is lower, it ranges from zero to the number of nodes on path. The amount of time a node is locked is on average also smaller. It is the time needed to check the nodes that in the meantime were inserted by other workers, if any, plus the time needed to allocate and initialize a new node.

TLWL avoids the TLNL memory usage problem by replacing trie node lock fields (`TrNode_lock`) with a global array of lock entries (`GLOBAL_locks`). A locking node operation is achieved by applying an hash algorithm (`hash_node()`) to the node address in order to index the global array entry that should be locked. This lock mechanism
preserves the TLNL lock semantics, whilst reducing the memory needed to implement locks to a fixed sized global array.

Lastly, we present the Table Lock at Write Level - Allocate Before Check (TLWL-ABC) scheme. The TLWL-ABC scheme is a variant of the TLWL scheme that follows the probable node insertion notion introduced in TLWL, but uses a different strategy on when to allocate a node. In order to reduce to a minimum the lock duration (minimum duration lock scheme), the TLWL-ABC scheme anticipates the allocation and initialization of nodes that are likely to be inserted in the table space to before locking. Note that, if in the meantime a different worker introduces first an identical node, we pay the cost of having pre-allocated an unnecessary node, that has to be additionally freed. Figure 6.11 presents the pseudo-code that implements the TLWL-ABC scheme.

```c
trie_node_check_insert(term t, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child
  while (child) {
    if (TrNode_entry(child) == t)
      return child
    child = TrNode_next(child)
  }
  // pre-allocate a node to represent t
  pre_alloc = new_trie_node(t, NULL, parent, NULL)
  lock(GLOBAL_locks[hash_node(parent)])
  child = TrNode_child(parent)
  TrNode_next(pre_alloc) = child
  while (child != initial_child) {
    if (TrNode_entry(child) == t) {
      // freeing the pre-allocated node
      free_struct(PAGES_trie_nodes, pre_alloc)
      unlock(GLOBAL_locks[hash_node(parent)])
      return child
    }
    child = TrNode_next(child)
  }
  // inserting the pre-allocated node
  TrNode_child(parent) = pre_alloc
  unlock(GLOBAL_locks[hash_node(parent)])
  return pre_alloc
}
```

Figure 6.11: Pseudo-code for trie_node_check_insert() with a TLWL-ABC scheme.

OPTYap supports all these table locking schemes. The TLWL scheme is the default scheme adopted for OPTYap. In Chapter 8 we present a detailed evaluation of the four alternative locking schemes, justifying our decision to choose TLWL as the default.
6.4 Data Frames Extensions

The or-frames, the subgoal frames and the dependency frames were the main data structures introduced to support the YapOr and YapTab models. To implement OPTYap, these data structures were extended to support parallel tabling.

6.4.1 Or-Frames

Or-frames synchronize access to the available alternatives for public choice points and support scheduling of work.

In the WAM, the choice point stack represents a single branch of the execution tree at a time. In the SLG-WAM, the choice point stack supports several different branches at a time. This leads to non-linearity in choice points. In other words, between two choice points for adjacent nodes in a branch there may exist several other choice points representing different branches. Hence, the notion of being public has to be clarified.

A worker can physically share a choice point \( C \), physically in the sense that it holds \( C \) on its stacks, while it is not logically sharing \( C \), logically in the sense that its current branch contains \( C \).

OPTYap considers that a physically shared choice point is a public choice point. When sharing work, the whole set of choice points being incrementally copied are made public, be they on the current branch of the sharing worker or not. This maximizes parallelism and simplifies the further management of suspended branches. The whole set of data structures representing the execution dependencies can be shared without changing its structure. However, the or-frame data structure has to store additional information to reflect the new choice point environment. Figure 6.12 shows an example that illustrates the new or-frame data fields.

The example is presented through three sub-figures. The sub-figure on the left shows the evaluation being considered. The sub-figure in the middle presents nodes dependencies at three different points of the evaluation. The nodes are presented linearly to reflect the physical choice point stack order. A link between two nodes indicates adjacent nodes on a branch. Situation (a) presents node dependencies after a worker \( \mathcal{P} \) had traversed nodes \( N_1 \) and \( N_2 \), suspended on \( N_2 \), backtracked to \( N_1 \), and traversed nodes \( N_3 \) and \( N_4 \). Situation (b) considers that worker \( \mathcal{P} \) accepted a sharing work
request from worker $Q$ and that it has made public the whole set of nodes. Last, situation (c) assumes that $P$ suspends on $N_4$, backtracks to $N_3$ and follows to node $N_5$ (note that $N_5$ is not public).

The sub-figure on the right presents or-frames dependencies at the end of situation (c). Observe that both workers hold the whole set of public nodes despite $N_2$ not being on either worker’s current branch and $N_4$ not being on $P$’s current branch. Remember that $\text{OrFr\_members}$ stores the set of workers which contain the choice point on their branch. A new or-frame data field, $\text{OrFr\_owners}$, stores the number of workers that hold the choice point on their stack, be it on their branch or not. The $\text{OrFr\_members}$ field allows worker $Q$, in situation (c), to determine that it is the unique worker with node $N_4$ on its branch. The $\text{OrFr\_members}$ field allows worker $Q$ to know that there is another worker holding $N_4$. That worker may, through a completion or answer resolution operation, include $N_4$ on its branch.

Figure 6.12 also shows two other fields, the $\text{OrFr\_next}$ and $\text{OrFr\_next\_on\_stack}$ fields, and two registers controlling or-frames, $\text{TOP\_OR\_FR}$ and $\text{TOP\_OR\_FR\_ON\_STACK}$. Remember that $\text{TOP\_OR\_FR}$ allows access to the youngest or-frame on the worker’s branch, and that $\text{OrFr\_next}$ points to the parent or-frame on branch. The $\text{TOP\_OR\_FR\_ON\_STACK}$
is a new register that allows access to the youngest or-frame on stack, while the
OrFr_next_on_stack is a new or-frame data field that points to the or-frame that
corresponds to the preceding choice point on stack, in such a way that the choice
point stack order can be obtained starting from TOP.OR.FR.ON_STACK and following
the OrFr_next_on_stack fields.

To allow support for suspension of SCCs, the or-frame data structure includes two ad­
ditional fields. The OrFr_suspensions field points to the suspended SCCs stored in the
frame. The OrFr_nearest_suspnode field points to the next or-frame in the worker’s
list of or-frames with suspended SCCs that corresponds to the nearest youngest choice
point on stack. The process of suspending SCCs and the role that these new fields
play in the process is detailed in section 6.7.

6.4.2 Subgoal and Dependency Frames

Remember that subgoal frames provide access to answer trie structures, while depen­
dency frames support the fixpoint check procedure. A detailed description of YapTab’s
subgoal and dependency frames was given in subsection 4.2.4. Next, we present the
extensions introduced to deal with the OPT model. Both subgoal and dependency
frames include three additional data fields.

For the subgoal frames, these fields are: SgFr_lock, SgFr_worker and SgFr.top_or_fr.
SgFr.lock is a lock that synchronizes concurrent updates to the frame fields. It
can also be used to support the TLEL table lock scheme. SgFr.worker stores the
identification number for the worker that allocated the frame. SgFr.top_or_fr points
to the generator or-frame, if the generator choice point is shared, and otherwise to the
or-frame that corresponds to the youngest shared choice point on the generator choice
point branch. Both SgFr.worker and SgFr.top_or_fr are used to compute the leader
node information (see section 6.5).

The new fields in the dependency frames are: DepFr_lock, DepFr_gen_on_stack and
DepFr_top_or_fr. DepFr.lock synchronizes concurrent updates to the frame fields.
DepFr_gen_on_stack is a boolean that indicates whether the generator choice point
for the correspondent leader choice point is on stack or not. In OPT, a consumer
node can have its generator on other worker’s branch. DepFr.top_or_fr points to
the consumer or-frame, if the consumer choice point is shared, and otherwise to the
or-frame that corresponds to the youngest shared choice point on the consumer choice point branch. Both $\text{DepFr}\_\text{gen}\_\text{on}\_\text{stack}$ and $\text{DepFr}\_\text{top}\_\text{or}\_\text{fr}$ support the fixpoint check procedure for shared nodes (see section 6.6).

### 6.5 Leader Nodes

Or-parallel systems execute alternatives early. As a result, it is possible that generators will execute earlier, and in a different branch than in sequential execution (as an example, please refer to Figure 5.3). In fact, different workers may execute the generator and the consumer goals. Workers may have consumer nodes while not having the corresponding generator nodes in their branches. Conversely, the owner of a generator node can have consumer nodes being executed by several different workers. This may induce complex dependencies between workers, therefore requiring a more elaborate completion operation that may involve the branches from several workers.

To clarify the dependencies between generator and consumer nodes we introduce a new concept, the *Generator Dependency Node* (or GDN). Its purpose is to signal the nodes that are candidates to be leader nodes, therefore representing a similar role as that of the generator nodes for sequential tabling. A GDN is calculated whenever a new consumer node, say $C$, is created. It is defined as the youngest node $V$ on the current branch of $C$, that is an ancestor of the generator node $Q$ for $C$. Obviously, if $G$ belongs to the current branch of $C$ then $G$ is the GDN. On the other hand, if the worker allocating $C$ is not the one that allocated $G$ then the youngest node $D$ is a public node, but not necessarily $G$.

Figure 6.13 presents three different situations that better illustrate the GDN concept. $WG$ is always the worker that allocated the generator node $G$, $WC$ is the worker that is allocating a consumer node $C$, and the node pointed by the black arrow is the GDN for the new consumer.

In situation (a), the generator node $G$ is on the branch of the consumer node $C$, and thus, $G$ is the GDN. In situation (b), nodes $N_1$ and $N_2$ are on the branch of $C$ and both contain a branch leading to the generator $G$. As $N_2$ is the youngest node of both, it is the GDN. In situation (c), $N_1$ is the unique node that belongs to $C$'s branch and that also contains $G$ in a branch below. $N_2$ contains $G$ in a branch below, but it is not
on C’s branch, while \( \mathcal{N}_3 \) is on C’s branch, but it does not contain \( G \) in a branch below. Therefore, \( \mathcal{N}_1 \) is the GDN. Notice that in both cases (b) and (c) the GDN can be a generator, a consumer or an interior node.

Sequential tabling performs only completion detection at generator nodes. Our parallel tabling design perform completion at GDNs. The procedure to compute the leader node information when allocating a dependency frame for a new consumer node now relies on the GDN concept. Remember that it is through leader node information stored in the dependency frames that a node can determine whether it is a leader node. The main difference from the sequential tabling algorithm is that now we first hypothesize that the leader node for the consumer node in hand is its GDN, and not its generator node. Figure 6.14 presents the modified pseudo-code for the `compute_leader_node()` procedure.

The parallel `compute_leader_node()` procedure can be divided in two main blocks. The first block computes the GDN, and the second block computes leader node
6.5. LEADER NODES

compute_leader_node(dependency frame dep_fr) {
  // start by computing the generator dependency node
  sg_fr = DepFr_sg_fr(dep_fr)
  if (SgFr_worker(sg_fr) == WORKER_ID) {
    leader_cp = SgFr_gen_cp(sg_fr)
    on_stack = TRUE
  } else {
    or_fr = SgFr_top_or_fr(sg_fr)
    while (WORKER_ID is not in OrFr_members(or_fr))
      or_fr = OrFr_next(or_fr)
    leader_cp = OrFr_node(or_fr)
    on_stack = (SgFr_gen_cp(sg_fr) == leader_cp)
  }
  // and then compute the leader node
  df = TOP_DEP_FR
  while (DepFr_cons_cp(df) is younger than leader_cp) {
    if (leader_cp is equal to DepFr_leader_cp(df)) {
      on_stack = DepFr_gen_on_stack(df)
      break
    } else if (leader_cp is younger than DepFr_leader_cp(df)) {
      leader_cp = DepFr_leader_cp(df)
      on_stack = DepFr_gen_on_stack(df)
      break
    }
    df = DepFr_next(df)
  }
  DepFr_leader_cp(dep_fr) = leader_cp
  DepFr_gen_on_stack(dep_fr) = on_stack
}

Figure 6.14: Modified pseudo-code for compute_leader_node().

information to be stored in the DepFr_leader_cp field. Note that the procedure now also computes the value of the DepFr_gen_on_stack field. This field is initialized to TRUE when the generator node for the computed leader node is on stack. Otherwise it is initialized to FALSE.

The first code block checks if the worker allocating the consumer node is the one that allocated the generator node. If so, then we assume the generator node is the GDN. Otherwise, we are in one of the situation presented in Figure 6.13 and we must traverse the chain of or-frames, starting from the one given by the SgFr_top_or_fr pointer relative to the subgoal in hand, until we reach one in the consumer branch. The node for the common or-frame corresponds to the GDN.

Regarding the second code block, we first check the consumer nodes younger than the newly found GDN for an older dependency. Remark that as soon as an older dependency \( D \) is found in a consumer node \( C' \), the remaining consumer nodes, older
than $C'$ but younger than the GDN, do not need to be checked because the leader node computation ensures that they do not contain older dependencies than $D$. The previous computation of the leader node information for the consumer node $C'$ already represents the oldest dependency that includes the remaining consumer nodes. This argument is similar to the one proved for sequential tabling (remember subsection 4.2.7).

By now, the reader may have spotted an inconsistency between the original GDN definition and the code block that computes it. If a consumer node was allocated by the same worker that allocated the generator node, then the procedure assumes that the GDN is the generator node. However, there are situations where this is not true. Observe for instance Figure 6.15. Node $G$ is the GDN computed by the procedure, despite $N_2$ being, by definition, the correct GDN. As explained next, this inconsistency is intentional to achieve code efficiency.

![Diagram](image)

Figure 6.15: The generator dependency node inconsistency.

The key observation is that backtracking over a generator node $G$ without completing, it only happens when there is a suspension point younger than $G$ that depends on a node older than $G$. We can therefore infer that there must exist a consumer node $N_3$ such that $N_2$, or an ancestor $N_1$, is its correspondent GDN. Thus, if we execute the remaining `compute_leader_node()` procedure we will eventually conclude that the leader of the SCC that includes $C$ is correctly determined, even if starting from an incorrect node for the generator dependency. This optimization avoids the
computation time required to detect the GDN, which can be quite significant in more complex situations.

As a final note we should remark that due to the dependency frame’s design, concurrency is not a problem for the `compute_leader_node()` procedure. Observe, for example, the situation from Figure 6.16. Two workers, \( W_1 \) and \( W_2 \), exploiting different alternatives from a common public node, \( \mathcal{N}_4 \), are allocating new private consumer nodes. They compute the leader node information for the new dependency frames without requiring any explicit communication between both and without requiring any synchronization if consulting the common dependency frame for node \( \mathcal{N}_3 \). The resulting dependency chain for each worker is illustrated on each side of the figure. Note that the dependency frame for consumer node \( \mathcal{N}_3 \) is common to both workers. It is illustrated twice only for simplicity.

![Diagram of dependency frames](image)

Figure 6.16: Dependency frames in the parallel environment.

A new consumer node is always a private node and a new dependency frame is always the youngest dependency frame for a worker. The leader information stored in a dependency frame denotes the resulting leader node at the time the correspondent consumer node was allocated. Thus, after computing such information it remains unchanged. If when allocating a new consumer node the leader changes, the new leader information is only stored in the dependency frame for the new consumer, therefore not influencing others. With this scheme each worker views its own leader
node independently from the execution being done by others. Determining the leader node where several dependent SCCs from different workers may be completed together is the problem that we address next.

6.6 The Flow of Control

OPTYap is a multi-sequential system where workers may be in engine mode, that is, doing work, or in scheduling mode, that is, looking for work. Actual execution control of a parallel tabled evaluation mainly flows through four procedures. The process of completely evaluating SCCs is accomplished by the completion() and answer_resolution() procedures, while parallel synchronization is achieved by the getwork() and scheduler() procedures.

Here we focus on the flow of control in engine mode, that is on the completion(), answer_resolution() and getwork() procedures, and leave scheduling for a following section. Figure 6.17 presents a general overview of how control flows between the three procedures in discussion and how it flows within each procedure. The design and implementation details for each procedure are presented in detail next.

6.6.1 Public Completion

Detection of completion in sequential tabling is a complex problem. With the introduction of parallelism the complexity increases even further. The correctness and efficiency of the completion algorithm appear to be one of the most important issues in the implementation of a parallel tabling system.

Different paths may be followed when a worker \( W \) reaches a leader node for a SCC \( S \). The simplest case is when the node is private. In this case, we should proceed as for sequential tabling. Hence, \( W \) enters the sequential completion() procedure previously presented in Figure 4.13. Otherwise, the node is public, and there may exist dependencies on branches explored by other workers. Therefore, even when all younger consumer nodes on \( W \)'s stacks do not have unconsumed answers, completion cannot be performed. The reason for this is that the other workers can still influence \( S \). For instance, these workers may find new answers for a consumer node in \( S \), in
which case the consumer must be resumed to consume the new answers. As a result, in order to allow \( \mathcal{W} \) to continue execution it becomes necessary to suspend the SCC at hand.

Suspending in this context is obviously different from suspending consumer nodes.
Consumer nodes are suspended due to tabling evaluation. SCCs are suspended due to or-parallel execution to enable the current worker to proceed executing work. Suspending a SCC includes saving the SCC's stacks to a proper space in the parallel data area, leaving in the leader node a reference to where the stacks were saved, and readjusting the freeze registers and the stack and frame pointers (more details in section 6.7). If the worker did not suspend the SCC, hence not saving the stacks to the parallel data area, any future sharing work operation might damage the SCC's stacks and therefore make delayed completion unworkable. An alternative would be for the worker to wait until no one else could influence it and only then complete the SCC. Obviously, this is not an efficient strategy.

To deal with the new particularities arising with concurrent evaluation a novel completion procedure, public_completion(), implements completion detection for public leader nodes. Most often, the public_completion() procedure executes through backtracking to a public generator node whose next available alternative leads to the completion instruction. Remember that the completion instruction follows a table.try_me.single or a table.trust.me instruction and that it forces completion detection when all alternatives have been exploited for a generator node. Note that the completion instruction only needs to get executed once for each particular generator node. Further executions of completion detection are triggered by the fixpoint check procedure. As a consequence, after a completion instruction gets loaded, the OrFr.alt field of the correspondent or-frame is set to NULL. Remember that for public nodes, the next available alternative is stored in the OrFr.alt field of the correspondent or-frame.

Looking back to Figure 6.17, it can be observed that there are two other situations from where the public_completion() procedure is directly invoked for execution (search for the 'goto public_completion()' statement). A first situation occurs when resuming a suspended SCC, we restart execution by performing public_completion() at the leader of the resumed SCC. A second situation occurs when failing to a public leader node. The exception is when the public leader node is a generator node for the current SCC and it contains unexploited alternatives. In such cases, the current SCC is not fully exploited, and therefore we should exploit first such alternatives. This last situation can be better understood in subsection 6.6.3.

Figure 6.18 introduces the pseudo-code for the public_completion() procedure. The
first step in the algorithm is to check for younger consumer nodes with unconsumed answers. If there is such a node, we resume the computation to it. In parallel tabling, resuming a computation to an younger consumer node $C$ includes: (i) updating the $\text{DepFr}\_\text{back}\_\text{cp}$ dependency frame field of $C$ when the leader detecting for completion is older than the current reference stored in $\text{DepFr}\_\text{back}\_\text{cp}$ (details about the $\text{DepFr}\_\text{back}\_\text{cp}$ semantics for public nodes in subsection 6.6.2); (ii) setting the worker’s bit in the $\text{OrFr}\_\text{member}$ field for the or-frames in the branch being resumed; and (iii) using the forward trail to restore the bindings for the branch being resumed.

```plaintext
public_completion(public node N) {
  if (N is the current leader node) {
    // remember that TOP\_OR\_FR points to N's or-frame
    owners = OrFr_owners(TOP\_OR\_FR) // keep N's owners
    df = TOP\_DEP\_FR
    while (DepFr\_cons\_cp(df) is younger than N)) {
      if (DepFr\_last\_ans(df) != SgFr\_last\_answer(DepFr\_sg\_fr(df))) {
        // dependency frame with unconsumed answers
        lock(DepFr\_lock(df))
        DepFr\_back\_cp(df) = oldest(N, DepFr\_back\_cp(df))
        unlock(DepFr\_lock(df))
        restore\_member\_info(TOP\_OR\_FR, DepFr\_top\_or\_fr(df))
        C = DepFr\_cons\_cp(df)
        restore\_bindings(CP\_TR(N), CP\_TR(C))
        goto answer\_resolution(C)
      }
      df = DepFr\_next(df)
    }
    L = youngest\_node\_holding\_a\_suspended\_SCC\_to\_resume()
    if (L is equal or younger than N) {
      // L belongs to the current SCC
      suspend\_SCC(N)
      resume\_SCC(L)
      goto public\_completion(L)
    }
    if (owners == 1) {
      // the current SCC is completely evaluated
      perform\_public\_completion()
    } else {
      // other workers can still influence the current SCC
      suspend\_SCC(N)
    }
    goto getwork(N)
  }
  goto scheduler()
}
```

Figure 6.18: Pseudo-code for public\_completion().

If the algorithm does not find any younger consumer node with unconsumed answers it must check for suspended SCCs in the scope of its SCC. A suspended SCC should
be resumed if it contains consumer nodes with unconsumed answers. To resume a suspended SCC a worker needs to copy the saved stacks to the correct position in its own stacks, and thus, it has to suspend its current SCC first.

We thus adopted the strategy of resuming suspended SCCs only when the worker finds itself at a leader node, since this is a decision point where the worker either completes or suspends the current SCC. Hence, if the worker resumes a suspended SCC it does not introduce further dependencies. This is not the case if the worker would resume a suspended SCC $R$ as soon as it reached the node where it had suspended. In that situation, the worker would have to suspend its current SCC $S$, and after resuming $R$ it would probably have to also resume $S$ to continue its execution. A first disadvantage is that the worker would have to make more suspensions and resumptions. Moreover, if we resume earlier, $R$ may include consumer nodes with unconsumed answers that are common with $S$. On the other hand, in a leader node position, we know that the consumer nodes belonging to $S$ have consumed all the answers currently available, and thus if $R$ has to be resumed it is because it has consumer nodes with unconsumed answers that do not belong to $S$. More importantly, suspending in non-leader nodes leads to further complexity. Answers can be found in upper branches for suspensions made in lower nodes, and this can be very difficult to manage.

A SCC $S$ is completely evaluated when (i) there are no unconsumed answers in any consumer node in its scope, that is, in any consumer node belonging to $S$ or in any consumer node within a SCC suspended in a node belonging to $S$; and (ii) there is only a single worker owning its leader node $L$. Condition (ii) has to be satisfied first, that is, before the worker $W$ performing completion starts checking for younger consumer nodes with unconsumed answers. Otherwise, other workers may find new answers in the meantime for the consumer nodes already checked by $W$ and these workers may retire from owning $L$ before $W$ ends checking. As a result, $S$ may be incorrectly considered completely evaluated.

When a SCC is found to be completely evaluated then it is completed. Completing a SCC includes marking all dependent subgoals as complete; releasing the dependency and or-frames belonging to the complete branches, including the branches in suspended SCCs; releasing the frozen stacks and the memory space used to hold the stacks from suspended SCCs; and finally readjusting the freeze registers and the whole set of stack and frame pointers.
Our public completion algorithm has two major advantages. One is that the worker checking for completion determines if its current SCC is completely evaluated or not without requiring any explicit communication or synchronization with other workers. The other is that it uses the SCC as the unit for suspension. This latter advantage is very important since it simplifies the management of dependencies arising from branches not on stack. A leader node determines the position from where dependencies may exist in younger branches. As a suspension unit includes the whole SCC and suspension only occurs in leader node positions, we can simply use the leader node to represent the whole scope of a suspended SCC, and therefore simplify its management (section 6.7 details this issue).

### 6.6.2 Answer Resolution

The answer resolution operation loads tabled answers from the table space to the execution stacks. The operation also support the fixpoint check procedure. Usually, the answer_resolution() procedure gets executed through failure to a consumer node, in which case execution jumps to the answer_resolution instruction through the CP_ALT choice point field. The execution can also flow directly to the answer_resolution() procedure when scheduling for a backtracking node during the fixpoint check procedure (these are the cases for the goto answer_resolution() statement in Figure 6.17).

Figure 6.19 shows the pseudo-code that implements the answer resolution operation for the parallel environment. Comparing with the procedure previously presented in Figure 4.14 for sequential tabling, it can be observed that the new answer_resolution() procedure extends the sequential algorithm to support the new situations arising with parallelism.

Initialy, the procedure checks the consumer node \( C \) for unconsumed answers to be loaded for execution. If we have answers, execution will jump to them. Otherwise, if there are no such answers, we schedule for a backtracking node. Remember that a valid reference \( B \) in the DepFr_back_cp field of the dependency frame associated with \( C \) indicates that we are in a fixpoint check procedure. Therefore, we search for a consumer node with unconsumed answers. If found then answer resolution gets re-executed. Otherwise, we backtrack to the youngest node between the current leader node and \( B \). For both situations, the OrFr_member bitmaps and the bindings for the
answer_resolution(consumer node C) {
    DEP_FR = CP_DEP_FR(C)
    if (DepFr_last_ans(DEP_FR) != SgFr_last_answer(DepFr_sg_fr(DEP_FR))) {
        // unconsumed answers in current dependency frame
        load_next_answer_from_subgoal(DepFr_sg_fr(DEP_FR))
        proceed
    }
    dep_back_cp = DepFr_back_cp(DEP_FR)
    if (dep_back_cp == NULL) {
        if (C is a public node)
            goto scheduler()
        else
            backtrack_to(CP_B(C))
    }
    back_cp = youngest(DepFr_leader_cp(TOP_DEP_FR), dep_back_cp)
    df = DepFr_next(DEP_FR)
    while (DepFr_cons_cp(df) is younger than back_cp) {
        if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
            // dependency frame with unconsumed answers
            lock(DepFr_lock(df))
            DepFr_back_cp(df) = oldest(DepFr_back_cp(df), dep_back_cp)
            unlock(DepFr_lock(df))
            restore_member_info(TOP_OR_FR, DepFr_top_or_fr(df))
            back_cp = DepFr_cons_cp(df)
            restore_bindings(CP_TR(C), CP_TR(back_cp))
            goto answer_resolution(back_cp)
        }
        df = DepFr_next(df)
    }
    restore_member_info(TOP_OR_FR, CP_OR_FR(back_cp))
    restore_bindings(CP_TR(C), CP_TR(back_cp))
    if (back_cp is a public node)
        goto getwork(back_cp)
    else
        goto completion(back_cp)
}

Figure 6.19: Pseudo-code for answer_resolution().

branch being resumed should be restored. Moreover, if we backtrack to a consumer
node, the correspondent DepFr_back_cp field should be updated.

There are two interesting aspects, both related with the fixpoint check procedure, that
should be pointed in the answer_resolution() procedure. One is that the youngest
node between the current leader node and the node given by the C’s DepFr_back_cp
is used by the procedure as the node that limits the search for youngest consumer
nodes with unconsumed answers. The other is that if a consumer node B is scheduled
for backtracking, the DepFr_back_cp field associated with B is updated to the oldest
node between its current reference node and the node given by C’s DepFr_back_cp
field. In order clarify these aspects, Figure 6.20 illustrates two different sequences for a complete loop over the fixpoint check procedure. Both sequences start with a worker $W$ in a leader node position, and assume that all younger consumer nodes have unconsumed answers.

![Diagram](image)

Figure 6.20: Scheduling for a backtracking node in the parallel environment.

Regarding situation (a), the computation initially moves from leader node $L_2$ to consumer node $C_2$, which includes updating the DepFr_back_cp field of $C_2$ to the leader reference $L_2$. Then, while worker $W$ was consuming the available unconsumed answers for $C_2$, another worker, also in a fixpoint check procedure, updates $C_1$’s DepFr_back_cp to its leader reference, $L_1$ in the case. Thus, after consuming all available answers in $C_2$, $W$ is scheduled to consumer node $C_1$, but $C_1$’s DepFr_back_cp remains unchanged because it holds an older leader reference. Last, when all available unconsumed answers for $C_1$ have been consumed, $W$ backtracks to $L_2$. Despite $C_1$ holding a DepFr_back_cp reference to $L_1$, meaning that all generator and interior nodes younger
than $L_1$ are necessarily exploited, the current leader node is younger than $L_1$, and therefore backtracking should be performed first to $L_2$ in order to avoid computation from flowing to nodes outside the current SCC.

Situation (b) presents a slightly different sequence. Worker $W$ starts from a leader node $L_2$ that resumes the computation to consumer node $C_2$. Next, a different worker updates the $\text{DepFr.back.cp}$ field of $C_1$ while $W$ is consuming the available answers for $C_2$. However, when exploiting an unconsumed answer for $C_2$, $W$ allocates a new consumer node and as a consequence, changes its current leader node to become $L_1$. After all available answers for $C_2$ have been consumed, $C_1$ is scheduled for backtracking. The interesting difference from situation (a) happens when, at the end, after all available answers for $C_1$ have been consumed, $C_1$ is scheduled for backtracking. This results not only from the fact that $L_1$ is the current leader node, but also from the $L_1$ reference in the $\text{DepFr.back.cp}$ field of $C_1$ that allows us to conclude that the branch between the initial leader node $L_2$ and the current leader node $L_1$ is already exploited. Note that, nothing can be concluded about $L_1$, for instance, $L_1$ can still have available alternatives. Hence, using $\text{public.completion()}$ to continue execution for $L_1$ would be incorrect. We therefore use $\text{getwork()}$ to ensure the correct behavior, as discussed next.

### 6.6.3 Getwork

Getwork is the last flow control procedure. It contributes to the progress of a parallel tabled evaluation by moving to effective work. Remark that, despite this procedure being related with the process of getting a new piece of work, it is independent from the process of scheduling for a new piece of work. More precisely, we use getwork for public nodes bordering private regions, that is, the youngest public nodes on each branch, while scheduling works over interior nodes in the public region of the search tree.

The usual way to execute $\text{getwork()}$ is through failure to the youngest public node on the current branch, in which case the $\text{getwork}$ instruction gets loaded for execution. However, there are three other cases from where $\text{getwork()}$ is directly invoked to continue the execution. One occurs in the fixpoint check procedure to ensure the correct behavior of the computation when the leader node is scheduled for backtrack-
ing. The other two occur in the public completion algorithm and both are related with situations where the SCC in hand is removed from the execution stacks, either because it is completed or suspended.

Figure 6.21 presents the pseudo-code that implements the `getwork()` procedure. We can distinguish two blocks of code. The first block detects completion points and therefore makes the computation flow to the `public_completion()` procedure. The second block corresponds to or-parallel execution. It checks the associated or-frame for available alternatives and executes the next one, if any. Otherwise, it invokes the scheduler. Remember that the TOP.OR.FR register points to the or-frame for the youngest public node on the current branch, that is, the or-frame related with $\mathcal{N}$.

```plaintext
getwork(public node N) {
    // code for detecting completion points
    if (DepFr_leader_cp(TOP_DEP_FR) == N &&
        (DepFr_gen_on_stack(TOP_DEP_FR) == FALSE || OrFr_alt(TOP_OR_FR) == NULL))
        goto public_completion(N)

    // original code inherited from YapOr
    if (OrFr_alt(TOP_OR_FR) != NULL) {
        load_next_alternative_from_frame(TOP_OR_FR)
        proceed
    } else
        goto scheduler()
}
```

Figure 6.21: Pseudo-code for `getwork()`.

The `getwork()` procedure detects a completion point when $\mathcal{N}$ is the leader node pointed by the top dependency frame. The exception is if $\mathcal{N}$ is itself a generator node for a consumer node within the current SCC (DepFr_gen_on_stack(TOP_DEP_FR) == TRUE) and it contains unexploited alternatives (OrFr_alt(TOP_OR_FR) != NULL). In such cases, the current SCC is not fully exploited. Hence, we should exploit first the available alternatives, and only then invoke completion.

Figure 6.22 illustrates the complete set of situations where computation flows from `getwork()` to `public_completion()`. It distinguishes two different cases: `goto situations`, and `load situations`. The goto situations correspond to the completion points detected by `getwork()`. A load situation occurs when completion is loaded for execution from a generator node whose next available alternative points to a completion instruction. This situation occurs independently of the generator being leader or not.
CHAPTER 6. OPYAP: THE OR-PARALLEL TABLING ENGINE

6.7 SCC Suspension

Whenever a worker executing a public completion operation determines that the current SCC depends on branches being exploited outside the SCC, it should delay completion until no more dependencies exist. To allow the worker to proceed with the execution of other work it is convenient to suspend the current SCC at this point. Remark that SCC suspension is absolutely necessary for an environment copy based implementation. Environment copy requires coherency between workers for the sub-stacks corresponding to shared regions. Delayed completion would be incorrect if a SCC is not suspended and an incremental copying operation damages its stacks.

The SCC suspension procedure includes saving the stacks segments relative to the SCC being suspended to a proper space in the parallel data area and leaving a reference to where the stacks were saved in the leader node. This reference corresponds to the suspension frame data structure. The suspension frame is a novel data structure introduced to allow for suspended SCCs to be resumed. Figure 6.23 presents an example of suspension that illustrates how suspension frames relate with suspended SCCs.

The process of suspending a SCC works as follows. Initially, the set of stack segments corresponding to the SCC being suspended is copied to the parallel data area. After
that, a new suspension frame is allocated and a reference to it is stored in the or-frame relative to the leader node of the SCC being suspended. Finally, the whole set of stack and frame pointers are readjusted in order to correctly reflect the resulting computation state.

A suspension frame holds the following data from a suspended SCC: the values from the TOP_SG.FR, TOP_DEP.FR, and TOP.OR.FR.ON_STACK registers at the time the SCC was suspended, the pointers to the beginning of each area where the segments were saved, and the size of each suspended segment. The data stored in a suspension frame plus the data stored in the node that holds the reference to the suspension frame are sufficient to restore a suspended SCC to its original computation state.

Notice that we never need to suspend a SCC $S$ that does not contain private nodes. Otherwise, $S$ will repeatedly suspend for each worker sharing it. This is a safe optimization because, at least, one of the workers, say $W$, sharing $S$ will later suspend or complete $S$, either because the current SCC of $W$ includes $S$ and further private nodes, or because $W$ will be the last worker executing public completion over $S$.

In order to access the suspension frames for a particular node, the or-frame data structure was extended with a OrFr_suspensions extra field to point to a linked list of suspension frames for the node. The linked list is maintained through a SuspFr_next.
A suspended SCC is resumed when a worker executing completion in a public leader node finds that a suspended SCC in the scope of its current SCC contains consumer nodes with unconsumed answers. In order to find out which suspended SCCs need to be resumed, each worker maintains a list of suspended SCCs that may contain consumer nodes with unconsumed answers. In order to avoid frequent and redundant checking operations for suspended SCCs, a worker only checks for suspended SCCs when it is the last worker backtracking from a node \( \mathcal{N} \). If there are suspended SCCs, the or-frame associated with \( \mathcal{N} \) is included in the worker’s list of or-frames with suspended SCCs. If the or-frame already belongs to another worker’s list, it is not collected. This guarantees that each or-frame only belongs to a worker’s list at a time.

Each worker holds a `TOP_SUSP.FR` register that points to the list \( \mathcal{L} \) of or-frames with suspended SCCs. The list always starts with the or-frame of \( \mathcal{L} \) that corresponds to the youngest choice point on stack. The list \( \mathcal{L} \) is maintained through a new field `OrFr.nearest.suspnode` in the or-frame. The field always points to the next or-frame of \( \mathcal{L} \) that corresponds to the nearest younger choice point on stack. In this way we guarantee that the list of or-frames belonging to \( \mathcal{L} \) is traversed in stack order.

Figure 6.24 illustrates how or-frames referring suspended SCCs are linked. The figure assumes two workers, \( \mathcal{W}_1 \) and \( \mathcal{W}_2 \), and four public nodes containing suspended SCCs. For simplicity of illustration, the figure only presents the segment of the local stack that is shared between both workers.

The figure shows that even if a worker contains several or-frames with suspended SCCs, it may not refer the complete list. It further shows that an or-frame with suspended SCCs may not be in any linking list. The or-frames relative to nodes \( \mathcal{N}_1 \) and \( \mathcal{N}_4 \) are in the list for \( \mathcal{W}_1 \), the or-frame relative to \( \mathcal{N}_3 \) is in the list for \( \mathcal{W}_2 \), while the or-frame for \( \mathcal{N}_2 \) is not in any list. An or-frame with suspended SCCs does not belong to any worker’s list either if there still exist workers in the node, or if it is already known that none of the suspended SCCs contain consumer nodes with unconsumed answers. However, this latter case does not guarantees that the SCCs are completely evaluated. As a result of a completion operation performed above, workers can still be scheduled to include nodes belonging to those SCCs.

A worker executing public completion follows its list of or-frames with suspended
SCCs in order to search for SCCs to be resumed. It starts searching the suspended SCCs in the or-frame given by the TOP_SUSP.FR register and then it follows the OrFr.nearest.suspnode chain until either a suspended SCC with unconsumed answers is found or until reaching an or-frame corresponding to a node younger than the leader node executing completion. At the end of the process, it updates the TOP_SUSP.FR register to the or-frame where searching was aborted, either because a SCC was resumed there or because it corresponds to a node older than the leader node.

Resuming a SCC includes copying the previously saved stack segments in the parallel data area to the correct stack positions of the worker resuming the SCC. Therefore, in order to protect the current stack’s data from being lost, the worker has to suspend its current SCC first.

Figure 6.25 illustrates the management of suspended SCCs when searching for SCCs to resume. The figure considers a worker W executing public completion in a leader node N1 and assumes that the worker's list of or-frames with suspended SCCs refers two or-frames in its current SCC S1.

The search for SCCs to be resumed starts at the or-frame given by TOP_SUSP.FR.
CHAPTER 6. OPTYAP: THE OR-PARALLEL TABLING ENGINE

Figure 6.25: Resuming a SCC.

Assuming that the suspended SCC $S_4$ does not contain unconsumed answers, the search continues in the next or-frame in the list. Here, suppose that SCC $S_2$ does not have consumer nodes with unconsumed answers, but SCC $S_3$ does. At this point, the current SCC $S_1$ must be suspended. This includes storing the correspondent reference in the or-frame relative to its leader node $N_1$, and updating $\text{TOP_SUSP.FR}$ to the or-frame referring to the SCC to be resumed. Now we can resume $S_3$.

Resuming $S_3$ includes copying the set of suspended stack segments from the parallel data area to the correct position in $W$'s stacks; updating the $\text{OrFr-members}$ and $\text{OrFr-owners}$ info for the or-frames below the previous leader node ($N_1$ in this case), including the or-frames from $S_3$; adjusting the whole set of stack and frame pointers in order to reflect the previous computation state of $S_3$; and releasing the suspension frame related to $S_3$.

Still regarding Figure 6.25, notice that the or-frame relative to node $N_3$ was removed from $W$'s list of or-frames with suspended SCCs. This happens because $S_3$ may not include $N_3$ in its stack segments. For simplicity and efficiency, instead of checking $S_3$'s segments, we simply remove $N_3$'s or-frame from $W$'s list. Note that this is a safe decision as a SCC only depends from branches below the leader node and thus, if $S_3$ does not include $N_3$ then no new answers can be found for $S_1$'s consumer nodes.
Otherwise, if this is not the case then \( W \) or other workers can eventually be scheduled to a node held by \( S_4 \) and find new answers for at least one of its consumer nodes. In this case, when failing, these workers will necessarily backtrack through \( N_3 \), \( S_4 \)'s leader. Therefore, the last worker backtracking from \( N_3 \) will collect the or-frame relative to \( N_3 \) for its own list of or-frames with suspended SCCs, which allows \( S_4 \) to be later resumed when public completion is being executed in an upper leader node.

Remember even when a worker does not find any suspended SCC to resume, it may not always perform completion. This occurs when it is not the unique owner of the current leader node. Remember that a worker \( W \) containing a node \( N \) in its stacks is an owner of \( N \). A problem arises if \( W \) suspends the SCC that includes \( N \). \( W \) then retires from owning \( N \). Nevertheless, if \( W \) later resumes the suspended SCC then \( W \) again owns \( N \). Execution would be incorrect if a worker would complete a SCC based on being the unique owner of the current leader node \( L \), and then a suspended SCC that includes \( L \) was resumed. To overtake this problem, we assume that the number of owners of a node \( N \) corresponds to the number of representations of \( N \) in the computational environment, be \( N \) represented in the execution stacks of a worker or be \( N \) in the suspended stack segments of a SCC. Therefore, whenever a SCC is suspended, the OrFr.owners field of the or-frames belonging to the SCC remains unchanged.

## 6.8 Scheduling Work

Scheduling work is the scheduler's task. It is about efficiently distributing the available work for exploitation between the running workers. In a parallel tabling environment we have the extra constraint of keeping the correctness of sequential tabling semantics. A worker enters in scheduling mode when it runs out of work and returns to execution whenever a new piece of unexploited work is assigned to it by the scheduler.

Subsection 3.2 presented the YapOr's scheduler algorithm: *when a worker runs out of work it searches for the nearest unexploited alternative in its branch. If there is no such alternative, it selects a busy worker with excess of work load to share work with. If there is no such a worker, the idle worker tries to move to a better position in the search tree.*
The scheduler for the OPTYap engine is mainly based on YapOr's scheduler. All the scheduler strategies implemented for YapOr were used in OPTYap. However, extensions were introduced in order to preserve the correctness of tabling semantics. These extensions allow support for leader nodes, frozen stack segments, and suspended SCCs.

Figure 6.26 presents two different situations that illustrate how leader node semantics influences the usual scheduling for the nearest node with unexploited alternatives within the current branch. Situation (a) considers that the current leader node is equal or older than the nearest node with unexploited alternatives, while situation (b) considers that the current leader node is younger than the nearest node with unexploited alternatives.

![Diagram showing scheduling for the nearest node with unexploited alternatives.](image)

Figure 6.26: Scheduling for the nearest node with unexploited alternatives.

A node has available work if the OrFr_alt field of its relative of-frame is not NULL. Besides the usual instructions corresponding to unexploited alternatives, this includes the cases where the OrFr_alt field points to a completion or to an answer_resolution instruction.
The OPTYap model was designed to enclose the computation within a SCC until the SCC was suspended or completely evaluated. Thus, OPTYap introduces the constraint that the computation cannot flow outside the current SCC, and workers cannot be scheduled to execute at nodes older than their current leader node. Therefore, when scheduling for the nearest node with unexploited alternatives, if it is found that the current leader node is younger than the potential nearest node with unexploited alternatives, then the current leader node is the node scheduled to proceed with the evaluation. This is the case illustrated in situation (b) of Figure 6.26.

The next case is when the process of scheduling for the nearest node with unexploited alternatives does not return any node to proceed execution. The scheduler then starts searching for busy workers that can be requested for work. If such a worker $B$ is found, then the requesting worker moves up to the lowest node that is common to $B$, in order to become partially consistent with part of $B$. Otherwise, no busy worker was found, and the scheduler moves the idle worker to a better position in the search tree. Therefore, we can enumerate three different situations for a worker to move up to a node $N$: (i) $N$ is the nearest node with unexploited alternatives; (ii) $N$ is the lowest node common with the busy worker we found; or (iii) $N$ corresponds to a better position in the search tree.

The process of moving up in the search tree from a current node $N_i$ to a target node $N_f$ is mainly implemented by the move_up_one_node() procedure. This procedure is invoked for each node that has to be traversed until reaching $N_f$. The presence of frozen stack segments or the presence of suspended SCCs in the nodes being traversed influences and can even abort the usual moving up process. Figure 6.27 presents the pseudo-code that implements the move_up_one_node() procedure for OPTYap.

The argument for the move_up_one_node() procedure is the node $N_i$ where the idle worker $W$ is currently positioned at and from where it wants to move up one node. Initially, the procedure checks for frozen nodes on the stack to infer whether $W$ is moving within a SCC. If so, $W$ is simply deleted from member of the or-frame relative to $N_i$ and if it is the last worker leaving the frame then it checks for suspended SCCs to be collected.

The interesting case is when $W$ is not within a SCC. If $N_i$ holds a suspended SCC, then $W$ can safely resume it. If resumption does not take place, the procedure proceeds to check whether $N_i$ is a consumer node. Being this the case, $W$ is deleted from the
move_up_one_node(public node N) {
    // remember that TOP_OR_FR points to N's or-frame
    lock(OrFR_lock(TOP_OR_FR))

    // frozen nodes on stack?
    if (B_FZ is younger than N) {
        delete_from_bitmap(OrFr_members(TOP_OR_FR), WORKER_ID)
        if (OrFr_members(TOP_OR_FR) is empty) {
            collect_suspended_SCCs(TOP_OR_FR)
        }
        unlock(OrFR_lock(TOP_OR_FR))
        return CP_B(N)
    }

    // suspended SCCs to resume?
    if (N holds a suspended SCC to resume) {
        unlock(OrFR_lock(TOP_OR_FR))
        restore_bindings(TR, CP_TR(N))
        resume_SCC(N)
        goto public_completion(N)
    }

    // N is a consumer node?
    if (B_FZ == N) {
        delete_from_bitmap(OrFr_members(TOP_OR_FR), WORKER_ID)
        if (OrFr_owners(TOP_OR_FR) == 1) {
            complete_suspended_SCCs(TOP_OR_FR)
            unlock(OrFR_lock(TOP_OR_FR))
            return CP_B(N)
        }
    }

    // unique owner?
    if (OrFr_owners(TOP_OR_FR) == 1) {
        complete_suspended_SCCs(TOP_OR_FR)
        if (SgFr_gen_cp(TOP_SG_FR) == N)
            mark_subgoal_as_completed(TOP_SG_FR)
        free_struct(PAGES_or_frames, TOP_OR_FR)
        return CP_B(N)
    }

    delete_from_bitmap(OrFr_members(TOP_OR_FR), WORKER_ID)
    OrFr_owners(TOP_OR_FR) --
    unlock(OrFR_lock(TOP_OR_FR))
    return CP_B(N)
}

Figure 6.27: Pseudo-code for move_up_one_node().

members bitmap of the or-frame relative to $N_i$ and if $W$ is the unique owner of $N_i$ then the suspended SCCs in $N_i$ can be completed. Completion can be safely performed over the suspended SCCs in $N_i$ not only because the SCCs are completely evaluated, as none was previously resumed, but also because no more dependencies exist, as there
are no more branches below $N_i$.

The reasons given to complete the suspended SCCs in $N_i$ hold even if $N_i$ is not a consumer node, as long as $W$ is the unique owner of $N_i$. In such case, as $W$ is the last owner leaving $N_i$, the or-frame for $N_i$ can be freed and if $N_i$ is a generator node then its correspondent subgoal can be also marked as completed. Otherwise, $W$ is simply deleted from being member and owner of the or-frame relative to $N_i$.

The scheduler extensions presented are mainly related with tabling support. Further work is needed to implement and experiment with proper scheduling strategies that can take advantage of the parallel tabling environment, as the scheduling strategies inherited from the YapOr's scheduler were designed for an or-parallel model, and not for an or-parallel tabling model. Next, we propose two new scheduling strategies that explicitly deal with the flow of a parallel tabling evaluation:

- When a worker is looking for others with available work, the scheduler must give higher priority to work that contains suspended SCCs. By doing so, suspended SCCs can be resumed sooner, and therefore we increase the probability of an early successful completion. Furthermore, we may avoid further dependencies that would occur if the subgoals involved were not completed early.

- The scheduler must avoid sharing branches with consumer nodes. Consumer nodes correspond to frozen segments, and frozen segments involve extra copying of stack segments. Moreover, we may generated suspended SCCs that in turn contain repeated stack segments corresponding to shared frozen segments.

We believe that these strategies can contribute to a more efficient distribution of work for parallel tabling and thus we intend to further implement and experiment the impact of these strategies in OPTYap’s performance.

### 6.9 Local Scheduling

All the implementations issues described above assume a batched scheduling strategy. In this section we present how the batched based implementation for parallel tabling can be straightforwardly extended to support local scheduling.
Support for local scheduling in the parallel environment includes the extensions previously presented in subsection 4.3 to support local scheduling for sequential tabling. Remember that a generator choice point is implemented as a consumer choice point and that this includes allocating a dependency frame when storing a generator node. One should also remember that when a generator node loads the completion instruction for execution, it also updates the field for the next available alternative to the answer_resolution instruction, in order to guarantee that, subsequently, the node will act like a consumer node and consume the found answers.

Full support for the parallel execution with local scheduling is attained by considering the novel situation where a generator turned consumer is both a public node and the youngest node on stack. Figure 6.28 illustrates the case in point. Note that node $N$ obviously corresponds to the local scheduling implementation for generator nodes, as this is the unique case where the $\text{DepFr}_\text{leader}_\text{cp}$ field of a node references itself.

![Diagram](image)

Figure 6.28: Local scheduling situation requiring special implementation support.

The problem arising with this kind of situation is that $N$ can be computed as a leader node. This happens because the $\text{DepFr}_\text{leader}_\text{cp}$ field of the dependency frame corresponding to the top consumer node, that is $N$, references $N$. However, $N$ should only execute completion when it is found that no unconsumed answers are available. Implementation support for this special situation requires slight changes to the $\text{getwork()}$, $\text{answer_resolution()}$ and $\text{public completion()}$ procedures.

Figure 6.29 presents the modified pseudo-code for the $\text{getwork()}$ procedure. It introduces a single modification in the block of code that detects for completion points, by replacing $\text{NULL}$ for $\text{answer_resolution}$ in the test involving the $\text{OrFr}_\text{alt}$ field of the $\text{TOP.OR.FR}$ register, and by adding a new test condition that avoids completion detection for nodes in the local scheduling special situation. The first change is because, in local scheduling, the last update operation to the $\text{OrFr}_\text{alt}$ field relative
6.9. LOCAL SCHEDULING

to a generator node is to answer_resolution and not to NULL. The second change forces the nodes in the local scheduling special situation to act like consumer nodes and consume the newly found answers.

getwork(public node N) {
// code for detecting completion points
    if (DepFr_leader_cp(TOP_DEP_FR) == N &&
        (DepFr_gen_on_stack(TOP_DEP_FR) == FALSE ||
         (OrFr_alt(TOP.OR.FR) == answer_resolution && B_FZ != N))) // changed
        goto public_completion(N)
} ...

Figure 6.29: Pseudo-code for getwork() with a local scheduling strategy.

We next present in Figure 6.30 the new pseudo-code for the answer_resolution() procedure. When the node C executing the procedure is also the current leader node then it is known that we are in the presence of the local scheduling special situation, because a leader node never executes answer_resolution(). Notice that in this case, C forms a SCC with a single node, and thus, computation cannot flow to upper nodes while C remains on stack. Hence, if it is found that no unconsumed answers are available for C, no work can be done for the current SCC. The new code for answer_resolution() detects this kind of situations and moves the flow of the computation to the public_completion() procedure, which is where they are resolved.

answer_resolution(consumer node C) {
    DEP.FR = CP_DEP.FR(C)
    if (DepFr_last_ans(DEP.FR) != SgFr_last_answer(DepFr_sg_fr(DEP.FR))) {
        load_next_answer_from_subgoal(DepFr_sg_fr(DEP.FR))
        proceed
    }
    if (DepFr_leader_cp(TOP_DEP.FR) == C) { // new
        goto public_completion(C) // new
    }
    dep_back_cp = DepFr_back_cp(DEP.FR)
} ...

Figure 6.30: Pseudo-code for answer_resolution() with a local scheduling strategy.

Figure 6.31 presents the extended pseudo-code for the public_completion() procedure. It includes the following modifications: adding a test condition to avoid getwork() when facing the local scheduling special situation; and introducing a new block of code to specifically process the situation. Notice that the new block of code is
CHAPTER 6. OPTYAP: THE OR-PARALLEL TABLING ENGINE

positioned after the code that implements completion or suspension for the previous SCC on stack because this is from where a local scheduling special situation can result.

```java
public_completion(public node N) {
    if (N is the current leader node) {
        if (DepFr_leader_cp(TOP_DEP_FR) != N) // new
            goto getwork(N)

        // start of new block of code due to local scheduling
        df = TOP_DEP_FR
        if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
            unlock(OrFr_lock(TOP_OR_FR))
            load_next_answer_from_subgoal(DepFr_sg_fr(df))
            proceed
        } // no unconsumed answers found
        lock(OrFr_lock(TOP_OR_FR))
        if (OrFr_owners(TOP_OR_FR) != 1) {
            // remove N from stack
            delete_from_bitmap(OrFr_members(TOP_OR_FR), WORKER_ID)
            OrFr_owners(TOP_OR_FR) =
            TOP_OR_FR_ON_STACK = OrFr_next_on_stack(TOP_OR_FR)
            TOP_DEP_FR = DepFr_next(df)
            unlock(OrFr_lock(TOP_OR_FR))
            if (SgFr_gen_cp(TOP_SG_FR) == N)
                TOP_SG_FR = SgFr_next(TOP_SG_FR)
                TOP_OR_FR = CP_OR_FR(CP_B(N))
                adjust_freeze_registers()
                backtrack_to(CP_B(N))
            } else {
                // make N an interior node
                OrFr_alt(TOP_OR_FR) = NULL
                unlock(OrFr_lock(TOP_OR_FR))
                TOP_DEP_FR = DepFr_next(df)
                free_struct(PAGES_dependency_frames, df)
                adjust_freeze_registers()
                goto scheduler()
            }
        } // end of new block of code
    }
    goto scheduler()
}
```

Figure 6.31: Pseudo-code for public_completion() with a local scheduling strategy.

The new block of code starts by checking node $N$ for unconsumed answers to proceed execution. If it is found that no unconsumed answers are available in $N$ then no work can be done for the current SCC and therefore execution only proceeds if the current SCC changes. If the worker $W$ executing the procedure is not the unique owner of $N$ then $N$ is removed from the stacks of $W$ and execution is backtracked to the parent
node on the branch. Otherwise, as $W$ is the unique owner, $N$ is made to be an interior node without available alternatives so that $W$ can enter in scheduling mode and get a new piece of work at a different node. Execution is not immediately backtracked in this second case because $W$ is the last worker leaving $N$ and therefore it must use the scheduler’s `move_up_one_node()` procedure to move in the search tree to guarantee that, for instance, $N$ is checked for suspended SCCs and the subgoal associated with $N$ is marked as completed.

6.10 Chapter Summary

This chapter introduced the OPTYap engine. To the best of our knowledge, OPTYap is the first implementation of a parallel tabling engine for logic programming systems. OPTYap extends Yap’s efficient sequential Prolog engine to support or-parallel execution of tabled logic programs. It follows OPT’s computation model for parallel tabling, and it builds on SLG-WAM for tabling and on environment copying for or-parallelism.

We discussed the complete set of major problems addressed during OPTYap’s development, which included: memory management; concurrent table access; public completion; scheduling decisions for parallel tabling; and SCC suspension. For each problem we presented and described the new data areas, data structures and algorithms introduced to efficiently solve them. We can emphasize the GDN concept of signalling nodes that are candidates to be leader nodes; the new algorithms to quickly compute and detect leader nodes; the novel termination detection scheme to allow completion in public nodes; the assumption of SCCs as the units for suspension; and the different locking schemes for concurrent table access.
CHAPTER 6. OPTYAP: THE OR-PARALLEL TABLING ENGINE
Chapter 7

Speculative Work

In [21], Cieplelewski defines speculative work as *work which would not be done in a system with one processor*. The definition clearly shows that speculative work is an implementation problem for parallelism, that must be addressed carefully in order to reduce its impact.

The presence of pruning operators during or-parallel execution introduces the problem of speculative work [52, 53, 8, 13]. Prolog has an explicit pruning operator, the *cut* operator. When a computation executes a cut operation, all branches to the right of the cut are pruned. Computations that can potentially be pruned are thus *speculative*. Earlier execution of such computations may result in wasted effort compared to sequential execution.

In this chapter, we discuss the problems arising with speculative computations and introduce the mechanisms used in YapOr and OPTYap to deal with it. Initially, we introduce the cut semantics and its particular behavior within or-parallel systems. After that we present the cut scheme currently implemented in YapOr and describe the main implementation details. Then we discuss speculative tabling execution and present the support actually implemented in OPTYap.
7.1 Cut Semantics

Cut is a system built-in predicate that is represented by the ! symbol. Its execution results in pruning all the branches to the right of the cut scope branch. The cut scope branch starts at the current node and finishes at the node corresponding to the predicate containing the cut. Cut is an asymmetric pruning operator because it only prunes branches at the right. Other parallel Prolog systems implement symmetric pruning operators, with a generic name of commit. The execution of commit results in pruning both to the left and to the right. YapOr and OPTYap do not yet support symmetric pruning operators.

Figure 7.1 gives a general overview of cut semantics by illustrating the left to right execution of a particular program containing cuts. The query goal a(X) leads the computation to the first alternative of predicate a and the query goal is replaced with the body of the first clause of a, where !(a) means a cut with the scope a. If !(a) gets executed, all the right branches until the node corresponding to predicate a, inclusively, should be pruned.

As execution continues, b(X) is called and its first alternative succeeds by binding X to value 1. The cut corresponding to the first alternative of a is invoked next and thus the remaining alternatives for predicates a and b are pruned. As a consequence, the nodes representing both predicates can be removed.
7.2 Cut within the Or-Parallel Environment

In a sequential system, cut only prunes alternatives whose exploitation has not been started yet. This does not hold for parallel systems, as cut can prune alternatives that are being exploited by other workers or that have already been completely exploited. Therefore, cut’s semantics in a parallel environment have a new dimension. First, a pruning operation cannot always be completely performed if the cut scope branch is not leftmost, because the operation itself may be pruned by the execution of other pruning operation in a branch to the left. Similarly, an answer for the query goal in a non-leftmost branch may not be valid. Last, pruning a branch puts out of work the workers exploiting such branch.

Ali [3] showed that speculative work can be completely banned from a parallel system if proper rules are applied. However, as such rules severely restrict the parallel exploitation of work, most or-parallel systems allow speculative work as it is their main source of parallelism. Speculative branches can be controlled more or less tightly. Ideally, we would prune all branches as soon as they become useless. In practice, deciding if a computation is still speculative or already useless can be quite complex when nested cuts with intersecting scopes are considered.

7.2.1 Our Cut Scheme

Implementing cut in an or-parallel system entails two main problems: the cut operation may have to prune work from the shared region of the search tree and the execution of the branch where the cut is found may itself be speculative. When implementing cut, the following rule must be preserved: *we cannot prune branches that would not be pruned if our own branch will be pruned by a branch to the left.*

YapOr currently implements a cut scheme based on the ideas presented by Ali and Karlsson [8] that prunes useless work as early as possible. The worker executing cut, must go up in the tree until it reaches either the cut scope choice point or a choice point with workers executing branches to the left. While going up it may find workers in branches to the right. If so, it sends them a signal informing them that their branches have been pruned. When receiving such a signal, workers must backtrack to the shared part of the tree and become idle workers again.
Note that a worker may not be able to complete a cut if there are workers in left branches, as they can themselves prune the current cut. In these cases, one says the cut was left pending. In YapOr, a cut is left pending on the first (youngest) node $N$ that has left branches. A pending cut can be resumed only when all workers to the left backtrack into the shared node $N$. It will then be the responsibility of the last worker backtracking to $N$ to continue the execution of the pending cut.

Even if a cut is left pending in a node $N$, there may be branches, older than $N$, that correspond to useless work according to the cut rule mentioned above. YapOr's cut scheme prunes these branches immediately. To illustrate how these branches can be detected we present in Figure 7.2 a small example taken from [8]. To better understand the example, we index the repeated calls to the same predicate by call order. For instance, the node representing the first call to predicate $p$ is referred as $p_1$, the second as $p_2$ and successively. We also write $p_n^{(i)}$ to denote the $ith$ alternative of node $p_n$. Notice also, that we use the mark $!$ in the branch of an alternative to indicate that it contains at least one cut predicate.

Figure 7.2: Pruning in the parallel environment.

Assume that a worker $W$, in a parallel execution environment, is computing the branch corresponding to $[p_1^{(1)}, q_1^{(1)}, p_2^{(1)}, q_2^{(2)}, p_3^{(2)}]$. There are only two branches to the left, corresponding to alternatives $p_3^{(1)}$ and $q_2^{(1)}$. If there are workers within alternative
7.2. **CUT WITHIN THE OR-PARALLEL ENVIRONMENT**

If $p_3^{(1)}$ then $\mathcal{W}$ cannot execute any pruning at all because $p_3^{(1)}$ is marked as containing cuts. A potential execution of a pruning operation in $p_3^{(1)}$ will invalidate any cut executed in $p_3^{(2)}$ by $\mathcal{W}$. Therefore, $\mathcal{W}$ saves a pending cut marker in $p_3$ and when the work in $p_3^{(1)}$ terminates, pruning for the pending cut is executed.

Let's now assume that there are no workers in alternative $p_3^{(1)}$, but there are in alternative $q_2^{(1)}$. Alternative $q_2^{(1)}$ is not marked as containing cuts, but the continuation of $q_2$ contains two pruning operations, $!(p_2)$ and $!(p_1)$. The worker $\mathcal{W}$ first executes $!(p_2)$ in order to prune $q_2^{(3)}$ and $p_2^{(2)}$. This is a safe pruning operation because any pruning from $q_2^{(1)}$ will also prune $q_2^{(3)}$ and $p_2^{(2)}$. At the same time, $\mathcal{W}$ stores a cut marker in $q_2$ to signal the pruning operation done.

Pursuing with the example, $\mathcal{W}$ executes $!(p_1)$ in order to prune $q_1^{(2)}$, $q_1^{(3)}$ and $p_1^{(2)}$. However, this is a dangerous operation. A worker in $q_2^{(1)}$ may execute the previous pruning operation, $!(p_2)$, pruning $\mathcal{W}$'s branch but not $q_1^{(2)}$, $q_1^{(3)}$ or $p_1^{(2)}$. Hence, there is no guarantee that the second pruning, $!(p_1)$, is safe. The cut marker stored in $q_2$ is a warning that this possibility exists. So, instead of doing pruning immediately, $\mathcal{W}$ updates the pending cut marker stored in $q_2$ to indicate the did not complete cut operation.

Figure 7.3 shows the effect of executing two pruning operations using our cut scheme. Initially, the pruning operations, $!(b)$ and $!(c)$, are respectively executed until nodes $f$ and $e$ as these are the closest nodes that contain unexploited alternatives in left branches. Therefore, cut markers are stored in nodes $f$ and $e$. A cut marker is a two field data structure consisting of the cut scope and the branch executing the cut.

However, we know that no branch to the left, except the ones marked with $!$, can invalidate further pruning for the current operations. Therefore $!(b)$ can execute up to node $d$ and $!(c)$ can fully execute till node $c$. The cut marker stored in $f$ indicates a pending cut operation, while the cut marker stored in $e$ prevents possible future pruning operations from the same branch.

### 7.2.2 Tree Representation

Supporting the cut predicate requires efficient mechanisms to represent the absolute and relative positions of each worker in the search tree. Checking whether the current
branch is leftmost or identifying workers working on branches to the left/right need to be very efficient operations.

The current YapOr implementation has the following representation of a Prolog search tree. We use a bi-dimensional matrix, \texttt{branch}[], to represent the current branch of each worker. Each entry \texttt{branch}[\mathcal{W}, \mathcal{D}] corresponds to the alternative taken by worker \( \mathcal{W} \) in the shared node with depth \( \mathcal{D} \) of its current branch. The advantage of this simple representation is that moving a worker in the search tree is a very efficient operation - neither locking nor extra overheads for maintaining the tree topology are needed.

Figure 7.4 presents a small example that clarifies the correspondence between a Prolog search tree and its matrix representation. Notice that we only represent the shared part of a search tree in the branch matrix. This is due to the fact that the position of each worker in the private part of the search tree is not helpful when computing relative positions.

To correctly consult or update the branch matrix, we need to know the depth of each shared node. To achieve this, we introduce a new data field in the or-frame data structure, the \texttt{OrFr\_depth} field, that holds the depth of the corresponding node. By using the \texttt{OrFr\_depth} field together with the \texttt{OrFr\_members} bitmap of each or-frame to consult the branch matrix, we can easily identify the workers in a node that are in branches at the left or at the right the current branch of a given worker.
7.2.3 Leftmostness

Let us suppose that a worker $W$ wants to check whether it is leftmost or at which node it ceases from being leftmost. $W$ should start from the bottom shared node $N$ on its branch, read the $\text{OrFr_members}$ bitmap from the or-frame associated with $N$ to determine the workers sharing the node, and investigate the branch matrix to determine the alternative number taken by each worker sharing $N$. If $W$ finds an alternative number less than its own, then $W$ is not leftmost. Otherwise, $W$ is leftmost in $N$ and will repeat the same procedure at the next upper node on branch and so on until reaching the root node or a node where it is not leftmost.

Two improvements were introduced [5] to obtain an efficient implementation. The first improvement reduces the number of workers to be consulted in each shared node, by avoiding consulting workers already known to be to the right. The second improvement reduces the number of nodes to be investigated in a branch, by associating with each shared node a new or-frame data field named $\text{OrFr_nearest_leftnode}$ pointing to the nearest upper node with branches to the left.

7.2.4 Pending Answers

With speculative work, a new answer for the query goal in a non-leftmost branch may not be valid since the branch where the answer was found may be pruned. To deal with this kind of situations, it is necessary to efficiently store the newly found answers in such a way that, by end of the computation, all valid answers are easily obtained.
YapOr stores a new answer in the first (youngest) shared node where the current branch is not leftmost. To accomplish this, a new data field was introduced in the or-frame data structure, the $\text{OrFr.qg.\_answers}$ field. This, to allow access to the set of pending answers stored in the corresponding node. New data structures were introduced to store the pending answers that are being found for the query goal in hand. Figure 7.5 details the data structures used to efficiently keep track of pending answers. Answers from the same branch are grouped into a common top data structure. The top data structures are organized by reverse branch order. This organization simplifies the pruning of answers that became invalid in consequence of a cut operation to the left.

![Figure 7.5: Dealing with pending answers.](image)

When a node $\mathcal{N}$ is fully exploited and its corresponding or-frame is being deallocated, the whole set of pending answers stored in $\mathcal{N}$ can be easily linked together and moved to the next node where the current branch is not leftmost. At the end, the set of answers stored in the root node are the set of valid answers for the given query goal.

### 7.2.5 Scheduling Speculative Work

We have seen that pruning speculative branches as soon as a cut to the left is executed is a key implementation issue in order to efficiently deal with speculative work in a parallel environment. Besides this important aspect, speculative work can be minimized if proper scheduling strategies are used. The Muse system implements a sophisticated strategy named *actively seeking the leftmost available work strategy* [8], that concentrates workers on the leftmost unexploited work of a search tree as long as
there is enough parallelism, in order to avoid workers entering into speculative work if less speculative work is available.

The set of unexploited alternatives in a search tree can be ordered according to their degree of speculativeness. Speculativeness decreases towards the bottom of the leftmost branch and increases towards the top of the rightmost one. Scheduling strategies that benefit the branches closer to the leftmost bottom corner of the execution tree should make useless work less probable.

The general idea of the actively seeking the leftmost available work strategy is to concentrate workers in the less speculative branches of the search tree in order to simulate the sequential Prolog execution as much as possible. The search tree is divided into two parts: the left part contains active work and the right part contains suspended work. Periodically, if there are no idle workers, all workers cooperate to compute their ordering and load information. Whenever there exists leftmost available work, the rightmost worker suspends all non-suspended alternatives to its right, including its current branch, and moves to the leftmost available alternative. When the amount of work to the left is not enough for the running workers, the leftmost suspended work to the right is taken and made active for exploitation.

Further work is still necessary to make YapOr's scheduler take full advantage of this kind of strategies.

### 7.3 Cut within the Or-Parallel Tabling Environment

The previous sections shown us that dealing with speculative work is not simple. Extending the or-parallel system to include tabling introduces complexity into cut’s semantics. During a tabled computation, not only the answers found for the query goal may not be valid, but also answers found for tabled predicates may be invalidated. The problem here is even more serious because tabled answers can be consumed elsewhere in the tree, which makes impracticable any late attempt to prune computations resulting from the consumption of invalid tabled answers. Indeed, consuming invalid tabled answers may result in finding more invalid answers for the same or other tabled predicates.
Notice that finding and consuming answers is the natural way to get a tabled computation going forward. Delaying the consumption of answers may compromise such flow. Therefore, tabled answers should be released as soon as it is found that they are safe from being pruned. Whereas for all-solution queries the requirement is that, at the end of the execution, we will have the set of valid answers, in tabling the requirement is to have the set of valid tabled answers released as soon as possible. Dealing with speculative tabled computations and guaranteeing the correctness of tabling semantics, without compromising the performance of the or-parallel tabling system, requires very efficient implementation mechanisms. Next, we discuss the OPTYap’s approach.

### 7.3.1 Inner and Outer Cut Operations

Allowing pruning operations in a tabling environment introduces a major design problem: how to deal with the operations that prune tabled nodes. We consider two types of cut operations in a tabling environment, cuts that do not prune tabled nodes — *inner cut* operations, and cuts that prune tabled nodes — *outer cut* operations. Figure 7.6 illustrates four different situations corresponding to inner and outer cut operations. Below each illustration we present a block of Prolog code that may lead to such situations. Predicates $t$ and $s$ correspond respectively to the tabled and scope nodes illustrated. Notice that the last situation only occurs if a parallel tabling environment is considered.

![Figure 7.6: The two types of cut operations in a tabling environment.](image-url)
Cut semantics for outer cut operations is still an open problem. The intricate dependencies in a tabled evaluation makes pruning a very complex problem. A major problem is that of pruning generator nodes. Pruning generator nodes cancels its further completion and puts the table space in an inconsistent state. This may lead dependent consumer nodes to incorrect computations as the set of answers found for the pruned generator node may be incomplete. A possible solution to this problem can lay on moving the generator's role to a not pruned dependent consumer node, if any, in order to allow further exploitation of the generator's unexploited branches. Such a solution will require that the other non-pruned consumer nodes recompute and update their dependencies relatively to the new generator node. Otherwise, if all dependent consumer nodes are also pruned, we can suspend the execution stacks and the table data structures of the pruned subgoal and try to resume them when the next variant call takes place. Scheduling also appears to be a problem. Applying different resolution strategies to return answers may lead to different pruning sequences that may influence the order that tabled nodes are pruned. Obviously, these are only simple preliminary ideas about the problems in discussion. Further research is still necessary in order to study the combination of pruning and tabling. Currently, OPTYap does not support outer cut operations. For such cases, execution is aborted.

7.3.2 Detecting Speculative Tabled Answers

As mentioned before, a main goal in the implementation of speculative tabling is to allow storing safe answers immediately. We would like to maintain the same performance as for the programs without cut operators. In this subsection, we introduce and describe the data structures and implementation extensions required to efficiently detect if a tabled answer is speculative or not.

We introduced a global bitmap register named \texttt{GLOBAL\_pruning\_workers} to keep track of the workers that are executing alternatives that contain cut operators and that, in consequence, may prune the current goal. Additionally, each worker maintains a local register, \texttt{LOCAL\_safe\_scope}, that references the bottommost (youngest) node that cannot be pruned by any pruning operation executed by itself.

The correct manipulation of these new registers is achieved by introducing the new WAM instruction \texttt{clause\_with\_cuts}. This new instruction marks the blocks of code...
that include cut instructions. During compilation, the WAM code generated for
the clauses containing cut operators was extended to include the `clause_with_cuts`
instruction so that it is the first instruction to be executed for such clauses. When
a worker loads a `clause_with_cuts` instruction, it executes the `clause_with_cuts()`
procedure.

Figure 7.7 details the pseudo-code that implements the `clause_with_cuts()` proce-
dure. It sets the worker’s bit of the global register `GLOBAL_pruning_workers`, and
updates the worker’s local register `LOCAL_safe_scope` to the oldest reference between
its current value and the current node. The current node is the resulting top node if
a pruning operation takes place from the clause being executed.

```c
clause_with_cuts() {
    if (LOCAL_safe_scope == NULL) {
        // first execution of clause_with_cuts
        insert_into_bitmap(GLOBAL_pruning_workers, WORKER_ID)
        LOCAL_safe_scope = B
    } else if (LOCAL_safe_scope is younger than B)
        // B is the local stack register
        LOCAL_safe_scope = B
    load_next_instruction
    proceed
}
```

Figure 7.7: Pseudo-code for `clause_with_cuts()`.

When a worker finds a new answer for a tabled subgoal call, it inserts the answer’s trie
representation into the table space and then it checks if the answer is safe from being
pruned. When this is the case, the answer is included in the chain of available answers
for the tabled subgoal, as usually. Otherwise, if it is found that the answer can be
pruned by another worker, its availability is delayed. Figure 7.8 presents the pseudo-
code that implements the checking procedure. When it is found that the answer being
checked can be speculative, the procedure returns the or-frame that corresponds to
the youngest node where the answer can be pruned by a worker in a left branch. That
or-frame is where the answer should be left pending. Otherwise, if is found the answer
is safe, the procedure returns NULL.

Note that the `speculative_tabled_answer()` procedure is only called when the gen-
erator node for the answer being checked in is public, as otherwise any pruning corre-
sponds to an outer cut operation. The procedure’s pseudo-code starts by determining
if there are workers that may execute pruning operations. If so, it checks the safeness
speculative_tabled_answer(generator node G) {
    // G is the generator node for the answer being checked
    prune_wks = GLOBAL_pruning_workers
    delete_from_bitmap(prune_wks, WORKER_ID)
    if (prune_wks is not empty) {
        // there are workers that may execute pruning operations
        or_fr = TOP_OR_FR
        depth = OrFr_depth(or_fr)
        scope_depth = OrFr_depth(CP_OR_FR(G))
        while (depth > scope_depth) {
            // checking the public branch till the generator node
            alt_number = branch(WORKER_ID, depth)
            for (w = 0; w < number_workers; w++) {
                if (w is in OrFr_members(or_fr) &&
                    branch(w, depth) < alt_number &&
                    w is in prune_wks &&
                    OrFr_node(or_fr) is younger than LOCAL_safe_scope(w))
                    // the answer can be pruned by worker w
                    return or_fr
            }
            or_fr = OrFr_next(or_fr)
            depth = OrFr_depth(or_fr)
        }
        // the answer is safe from being pruned
        return NULL
    }
}

Figure 7.8: Pseudo-code for speculative_tabled_answer().

of the branch where the tabled answer was found. The branch only needs to be
checked until the corresponding generator node, as otherwise it would be an outer
cut operation. A branch is found to be safe if it is leftmost, or if the workers in the
branches to the left cannot prune it.

The speculative_tabled_answer() procedure is similar to the leftmost check pro-
cedure described before. Hence, the implementation improvements mentioned for
the leftmost check procedure can also be used here to improve the efficiency of the
speculative_tabled_answer() procedure. However, for simplicity of presentation,
none of those improvements were included in the pseudo-code.

### 7.3.3 Pending Tabled Answers

If a tabled answer is speculative, its availability is delayed. A speculative answer
should remain in a pending state until it is pruned by a left branch or until it is found
that it is safe from being pruned. In the latter case it should be released as a valid answer. Dealing with pending tabled answers requires efficient support to allow that the operations of pruning or releasing pending answers are efficiently performed.

Remember that pending answers are stored in a node. To allow access to the set of pending answers for a node, a new data field was introduced in the or-frame data structure, the OrFr_tg_answers field. New data structures were also introduced to efficiently keep track of the pending answers being found for the several tabled subgoal calls. Figure 7.9 details that data structure organization.

![Diagram of OrFr_tg_answers](image)

**Figure 7.9:** Dealing with pending tabled answers.

The figure shows a situation where three tabled answers, answer-x, answer-y and answer-z, were found to be speculative and in consequence have all been left pending in a common node \( N \). \( N \) is the bottommost node where a worker in a left branch, \( W \) in the figure, holds a LOCAL_safe_scope register pointing to a node older than \( N \).

Pending answers found for the same subgoal and from the same branch are addressed by a common top frame data structure. As the answers in the figure were found in different subgoal/branch pairs, three top frames were required. answer-x, answer-y and answer-z were found respectively in branches 2, 3 and 3 for the subgoals corresponding to generator nodes \( G_1, G_1 \) and \( G_2 \). The top frames are organized in order to younger generator order and by reverse branch order when there are several frames for the same generator. Hence, each frame contains two types of pointers to follow the chain of frames, one points to the frame that corresponds to the next younger generator
node, while the other points to the frame that corresponds to the next branch within
the same generator.

Blocks of answers address the set of pending answers for a subgoal/branch pair. Each
block points to a fixed number of answers. By linking the blocks we can have a
large number of answers for the same subgoal/branch pair. Note that the block data
structure does not hold the representation of a pending answer, only a pointer to the
leaf answer trie node of the answer trie structure representing the pending answer.
This happens because tabled answers are inserted in advance into the table space even
if they are to be pruned later.

As already mentioned, a key point in the implementation support for pending answers
is the efficiency of the procedure to release answers. OPTYap implements the fol­
lowing algorithm: the last worker \( W \) leaving a node \( N \) with pending tabled answers,
determines the next node \( M \) on its branch that can be pruned by a worker to the left.
The pending answers from \( N \) that correspond to generator nodes equal or younger
than \( M \) are made available, while the remaining are moved from \( N \) to \( M \). Notice
that \( W \) only needs to check for the existence of a node \( M \) up to the oldest generator
node in the pending answers stored in \( N \). To simplify finding the oldest generator
node we organized top frames in older to younger generator order.

Last, in order to correctly implement direct compiled code execution in OPTYap,
it is required that the answer trie nodes representing pruned answers are removed
from the trie structure. For simplicity and efficiency, this is performed by the tabled
subgoal call that first calls the tabled subgoal after it has been completed because it
requires traversing the whole answer trie structure. The code for direct compiled code
execution is therefore computed while traversing the answer trie structure.

7.4 Chapter Summary

This chapter discussed the problems behind the management of speculative computa-
tions. A computation is named speculative if it can potentially be pruned during
parallel evaluation, therefore resulting in wasted effort when compared to sequential
execution.

We started by introducing the semantics for the standard pruning operator – cut,
and then we discussed its behavior for parallel execution. Next we presented YapOr’s approach to efficiently deal with speculative work and described the supporting data structures and algorithms for its implementation.

Lastly, we motivated the problems of combining pruning with tabling and distinguished two different types of cut operations in a tabling environment, cuts that do not prune tabled nodes—inner cuts, and cuts that prune tabled nodes—outer cuts. Cut semantics for outer cuts is still an open problem. We thus focused on the support for inner cuts and described OPTYap’s approach to efficiently deal with speculative tabled answers.
Chapter 8

Performance Analysis

The overall goal of research in parallel logic programming is to achieve of higher performance through parallelism. The initial implementations of successful or-parallel Prolog systems, such as Aurora and Muse, relied on a detailed knowledge of a specific Prolog system, SICStus Prolog [18], and on the evaluation attained from original shared memory machines, such as the Sequent Symmetry. Modern Prolog systems, although WAM based, have made substantial improvements in sequential execution. These improvements largely result from the development of new and refined optimizations not found in the original SICStus Prolog. Besides, the impressive improvements on CPU performance over the last years have not been followed by similar gains in bus and memory performance. As a result, modern parallel machines show a much higher memory latency, as measured by the number of CPU clock cycles, than original Sequent style machines.

The question therefore arises of whether the good results previously obtained with Aurora and Muse in Sequent style machines are still reachable with current Prolog systems in modern parallel architectures. In particular, we can question whether such results extend to parallel tabling implementations as tabling, by nature, reduces the potential non-determinism available in logic programs. Also notice that accomplishing good speedups may not necessarily translate to a corresponding improvement in performance with respect to state of the art sequential implementations. The cost of managing parallelism can make the performance of the parallel implementation with a single worker considerably worst than the base sequential implementation.
To assess the efficiency of our parallel tabling implementation and thus respond to the questions just raised, we present next a detailed analysis of OPTYap’s performance. We start by presenting an overall view of the overheads of supporting several Yap extensions: YapOr, YapTab and OPTYap. Then, we compare YapOr’s parallel performance with that of OPTYap for a set of non-tabled programs. Next, we use a set of tabled programs to measure the sequential behavior of YapTab, OPTYap and XSB, and to assess OPTYap’s performance when running the tabled programs in parallel. At last, we study the impact of using the alternative locking schemes from subsection 6.3.2 to deal with concurrent accesses to the table space data structures.

YapOr, YapTab and OPTYap are based on Yap’s 4.2.1 engine\(^1\). We used the same compilation flags for Yap, YapOr, YapTab and OPTYap. Concerning YapTab and OPTYap, we studied performance under both batched and local scheduling strategies. Regarding XSB Prolog, we used version 2.3 with the default configuration and the default execution parameters (chat engine and batched scheduling) for batched scheduling, and version 2.4 with the default configuration and the default execution parameters (chat engine and local scheduling) for local scheduling.

The environment for our experiments was oscar, a Silicon Graphics Cray Origin2000 parallel computer from the Oxford Supercomputing Centre. Oscar consists of 96 MIPS 195 MHz R10000 processors each with 256 Mbytes of main memory (for a total shared memory of 24 Gbytes) and running the IRIX 6.5.12 kernel. While benchmarking, the jobs were submitted to an execution queue responsible for scheduling the pending jobs through the available processors in such a way that, when a job is scheduled for execution, the processors attached to the job are fully available during the period of time requested for execution. We have limited our experiments to 32 processors because the machine was always with a very high load and we were limited to a guest-account.

### 8.1 Performance on Non-Tabled Programs

To place our performance results in perspective we first evaluate how the original Yap Prolog engine compares against the several Yap extensions we implemented and

\(^1\)Note that sequential execution would be somewhat better with more recent Yap engines.
against the most well-known tabling engine, XSB Prolog. Since OPTYap is based on the same environment model as the one used by YapOr, we then compare OPTYap’s performance with that of YapOr on a similar set of non-tabled programs.

8.1.1 Non-Tabled Benchmark Programs

We use a set of standard non-tabled logic programming benchmarks [100, 56, 92, 32]. The set includes the following benchmark programs:

- cubes: solves the N-cubes or instant insanity problem from Tick’s book [104]. It consists of stacking 7 colored cubes in a column so that no color appears twice within any given side of the column.
- ham: finds all hamiltonian cycles for a graph consisting of 26 nodes with each node connected to other 3 nodes.
- map: solves the problem of coloring a map of 10 countries with five colors such that no two adjacent countries have the same color.
- nsort: naive sort algorithm. It sorts a list of 10 elements by brute force starting from the reverse order (and worst) case.
- puzzle: places numbers 1 to 19 in an hexagon pattern such that the sums in all 15 diagonals add to the same value (also taken from Tick’s book [104]).
- queens: a non-naive algorithm to solve the problem of placing 11 queens on a 11x11 chess board such that no two queens attack each other.

All benchmarks find all the answers for the problem. Multiple answers are computed through automatic failure after a valid answer has been found. To measure total execution time we used the Prolog code that follows.

```prolog
:- sequential run/0.

go :- statistics(walltime, [Start,_]),
    run,  
    statistics(walltime, [End,_]),
```

2 The Prolog code for these benchmark programs is included as Appendix A.1.
Time is End-Start,
write('WallTime is '), write(Time), nl.
run :- benchmark, fail.
run.
benchmark :- ...

The \texttt{go/0} predicate is the top query goal. For each particular benchmark, \texttt{benchmark/0} is the predicate that triggers the benchmark’s execution. The \texttt{run/0} predicate is defined by two clauses. The first clause implements the automatic failure mechanism, while the second accomplishes successfully completes execution of the top query goal. Note that for parallel execution one needs to declare the \texttt{run/0} predicate as sequential in order to ensure that the second clause only gets executed after the whole search space for the benchmark in hand has been exploited.

### 8.1.2 Overheads over Standard Yap

Fundamental criteria to judge the success of an or-parallel, tabling, or of a combined or-parallel tabling model includes measuring the overhead introduced by these models when running programs that do not take advantage of the particular extension. Ideally, a program should not pay a penalty for or-parallel or tabling mechanisms that it does not require. Therefore, in order to develop a successful or-parallel, tabling, or or-parallel tabling engine such overheads should be minimal.

Table 8.1 shows the base execution time, in seconds, for Yap, YapOr, YapTab, OPTYap and XSB for our set of non-tabled benchmark programs. In parentheses, it shows the overhead over the Yap execution time. Obviously, the timings reported for YapOr and OPTYap correspond to the execution with a single worker. For simplicity, in this section, we will not distinguish between batched and local scheduling when reporting to YapTab, OPTYap and XSB, as for non-tabled programs there are no execution differences between both strategies.

The results indicate that YapOr, YapTab and OPTYap introduce, on average, an overhead of about 10%, 5% and 17% respectively over standard Yap. YapOr overheads result from handling the work load register and from testing operations that (i) verify whether the bottommost node is shared or private, (ii) check for sharing requests, and (iii) check for backtracking messages due to cut operations. On the other hand,
8.1. PERFORMANCE ON NON-TABLED PROGRAMS

<table>
<thead>
<tr>
<th>Program</th>
<th>Yap</th>
<th>YapOr</th>
<th>YapTab</th>
<th>OPTYap</th>
<th>XSB 2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cubes</td>
<td>1.97</td>
<td>2.06(1.05)</td>
<td>2.05(1.04)</td>
<td>2.16(1.10)</td>
<td>4.81(2.44)</td>
</tr>
<tr>
<td>ham</td>
<td>4.04</td>
<td>4.61(1.14)</td>
<td>4.28(1.06)</td>
<td>4.95(1.23)</td>
<td>10.36(2.56)</td>
</tr>
<tr>
<td>map</td>
<td>9.01</td>
<td>10.25(1.14)</td>
<td>9.19(1.02)</td>
<td>11.08(1.23)</td>
<td>24.11(2.68)</td>
</tr>
<tr>
<td>nsort</td>
<td>33.05</td>
<td>37.52(1.14)</td>
<td>35.85(1.08)</td>
<td>39.95(1.21)</td>
<td>83.72(2.53)</td>
</tr>
<tr>
<td>puzzle</td>
<td>2.04</td>
<td>2.22(1.09)</td>
<td>2.19(1.07)</td>
<td>2.36(1.16)</td>
<td>4.97(2.44)</td>
</tr>
<tr>
<td>queens</td>
<td>16.77</td>
<td>17.68(1.05)</td>
<td>17.58(1.05)</td>
<td>18.57(1.11)</td>
<td>36.40(2.17)</td>
</tr>
</tbody>
</table>

Table 8.1: Yap, YapOr, YapTab, OPTYap and XSB execution time on non-tabled programs.

YapTab overheads are due to the handling of the freeze registers and support of the forward trail. OPTYap overheads inherits both sources of overheads. Considering that Yap Prolog is one of the fastest Prolog engines currently available, the low overheads achieved by YapOr, YapTab and OPTYap are very good results.

Regarding XSB, the results from Table 8.1 show that, on average, XSB is 2.47 times slower than Yap. This is a result mainly due to the faster Yap engine.

8.1.3 Speedups for Parallel Execution

To assess the performance of OPTYap’s or-parallel engine when executing non-tabled programs in parallel, we ran OPTYap with a varying number of workers for the set of non-tabled benchmark programs.

The results reported in previous work [78, 81], for parallel execution of non-tabled programs, showed that YapOr is very efficient in exploiting or-parallelism and that it obtains better speedup ratios than Muse with the increase in the number of workers. This was a surprising result given that YapOr has better base performance. Note, however, that Muse under SICStus is a more mature system that implements some functionalities that are still lacking in YapOr. Since OPTYap is based on YapOr’s engine, we also tested YapOr against the same set of benchmark programs to get a better perspective of OPTYap’s results.

Table 8.2 shows the speedups relative to the single worker case for YapOr and OPTYap with 4, 8, 16, 24 and 32 workers. Each speedup corresponds to the best execution time
obtained in a set of 3 runs.

<table>
<thead>
<tr>
<th>Program</th>
<th>YapOr</th>
<th></th>
<th></th>
<th></th>
<th>OPTYap</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>ham</td>
<td>3.93</td>
<td>7.61</td>
<td>13.71</td>
<td>15.62</td>
<td>15.75</td>
<td>3.92</td>
<td>7.64</td>
<td>13.54</td>
</tr>
<tr>
<td>map</td>
<td>3.98</td>
<td>7.73</td>
<td>14.03</td>
<td>17.11</td>
<td>18.28</td>
<td>3.98</td>
<td>7.88</td>
<td>13.74</td>
</tr>
<tr>
<td>nsort</td>
<td>3.98</td>
<td>7.92</td>
<td>15.62</td>
<td>22.90</td>
<td>29.73</td>
<td>3.96</td>
<td>7.84</td>
<td>15.50</td>
</tr>
<tr>
<td>queens</td>
<td>4.00</td>
<td>7.95</td>
<td>15.39</td>
<td>21.69</td>
<td>25.69</td>
<td>3.99</td>
<td>7.93</td>
<td>15.41</td>
</tr>
<tr>
<td>Average</td>
<td>3.97</td>
<td>7.76</td>
<td>14.52</td>
<td>19.13</td>
<td>21.00</td>
<td>3.96</td>
<td>7.76</td>
<td>14.34</td>
</tr>
</tbody>
</table>

Table 8.2: Speedups for YapOr and OPTYap on non-tabled programs.

The results show that YapOr and OPTYap achieve identical effective speedups in all benchmark programs. Despite that OPTYap includes all the machinery required to support tabled programs, these results allow us to conclude that OPTYap maintains YapOr’s behavior in exploiting or-parallelism in non-tabled programs.

### 8.2 Performance on Tabled Programs

In this section we start by describing the set of tabled benchmark programs that we used to assess performance for tabling execution. We then measure the performance of YapTab and OPTYap for sequential execution and compare the results with those of XSB. Next, we assess OPTYap’s performance for parallel execution on these tabled programs and discuss various statistics gathered during execution so that the results obtained can be better understood. At last, we study the impact of using alternative locking schemes to access the table space during parallel execution.
8.2. PERFORMANCE ON TABLED PROGRAMS

8.2.1 Tabled Benchmark Programs

The tabled benchmark programs were obtained from the XMC\textsuperscript{3} [45] and XSB [44] world wide web sites and are frequently used in the literature to evaluate such systems. The benchmark programs are\textsuperscript{4}:

sieve: the transition relation graph for the sieve specification\textsuperscript{5} defined for 5 processes and 4 overflow prime numbers.

leader: the transition relation graph for the leader election specification defined for 5 processes.

iproto: the transition relation graph for the i-protocol specification defined for a correct version (fix) with a huge window size ($w = 2$).

samegen: solves the same generation problem for a randomly generated $24 \times 24 \times 2$ cylinder. The cylinder data can be thought of as a rectangular matrix of $24 \times 24$ elements where each element in row $n$ (except the last) is connected to two elements in row $n + 1$. A pair of nodes is said to belong to the same generation when they are the same or when each one holds a connection to nodes that are in the same generation. This benchmark is very interesting because for sequential execution it does not allocate any consumer choice point. Variant calls to tabled subgoals only occur when the subgoals are already completed.

lgrid: computes the transitive closure of a $25 \times 25$ grid using a left recursion algorithm. A link between two nodes, $n$ and $m$, is defined by two different relations; one indicates that we can reach $m$ from $n$ and the other indicates that we can reach $n$ from $m$.

lgrid/2: the same as lgrid but it only requires half the relations to indicate that two nodes are connected. It defines links between two nodes by a single relation, and

\textsuperscript{3}The XMC system [70] is a model checker implemented atop the XSB system which verifies properties written in the alternation-free fragment of the modal $\mu$-calculus [60] for systems specified in XL, an extension of value-passing CCS [66].

\textsuperscript{4}The Prolog code for these benchmark programs is included as Appendix A.2.

\textsuperscript{5}We are thankful to C. R. Ramakrishnan for helping us in dumping the transition relation graph of the automata corresponding to each given XL specification, and in building runnable versions out of the XMC environment.
it uses a predicate to achieve symmetric reachability. This modification alters the order by which answers are found, therefore leading to a more randomly distributed answers. This modification also lead the benchmark to take more time to execute, and in consequence we reduced the search space to a 20x20 grid.

**rgrid/2:** the same as lgrid/2 but it computes the transitive closure using a right recursion algorithm.

Similarly to what was done for non-tabled benchmark programs, here we use the same mechanisms to search for all answers for each problem and to measure the execution time necessary to fully search the execution tree of a particular benchmark.

### 8.2.2 Timings for Sequential Execution

In order to place OPTYap’s results in perspective we start by analyzing the overheads introduced to extend YapTab to parallel execution and by measuring YapTab and OPTYap behavior when compared with the latest versions of the XSB system.

Table 8.3 shows the execution time, in seconds, for YapTab, OPTYap and XSB using batched and local scheduling strategies for the tabled benchmark programs. In parentheses it shows the overheads, respectively, over the YapTab Batched and YapTab Local execution time. The execution time reported for OPTYap correspond to the execution with a single worker. We used the TLWL locking scheme for OPTYap Batched and the TLWL-ABC locking scheme for OPTYap Local. We choose these schemes as a result of the performance study that we present in subsection 8.2.5. In what follows, if nothing is said, when reporting OPTYap Batched or OPTYap Local we assume the locking schemes mentioned above. Regarding XSB, we used version 2.3 for batched scheduling and version 2.4 for local scheduling, as referred in the beginning of this chapter. Notice that the average result obtained for XSB using local scheduling is clearly influenced by the strange behavior showed for the *rgrid/2* benchmark. If we do not consider such benchmark then the average result is 1.97.

The results indicate that, for these set of tabled benchmark programs, OPTYap introduces, on average, an overhead of about 15% over YapTab for both batched and local scheduling strategies. This overhead is very close to that observed for non-tabled programs (11%). The small difference results from locking requests to handle
8.2. PERFORMANCE ON TABLED PROGRAMS

<table>
<thead>
<tr>
<th>Program</th>
<th>Batched Scheduling</th>
<th>Local Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YapTab</td>
<td>OPTYap</td>
</tr>
<tr>
<td>sieve</td>
<td>235.31</td>
<td>268.13(1.14)</td>
</tr>
<tr>
<td>leader</td>
<td>76.60</td>
<td>85.56(1.12)</td>
</tr>
<tr>
<td>ipproto</td>
<td>20.73</td>
<td>23.68(1.14)</td>
</tr>
<tr>
<td>samangen</td>
<td>23.56</td>
<td>26.00(1.11)</td>
</tr>
<tr>
<td>lgrid</td>
<td>3.55</td>
<td>4.28(1.21)</td>
</tr>
<tr>
<td>lgrid/2</td>
<td>59.53</td>
<td>69.02(1.16)</td>
</tr>
<tr>
<td>rgrid/2</td>
<td>6.24</td>
<td>7.51(1.20)</td>
</tr>
</tbody>
</table>

Table 8.3: YapTab, OPTYap and XSB execution time on tabled programs.

The data structures introduced by tabling. Locks are required to insert new trie nodes into the table space, and to update subgoal and dependency frame pointers to tabled answers. These locking operations are all related with the management of tabled answers. Therefore, the benchmarks that deal with more tabled answers are the ones that potentially can perform more locking operations. This causal relation seems to be reflected in the execution times showed in Table 8.3, because the benchmarks that show higher overheads are also the ones that find more answers. The answers found by each benchmark are presented in Table 8.4. In this table we can observe that lgrid and rgrid/2 are the benchmarks that find more answers, followed by the ipproto and lgrid/2 benchmarks.

The results also confirm previous results from Freire et al. [40] where local scheduling performs worst than batched scheduling. Regardless, our results did show a smaller slowdown. YapTab Local is only about 3% slower than YapTab Batched (this overhead is not included in the table). Moreover, there is one benchmark, rgrid/2, where local scheduling performs slightly better than batched.

Table 8.3 shows that YapTab is on average about twice as fast as XSB for these set of benchmarks. This may be partly due to the faster Yap engine, as seen in Table 8.1, and also to the fact that XSB implements functionalities that are still lacking in YapTab and that XSB may incur through overheads in supporting those functionalities. Independently of the scheduling strategy, the average execution time for the single worker case proved that OPTYap runs as fast or faster than current XSB.
We believe that these results clearly show that we have accomplished our initial aim of implementing an or-parallel tabling system that compares favorably with current state of the art technology. Hence, we believe the following evaluation of the parallel engine is significant and fair.

8.2.3 Characteristics of the Benchmark Programs

In order to achieve a deeper insight on the behavior of each benchmark, and therefore clarify some of the results that are presented next, we first present in Table 8.4 data on the benchmark programs. The columns in Table 8.4 have the following meaning:

**first:** is the number of first calls to subgoals corresponding to tabled predicates. It corresponds to the number of generator choice points allocated.

**nodes:** is the number of subgoal/answer trie nodes used to represent the complete subgoal/answer trie structures of the tabled predicates in the given benchmark. For the answer tries, in parentheses, it shows the percentage of saving that the trie’s design achieves on these data structures. Given the total number of nodes required to represent individually each answer and the number of nodes used by the trie structure, the saving can be obtained by the following expression:

$$saving = \frac{total - used}{total}$$

As an example, consider two answers whose single representation requires respectively 12 and 8 answer trie nodes for each. Assuming that the answer trie representation of both answers only requires 15 answer trie nodes, thus 5 of those being common to both paths, it achieves a saving of 25%. Higher percentages of saving reflect higher probabilities of lock contention when concurrently accessing the table space.

**depth:** is the number of nodes required to represent a path through a subgoal/answer trie structure. In other words, it is the number of nodes required to represent a subgoal call or to represent an answer. It is a three value column. The first and third values correspond, respectively, to the minimum and maximum depth of a path in the whole subgoal/answer tries. The second value is the average depth of the whole set of paths in the corresponding subgoal/answer trie structures.
Trie structures with smaller average depth values are more amenable to higher lock contention.

**unique:** is the number of non-redundant answers found for tabled subgoals. It corresponds to the number of answers stored in the table space.

**repeated:** is the number of redundant answers found for tabled subgoals. A high number of redundant answers can degrade the performance of the parallel system when using table locking schemes that lock the table space without taking into account whether writing to the table is, or is not, likely.

<table>
<thead>
<tr>
<th>Program</th>
<th>Subgoal Tries</th>
<th>New Answers</th>
<th>Answer Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>first nodes</td>
<td>depth</td>
<td>unique</td>
</tr>
<tr>
<td>sieve</td>
<td>1</td>
<td>7</td>
<td>6/6/6</td>
</tr>
<tr>
<td>leader</td>
<td>1</td>
<td>5</td>
<td>4/4/4</td>
</tr>
<tr>
<td>iprot0</td>
<td>1</td>
<td>6</td>
<td>5/5/5</td>
</tr>
<tr>
<td>samegen</td>
<td>485</td>
<td>971</td>
<td>2/2/2</td>
</tr>
<tr>
<td>lgrid</td>
<td>1</td>
<td>3</td>
<td>2/2/2</td>
</tr>
<tr>
<td>lgrid/2</td>
<td>1</td>
<td>3</td>
<td>2/2/2</td>
</tr>
<tr>
<td>rgrid/2</td>
<td>626</td>
<td>1253</td>
<td>2/2/2</td>
</tr>
</tbody>
</table>

Table 8.4: Characteristics of the tabled programs.

By observing Table 8.4 it seems that *sieve* and *leader* are the benchmarks least amenable to table lock contention because they are the ones that find the least number of answers and also the ones that have the deepest trie structures. In this regard, *lgrid*, *lgrid/2* and *rgrid/2* correspond to the opposite case. They find the largest number of answers and they have very shallow trie structures. However, *rgrid/2* is a benchmark with a large number of first subgoals calls which can reduce the probability of lock contention because answers can be found for different subgoal calls and therefore be inserted with minimum overlap. Likewise, *samegen* is a benchmark that can also benefit from its large number of first subgoal calls, despite also presenting a very shallow trie structure. Finally, *iproto* is a benchmark that can also lead to higher ratios of lock contention. It presents a deep trie structure, but it inserts a huge number of trie nodes in the table space. Moreover, it is the benchmark showing the highest percentage of saving.
8.2.4 Parallel Execution Study

To assess OPTYap’s performance when running tabled programs in parallel, we ran OPTYap with varying number of workers for the set of tabled benchmark programs. We start by studying parallel execution with batched scheduling.

Parallel Execution with Batched Scheduling

Table 8.5 presents the speedups for OPTYap with 2, 4, 6, 8, 12, 16, 24 and 32 workers using batched scheduling. The speedups are relative to the single worker case of Table 8.3. They correspond to the best speedup obtained in a set of 3 runs. The table is divided in two main blocks: the upper block groups the benchmarks that showed potential for parallel execution, whilst the bottom block groups the benchmarks that do not show any gains when run in parallel.

![Table 8.5: Speedups for OPTYap using batched scheduling on tabled programs.](table)

The results show superb speedups for the XMC sieve and the leader benchmarks up to 32 workers. These benchmarks reach speedups of 31.5 and 31.18 with 32 workers! Two other benchmarks in the upper block, samegen and lgrid/2, also show excellent speedups up to 32 workers. Both reach a speedup of 24 with 32 workers. The remaining benchmark, iproto, shows a good result up to 16 workers and then it slows down with 24 and 32 workers. Globally, the results for the upper block are quite good, especially considering that they include the three XMC benchmarks that are more representative.
of real-world applications.

On the other hand, the bottom block shows almost no speedups at all. Only for rgrid/2 with 6 and 8 workers we obtain a slight positive speedup of 1.01 and 1.15. The worst case is for lgrid with 32 workers, where we are about 2.5 times slower than execution with a single worker. In this case, surprisingly, we observed that for the whole set of benchmarks the workers are busy for more than 95% of the execution time, even for 32 workers. The actual slowdown is therefore not caused because workers became idle and start searching for work, as usually happens with parallel execution of non-tabled programs. Here the problem seems more complex: workers do have available work, but there is a lot of contention to access that work.

The parallel execution behavior of each benchmark program can be better understood through the statistics described in the tables that follows. The columns in these tables have the following meaning:

**variant**: is the number of variant calls to subgoals corresponding to tabled predicates. It matches the number of consumer choice points allocated.

**complete**: is the number of variant calls to completed tabled subgoals. It is when the *completed table optimization* takes places, that is, when the set of found answers is consumed by executing compiled code directly from the trie structure associated with the completed subgoal.

**SCC suspend**: is the number of SCCs suspended.

**SCC resume**: is the number of suspended SCCs that were resumed.

**contention points**: is the total number of unsuccessful first attempts to lock data structures of all types. Note that when a first attempt fails, the requesting worker performs arbitrarily locking requests until it succeeds. Here, we only consider the first attempts.

**subgoal frame**: is the number of unsuccessful first attempts to lock subgoal frames. A subgoal frame is locked in three main different situations: (i) when a new answer is found which requires updating the subgoal frame pointer to the last found answer; (ii) when marking a subgoal as completed;
(iii) when traversing the whole answer trie structure to remove pruned answers and compute the code for direct compiled code execution.

**dependency frame:** is the number of unsuccessful first attempts to lock dependency frames. A dependency frame has to be locked when it is checked for unconsumed answers.

**trie node:** is the number of unsuccessful first attempts to lock trie nodes. Trie nodes must be locked when a worker has to traverse a trie structure to check/insert for new subgoal calls or answers.

To accomplish these statistics it was necessary to introduce in the system a set of counters to measure the several parameters. Although, the counting mechanism introduces an additional overhead in the execution time, we assume that it does not significantly influence the parallel execution pattern of each benchmark program.

Tables 8.6 and 8.7 show respectively the statistics gathered for the group of programs with and without parallelism. We do not include the statistics for the leader benchmark because its execution behavior showed to be identical to the observed for the sieve benchmark.

The statistics obtained for the sieve benchmark support the excellent performance speedups showed for parallel execution. It shows insignificant amounts of contention points, it only calls a variant subgoal, and despite the fact that it suspends some SCCs it successfully avoids resuming them. In this regard, the samegen benchmark also shows insignificant amounts of contention points. However the number of variant subgoals calls and the number of suspended/resumed SCCs indicate that it introduces more dependencies between workers. Curiously, for more than 4 workers, the amount of variant calls and the amount of suspended SCCs seems to be stable. The only parameter that slightly increases is the number of resumed SCCs. Regarding iproto and lgrid/2, lock contention seems to be the major problem. Trie nodes show identical lock contention, however iproto inserts about 10 times more answer trie nodes than lgrid/2. Subgoal and dependency frames show an identical pattern of contention, but iproto presents higher contention ratios. Moreover, if we remember from Table 8.3 that iproto is about 3 times faster than lgrid/2 to execute, we can conclude that the contention ratio for iproto is obviously much higher per time unit, which justifies its worst behavior.
8.2. PERFORMANCE ON TABLED PROGRAMS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number of Workers</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>sieve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>20/0</td>
<td>70/0</td>
<td>136/0</td>
<td>214/0</td>
<td>261/0</td>
<td></td>
</tr>
<tr>
<td>contention points</td>
<td>108</td>
<td>329</td>
<td>852</td>
<td>1616</td>
<td>3040</td>
<td></td>
</tr>
<tr>
<td>subgoal frame</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>dependency frame</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>trie node</td>
<td>96</td>
<td>188</td>
<td>415</td>
<td>677</td>
<td>1979</td>
<td></td>
</tr>
<tr>
<td>iproto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td></td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>5/0</td>
<td>9/0</td>
<td>17/0</td>
<td>26/0</td>
<td>32/0</td>
<td></td>
</tr>
<tr>
<td>contention points</td>
<td>7712</td>
<td>22473</td>
<td>60703</td>
<td>120162</td>
<td>136734</td>
<td></td>
</tr>
<tr>
<td>subgoal frame</td>
<td>3832</td>
<td>9894</td>
<td>21271</td>
<td>33162</td>
<td>33307</td>
<td></td>
</tr>
<tr>
<td>dependency frame</td>
<td>678</td>
<td>4685</td>
<td>25006</td>
<td>66334</td>
<td>81515</td>
<td></td>
</tr>
<tr>
<td>trie node</td>
<td>3045</td>
<td>6579</td>
<td>10537</td>
<td>11816</td>
<td>11736</td>
<td></td>
</tr>
<tr>
<td>samegen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>485/1067</td>
<td>1359/193</td>
<td>1355/197</td>
<td>1384/168</td>
<td>1363/189</td>
<td></td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>187/2</td>
<td>991/11</td>
<td>1002/20</td>
<td>1024/25</td>
<td>1020/34</td>
<td></td>
</tr>
<tr>
<td>contention points</td>
<td>255</td>
<td>314</td>
<td>743</td>
<td>1160</td>
<td>1607</td>
<td></td>
</tr>
<tr>
<td>subgoal frame</td>
<td>8</td>
<td>52</td>
<td>112</td>
<td>283</td>
<td>493</td>
<td></td>
</tr>
<tr>
<td>dependency frame</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>trie node</td>
<td>154</td>
<td>119</td>
<td>201</td>
<td>364</td>
<td>417</td>
<td></td>
</tr>
<tr>
<td>lgrid/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
<td></td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>4/0</td>
<td>8/0</td>
<td>16/0</td>
<td>24/0</td>
<td>32/0</td>
<td></td>
</tr>
<tr>
<td>contention points</td>
<td>4004</td>
<td>10072</td>
<td>28669</td>
<td>59283</td>
<td>88541</td>
<td></td>
</tr>
<tr>
<td>subgoal frame</td>
<td>167</td>
<td>1124</td>
<td>7319</td>
<td>17440</td>
<td>27834</td>
<td></td>
</tr>
<tr>
<td>dependency frame</td>
<td>98</td>
<td>1209</td>
<td>5987</td>
<td>23357</td>
<td>35991</td>
<td></td>
</tr>
<tr>
<td>trie node</td>
<td>2958</td>
<td>5292</td>
<td>10341</td>
<td>12870</td>
<td>12925</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.6: Statistics of OPTYap using batched scheduling for the group of programs with parallelism.

The statistics gathered for the second group of programs present very interesting results. Remember that lgrid and rgrid/2 are the benchmarks that find the largest
number of answers per time unit (please refer to Tables 8.3 and 8.4). Regarding lgrid’s statistics it shows high contention ratios in all parameters considered. Closer analysis of its statistics allows us to observe that it shows an identical pattern when compared with lgrid/2. The problem is that the ratio per time unit is significantly worst for lgrid. This reflects the fact that most of lgrid’s execution time is spent in massively accessing the table space to insert new answers and to consume found answers.

The sequential order by which answers are accessed in the trie structure is the key issue that reflects the high number of contention points in subgoal and dependency frames. When inserting a new answer we need to update the subgoal frame pointer to point at the last found answer. When consuming a new answer we need to update the dependency frame pointer to point at the last consumed answer. For programs that find a large number of answers per time unit, this obviously increases contention when accessing such pointers. Regarding trie nodes, the small depth of lgrid’s answer trie structure (2 trie nodes) is one of the main factors that contributes for the high number of contention points when massively inserting trie nodes. Trie structures are
8.2. PERFORMANCE ON TABLED PROGRAMS

a compact data structure. Therefore, obtaining good parallel performance in the presence of massive table access will always be a difficult task.

Analyzing the statistics for \textit{rgrid/2}, the number of variant subgoals calls and the number of suspended/resumed SCCs suggest that this benchmark leads to complex dependencies between workers. Curiously, despite the large number of consumer nodes that the benchmark allocates, contention in dependency frames is not a problem. On the other hand, contention for subgoal frames seems to be a major problem. The statistics suggest that the large number of SCC resume operations and the large number of answers that the benchmark finds are the key aspects that constrain parallel performance. A closer analysis shows that the number of resumed SCCs is approximately constant with the increase in the number of workers. This may suggest that there are answers that can only be found when other answers are also found, and that the process of finding such answers cannot be anticipated. In consequence, suspended SCCs have always to be resumed to consume the answers that cannot be found sooner. We believe that the sequencing in the order that answers are found is the other major problem that restrict parallelism in tabled programs.

Another aspect that can negatively influence this benchmark is the number of completed calls. Before executing the first call to a completed subgoal we need to traverse the trie structure of the completed subgoal. When traversing the trie structure the correspondent subgoal frame is locked. As \textit{rgrid/2} stores a huge number of answer trie nodes in the table (please refer to Table 8.4) this can lead to longer periods of lock contention.

Next, we present an identical study for parallel execution of OPTYap using local scheduling.

\textbf{Parallel Execution with Local Scheduling}

Table 8.8 presents the speedups for parallel execution of OPTYap with 2, 4, 6, 8, 12, 16, 24 and 32 workers using local scheduling. The speedups are relative to the single worker case of Table 8.3 and they correspond to the best speedup obtained in a set of 3 runs. As for batched, we group the benchmarks in two main blocks.

On average the results for local scheduling are worst than those obtained for batched.
Table 8.8: Speedups for OPTYap using local scheduling on tabled programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>sieve</td>
<td>2.00</td>
</tr>
<tr>
<td>leader</td>
<td>1.99</td>
</tr>
<tr>
<td>ipproto</td>
<td>1.68</td>
</tr>
<tr>
<td>samegen</td>
<td>1.93</td>
</tr>
<tr>
<td>rgrid/2</td>
<td>1.84</td>
</tr>
<tr>
<td>Average</td>
<td>1.89</td>
</tr>
</tbody>
</table>

Generally, the benchmarks that find more answers are the ones that introduce further overheads and obtain lesser speedups with local scheduling. These are the cases of the ipproto and rgrid/2 benchmarks for the upper block and the lgrid and rgrid/2 for the lower block. In order to understand what extra overheads local scheduling introduces for parallel execution, we present in Table 8.9 some statistics gathered during parallel execution of these four benchmarks. We do not include the statistics for the sieve, leader and samegen benchmarks because their execution behavior showed to be similar to the observed for batched scheduling.

A closer analysis of the statistics obtained in Table 8.9 for the four benchmarks in discussion clearly shows that the worst results obtained for local scheduling relate with a higher rate of contention in dependency frames. In particular, the difference is most obvious on the rgrid/2 benchmark. The rest of the parameters show comparable results to those obtained for batched scheduling.

We remember the reader that in local scheduling after a leader subgoal is completed we need to consume the answers that were prevented from being returned to the caller environment. For sequential execution this is done by executing compiled code directly from the trie data structure associated with the completed subgoal. Unfortunately, this optimization is not possible on our parallel implementation of local scheduling. The problem is that workers may start consuming answers before subgoals were completed. This occurs for workers where the subgoals are not leaders. Hence, when a leader subgoal is completed we just act like a consumer node and start consuming answers.
Table 8.9: Statistics of OPTYap using local scheduling for the group of programs showing worst speedups than for batched scheduling.

The results presented in Table 8.9 suggest that in some cases this may be incompatible with good performance. The typical situation is when a leader subgoal with a large

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Number of Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>ipproto</td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>7/0</td>
</tr>
<tr>
<td>contention points</td>
<td>36706</td>
</tr>
<tr>
<td>subgoal frame</td>
<td>3506</td>
</tr>
<tr>
<td>dependency frame</td>
<td>31235</td>
</tr>
<tr>
<td>trie node</td>
<td>1208</td>
</tr>
<tr>
<td>lgrid/2</td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>4/0</td>
</tr>
<tr>
<td>contention points</td>
<td>50723</td>
</tr>
<tr>
<td>subgoal frame</td>
<td>227</td>
</tr>
<tr>
<td>dependency frame</td>
<td>44217</td>
</tr>
<tr>
<td>trie node</td>
<td>4153</td>
</tr>
<tr>
<td>lgrid</td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>1/0</td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>4/0</td>
</tr>
<tr>
<td>contention points</td>
<td>246749</td>
</tr>
<tr>
<td>subgoal frame</td>
<td>18051</td>
</tr>
<tr>
<td>dependency frame</td>
<td>157773</td>
</tr>
<tr>
<td>trie node</td>
<td>56866</td>
</tr>
<tr>
<td>rgrid/2</td>
<td></td>
</tr>
<tr>
<td>variant/complete</td>
<td>3018/1157</td>
</tr>
<tr>
<td>SCC suspend/resume</td>
<td>1711/509</td>
</tr>
<tr>
<td>contention points</td>
<td>155099</td>
</tr>
<tr>
<td>subgoal frame</td>
<td>63247</td>
</tr>
<tr>
<td>dependency frame</td>
<td>87115</td>
</tr>
<tr>
<td>trie node</td>
<td>766</td>
</tr>
</tbody>
</table>
number of answers completes and its answers start being heavily consumed by the available workers, therefore leading to high ratios of contention in the dependency frames. We believe that this is a very hard problem to be solved even if different parallel tabling approaches were developed.

The statistics presented in the tables above clearly illustrate some of the problems behind parallel tabled evaluation. They are thus an excellent source for further study in order to improve and/or reformulate some of the implementation issues that showed to be less suitable for parallel execution.

Two major conclusions can be highlighted from the performance analysis done in this section. First, there are table applications that can achieve very high performance through parallelism. Second, batched scheduling showed to be more adequate than local scheduling for parallel execution.

### 8.2.5 Locking the Table Space

OPTYap implements four alternative locking schemes to deal with concurrent accesses to the table space data structures. These schemes were described in subsection 6.3.2 and were referred as: **TLEL** (Table Lock at Entry Level); **TLNL** (Table Lock at Node Level); **TLWL** (Table Lock at Write Level); and **TLWL-ABC** (Table Lock at Write Level - Allocate Before Check).

To evaluate the impact that different approaches to locking the table space may produce during parallel execution, we ran OPTYap using the four alternative locking schemes for the tabled benchmark programs that showed significant speedups for parallel execution. Table 8.10 shows the speedups for the four alternative locking schemes with varying number of workers for batched and local scheduling. The speedups are relative to the single worker case and they correspond to the best speedup obtained in a set of 3 runs.

Two main conclusions can be easily drawn from the speedups showed in Table 8.10. First, all benchmarks show identical patterns with the increase in the number of workers for both batched and local scheduling. Apparently, this suggests that scheduling does not significantly influence lock contention in table access. Second, TLWL and TLWL-ABC are the locking schemes that present the best speedup ratios and they are
### Table 8.10: OPTYap execution time with different locking schemes for the group of programs with parallelism.

<table>
<thead>
<tr>
<th>Locking Scheme</th>
<th>Batched Scheduling</th>
<th>Local Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td><strong>sieve</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLEL</td>
<td>3.79</td>
<td>7.35</td>
</tr>
<tr>
<td>TLNL</td>
<td>3.80</td>
<td>7.24</td>
</tr>
<tr>
<td>TLWL</td>
<td>3.99</td>
<td>7.97</td>
</tr>
<tr>
<td><strong>leader</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLEL</td>
<td>3.80</td>
<td>6.16</td>
</tr>
<tr>
<td>TLNL</td>
<td>3.49</td>
<td>6.32</td>
</tr>
<tr>
<td>TLWL</td>
<td>3.98</td>
<td>7.92</td>
</tr>
<tr>
<td>TLWL-ABC</td>
<td>3.98</td>
<td>7.94</td>
</tr>
<tr>
<td><strong>iproto</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLEL</td>
<td>1.66</td>
<td>1.41</td>
</tr>
<tr>
<td>TLNL</td>
<td>1.68</td>
<td>2.65</td>
</tr>
<tr>
<td>TLWL</td>
<td>3.05</td>
<td>5.08</td>
</tr>
<tr>
<td>TLWL-ABC</td>
<td>3.10</td>
<td>5.13</td>
</tr>
<tr>
<td><strong>samegen</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLNL</td>
<td>3.68</td>
<td>7.23</td>
</tr>
<tr>
<td><strong>lgrid/2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLEL</td>
<td>3.74</td>
<td>7.17</td>
</tr>
<tr>
<td>TLNL</td>
<td>3.48</td>
<td>6.79</td>
</tr>
<tr>
<td>TLWL-ABC</td>
<td>3.60</td>
<td>6.95</td>
</tr>
</tbody>
</table>

The only schemes showing scalability. Despite none of both schemes shows to clearly outperform the other, TLWL seems slightly better for batched scheduling and TLWL-ABC for local scheduling. In order to avoid choosing only one, we decided to use TLWL for OPTYap Batched and TLWL-ABC for OPTYap Local in the performance study described during this chapter.
Closer analysis to Table 8.10 allows us to observe other interesting aspects: all schemes show identical speedups for the *samegen* benchmark, and the TLEL and the TLNL schemes clearly slow down for more than 16 workers. The reason for the good behavior of all schemes with the *samegen* benchmark arises from the fact that this benchmark calls 485 different tabled subgoals. This increases the number of entries where answers can be stored and thus reduces the probability of two workers accessing simultaneously the same answer trie structure.

The slow-down of TLEL and TLNL schemes is related to the fact that these schemes lock the table space even when writing is not likely. In particular, for repeated answers they pay the cost of performing locking operations without inserting any new trie node. For these schemes the number of potential contention points is proportional to the number of answers found during execution, being they unique or redundant. This explains the slow-down presented by these schemes for the *sieve* and *leader* benchmarks. These benchmarks find a smaller number of unique answers, but have large number of redundant answers (please refer to Table 8.4). Curiously, for some benchmarks TLEL obtains better speedups than TLNL with the increase of workers. This suggests that for certain circumstances it is better to lock the whole trie and traverse it more quickly than lock node by node and increase the points of contention and the time spent to traverse the trie.

### 8.3 Chapter Summary

In this chapter we have presented a detailed analysis of OPTYap’s performance. We started by presenting an overall view of OPTYap’s performance for execution of non-tabled programs. Then, we measured the sequential tabling behavior of OPTYap and compared it with current XSB. Next, we assessed OPTYap’s performance when running tabled programs in parallel and discussed its execution behavior. At last, we studied the impact of using alternative locking schemes to concurrently access the table space.

The initial results obtained for OPTYap shows that it introduces low overheads over Yap and YapTab for sequential execution of non-tabled and tabled programs, and that it compares favorably with current versions of XSB. Moreover, the results showed that OPTYap maintains YapOr’s effective speedups in exploiting or-parallelism in non-
tabled programs. For parallel execution of tabled programs, OPTYap showed superb results for two benchmarks and quite good results globally. However, there are tabled programs where OPTYap may not speedup execution. Our study suggested that parallel execution of tabled programs is more natural for a batched scheduling strategy and for a TLWL or a TLWL-ABC locking scheme.
Chapter 9

Concluding Remarks

This long journey is about to end. In this final chapter, we begin by summarizing the main contributions of the thesis and then we suggest several directions for further travel. At the end, a final remark ceases the chapter and the thesis.

9.1 Main Contributions

The work described in this thesis can be stated as the design, implementation and evaluation of the OPTYap system. To the best of our knowledge, OPTYap is the first engine that exploits or-parallelism and tabling from logic programs. A major guideline for OPTYap was concerned with making best use of the excellent technology already developed for previous systems. In this regard, OPTYap uses Yap’s efficient sequential Prolog engine [29, 31] as its starting framework, and the SLG-WAM [86, 89, 87] and environment copying [5, 56] approaches, respectively, as the basis for its tabling and or-parallel components.

We then summarize the main contributions of our work.

Novel computational models for parallel tabling. We have proposed two novel computational models, the Or-Parallelism within Tabling (OPT) and Tabling within Or-Parallelism (TOP) models, that exploit implicit or-parallelism from tabled logic programs by considering all subgoals as being parallelizable, be they from tabled or non-tabled predicates.
The YapTab sequential tabling engine. We have presented the design and implementation of YapTab, an extension to the Yap Prolog system that implements sequential tabling. YapTab reuses the principles of the SLG-WAM, whilst innovating by separating the tabling suspension data in a single space, the dependency space, and by proposing a new completion detection algorithm not based on the intrinsically sequential completion stack. YapTab has been implemented from scratch and it was developed to be used as the basis for OPTYap’s tabling component. YapTab showed low overheads over standard Yap when executing non-tabled programs, and excellent results for tabling benchmarks if compared with the more mature XSB system [44].

The OPTYap or-parallel tabling engine. OPTYap’s execution framework was a first step to study and understand the behavior and implications of exploiting parallelism from tabled logic programs. During this thesis, we have presented novel data structures, algorithms and implementation techniques to efficiently solve the challenging issues that a project of this size encompasses. These contributions can be used as a reference guide for other approaches that may follow. Next, we enumerate the most relevant contributions.

- The dependency frame data structure and the idea of keeping apart, in a common shared space, the whole data related with tabling suspensions.
- The generator dependency node (GDN) concept of signalling nodes that are candidates to be leader nodes.
- New algorithms to quickly compute and detect leader nodes.
- The novel termination detection scheme to allow completion in public nodes.
- The support for suspension of strongly connected components (SCCs) and the assumption of SCCs as the units for suspension.
- Newer scheduler heuristics to support tabling that explicitly deal with the flow of a parallel tabled evaluation and achieve a more efficient distribution of work in such evaluations.
- The implementation techniques to deal with concurrent table access and the TLEL, TLNL, TIWL and TLWL-ABC locking schemes.
- The distinction between inner and outer cut operations in a parallel tabling environment and the support for speculative tabled answers.
Performance study. We have performed a detailed study to assess the performance of the or-parallel tabling engine over a large number of parameters. During evaluation, the system was examined against a selected set of benchmark programs that we believe are reasonably representative of existing applications. From the results obtained, the following observations can be enumerated.

- Sequential execution of non-tabled programs showed that YapOr, YapTab and OPTYap introduce, on average, respectively an overhead of about 10%, 5% and 17% over standard Yap. Considering that Yap Prolog is one of the fastest Prolog engines currently available, these results are quite satisfactory.

- Parallel execution of non-tabled programs showed that YapOr and OPTYap achieve, on average, identical speedups up to 32 workers. This result suggests that OPTYap do not introduces further overheads for parallel execution of non-tabled programs, despite the fact that it includes all the machinery required to support tabled programs.

- Sequential execution of tabled programs indicate that OPTYap introduces, on average, an overhead of about 15% over YapTab for both batched and local scheduling strategies, which is very close to the overhead observed for non-tabled programs, about 11%. The small difference results from locking requests to the data structures introduced with tabling. The results also showed that we successfully accomplished our initial goal of comparing favorably with current state of the art technology since, on average, YapTab showed to be about twice as fast as XSB.

- Parallel execution of tabled programs showed that the system was able to achieve excellent speedups up to 32 workers for applications with coarse grained parallelism and quite good speedups for applications with medium parallelism. Our results suggested that parallel execution of tabled programs is more natural for batched scheduling than for local scheduling and that concurrent table access is best handled by schemes that lock table data structures only when writing to the table is likely. On the other hand, there are applications where OPTYap was not able to speedup their execution. This is the case with applications whose evaluation is mostly deterministic or whose main execution operations rely on massive accesses.
to the table space. The parameters evaluated during execution suggested that the slowdown for these applications is not caused by workers becoming idle, but because there is a lot of contention in handling tabled answers. In general, tabling tends to decrease the height of the search tree, whilst increasing its breadth. We therefore believe that further improvements in scheduling and on concurrent access to the data structures introduced to support parallel tabling may be fundamental to achieve even better scalability.

Through this research we aimed at showing that the models developed to exploit implicit or-parallelism in standard logic programming systems can also be used to successfully exploit implicit or-parallelism in tabled logic programming systems. Initial results show that OPTYap can indeed speed up well known tabled programs without programmer intervention. The results reinforced our belief that tabling and parallelism are a very good match that can contribute to expand the range of applications for Logic Programming.

9.2 Further Work

We hope that the work resulting from this thesis will be a basis to conduct further improvements and further research in this area. OPTYap has achieved our initial goal. Even so, the system still has some limitations that may reduce its use elsewhere and its contribution in the support of realistic applications. Current limitations relate to issues not within the scope of the present work, but that are very important for wider use throughout the logic programming community. These include:

**Further experimentation.** The current implementation needs to be tested more intensively with a wider range of applications. Many opportunities for refining the system exist, and more will almost certainly be uncovered with profound experimentation of the system. We gratefully acknowledge the generosity of tabling logic programming community by providing us access to several interesting applications, such as XMC. We are experimenting with other tabled logic programming applications and differently platforms.
Scheduling strategies. OPTYap scheduling strategies are essentially inherited from YapOr's scheduler. Further work is still needed to implement and experiment with proper scheduling strategies that can take advantage of the tabling environment. In subsection 6.8 we have proposed novel scheduling strategies that we believe should contribute for a more efficient work distribution strategy in an or-parallel tabled evaluation.

Speculative work limitations. For certain groups of applications, such as best-solution kind of problems, speculative computations represent a major problem. OPTYap prunes speculative computations as soon as a cut causing their speculative nature is executed. However, it does not implement any scheduling strategy that makes speculative computations less likely. To some extent, these limitations can be addressed by implementing Muse's sophisticated strategy — actively seeking leftmost available work [8], to voluntary suspend rightmost computations and thus reduce the degree of speculativeness of the work being done to obtain high performance (please refer to subsection 7.2.5 for more details).

In the presence of tabling, pruning is an even more delicate issue. A deeper understanding of the interaction between pruning and tabling is required. We need to do it correctly, that is, in such a way that the system will not break but instead produce sensible answers, and well, that is, allow useful pruning with good performance.

Support for full Prolog. To support full Prolog semantics, the system still needs more development, specially to support side-effects effectively. To ensure sequential Prolog semantics, side-effects must be executed by leftmost workers. Full support for side-effects in YapOr can be achieved by extending some of the data structures used to support the cut predicate and to support SCC suspension for parallel tabling. One interesting problem is the management of the internal database, as many applications require concurrency in database updates. Yap already includes the base machinery to allow such concurrency, however further work is need to make it usable by programmers. Several ideas about efficient side-effects implementation can be found elsewhere [53, 5, 21, 101, 57, 102].

Tabling is a more complex problem. Semantics are different and side-effects are not Prolog compatible in tabling, as they may depend on scheduling order. What do programmers expect from side-effects in a tabling environment is still
an open problem.

**Dynamic memory expansion.** OPTYap allows to indicate the amount of memory required for each data area. However, during execution one may discover that the memory initially requested was insufficient. We would like to lift that burden from the user by allowing dynamic memory expansion. Unfortunately, dynamic memory expansion is a very complex operation when supporting an environment copying based implementation. Accomplishing efficiency is even more laborious. Proposals for novel memory organization schemes enabling efficient dynamic memory expansion operations are therefore required.

**Garbage collection.** By nature, garbage collection is a heavy cost operation. For an environment copying based system, garbage collection may also lead to inconsistency between the execution stacks of the running workers. Special care is not taken when incremental copying is used to share work. Although YapTab supports garbage collection, OPTYap does not implement garbage collection at all. In [4] K. Ali proposes some interesting mechanisms to deal with garbage collection for environment copying systems.

**Support for negation.** A wide range of applications that use tabling require the expressiveness granted by the possibility of manipulating negative subgoals. OPTYap does not currently implement support for negation. Extending OPTYap to efficiently support negation will certainly be one major step forward to make OPTYap usable by a larger community.

### 9.3 Final Remark

Clearly, the research we present in this thesis is built on the vigorous research effort made by preceding researchers. Their ideas brought us the flame that has lighted up our way. With our work, we hope to shed at least a ray of light to someone else that may follow.

Much work still remains to be done. A large amount of this available work will be exploited in parallel by many different research workers all over the world. Sometimes, much of the clues to pursue such work have already been tabled by other researchers when studying variant problems. The question therefore is how to efficiently distribute
the available tasks through the available workers in such a way that we avoid *speculative work* and redundant *answers* for the *subgoals* of the ultimate *query goal*:

\[ ?- \text{develop\_system}(S), \text{least\_development\_cost}(S), \text{best\_achievable\_performance}(S). \]
Appendix A

Benchmark Programs

This appendix contains the benchmark programs used in Chapter 8 to assess OPTYap’s performance. For the set of non-tabled benchmark programs we provide the full Prolog code. On the other hand, as the tabled benchmark programs are quite lengthy, we only show parts of the code. The author may be contacted for the full Prolog code of these programs.

A.1 Non-Tabled Benchmark Programs

cubes

benchmark :- cubes7(_,).
cubes7(Sol) :-
cubes7(SQ), solve(SQ, [], Sol).
cubes7([q(p(5,1),p(0,5),p(3,1)),
q(p(2,3),p(1,4),p(4,0)),
q(p(3,6),p(0,0),p(2,4)),
q(p(6,4),p(6,1),p(0,1)),
q(p(1,6),p(3,2),p(5,2)),
q(p(5,0),p(2,3),p(4,5)),
q(p(4,2),p(2,6),p(0,3))]).
solve([], Rs, Rs).
solve([C1|Cs], Ps, Rs) :-
solve(Cs, [P|Ps], Rs).
check([P], P).
check([q(A1,B1,C1,D1)|Ps], P) :-
P = q(A2,B2,C2,D2),
A1\=A2,
B1\=B2,
C1\=C2,
D1\=D2,
check(Ps, P).
set(q(A1,B1,C1,D1), P).
rotate(P2,P1,P).
**APPENDIX A. BENCHMARK PROGRAMS**

### ham

benchmark :- ham(_).

ham(H) :-
  cycle_ham([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z],H).

cycle_ham([X|Y], [X,T|L]) :-
  chain_ham([XIY], [], [TIL]),
  ham_edge(T,X).

chain_ham([X|Y], [X,T|L]) :-
  ham_del(Z,Y,T),
  ham_edge(X,Z),
  chain_ham([Z|T], [X|K], L).

ham_del(X, [X|Y], Y).

ham_del(X, [U|Y], [U|Z]) :-
  ham_del(X, Y, Z).

ham_edge(X, Y) :-
  ham_connect(X, L),
  ham_el(Y, L).

ham_el(X, [X|J]).

ham_el(X, [_|L]) :-
  ham_el(X, L).

ham_connect(a, [b,n,m]).

### map

benchmark :- map(_).

map(M) :-
  my_map(M),
  map_colours(C),
  colour_map(M, C).

my_map([country(a, A, Adjacents), country(b, B, Adjacents), country(c, C, Adjacents), country(d, D, Adjacents), country(e, E, Adjacents), country(f, F, Adjacents), country(g, G, Adjacents), country(h, H, Adjacents), country(i, I, Adjacents), country(j, J, Adjacents), country(k, K, Adjacents), country(l, L, Adjacents), country(m, M, Adjacents), country(n, N, Adjacents), country(o, O, Adjacents), country(p, P, Adjacents), country(q, Q, Adjacents), country(r, R, Adjacents), country(s, S, Adjacents), country(t, T, Adjacents), country(u, U, Adjacents), country(v, V, Adjacents), country(w, W, Adjacents), country(x, X, Adjacents), country(y, Y, Adjacents), country(z, Z, Adjacents)]).

colour_map([], _).

colour_map([Country|Map], Colourlst) :-
  colour_country(Country, Colourlst),
  colour_map(Map, Colourlst).

colour_country(country(_, C, Adjacents), Colourlst) :-
  map_del(C, Colourlst, CL),
  map_subset(Adjacents, CL).

map_subset([], _).

map_subset([CICs] . Colourlst) :-
  map_del(C, Colourlst, _),
  map_subset(Cs, Colourlst).

map_colours([red, green, blue, white, black]).

### nsort

benchmark :- nsort(_).

nsort(L) :-
  go_nsort([10,9,8,7,6,5,4,3,2,1], L).

go_nsort(L1, L2) :-
  nsort_permutation(L1, L2),
  nsort_sorted(L2).

nsort_permutation([], []).

nsort_permutation([L|L2], [H|T]) :-
  X=<Y, nsort_permutation([Y|L2], [H|T]),
  nsort_permutation(L, [X|T]).
A.1. NON-TABLED BENCHMARK PROGRAMS

puzzle

benchmark :- puzzle(_).
    List=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],
    member(A,List,La),
    member(B,La,Lb),
    C is 38-A-B,
    member(C,Lb,Lc),
    A<C,
    member(D,Lc,Ld),
    H is 38-A-D,
    member(H,Ld,Lh),
    member(E,Lh,Le),
    member(F,Le,Lf),
    G is 38-D-E-F,
    member(G,Lf,Lg),
    L is 38-C-G,
    member(L,Lg,Ll),
    A<L,
    member(I,Ll,Li),
    M is 38-B-E-I,
    member(M,Li,Lm),
    Q is 38-H-M,
    member(G,Lm,Lq),
    A<Q,
    member(J,Lq,Lj),
    N is 38-C-F-J-Q,
    member(K,Lj,Lk),
    P is 38-H-I-J-L,
    member(P,Lk,Lp),
    S is 38-I-P,
    member(S,Lp,Le),
    A<S,
    R is 38-Q-S,
    member(Q,Le,Lr),
    38 is D=I=E+R,
    member(U,Lr,Lo),
    38 is N+O+P+F,
    38 is A+K+O+S,
    38 is G+K+O+N.
    member(X,[X|Y],Y).
    member(X,[X,Y],[X2,Y2]).

queens

benchmark :- queens(_).
queens(S) :-
    get_solutions(11,S).
get_solutions(Board_size,Soln) :-
    solve(Board_size,[],Soln).
solve(Board_size,Initial,Final) :-
    newsquare(Initial,Next),
    solve(Board_size,[Next|Initial],Final).
solve(Boards_size,[square(I,J)|L],Square) :- size(Boards_size).
    newsquare(squares(I,J)|Rest),square(X,Y)) :-
        X is I+1,
        safe(X,Y),
        not_threatened(I,J,X,Y),
        safe(X,Y,Rest).
    newsquare([],square(1,1)) :-
        snint(X).
    not_threatened(I,J,X,Y) :-
        I=\=1,
        J=\=Y,
        I=\=J,
        square(X,Y,[square(I,J)|L]) :-
        not_threatened(I,J,X,Y),
        safe(X,Y,Rest).
    size(11).
    snint(1).
    snint(2).
    snint(3).
    snint(4).
    snint(5).
    snint(6).
    snint(7).
    snint(8).
    snint(9).
    snint(10).
    snint(11).
A.2 Tabled Benchmark Programs

sieve

```
benchmark :- reach(sieve_0(5,4,27,end), T).
:= table reach/2.
reach(S,T) :-
    trans(S,_,T).
reach(S,T) :-
    reach(S,N),
    trans(N,_,T).
```

```
% the transition relation graph
trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
    (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(generator_0(A,B,C,D),out(A,B),D) :-
    E is B+1, not B=C.
```

```
% auxiliary predicates
```

leader

```
benchmark :- reach(systemLeader_0(5,end), T).
:= table reach/2.
reach(S,T) :-
    trans(S,_,T).
reach(S,T) :-
    reach(S,N),
    trans(N,_,T).
```

```
% the transition relation graph
trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
    (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(medium_0(A,B,C,D),in(A,E),medium_0(A,B,[E|C],D)).
```

```
% auxiliary predicates
```

iproto

```
benchmark :- reach(iproto_0(_,_,end), T).
:= table reach/2.
reach(S,T) :-
    trans(S,_,T).
reach(S,T) :-
    reach(S,N),
    trans(N,_,T).
```

```
% the transition relation graph
trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
    (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(medium_0(A,B,C,D),in(A,E),medium_0(A,B,[E|C],D)).
```

```
% auxiliary predicates
```

samegen

```
benchmark :- same_generation(_,_).
:= table same_generation/2.
same_generation(X,Y) :-
    cyl(X,Z),
    same_generation(Z,W),
    cyl(Y,W).
same_generation(X,X).
```

```
% the cylinder data
cyl(1,30).
cyl(1,40).
cyl(2,43).
...```
A.2. TABLED BENCHMARK PROGRAMS

lgrid

benchmark :- lpath(_,_).
:- table lpath/2.
lpath(X,Y) :-
lpath(X,Z),
    link(Z,Y).
lpath(X,Y) :-
    link(X,Y).

% the 25x25 grid
link(1,2).
link(2,3).
link(3,2).
...
link(575,600).
link(600,625).
link(625,600).

lgrid/2

benchmark :- lpath(_,_).
:- table lpath/2.
lpath(X,Y) :-
lpath(X,Z),
    arc(Z,Y).
lpath(X,Y) :-
    arc(X,Y).
arc(X,Y) :-
    link(X,Y).
arc(X,Y) :-
    link(Y,X).

% the 20x20 grid
link(1,2).
link(2,3).
link(3,4).
link(4,5).
link(5,6).
...
link(300,320).
link(320,340).
link(340,360).
link(360,380).
link(380,400).

rgrid/2

benchmark :- rpath(_,_).
:- table rpath/2.
rpath(X,Y) :-
arc(X,Y).
rpath(X,Y) :-
arc(X,Z),
rpath(Z,Y).
arc(X,Y) :-
    link(X,Y).
arc(X,Y) :-
    link(Y,X).

% the 20x20 grid
link(1,2).
link(2,3).
link(3,4).
link(4,5).
link(5,6).
...
link(300,320).
link(320,340).
link(340,360).
link(360,380).
link(380,400).
References


REFERENCES


REFERENCES

Processing, number 1900 in LNCS, pages 744–753. Springer-Verlag, September 2000.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


