Faculdade de Ciências e Tecnologia
Departamento de Informática

Concurrent Tabling: Algorithms and Implementation

Rui Filipe Pereira Marques

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(2006)

This dissertation was prepared under the supervision of
Professor Terrance Lee Swift
of State University of New York at Stony Brook
and
Professor José Cardoso e Cunha,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To the memory of José Carlos Pereira Marques, a fighter for life

Acknowledgments

First I'd like to thank both my advisers for their support. Terrance Swift was most committed in helping me through the dissertation. He didn't stop short of meeting with me by phone on Sundays, though he was 6000 km away. He put my code under the microscope and designed most of the multi-threaded tests. He went as far as doing some programming, which is mentioned where appropriate. Although he was not as involved with the implementation work as Terrance Swift, José Cunha's support also proved invaluable. He supported me within the department in all situations. He's help with multi-threading was also important – I take the chance to mention he was one of the first people to propose multi-threaded Prolog, proposal which first took form in my graduation project, back in 1992.

I also have to thank David S. Warren. There's no need to mention his importance in the logic programming field. In the few situations that we've met he was always a great source of inspiration. He did some implementation work that helped enriching this thesis; he also help with some random bugs, either reporting them, or actually fixing them! I have to thank also the people from XSB research group, mostly Kostis Sagonas, Prasad Rao, Juliana Freire, Rui Hu and Luis de Castro, most of them who have left the group long ago, for their support and introduction to the XSB environment. I also thank the XSB user community for their feedback.

I have to thank my university teaching colleagues, Vítor Duarte, João Lourenço, Pedro Medeiros and Cecília Gomes, for their understanding and support, which ranged from lightning my teaching workload to encouraging and giving useful tips to me, during the elaboration of the dissertation.

The research done in this dissertation was financially supported by the Fundaçcão para a Ciência e Tecnologia and by research project TARDE whose principal researcher was of Carlos Damásio.

Summary

In the past few years Tabling has emerged as a powerful logic programming model. The integration of concurrent features into the implementation of Tabling systems is demanded by need to use recently developed tabling applications within distributed systems, where a process has to respond concurrently to several requests. The support for sharing of tables among the concurrent threads of a Tabling process is a desirable feature, to allow one of Tabling's virtues, the re-use of computations by other threads and to allow efficient usage of available memory. However, the incremental completion of tables which are evaluated concurrently is not a trivial problem.

In this dissertation we describe the integration of concurrency mechanisms, by the way of multi-threading, in a state of the art Tabling and Prolog system, XSB. We begin by reviewing the main concepts for a formal description of tabled computations, called SLG resolution and for the implementation of Tabling under the SLG-WAM, the abstract machine supported by XSB. We describe the different scheduling strategies provided by XSB and introduce some new properties of local scheduling, a scheduling strategy for SLG resolution.

We proceed to describe our implementation work by describing the process of integrating multi-threading in a Prolog system supporting Tabling, without addressing the problem of shared tables. We describe the trade-offs and implementation decisions involved.

We then describe an optimistic algorithm for the concurrent sharing of completed tables, Shared Completed Tables, which allows the sharing of tables without incurring in deadlocks, under local scheduling. This method relies on the execution properties of local scheduling and includes full support for negation. We provide a theoretical framework and discuss the implementation's correctness and complexity.

After that, we describe a method for the sharing of tables among threads that allows parallelism in the computation of inter-dependent subgoals, which we name Concurrent Completion. We informally argue for the correctness of Concurrent Completion.

We give detailed performance measurements of the multi-threaded XSB systems over a variety of machines and operating systems, for both the Shared Completed Tables and the Concurrent Completion implementations. We focus our measurements in

the overhead over the sequential engine and the scalability of the system.

We finish with a comparison of XSB with other multi-threaded Prolog systems and we compare our approach to concurrent tabling with parallel and distributed methods for the evaluation of tabling. Finally, we identify future research directions.

Sumário

Nos últimos anos a Tabulação têm-se destacado como um modelo de programação em lógica poderoso. A integração de mecanismos de concorrência na implementação de sistemas tabulados é requerida pelo recente desenvolvimento de aplicações tabuladas que precisam de ser integradas em sistemas distribuídos, onde um processo tem de responder em simultâneo a vários pedidos. O suporte à partilha de tabelas entre os diversos processos leves é desejável, pois permite explorar uma das vantagens da Tabulação, a re-utilização de resultados das computações prévias de outros processos leves e a utilização eficiente da memória disponível. No entanto, a completação incremental de tabelas avaliadas em concorrência não é um problema trivial.

Nesta dissertação descrevemos a integração de mecanismos de concorrência, através do suporte de processos leves, num sistema de Prolog e Tabulação que representa o estado da arte da tecnologia desta, o sistema XSB. Começamos por rever os conceitos principais usados na descrição formal da execução tabulada de programas, a resolução SLG, e da implementação da tabulação através do modelo SLG-WAM, a máquina abstracta suportada pelo XSB. Descrevemos as diferentes políticas de escalonamento utilizadas no sistema XSB e introduzimos novas propriedades do escalonamento local, uma política de escalonamento da resolução SLG.

Seguidamente, descrevemos o trabalho de concepção e implementação, abordando o processo de integração dos processos leves num sistema Prolog com suporte à tabulação, sem abordar a partilha de tabelas. Descrevemos os compromissos e as decisões de implementação tomadas.

Depois descrevemos um algoritmo optimista para a partilha de tabelas entre processos leves, as Tabelas Partilhadas Completas, que permite aos processos leves partilharem tabelas completas sem a ocorrência de deadlocks, sob o escalonamento local. Este método depende das propriedades do escalonamento local e inclui suporte à negação. Providenciamos um enquadramento teórico e discutimos a correcção e complexidade da implementação.

Em seguida descrevemos um método para partilhar tabelas que permite paralelismo na avaliação de subgolos inter-dependentes, que designamos como Completção Concorrente. Discutimos informalmente a correcção da Completação Concor-

rente.

Apresentamos resultados das medições de desempenho detalhadas obtidas com os sistemas XSB com processos leves, sobre uma variedade de máquinas e sistemas operativos, para ambas as implementações, Tabelas Completas Partilhadas e Completação Concorrente. Concentramos as nossas medidas no impacto dos processos leves em comparação com a implementação sequencial e na escalabilidade do sistema.

Terminamos com uma comparação do XSB com outros Prologs com processos leves e comparamos a nossa abordagem à tabulação concorrente com outras abordagens sobre a execução paralela e distribuída para a computação da tabulação. E finalmente, identificamos direcções de trabalho futuro.

Contents

1 Introduction **1**

1.1 The Demand for Multi-Threaded Tabling 4

1.2 Structure of the Dissertation 5

2 Background **7**

2.1 Introduction to Tabling 7

2.2 SLG Resolution 9

2.2.1 An Operational Semantics for Tabling 9

2.2.2 Example of SLG Resolution 14

2.2.3 Local SLG Evaluations 16

2.3 The SLG-WAM for Definite Programs 20

2.3.1 The WAM 20

2.3.2 SLG-WAM Stacks and Registers 22

2.3.3 Table Space 26

2.3.4 The SLG-WAM Instruction Set 27

2.3.5 Scheduling 30

2.4 Support for Negation in the SLG-WAM under Local Scheduling 33

2.4.1 Support for Stratified Negation 34

2.4.2 Support for Delay and Simplification 35

3 The Multi-Threaded Engine **39**

3.1 Multi-Threaded Programming 39

3.1.1 Multi-Threaded Prolog 41

3.1.2 Multi-Threaded Tabling 43

3.2 Design of Multi-Threaded XSB 43

3.2.1 Multi-Threaded API 45

3.3 Implementation Details 51

3.3.1 Multi-Threading for Pure Prolog 52

3.3.2 Support for Thread Private Tables 55

4 Shared Completed Tables **61**
4.1 Changing the SLG-WAM to support Shared Completed Tables 62
4.1.1 Data Structures 63
4.1.2 The Table_Try Instruction 64
4.1.3 Detecting a Deadlock 66
4.1.4 Resetting the other Threads in the Deadlock 67
4.1.5 The Check_Complete Instruction 68
4.2 A Semantics for Shared Completed Tables 70
4.2.1 SLG_{sct}: an Operational Semantics for Shared Completed Tables . 70
4.2.2 Local Multi-Threaded Evaluations 74
4.2.3 Complexity of SLG_{sct} 75
4.3 Example of Execution under Shared Completed Tables 76
4.4 Correctness of Shared Completed Tables 79
4.4.1 Correctness of the Implementation of the USURPATION Operation 79
4.4.2 Complexity of Shared Completed Tables 80

5 Concurrent Completion **83**
5.1 Changing the SLG-WAM to Support Concurrent Completion 84
5.1.1 Data Structures 84
5.1.2 Changes to the Table_Try Instruction 86
5.1.3 The New Check_Complete Instruction 88
5.1.4 Scheduling Answers 90
5.1.5 TDL Handling Procedures 91
5.1.6 Completion of a Single Thread 93
5.2 Example of Concurrent Completion 95
5.3 Correctness of Concurrent Completion 103
5.3.1 Completeness 103
5.3.2 Liveness 106
5.3.3 Note on Complexity for Concurrent Completion 110
5.4 Open Aspects of Concurrent Completion 111

6 Related Work **113**
6.1 Multi-Threaded Prolog 114
6.2 Parallel Tabling over Shared Memory Environments 122
6.2.1 Proposals for Parallel Evaluation of Tabling 124
6.2.2 OPTYap - a Parallel Tabling System 124
6.3 Tabling over Distributed Memory Environments 126

7 Performance Analysis **129**
7.1 Overhead for Execution of One Thread 130
7.1.1 Prolog Execution 130

7.1.2 Transitive Closure Benchmarks 132
7.1.3 Abstract Interpretation Benchmarks 137
7.2 Scalability of Private Tables on a Multi-Processor 142
7.3 Experiments with Shared Tables 145
7.3.1 Memory Usage 146
7.3.2 Occurrence of Deadlocks 147
7.3.3 Performance in a Dual Core System 148
7.3.4 Scalability of Shared Tables on a Multi-Processor 152

8 Conclusions **155**
8.1 Contributions of this Work 155
8.2 Future Work 157

List of Figures

1.1 Ancestor using right and left recursion 2
1.2 Fibonacci in Prolog 2

2.1 Program $P_{2.1}$ 7
2.2 SLD evaluation of program $P_{2.1}$ 8
2.3 Tabled evaluation of program $P_{2.1}$: deriving `p(s,b)` 8
2.4 Tabled evaluation of program $P_{2.1}$: deriving `p(b,c)` 9
2.5 Tabled evaluation of program $P_{2.1}$: `p(c,Z')` completes 10
2.6 Program $P_{2.2}$ 14
2.7 SLG evaluation of program $P_{2.2}$: the delay operation 15
2.8 SLG evaluation of program $P_{2.2}$: a conditional answer 15
2.9 SLG evaluation of program $P_{2.2}$: final forest 15
2.10 SDG for forests in Figures 2.5 and 2.8. 16
2.11 Format of generator choice points. 24
2.12 Format of consumer choice points. 25
2.13 Format of completion stack frames. 25
2.14 Data structures for a tabled predicate 26
2.15 Trie for terms `p(a(X),c)`,`p(a(X),Y)`, `p(b,X')` and `p(X",d(X"))` 26
2.16 Format of subgoal frames. 27
2.17 Answer trie for answers of `p(a(X),Y)`: `(X=a,Y=b)`; `(X=a,Y=c(X'))`; `(X=d,Y=Y')` 27
2.18 SLG-WAM code for predicate `p/2` of program $P_{2.1}$. 28
2.19 The table_try instruction. 29
2.20 Updating ASCC information on encountering consumer subgoals. . . . 30
2.21 The check_complete instruction for definite programs (Batched scheduling). 30
2.22 The fixpoint_check function. 31
2.23 The schedule_resumes function. 31
2.24 The check_complete instruction for definite programs (Local scheduling). 32
2.25 Format of negation suspension frames. 33
2.26 Implementation of the tabled negation predicate (`tnot/1`) 34

2.27 The check_complete instruction for normal programs (local scheduling). . 36
2.28 The ProcNegSusps function. 36

3.1 Multi-threaded execution model of a Prolog process 42
3.2 Multi-threaded memory layout of a XSB process 45
3.3 A multi-threaded goal server in XSB 49
3.4 A multi-threaded program to generate prime numbers in XSB using term queues 50
3.5 A multi-threaded program to generate prime numbers in XSB using the exit/join mechanism 51
3.6 Data structures for thread private tabled predicates 57
3.7 Data structures for thread private tabled predicates 58

4.1 Program $P_{4.1}$ 61
4.2 Program $P_{4.2}$ 62
4.3 The table_try instruction. 65
4.4 The would_deadlock function. 66
4.5 The reset_other_threads procedure. 67
4.6 The reset_thread procedure. 68
4.7 The ReclaimDSandMarkReset procedure. 68
4.8 The bottom_leader function. 69
4.9 The check_complete instruction. 69
4.10 Program $P_{4.3}$ 76
4.11 Concurrent execution of $P_{4.3}$: State 1 77
4.12 Concurrent execution of $P_{4.3}$: State 2 77
4.13 Concurrent execution of $P_{4.3}$: State 3 77
4.14 Concurrent execution of $P_{4.3}$: State 4 78
4.15 Concurrent execution of $P_{4.3}$: State 5 78
4.16 Concurrent execution of $P_{4.3}$: State 6 79
4.17 Concurrent execution of $P_{4.3}$: State 7 79

5.1 The external consumer choice point 84
5.2 The table_try instruction. 87
5.3 The check_complete instruction. 89
5.4 The fixpoint_check Procedure. 90
5.5 The UpdateDeps procedure. 91
5.6 The CheckForSCC function. 92
5.7 The MayHaveAnswers function. 92
5.8 The CompleteOtherThreads procedure. 92
5.9 The WakeOtherThreads procedure. 93
5.10 The CompleteTop procedure. 93
5.11 The WakeDependentThreads procedure. 94

5.12 Program $P_{5.1}$ 94
5.13 Concurrent execution of $P_{5.1}$: State 1 95
5.14 Concurrent execution of $P_{5.1}$: State 2 96
5.15 Concurrent execution of $P_{5.1}$: State 3 97
5.16 Concurrent execution of $P_{5.1}$: State 4 97
5.17 Concurrent execution of $P_{5.1}$: State 5 98
5.18 Concurrent execution of $P_{5.1}$: State 6 99
5.19 Concurrent execution of $P_{5.1}$: State 7 100
5.20 Concurrent execution of $P_{5.1}$: State 8 101
5.21 Concurrent execution of $P_{5.1}$: State 9 102
5.22 Concurrent execution of $P_{5.1}$: State 10 102
5.23 Concurrent execution of $P_{5.1}$: Final State 103

6.1 A multi-threaded program to generate prime numbers in IC-Prolog II . . 115
6.2 A multi-threaded program to generate prime numbers in PVM-Prolog . 116
6.3 A multi-threaded program to generate prime numbers in SICStus MT . . 117
6.4 A multi-threaded program to generate prime numbers in BinProlog . . . 117
6.5 A multi-threaded program to generate prime numbers in Ciao 118
6.6 A multi-threaded program to generate prime numbers in Qu-Prolog . . 119
6.7 A multi-threaded program to generate prime numbers in SWI-Prolog . . 120
6.8 A multi-threaded program to generate prime numbers in P# 121
6.9 A multi-threaded program to generate prime numbers in Logtalk 122

7.1 Ancestor with right recursion 132
7.2 Ancestor with left recursion 132
7.3 “Win not win” program 133

List of Tables

6.1 Summary of the features supported by several Multi-Threaded Prolog systems . . . 123

7.1 Lock usage for Prolog benchmarks . . . 131

7.2 Overheads of the multi-threaded engine for Prolog benchmarks for one thread in Linux . . . 131

7.3 Overheads of the multi-threaded engine for Prolog benchmarks for one thread in the Mac OS X system . . . 132

7.4 Lock usage for Transitive Closure benchmarks with Local scheduling in the Shared Completed Tables engine. . . . 134

7.5 Times for the Transitive Closure benchmarks for one thread using Local scheduling in the Shared Completed Tables engine in Linux . . . 134

7.6 Gain for taking out the locks in the multi-threaded engine with Local scheduling in the Shared Completed Tables engine for Transitive Closure benchmarks on Linux . . . 135

7.7 Times for the Transitive Closure benchmarks for one thread using Local scheduling under the Shared Completed tables engine in Mac OS X . . . 135

7.8 Lock usage for Transitive Closure benchmarks using Batched scheduling in the Concurrent Completion engine. . . . 136

7.9 Times for the Transitive Closure benchmarks for one thread using Batched scheduling in the Concurrent Completion engine in Linux . . . 136

7.10 Gain for taking out the locks in the multi-threaded engine with Batched scheduling in the Concurrent Completion engine in Linux . . . 137

7.11 Times for the Transitive Closure benchmarks for one thread using Batched scheduling in the Concurrent Completion engine in Mac OS X . 137

7.12 Lock usage for Abstract Interpretation programs for Local scheduling in the Shared Completed Tables engine . . . 138

7.13 Times for the Abstract Interpretation programs for one thread using Local scheduling in the Shared Completed tables engine in Linux . . . 138

7.14 Gain for taking out the locks in the multi-threaded engine with Local scheduling in the Shared Completed Tables engine in Linux 139
7.15 Times for the Abstract Interpretation programs for one thread using Local scheduling with the Shared Completed Tables engine in Mac OS X . 139
7.16 Lock usage for Abstract Interpretation programs for Batched scheduling in the Concurrent Completion engine 140
7.17 Times for the Abstract Interpretation programs for one thread using Batched scheduling for the Concurrent Completion engine in Linux . . . 140
7.18 Gain for taking out the locks in the multi-threaded engine with Batched scheduling using the Concurrent Completion engine in Linux 141
7.19 Times for the Abstract Interpretation programs for one thread using Batched Scheduling in the Concurrent Completion engine in Mac OS . . 141
7.20 Times for scalability of Prolog benchmarks for Solaris 143
7.21 Times for scalability of Transitive Closure benchmarks for private tables using Local scheduling for Solaris 143
7.22 Times for scalability of Transitive Closure benchmarks for private tabled using Batched scheduling for Solaris 144
7.23 Times for scalability of Abstract Interpretation benchmarks for private tables using Local scheduling for Solaris 144
7.24 Times for scalability of Abstract Interpretation benchmarks for private tables using Batched scheduling for Solaris 145
7.25 Peak memory usage for Right Recursion benchmarks with Local scheduling under the Shared Completed Tables engine in Linux 147
7.26 Peak memory usage for the Right Recursion program with Batched scheduling under the Concurrent Completion engine in Linux 147
7.27 Number of deadlocks for the Right Recursion program with Local scheduling under the Shared Completed Tables engine in both Linux Systems 148
7.28 Lock usage for random graph benchmarks using Local scheduling under the Shared Completed Tables engine 149
7.29 Lock usage for the Random Graph benchmarks using Batched scheduling under the Concurrent Completion engine 149
7.30 Times for the Left Recursion program on the Dual Core system using Local scheduling under the Shared Completed Tables engine 150
7.31 Times for the Left Recursion program on the Dual Core system using Batched scheduling under the Concurrent Completion engine 150
7.32 Times for the Right Recursion program on the Dual Core system using Local scheduling under the Shared Completed Tables engine 151
7.33 Times for the Right Recursion program on the Dual Core system using Batched scheduling under the Concurrent Completion engine 152

7.34 Times for the Left Recursion program on the Multi-processor system using Local scheduling under the Shared Completed Tables engine 152
7.35 Times for the Left Recursion program on the Multi-processor system using Batched scheduling under the Concurrent Completion engine 153

1

Introduction

In this dissertation we pursue the goal of enhancing the capabilities of existing logic programming systems to enable cooperation in concurrent and distributed environments. Such capabilities are essential in today's computing world, where large bandwidth networking is available for low prices. As evidence of this need for more advanced computing models, topics like grid computing, collaborative agents and distributed workflows have evolved as research fields themselves.

While in 1999, when this work started, those topics were not as pressing as today, the need for supporting distributed computing was already a reality. Our initial goal was to provide a programming system that would allow to apply the power of logic programming to the growing need for distributed applications.

While the ultimate goal was to provide a distributed logic programming system, we realized that the need for concurrent programming within a logic programming system was a fundamental step in the right direction. Thus the integration of multi-threaded programming in a logic programming system was an important goal.

By the time we started, several Prolog systems offered multi-threaded programming interfaces, but Prolog itself was already being "outclassed", or, to put it in a more correct way, complemented by a new logic programming execution model: tabling.

Prolog's procedural interpretation allows the programmer full control over the execution, while also allowing the logic interpretation of programs. However in the second case, as all Prolog programmers know, not all programs are correctly evaluated by Prolog systems.

Take the program in Figure 1.1, where `left_ancestor/2` declares the ancestor relation using left recursion and `right_ancestor/2` declares the ancestor relation using right recursion. The declarative meaning of ancestor is well defined, and clearly the

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

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

Figure 1.1: Ancestor using right and left recursion

```
fib(0,0).
fib(1,1).
fib(N,Res) :-
    N > 1,
    N1 is N - 1, N2 is N - 2,
    fib(N1,Res1),
    fib(N2,Res2),
    Res is Res1 + Res2.
```

Figure 1.2: Fibonacci in Prolog

same, whether it is defined through right recursion or left recursion. Yet the former will be correctly computed by a traditional Prolog system, while the other will loop. With tabling, both definitions are correctly computed.

On the other hand some programs may be correctly interpreted but the efficiency of the execution may suffer. Consider the program Fibonacci program in Figure 1.2.

This program's execution complexity is exponential in N. Of course, the program can be written in a more efficient manner, given Prolog's procedural semantics, but the clarity of the solution would be compromised.

Tabling avoids this problem, being able to evaluate the program with an acceptable complexity. While Prolog would loop for many programs, tabling guarantees termination and correct interpretation for all datalog programs. When the programs include negation, tabling allows evaluation with respect to the well founded semantics [83].

Tabling has been completely integrated in some Prolog systems, allowing the same program to combine the best of both worlds – integrating a declarative semantics with full procedural control. However not all the features of the procedural execution model have been integrated with tabling, namely multi-threading.

We presented an early proposal to integrate multi-threading in XSB [47], the state of the art tabling system at the time, and still a leading tabling system at the moment of this writing. That proposal included extending XSB Prolog to support multi-threading and integrating tabling in a way that tables could be shared and completed concurrently by a set of threads. The sharing of tables would allow for one of the nice features of tabling, the re-use of previous computations, to be accomplished, and to rationalize the use of existing memory on the system.

Supporting the multi-threading of Prolog reliably in XSB proved to be a major im-

plementation project. The reliance of XSB's implementation on global variables was a major headache - we had to consider a couple of hundreds of global variables that would make the code non-re-entrant, and had to deal with them in the appropriate way, either making them private to the thread, protecting them with locks, and even in some cases just making them regular local variables. Many of these problems stem from the fact that XSB supports tabling with well founded negation, asserted tries, subsumptive tabling and more. Meanwhile, the sequential XSB system had to be kept intact as it is used by thousands of people. The complexity of this process led us to reconsider our goals. The sharing of tables among threads would still be the foremost of the work, instead of addressing the distribution issues.

After having designed and implemented the support of multi-threading for Prolog, we proceeded to support tabling without allowing threads to share tables. Support was given to efficient access to private tables and private dynamic predicates. Interaction among threads in manipulating this private data structures was thoroughly eliminated, ensuring the scalability of the system for many threads executing in a multiprocessor. An API was designed to match the still evolving ISO standard proposal. This implementation is described in Chapter 3.

As the sharing of tables is concerned we intended to implement the completion algorithm described in [47]. However, in the early stages of implementation it was found that the algorithm was incorrect. While trying to figure out a correct completion algorithm, the idea of only sharing completed tables arose. Of course, as only completed tables would be shared, if a thread needed an answer from a table owned by another thread and vice-versa a deadlock would occur. However in the centralized context of a multi-threaded engine, detection of such a deadlock is straightforward. There remained the problem of how to break the deadlock. It turns out, that under local scheduling, the default scheduling of XSB for some years now, it is possible to reset a tabled computation and to restart the subgoal as if nothing happened (of course modulo Prolog side effects – see Chapter 2 for details). Thus the Shared Completed Tables were born, which are presented in Chapter 4.

We finally proceeded with the development of the concurrent completion algorithm. It has been reported several times that detecting the completion of tables is the most difficult problem in the implementation of tabling, and this is specially true in a concurrent environment. We managed to get an algorithm that allowed to concurrently complete tables owned by different threads, while allowing for inter-dependent subgoals to be evaluated concurrently. We implemented the algorithm under batched scheduling, as we already had support for shared tables under local scheduling. The algorithm and its implementation are not used by default in XSB (the default is local scheduling with Shared Completed Tables) and the implementation is not yet robust enough for widespread use, but we are confident that it is fundamentally correct and that it will at some point allow for parallelism to be exploited with tabling. This is

reported in Chapter 5.

Although during the process of the dissertation new implementations of tabling appeared and multi-threaded support become widespread in Prolog, no effort, as far as we know, has been made to support a clean semantics for the integration of the tabling logic programming model with multiple threads of execution.

1.1 The Demand for Multi-Threaded Tabling

In the last few years many applications that use tabling have been developed. A whole company, *XSB Inc.*[1], has was launched that develops applications in XSB. Other companies, like Ontology Works (`www.ontologyworks.com`), Medical Decision Logics, Inc (`mdlogix.com`) and BBN Inc have products that depend on XSB, as do many other companies. Heavy weight applications have been developed. CDF [77] is an ontology management system that has been used to develop commercial applications by XSB Inc. Flora-2 is an object oriented knowledge base language developed by Michael Kifer and is colleagues that supports the language of F-logic [41], Hilog [12] and Transaction Logic [6]. It makes extensive use of negation. XMC [55,56] is a model checking tool for XSB that has been heavily used for a variety of model checking problems, such as [24] and [54]. For many of such applications the need to interact with users through web pages conflicts with the restriction to sequential execution imposed by XSB. There is an interface for XSB with Java, InterProlog [8], but if the XSB process is restricted to sequential execution it can't respond concurrently to several Java threads. Needless to say, the use of heavyweight processes to solve this problem is a performance killer. As a result, version 3.0 of InterProlog (in preparation) is based on the multi-threaded engine developed in this thesis. Though many of the main queries that must be supported by these applications do not need to be amortized, and as such, adequately supported by private tables, some of the sub-queries are repeated ad-infinitum and as such are excellent candidates to share tables. One example are inheritance queries for the object oriented languages (e.g. CDF and Flora-2) that would benefit from the implicit caching offered by shared tables.

Shared tables are also needed when a large number threads, running at the same time, need to compute the same large tables. In that case, the use of private tables, while semantically correct, would multiply the memory requirements for table storage by the number of threads running. In such cases, memory could easily become exhausted, which is unacceptable.

With the emergence of the multi-core processors, parallel execution has become a mainstream feature, and expensive multi-processor systems are no longer the only option to support parallel execution. Dual core processors are today's standard for personal computers, while processors with more cores are already on the market, and the

[1] `www.xsb.com`

number of cores per processor on a standard personal computer is expected to grow gradually over the next years (according to Moore's Law, it should double every two years). This makes multi-threading of tabling more appealing, and even of imperative use, as the use of the explicit parallelism offered by multi-threaded programming model is a sure way to increase the performance of the tabling applications. We also think of the support for explicit parallelism with sharing of tables and concurrent completion of tables as a starting point to support implicit table parallelism, as defined in [33], where each thread computes a set of tables in a tabled program, and all threads contribute in parallel to the computation of the full program. This would increase the performance of tabling applications that don't require explicit concurrent actions without explicit intervention from the programmer.

1.2 Structure of the Dissertation

Chapter Two introduces the main concepts for the semantics and implementation of tabling. Section 2.1 introduces the concept of tabled evaluations. Section 2.2 presents a formalization of tabled evaluations and introduces the batched and local scheduling strategies, and discusses some properties of local evaluation. The formalization of local scheduling at Section 2.2.3 is original, while the general formalization in Section 2.2.1 follows previous work. Section 2.3 presents the fundamental of the SLG-WAM that are required to understand further chapters.

Chapter Three describes the design and implementation of the multi-threaded engine with thread private tables. Section 3.1 recalls multi-threaded programming and its mapping into the Prolog language. Section 3.2 discusses the design considerations for the multi-threaded XSB execution and programming model. Section 3.3 describes the actual implementation.

Chapter Four describes Shared Completed Tables. Section 4.1 presents the changes to the SLG-WAM to implement Shared Completed Tables. Section 4.2 presents a formal semantics for the concurrent execution of tabling through Shared Completed Tables and proofs its correctness w.r.t. to SLG. Section 4.3 gives an example of concurrent execution with deadlocks occurring and being resolved. Finally section 4.4 argues for the properties of the implementation vs the formal semantics.

Chapter Five describes the Concurrent Completion implementation. Section 5.1 describes the changes to the SLG-WAM to support Concurrent Completion. Section 5.2 presents an example of execution of Concurrent Completion. Section 5.3 informally proofs the completeness and liveness of the Concurrent Completion algorithm and also covers some loose ends.

Chapter Six discusses related work. Section 6.1 presents some multi-threaded Prolog systems. Section 6.2 discusses proposals and systems that allow the parallel computation of tables. Section 6.3 discusses experiments to support the distributed execution of tabling.

Chapter Seven presents the performance results. Section 7.1 considers the overhead of multi-threading vs. the sequential engine. Section 7.2 presents scalability results for private tables. Section 7.3 presents the results of some experiments involving shared tables.

Chapter Eight presents some conclusions and future work.

2

Background

In this chapter we introduce the main concepts to be developed throughout the dissertation. In Section 2.1 we introduce tabling informally. In Section 2.2 we formalize tabling by presenting an operational semantics for it, including well-founded negation. We formalize local scheduling and state some of its properties. In Section 2.3 we present the main concepts of the implementation of tabling for definite programs, namely the SLG-WAM's data structures, its instruction set and the implementation of Batched and Local scheduling.

2.1 Introduction to Tabling

It's well known that the evaluation of logic programs under Prolog may not reflect its declarative semantics. For example consider program $P_{2.1}$ in Figure 2.1 and the query `:- p(s,X).`

A SLD evaluation [43], as done by a standard Prolog system is shown in Figure 2.2.

```
:- table p/2.

p(X,Z) :- p(X,Y),p(Y,Z).
p(X,Z) :- e(X,Z), q(Z).

e(s,b).  e(s,d).  e(b,c).  e(b,f)

q(b).  q(d).  q(c).
```

Figure 2.1: Program $P_{2.1}$

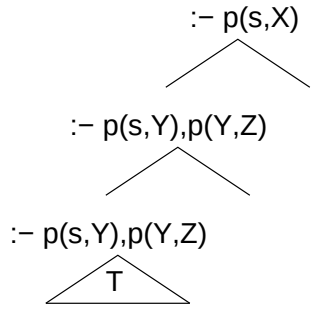

Figure 2.2: SLD evaluation of program $P_{2.1}$

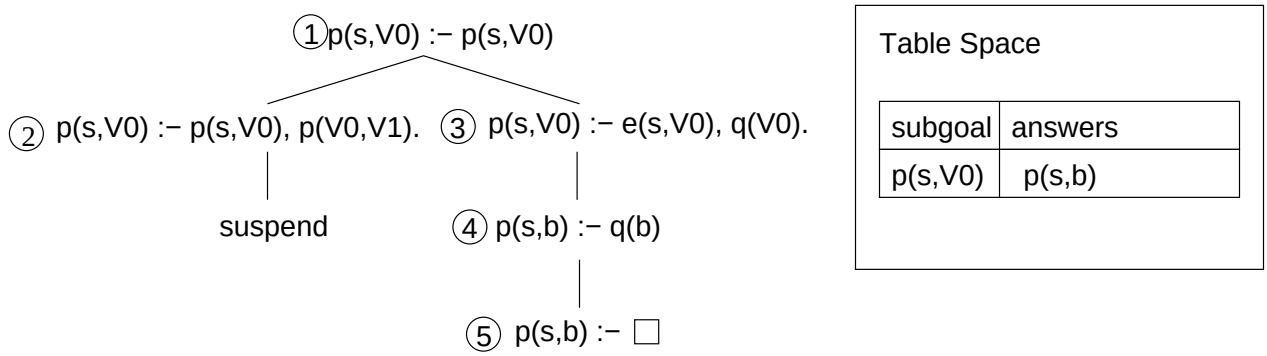

Figure 2.3: Tabled evaluation of program $P_{2.1}$: deriving `p(s,b)`

Tree T is obviously infinite and the Prolog system will never terminate evaluating the query. However `p(s,b)` is clearly an answer to that query under the program's declarative semantics. Indeed there is a non-infinite path in the SLD tree that would allow the derivation of that answer, but Prolog's fixed textual order of evaluation of clauses prevents that. And as there are infinite paths on the tree, SLD resolution itself cannot be used to find all the answers to the query.

Tabling considers a table where answers are stored as they are derived by the evaluation mechanism. When a subgoal that is not present on table is called, the NEW SUBGOAL operation is performed, and a new entry on the table is created for that subgoal together with the corresponding tree to evaluate that subgoal. Clauses are resolved against tree nodes by the PROGRAM CLAUSE RESOLUTION operation like in SLD. When a variant of a tabled subgoal that is present on the table is called and there are no answers for that subgoal, the computation suspends and the next alternative on the search tree is explored. When an answer is found to the tabled subgoal the operation NEW ANSWER is performed. After deriving the answer `p(s,b)` (5) the query is satisfied as shown on Figure 2.3.

If more answers were required, the suspended branch in the search tree would be resumed. It would be found that there is one new answer for the suspended call to `p(s,V0)` (2) and execution would proceed by returning answer `p(s,b)`, performing the POSITIVE RETURN operation. This would lead to the unification of $V0$ and `b`, and the

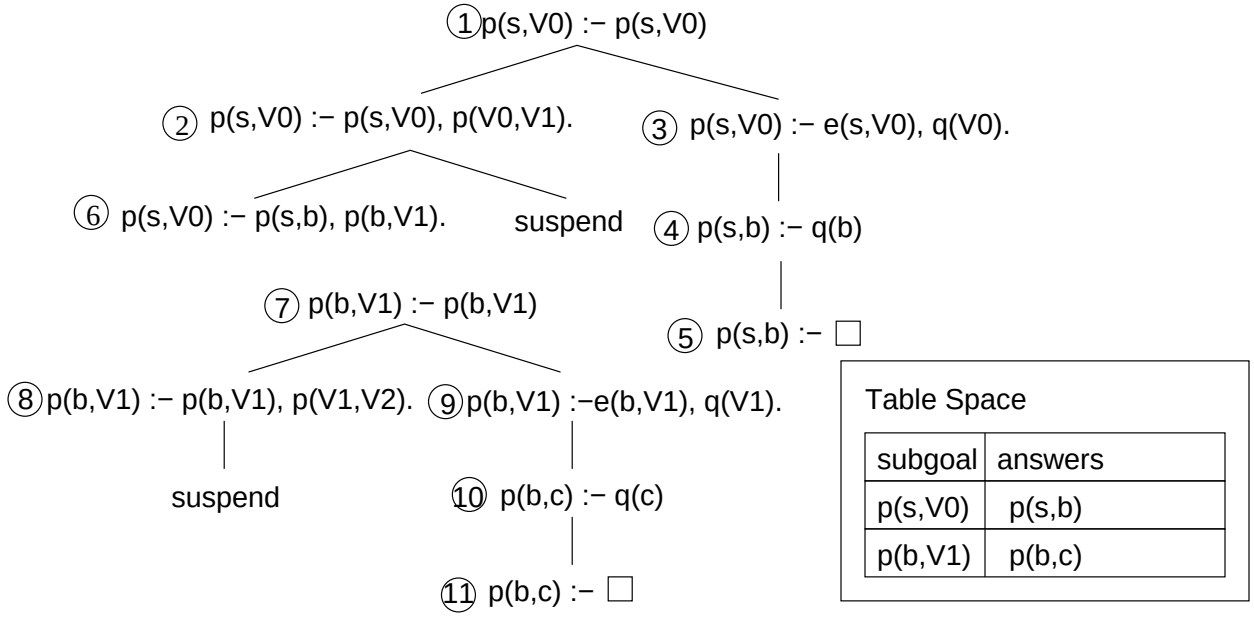

Figure 2.4: Tabled evaluation of program $P_{2.1}$: deriving `p(b,c)`

second call in the clause would become a call to subgoal p(b,V1). As subgoal p(b,V1) is not in the table, a new derivation tree for it would be created, together with an entry on the table by the NEW SUBGOAL operation (7). The answer p(b,c) would be derived in the same manner as p(s,b) above. This leads us to the state represented in Figure 2.4

The answer `p(b,c)` is returned by the POSITIVE RETURN operation to the call `p(b,V1)` and the call to `p(c,V2)` is made, performing the NEW SUBGOAL operation. A tree to compute `p(c,V2)` is created. After expanding all the sub-trees for subgoal `p(c,V2)` the call to `e(c,V2)` fails and the situation shown in Figure 2.5 is reached.

At this point no answers can be returned to any suspended call to `p(c,V1)`. As no more operations can be performed on any node of the tree for subgoal `p(c,V1)` it can be completed by the COMPLETION operation. Further operations would follow a similar course, until all answers are derived and all subgoals completed.

2.2 SLG Resolution

2.2.1 An Operational Semantics for Tabling

Several semantics have been proposed for tabled evaluation of logic programs such as OLDT [78] and SLG [13] which considers programs with well-founded negation [83]. In this section we follow the reformulated SLG proposed in [74].

Terminology and assumptions We assume the standard terminology of logic programming (see, e.g. [43]) and an understanding of the well-founded semantics (see [83]). We assume that any program is defined over a countable language $\mathcal{L}$ of pred-

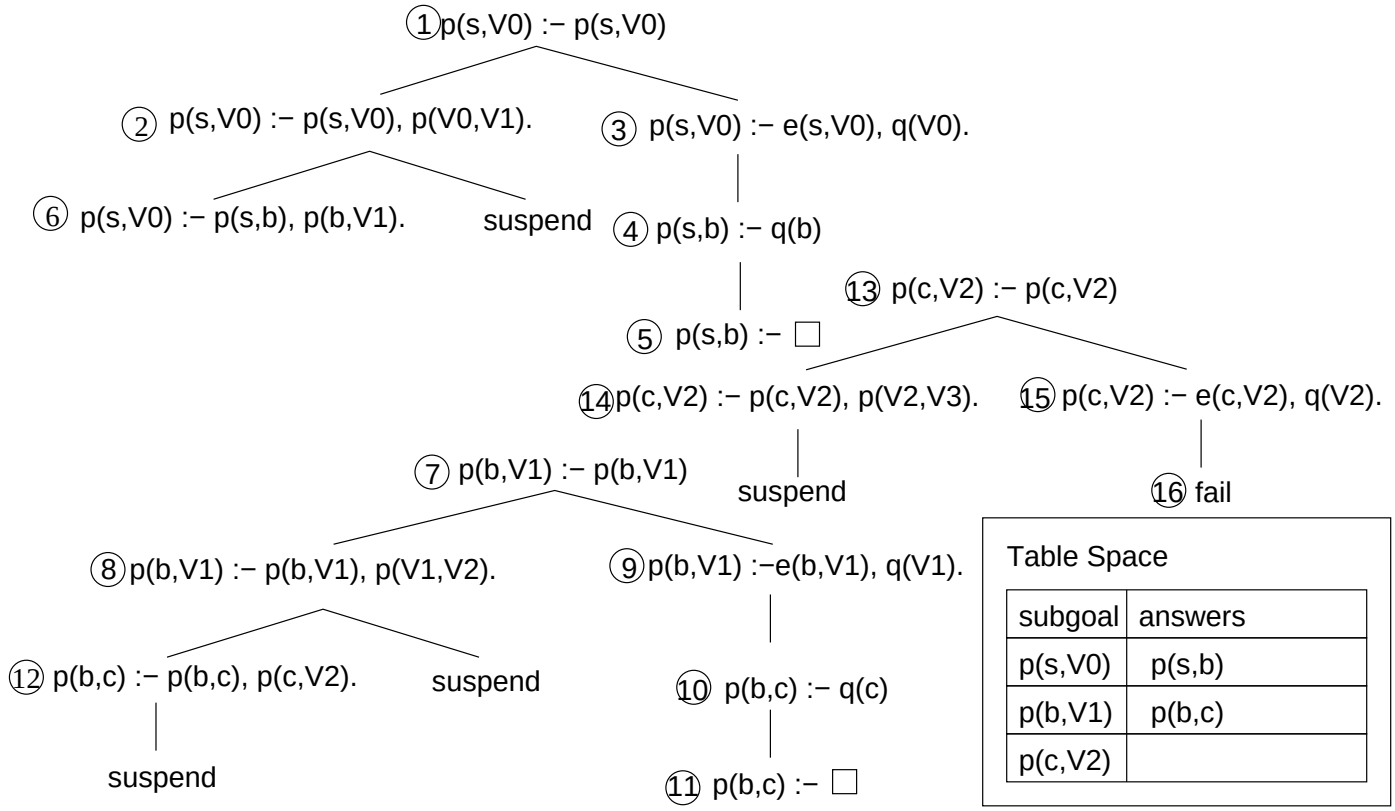

Figure 2.5: Tabled evaluation of program $P_{2.1}$: `p(c,Z')` completes

icates and function symbols. If L is a literal, then $vars(L)$ denotes the set of variables in L. The *Herbrand base* H_P of a program P is the set of all ground atoms formed from $\mathcal{L}$. By a *3-valued interpretation* I of a program P we mean a set of literals defined over H_P. For $A \in H_P$, if $A \in I$, A is true in I, and if *not* $A \in I$, A is false in I. When I is an interpretation and A is an atom, $I|_A$ refers to

$$\{L \mid L \in I \text{ and } (L = G \text{ or } L = \textit{not } G) \text{ and } G \text{ is in the ground instantiation of } A\}$$

The Well-Founded Model of a program P is denoted as $WFM(P)$. In the following sections, we use the terms *goal*, *subgoal*, and *atom* interchangeably. Variant terms are considered to be identical. *SLG* evaluations allow arbitrary, but fixed literal selection strategies. For simplicity, throughout this paper we assume that literals are selected in a left-to-right order.

Definition 2.2.1 (*SLG* Trees and Forest) *An SLG forest consists of a forest of SLG trees. Nodes of SLG trees have the forms:*

$$\textit{Answer\_Template :- Delay\_Set}|\textit{Goal\_List}$$

or

$$fail$$

In the first form, the Answer_Template is an atom, the Delay_Set is a set of delayed literals (see Definition 2.2.2) and Goal_List is a sequence of literals. The second form is called a failure node. *The root node of an SLG tree may be marked with the token* complete.

We call a node N an answer *when it is a leaf node for which* Goal_List *is empty. If the Delay_Set of an answer is empty it is termed an* unconditional answer, *otherwise, it is a* conditional answer.

Definition 2.2.2 specifies the exact formulation of delay literals. Definition 2.2.8 will ensure that the root node of a given SLG tree, T, has the form S :- $|S$, where S is a subgoal. If T is an SLG tree in a forest $\mathcal{F}$ whose root node is S :- $|S$ (possibly marked as complete), then we use the following terminology. S is the root node for T or that T is the tree for S, and S is in $\mathcal{F}$.

Definition 2.2.2 (Delay Literals) *A negative delay literal in the* Delay_Set *of a node N has the form not A, where A is an ground atom. Positive delay literals have the form* A^{Call}_{Answer}, *where A is an atom whose truth value depends on the truth value of some answer Answer for the subgoal Call. If* θ *is a substitution, then* $(A^{Call}_{Answer})\theta = (A\theta)^{Call}_{Answer}$.

Positive delay literals contain information so that they may be simplified when a particular answer to a particular call becomes unconditionally true or false. It is useful to define answer resolution so that it takes into account the form of delay literals.

Definition 2.2.3 (Answer Resolution) *Let N be a node* A :- $D|L_1, ..., L_n$, *where* $n > 0$. *Let* $Ans = A'$:- $D'|$ *be an answer whose variables have been standardized apart from N. N is* SLG resolvable *with Ans if* $\exists i$, $1 \leq i \leq n$, *such that* L_i *and* A' *are unifiable with an mgu* θ. *The SLG resolvent of N and Ans on* L_i *has the form*

$$(A \text{ :- } D|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

if D' *is empty, and*

$$(A \text{ :- } D, \overline{D}|L_1, ..., L_{i-1}, L_{i+1}, ..., L_n)\theta$$

otherwise, where $\overline{D} = L_i$ *if* L_i *is negative, and* $\overline{D} = L_{i\,A'}^{L_i}$ *otherwise.*

A set of subgoals is completely evaluated when it can produce no more answers. Formally,

Definition 2.2.4 (Completely Evaluated) *A set* $\mathcal{S}$ *of subgoals in a forest* $\mathcal{F}$ *is* completely evaluated *if at least one of the conditions holds for each* $S \in \mathcal{S}$

1. *The tree for S contains an answer S :- |; or*

2. *For each node N in the tree for S:*

   (a) *The selected literal* L_S *of N is completed or in* $\mathcal{S}$*; or*

(b) *There are no applicable* NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN, DELAYING, *or* NEGATIVE RETURN *operations (Definition 2.2.8) for N.*

Once a set of subgoals is determined to be completely evaluated, the COMPLETION operation marks the root node of the trees for each subgoal (Definition 2.2.1).

Definition 2.2.5 (Supported Answer) *Let $\mathcal{F}$ be a SLG forest, S a subgoal in $\mathcal{F}$, and Answer be an atom that occurs in the head of some answer of S. Then* Template *is supported by S in $\mathcal{F}$ if and only if:*

1. *S is not completely evaluated; or*

2. *there exists an answer node Answer :- Delay_Set| of S such that for every positive delay literal D^{Call}_{Ans}, Ans is supported by Call.*

As an aside, we note that unsupported answers appear to be uncommon in practical evaluations which minimize the use of delay such as [66].

An *SLG* evaluation consists of a (possibly transfinite) sequence of SLG forests. In order to define the behavior of an evaluation at a limit ordinal, we define a notion of a least upper bound for a set of SLG trees. If a global ordering on literals is assumed, then the elements in the *Delay_Set* of a node can be uniformly ordered, and under this ordering a node of a tree can be taken as a term to which the usual definition of variance apply. In particular, nodes of SLG trees are treated as identical when they are variant.

A rooted tree can be viewed as a partially ordered set in which each node N is represented as $\{N, P\}$ in which P is a tuple representing the path from N to the root of the tree [28]. When represented in this manner, it is easily seen that when T_1 and T_2 are rooted trees, $T_1 \subseteq T_2$ iff T_1 is a sub-tree of T_2, and furthermore, that if T_1 and T_2 have the same root, their union can be defined as their set union, for T_1 and T_2 taken as sets.

Definition 2.2.6 (Tabled Evaluation) *Given a program P, an atomic query Q and a set of tabling operations (from Definition 2.2.8), a tabled evaluation $\mathcal{E}$ is a sequence of SLG forests $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_\beta$, such that:*

- *$\mathcal{F}_0$ is the forest containing a single tree* Q :- | Q

- *For each successor ordinal, $n+1 \leq \beta$, $\mathcal{F}_{n_1}$ is obtained from $\mathcal{F}_n$ by an application of a tabling operation.*

If no operation is applicable to $\mathcal{F}_\alpha$, $\mathcal{F}_\alpha$ is called a final forest *of $\mathcal{E}$. If $\mathcal{F}_\beta$ contains a leaf node with a non-ground selected negative literal, it is* floundered.

SLG forests are related to interpretations in the following manner.

Definition 2.2.7 *Let $\mathcal{F}$ be a forest. Then the* interpretation induced by $\mathcal{F}$, $I_{\mathcal{F}}$ *is the smallest set such that:*

- *A (ground) literal $A \in I_{\mathcal{F}}$ iff A is in the ground instantiation of some unconditional answer Ans :- \| in $\mathcal{F}$.*
- *A (ground) literal not $A \in I_{\mathcal{F}}$ iff A is in the ground instantiation of a completely evaluated subgoal in $\mathcal{F}$, and A is not in the ground instantiation of any answer in $\mathcal{F}$.*

An atom S is successful in $\mathcal{F}$ *if the tree for S has an unconditional answer S. S is* failed in $\mathcal{F}$ *if S is completely evaluated in $\mathcal{F}$ and the tree for S contains no answers. An atom S is* successful (failed) *in $I_{\mathcal{F}}$ if S' (not S') is in $I_{\mathcal{F}}$ for every S' in the ground instantiation of S. A negative delay literal not D is successful (failed) in a forest $\mathcal{F}$ forest if D is (failed) successful in $\mathcal{F}$. Similarly, a positive delay literal D^{Call}_{Ans} is successful (failed) in a $\mathcal{F}$ if Call has an unconditional answer Ans :- \| in $\mathcal{F}$.*

These operations are as follows.

Definition 2.2.8 (*SLG* Operations) *Given a forest $\mathcal{F}_n$ of a SLG evaluation of program P and query Q, where n is a non-limit ordinal, $\mathcal{F}_{n+1}$ may be produced by one of the following operations.*

1. NEW SUBGOAL*: Let $\mathcal{F}_n$ contain a non-root node*

$$N = Ans \text{ :- } DelaySet|G, Goal\_List$$

   where G is the selected literal S or not S. Assume $\mathcal{F}_n$ contain no tree with root subgoal S. Then add the tree S :- \|S to $\mathcal{F}_n$.

2. PROGRAM CLAUSE RESOLUTION*: Let $\mathcal{F}_n$ contain a root node $N = S$:- \|S and C be a program clause Head :- Body such that Head unifies with S with mgu θ. Assume that in $\mathcal{F}_n$, N does not have a child $N_{child} = (S \text{ :- } |Body)\theta$. Then add N_{child} as a child of N.*

3. POSITIVE RETURN*: Let $\mathcal{F}_n$ contain a non-root node N whose selected literal S is positive. Let Ans be an answer node for S in $\mathcal{F}_n$ and N_{child} be the SLG resolvent of N and Ans on S. Assume that in $\mathcal{F}_n$, N does not have a child N_{child}. Then add N_{child} as a child of N.*

4. NEGATIVE RETURN*: Let $\mathcal{F}_n$ contain a leaf node*

$$N = Ans \text{ :- } DelaySet|not\ S, Goal\_List.$$

   whose selected literal not S is ground.

   (a) NEGATION SUCCESS*: If S is failed in $\mathcal{F}$, then create a child for N of the form: Ans :- DelaySet\|Goal_List.*

   (b) NEGATION FAILURE*: If S succeeds in $\mathcal{F}$, then create a child for N of the form* fail.

```
:- table p/1, q/1, r/1.

p(X) :- not q(X), r(X).

q(X) :- not p(X).
q(a).

r(a).
```

Figure 2.6: Program $P_{2.2}$

5. DELAYING: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|not\ S, Goal\_List$, such that S is ground, in $\mathcal{F}_n$, but S is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form Ans :- $DelaySet, not\ S|Goal\_List$.*

6. SIMPLIFICATION: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|$, and let $L \in DelaySet$*

   (a) *If L is failed in $\mathcal{F}$ then create a child* fail *for N.*

   (b) *If L is successful in $\mathcal{F}$, then create a child Ans :- $DelaySet'|$ for N, where $Delay\_Set' = Delay\_Set - L$.*

7. COMPLETION: *Given a completely evaluated set $\mathcal{S}$ of subgoals (Definition 2.2.4), mark the trees for all subgoals in S as completed.*

8. ANSWER COMPLETION: *Given a set of unsupported answers $\mathcal{UA}$, create a failure node as a child for each answer $Ans \in \mathcal{UA}$.*

An interpretation induced by a forest (Definition 2.2.7) has its counterpart for SLG, (Definition 5.2 of [13]). Using these concepts, we can relate SLG to *SLG* evaluations.

Theorem 2.2.1 *Let $\mathcal{F}$ a forest in a terminated SLG evaluation of a query Q to a program P, and A an atom such that A :- A is the root of some tree in $\mathcal{F}$. Then*

$$WFM(P)|_A = I_{\mathcal{F}}|_A$$

For the proof see [74]. This theorem implies the correctness of SLG, and as such, proving correctness w.r.t. to SLG will ensure the correctness w.r.t. to the well founded model.

2.2.2 Example of SLG Resolution

In this section we give an example of SLG resolution as defined in the previous section, stressing the features of SLG that support well-founded negation.

Consider program $P_{2.2}$ in Figure 2.6 and the query `:- p(a)`. The initial forest for the SLG evaluation consists of the tree with the node $p(a)$:- $p(a)$. After the application

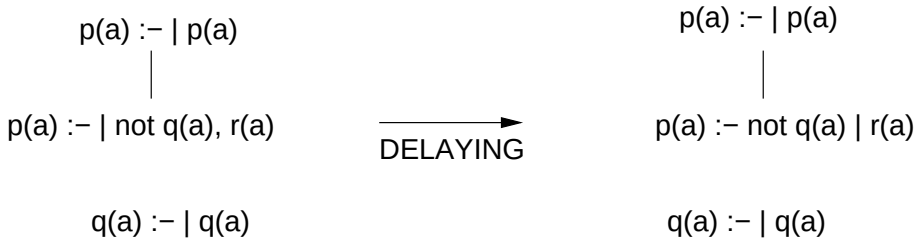

Figure 2.7: SLG evaluation of program $P_{2.2}$: the delay operation

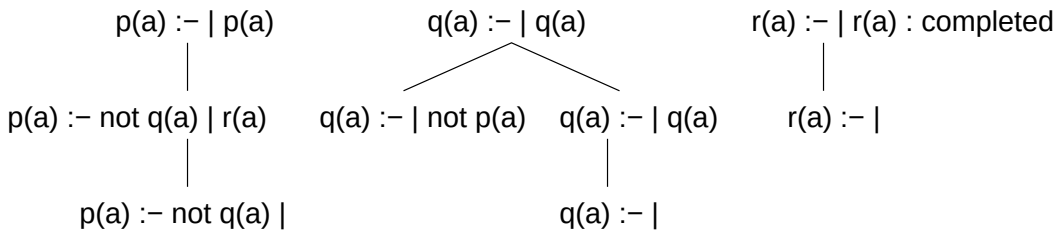

Figure 2.8: SLG evaluation of program $P_{2.2}$: a conditional answer

of PROGRAM CLAUSE RESOLUTION and NEW SUBGOAL for `q(a)` the forest at the left side Figure 2.7 is obtained.

At this point the DELAYING operation is applied giving the forest at the right of the figure. After applying a set of NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN and COMPLETION operations we get the forest represented in Figure 2.8.

At this point we have an unconditional answer for `p(a)`. In this case it is possible to apply the operation SIMPLIFICATION to literal `not q(a)` because there is an answer `q(a)`. `p(a)` fails and NEGATION SUCCESS can be applied to the literal `not p`. No more operations can be executed for the trees of `p(a)` and `q(a)`, so they are marked completed by the COMPLETION operation and the final forest represented in Figure 2.9 is reached.

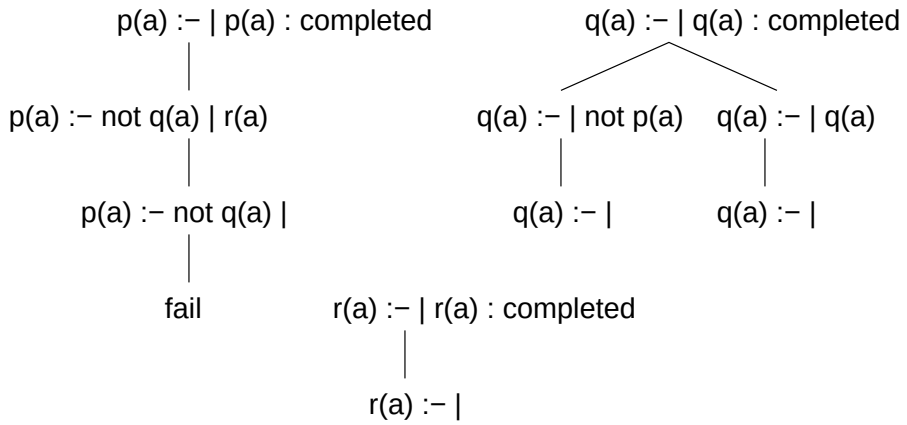

Figure 2.9: SLG evaluation of program $P_{2.2}$: final forest

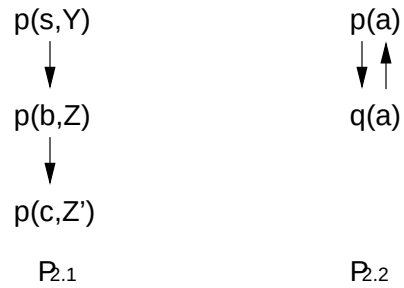

Figure 2.10: SDG for forests in Figures 2.5 and 2.8.

2.2.3 Local SLG Evaluations

Definition 2.2.9 (Subgoal Dependency Graph) *Let $\mathcal{F}$ be a forest. We say that a tabled subgoal $subgoal_1$* directly depends on *a tabled subgoal $subgoal_2$ in $\mathcal{F}$ iff $subgoal_1$ and $subgoal_2$ are* non-completed *and $subgoal_2$ or $not(subgoal_2)$ is the selected literal of some node in the tree for $subgoal_1$ in $\mathcal{F}$. If $subgoal_2$ ($not(subgoal_2)$) is the selected literal of some node in the tree for $subgoal_1$ in $\mathcal{F}$, then we say that $subgoal_1$ positively (negatively) directly depends on $subgoal_2$.*

The relation "depends on" *is the transitive closure of the relation* "directly depends on". *That is, $subgoal_1$* depends on *$subgoal_k$ iff there exists a sequence, $subgoal_1$, $subgoal_2$, ..., $subgoal_k$, such that $\forall\ i,\ k-1 \geq i \geq 1$, $subgoal_i$ directly depends on $subgoal_{i+1}$.*

The Subgoal Dependency Graph SDG($\mathcal{F}$) = (V,E) *of $\mathcal{F}$ is a directed graph in which V is the set of root goals for trees in $\mathcal{F}$ and $(s_i, s_j) \in E$ iff subgoal s_i directly depends on subgoal s_j.*

An SDG can be partitioned in disjoint sets of mutually dependent subgoals, or SCCs (strongly connected components).

Definition 2.2.10 (Independent SCC) *A strongly connected component SCC is independent if $\forall S \in SCC$, if S depends on some S', then $S' \in SCC$.*

In Figure 2.10 we show the SDGs for the forest represented in Figures 2.5 and 2.8. We consider the forest in Figure 2.5 as an informal SLG forest for definite programs. That SDG contains three SCCs, one for each tabled subgoal. Only the SCC for `p(c,Z')` is independent. The SDG for the forest of $P_{2.2}$ has only one independent SCC containing both subgoals.

We define local scheduling as a restriction to a SLG evaluation so that it satisfies the *locality property*.

Definition 2.2.11 (Locality) *A SLG evaluation satisfies the locality property if all SLG operations (definition 2.2.8) are applied to trees whose root subgoal is in an independent SCC.*

Furthermore operation COMPLETION *can only be applied to all the subgoals in the independent SCC.*

We say that $\mathcal{E}$ is a local SLG evaluation if $\mathcal{E}$ satisfies the locality property.

We now formulate the following theorem:

Theorem 2.2.2 *If an SLG evaluation $\mathcal{E} = \mathcal{F}_0 \ldots \mathcal{F}_n$ satisfies the locality property than $\forall 0 \leq k \leq n : \mathrm{SDG}(\mathcal{F}_k)$ has one and only one independent SCC.*

Proof: Let $\mathcal{E} = \mathcal{F}_0 \ldots \mathcal{F}_n$ be a local SLG evaluation. We prove by induction that $\forall 0 \leq k \leq n : SDG(\mathcal{F}_k)$ has one and only one independent SCC.

Induction base: $\mathcal{F}_0 = \{Q : -|Q\}$ has only the tree for subgoal Q. $SDG(\mathcal{F}_0) = (Q, \emptyset)$ which is itself an independent SCC as Q doesn't depend on any subgoal.

Induction hypothesis: $\forall 0 \leq i \leq k : SDG(\mathcal{F}_i)$ has one and only one independent SCC.

Induction thesis: $\forall 0 \leq i \leq k+1 : SDG(\mathcal{F}_i)$ has one and only one independent SCC.

Given the induction hypothesis we only need to prove that $SDG(\mathcal{F}_{k+1})$ has one and only one independent SCC to prove the induction thesis.

From $\mathcal{F}_k$ for which $SDG(\mathcal{F}_k)$ has only one independent SCC (induction hypothesis) we obtain $\mathcal{F}_{k+1}$ by applying one of the SLG operations of definition 2.2.8 to one of the trees whose root is in the independent SCC of $SDG(\mathcal{F}_k)$ (local SLG evaluation). This may be:

1. NEW SUBGOAL: This creates a new tree in $\mathcal{F}_{k+1}$ which means a new node in $SDG(\mathcal{F}_{k+1})$. This node has no outgoing edges (the tree has only its root) but will have at least one incoming edge (from the selected literal which originated the new subgoal.) So a new independent SCC with only one node (the new subgoal) is created. There was only one independent SCC in $SDG(\mathcal{F}_k)$ (induction hypothesis) and this will depend on the new node and will not be independent in $SDG(\mathcal{F}_{k+1})$. So $SDG(\mathcal{F}_{k+1})$ will have only one independent SCC.

2. PROGRAM CLAUSE RESOLUTION: This either leaves the $SDG(\mathcal{F}_k)$ unchanged in which case $SDG(\mathcal{F}_{k+1}) = SDG(\mathcal{F}_k)$ (in the case that there was another clause with the same subgoal) so it has only one independent SCC by the induction hypothesis, or adds a new edge to it departing from one of the nodes of its only independent SCC to create $SDG(\mathcal{F}_{k+1})$. There are two cases:

   - The destination node belongs to the independent SCC in which case it will not be changed.
   - The destination node belongs to a not independent SCC. In this case the not independent SCC will depend on the independent SCC (as this is the only one on the graph, all other SCCs will depend on it) and there will be a merge of SCCs forming one and only one new independent SCC.

In both cases $SDG(\mathcal{F}_{k+1})$ still has one independent SCC.

3. POSITIVE RETURN: In this case a new leaf node with (possibly) a new selected literal is added to a tree of $\mathcal{F}_k$. If there's no new selected literal $SDG(\mathcal{F}_k) = SDG(\mathcal{F}_{k+1})$. Otherwise, like in the above case a new edge may be created in $SDG(\mathcal{F}_k)$ departing from the independent SCC. The prove that $SDG(\mathcal{F}_{k+1})$ has only one independent SCC is the same.

4. NEGATIVE RETURN

   (a) NEGATION SUCCESS: This case is identical to the previous one. A new node is created with (possibly) a new selected literal. The proof is identical.

   (b) NEGATION FAILURE: In this case a new fail node is added to a tree of $\mathcal{F}_k$. $SDG(\mathcal{F}_{k+1}) = SDG(\mathcal{F}_k)$.

5. DELAYING: This case is identical to POSITIVE RETURN and NEGATION SUCCESS with respect to the SDG as a new node with (possibly) a new selected literal is created.

6. SIMPLIFICATION: In this case the new node added to the tree has the same selected literal (if any). $SDG(\mathcal{F}_{k+1}) = SDG(\mathcal{F}_k)$.

7. COMPLETION: Let $\mathcal{F}_i, i < k$ be the forest in $\mathcal{E}$ before the SLG operation that added the first node in the independent SCC of $SDG(\mathcal{F}_k)$. All posterior SLG operations applied on $\mathcal{F}_j, j > i$ either (local SLG evaluation):

   - (NEW SUBGOAL) added a new node to $SDG(\mathcal{F}_j)$. As this will be the only new independent SCC (induction hypothesis), the independent SCC of $SDG(\mathcal{F}_{i+1})$ will depend on it. This new node will be later either be merged into the independent SCC of $SDG(\mathcal{F}_k)$ or removed by a COMPLETION operation.
   - (COMPLETION) removed the only independent SCC of $SDG(\mathcal{F}_j)$ (induction hypothesis) which the independent SCC of $SDG(\mathcal{F}_{i+1})$ depends on.
   - (others) added a new edge to a node of the independent SCC of $SDG(\mathcal{F}_j)$ which the independent SCC of $SDG(\mathcal{F}_{i+1})$ depends on (as its the the only one independent SCC of $SDG(\mathcal{F}_j)$ – induction hypothesis). The independent SCC of $SDG(\mathcal{F}_j)$ will either be merged into the independent SCC of $SDG(\mathcal{F}_k)$ or removed by a COMPLETION operation.
   - (others) left $SDG(\mathcal{F}_j)$ unchanged.

   Removing the independent SCC from $SDG(\mathcal{F}_k)$ will thus give us $SDG(\mathcal{F}_i)$.

   As we are assuming (induction hypothesis) that $\forall i \leq k : SDG(\mathcal{F}_i)$ has only one independent SCC, $SDG(\mathcal{F}_{k+1}) = SDG(\mathcal{F}_i)$ must have only one independent SCC.

8. ANSWER COMPLETION: This will create failure nodes and not affect the SDG. Thus $SDG(\mathcal{F}_{k+1}) = SDG(\mathcal{F}_k)$.

∎

Corollary 2.2.1 *In the SDG for a forest of a local SLG evaluation there is at most one incoming edge for each SCC.*

Proof: By theorem 2.2.2 there can only be one independent SCC. As such the only incoming edge into that SCC has to be added at its creation, as by the locality property (Definition 2.2.11) no operation may be applied to a tree outside the independent SCC once it has been created. ∎

Corollary 2.2.2 *In a SLG evaluation a subgoal may only return answers out of its SCC once its completed.*

Proof:

By theorem 2.2.2 there can only be one independent SCC, I. As such no POSITIVE RETURN or NEGATIVE RETURN operation can be applied to trees outside of I. Its only after I is completed that POSITIVE RETURN or NEGATIVE RETURN operations can be applied to trees of another SCC.

∎

Theorem 2.2.3 (Completeness) *Let P be a finite program and Q an atomic query. Then there exists a finite SLG evaluation of Q against P with final SLG forest $\mathcal{F}$ if and only if there exists a finite local SLG evaluation of Q against P with final state $\mathcal{F}^L$.*

Proof:

If Part: Trivial, as a local SLG evaluation is a SLG evaluation.

Only If Part: In every SDG for an SLG forest, $\mathcal{F}_n$, there is an independent SCC, I. If there is a SLG evaluation $\mathcal{E} = \mathcal{F}_n \ldots \mathcal{F}_{final}$, where $\mathcal{F}_{final}$ is a final forest, according to Definition 2.2.6, then there always at least one SLG operation, from the ones applied in $\mathcal{E}$, that involves the trees in I – if none of the others, the COMPLETION for operation for all the trees in I can be applied. When the COMPLETION operation is applied I is removed from the SDG, and there is a new independent SCC, I', where again at least one SLG operation of $\mathcal{E}$ can be applied to I' in $\mathcal{F}_{n+1}$. This can be repeated until all trees are completely evaluated, i.e. a final SLG forest is reached by a local SLG sub-evaluation $\mathcal{E}_{local} = \mathcal{F}_n \ldots \mathcal{F}_{final}$. By making $\mathcal{F}_n = \mathcal{F}_0$, an initial forest containing one single tree, which has only one independent SCC, we have that for every finite SLG evaluation $\mathcal{E} = \mathcal{F}_0 \ldots \mathcal{F}_{final}$ there is a finite local SLG evaluation $\mathcal{E}_{local} = \mathcal{F}_0 \ldots \mathcal{F}_{final}$.

This theorem ensures that by restricting ourselves to local evaluations of SLG we don't lose any of the power of SLG to evaluate queries.

2.3 The SLG-WAM for Definite Programs

The SLG-WAM is the abstract machine that implements tabling at the abstract machine level. It is an extension of the WAM, with new registers data structures and instructions. It also implies some modifications in proper WAM data structures and instructions. In this section we only cover the part of the SLG-WAM that deals with definite programs (we don't consider negation). For a complete description of the SLG-WAM including negation, consult [64].

Unlike the WAM, the SLG-WAM needs to be able to suspend and resume computations, whenever a tabled subgoal's computation path can't generate more answers and an alternative must be explored. This may lead to answers to be returned to the previous computation path, which later will have to be resumed. This implies the problem of scheduling: to choose which computation path will be resumed next.

There's also the need to incrementally complete tabled subgoals, both for space efficiency and to allow the completed subgoal call optimization: if a completed subgoal is called a more efficient way of traversing the answers using compiled *tries*, can be used.

Since the introduction of the SLG-WAM other methods have been tried to implement tabling. CAT [25] uses copying of the stacks instead of freezing, to suspend and resume computations. It has been implemented in Mercury [70]. CHAT [26, 27] is an hybrid approach between copying and freezing. It has been implemented in XSB but results reported in [11] showed the SLG-WAM to perform better and it is no longer supported. YapTab [59] implements the SLG-WAM without the completion stack and is further discussed on Chapter 6. A very different scheme of the implementation is *linear tabling* which doesn't require the suspension and resuming of computation. Two approaches of linear tabling were used in B-Prolog [88, 89] and ALS [35, 36]. A recent work [69] reproduces and compares these new approaches. A brief explanation of linear tabling is given in Chapter 6.

2.3.1 The WAM

The WAM is the standard abstract machine used to implement many Prolog systems. It supports backtracking and unification at the abstract machine level. We give a brief account of its data structures and instruction set as implemented by XSB. For a full description see [1].

Data Structures The XSB implementation of the WAM uses four stacks:

- The **environment stack** sometimes called the *local stack* keeps track of the procedure activation records for Prolog calls. This include the local variables for a predicate invocation and the return address of the call, very much like in procedural languages implementations.

- The **choice point stack** keeps the choice points created by the execution of predicates with multiple clauses, that require a choice to be made in the derivation tree. It's used to control backtracking and contains the *Failure Continuation* – the next clause to be executed in the case of a failure into that choice point – and the argument registers as well as the abstract machine's registers that must be saved for the procedure to be re-invoked.

- The **heap** is used to store the global variables and data structures (lists and terms). Sometimes this is referred to as the *global stack*.

- The **trail** is used to record conditional bindings that must be undone when backtracking occurs.

The predicate space is a separate space that contains the WAM code for the Prolog predicates. The atom table, sometimes called the heap, is a separate space that contains the values for constants and functors.

The WAM uses the following registers:

- **pcreg** the program counter.

- **cpreg** the return pointer register; used to save the pcreg value to where execution may return after successful execution of a predicate.

- **breg, hreg, ereg, trreg** the top of the stacks

- **hbreg** the frame of the heap that corresponds to the top of the choice point stack. Used to save a segment of the heap for backtracking.

- **ebreg** the same as *hbreg*, but for the environment stack.

- **regs** named in our WAM reference as the **A** registers, these are used to store the arguments for a predicate call.

In many Prolog implementations the Environment and Choice Point stacks are collapsed into a single stack. In such implementations the ebreg is not needed.

Instruction Set The WAM includes the following instruction groups:

- **Unification instructions**: the get, put, uni and bld families of instructions. These allow for the construction, binding and unification of variables and data structures on top of the WAM's data structures. There are specialized instructions for variables, constants, functors and lists. There is support for conditional (trailed) bindings and unconditional (non trailed) bindings. A binding is said to be conditional if it refers to a variable deeper in the stacks than the youngest choice point and has to be undone in backtracking.

- **Predicate call related instructions** In this group we consider the call and instructions. We also consider the instructions for allocation and de-allocation of environments, allocate and deallocate.

- **Choice point instructions** In this group we consider the try, retry and trust families of instructions. These instructions are used to allocate choice points for predicates with multiple clauses. They usually form a chain, in which, during execution, the first try instruction calls a clause and sets the choice point's failure continuation for the next clause's retry instruction and so on.

- **Indexing instructions** The switch family of instructions allows the indexing of a predicate with multiple clauses, in a way that depending on value of the arguments of the call an appropriate clause is selected without having to traverse a try–retry chain. It works very much like a "C" language switch statement vs an if–else if chain. This instructions are clearly important to speedy access to large multi-fact predicates.

We finish by noting that many WAM implementations provide macro-instructions that correspond to the most common sequences of compiler generated WAM instructions. With this optimization they eliminate multiple fetch and decode cycles, saving considerable execution time, and also achieve a more compact WAM code. XSB doesn't do this optimization. For some of the developments that followed the WAM see [84].

2.3.2 SLG-WAM Stacks and Registers

Due to the need to suspend and resume computations, implemented by the freeze registers, the SLG-WAM stacks are not proper stacks, being more properly described as *cactus stacks*. They represent a view of the tree of computation in progress with more than one branch at each time. This is needed because in the SLG-WAM there is a need to suspend and resume computations, unlike in the WAM where the resolution tree is searched in a strictly depth first strategy. The representation of more than just one branch of the tree in the stacks is achieved through the *freeze mechanism* which consists of freezing the contents of the stacks when a computation is suspended. For this, the

SLG-WAM has special purpose registers, the *freeze registers* which record which part of the stacks is frozen.

SLG-WAM Registers Other than all of the WAM registers the SLG-WAM includes the following:

- **hfreg, bfreg, efreg, trfreg** The freeze registers. These registers are used in the freezing mechanism, to preserve the status of suspended computations in the SLG-WAM cactus like stacks (see next section).
- **openreg** The top of the completion stack

The Choice Point Stack The choice point stack is used in the WAM to control backtracking. In the SLG-WAM it also controls the suspension and resuming of subgoals. There are three types of choice points in the SLG-WAM's choice point stack:

- **Internal choice points** These are really Prolog choice points used to control backtracking in program clause resolution. They are equivalent to regular WAM choice points.
- **Generator choice points** When a tabled subgoal is first encountered a *generator choice point* is laid out. This choice point is also used to do resolve program clauses whose answers will be stored on the table. In Figure 2.11 and others, the * marks a field which is not present in the WAM. The first section corresponds to state information that the SLG-WAM must restore on backtracking for any subgoal, tabled or not, and are also present in internal choice points. The middle part doesn't appear in internal choice points. It contains a pointer to the table entry of the subgoal (the *Subgoal Frame*, see Section 2.3.3) and the values of the freeze registers upon creation of the choice point. These are needed to free the frozen areas of the stacks when the subgoal completes. The bottom section contains the argument registers along with its *substitution factor*, the set of free variables which exist in the terms in the argument registers. As explained in Section 2.3.3, the substitution factor is used to reduce copying information into and out of the answer tables.
- **Consumer choice points** These are laid out when the subgoal has been already called and used by the *answer return* instruction to fetch answers from the table. The consumer choice point contains a pointer to the last answer that has been consumed from the table, so that on backtracking further answers can be consumed. A list of the consumer choice points for a tabled subgoal is kept, and the field *PrevCCP* points to the previous consumer choice point in this list. Instead of the argument registers the consumer choice point stores the substitution factor, which is used to retrieve answers from the table (see Section 2.3.3).

FailCont	The Failure Continuation
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Success Continuation for Subgoal
Ereg	Parent Environment
*Breg_Chain**	Failure Continuation on Backtracking *out* of this CP
*SubgFr**	Pointer to the Subgoal Frame
*BFreg**	Choice Point Freeze Register
*HFreg**	Heap Freeze Register
*TRFreg**	Trail Freeze Register
*EFreg**	Local Stack Freeze Register
A_n	Argument Register *n*
⋮	⋮
A_1	Argument Register 1
⋮	⋮
A_1	Argument Register 1
*VarNum**	Number of Variables: *m*
V_m*	Substitution Factor Variable *m*
⋮	⋮
V_1*	Substitution Factor Variable 1

Figure 2.11: Format of generator choice points.

The Trail The trail records the conditional bindings that must be undone in backtracking. In the SLG-WAM because of the suspend and resume mechanism, the values of the bindings have to be remembered, for the case that a computation path has to be restored. This is sometimes referred to as a *forward trail*. The freeze mechanism is also used for the trail itself, which implies that the previous trail frame in the current computation path may not be adjacent. This means adding a back pointer to every trail frame.

Summing up, the SLG-WAM trail frames have three fields: the variable address, the variable value and the previous trail frame.

The trail is used to restore the bindings when switching of environments takes place. This is done by the restore_bindings procedure given in [64].

Note on Conditional Bindings Because the suspend and resume mechanisms may freeze the stacks when some traditional WAM unconditional bindings have taken place, those bindings may still have to be undone and redone later, and can no longer be conditional. This leads to the restriction of the situations where bindings may be unconditional in the SLG-WAM. See [75] for details.

The Completion Stack The SLG-WAM has a new stack, the *completion stack* which is a proper stack (it doesn't require the freeze mechanism). The completion stack records

FailCont	Pointer to answer_return Instruction
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of Trail
CPreg	Success Continuation
Ereg	Parent Environment
*Breg_Chain**	Failure Continuation on Backtracking *out* of this CP
*LastAnswer**	Pointer to Last Consumed Answer
*PrevCCP**	Pointer for Consumer Choice Point Chain
*VarNum**	Number of Variables: m
V_m*	Substitution Factor Variable m
⋮	⋮
V_1*	Substitution Factor Variable 1

Figure 2.12: Format of consumer choice points.

which tabled goals are being solved at the moment and an approximation of their inter-dependencies. The *DirLink* field (see Figure 2.13) indicates the deepest subgoal that this subgoal depends on. For a subgoal S, we define $MinLink(S)$ as the deepest *DirLink* value for all the frames in the completion stack younger than the one for S. We define a S as a *leader of a scheduling ASCC* if and only if the completion frame associated with S is either the deepest one in the completion stack or satisfies the condition

$$DFN(S_{prev}) < min(DirLink(S), MinLink(S))$$

where S_{prev} is the predecessor of S on the completion stack.

The completion stack can be thus partitioned in scheduling ASCCs, $A_1 \ldots A_n$, with the property that no subgoal in a given scheduling ASCC depends on any subgoal in a scheduling ASCC deeper in the stack.

The scheduling ASCCs are thus approximations of *strongly connect components* (SCCs) in the subgoal dependency graph (see Section 2.2.3), in that they consist of a set of SCCs[1].

The DirLink fields are updated and the ASCCs maintained by the procedure adjust_levels (Figure 2.20 in Section 2.3.4) which is called by the SLG-WAM instruction table_try.

SubgFr	Pointer to Subgoal Frame
DFN	Depth First Number
DirLink	Deepest Direct Dependency

Figure 2.13: Format of completion stack frames.

[1]With Local scheduling an ASCC correspond to one and only one SCC.

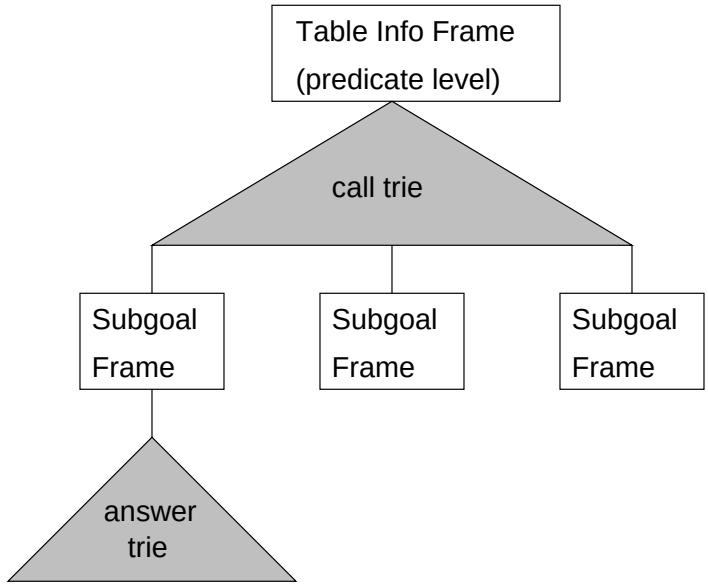

Figure 2.14: Data structures for a tabled predicate

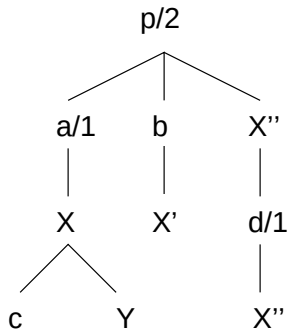

Figure 2.15: Trie for terms `p(a(X),c)`,`p(a(X),Y)`, `p(b,X')` and `p(X'',d(X''))`

2.3.3 Table Space

The SLG-WAM adds a new space to the WAM, the table space, used to store the tables of subgoals with their answers.

The table space is laid out one entry for each predicate (see Figure 2.14). The *Table Information Frame* represents a predicate and gives the entry point for the subgoals associated with that predicate by pointing into the *call trie*. The call trie represents the Prolog terms that correspond to the subgoal calls and can be traversed to give access to the *Subgoal Frame*.

Figure 2.15 shows the trie representation for several subgoals of a predicate `p/2`. This representation is compact and allows easy implementation of the check/insert operation of a subgoal in the table, by traversing the trie from the root to the leaves.

For each subgoal there is a subgoal frame (see Figure 2.16), which has a pointer to the root of the *answer trie*. The bottom elements of the figure, such as the *Answer Return List* are used for incomplete subgoals and can be disposed of when the subgoal is completed.

AnsTrieRoot	Pointer to the Root of the Answer Trie
IsCompleted	True if Subgoal is completed
NextSF	Pointer to Next Subgoal Frame
PreviousSF	Pointer to Previous Subgoal Frame
ComplSF	Pointer to the Associated Completion Stack Frame
AnsRetListH	Pointer to the Head of the Answer Return List
AnsRetListT	Pointer to the Tail of the Answer Return List
CCP_Chain	Pointer to the Head of the Consumer Choice Point Chain

Figure 2.16: Format of subgoal frames.

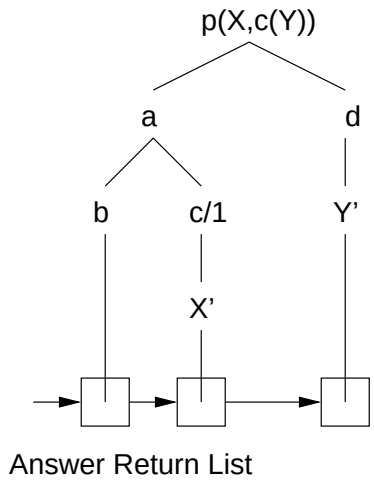

Figure 2.17: Answer trie for answers of `p(a(X),Y)`: `(X=a,Y=b)`; `(X=a,Y=c(X'))`; `(X=d,Y=Y')`

The answer trie (see Figure 2.17) corresponds to a trie where the answers of the subgoals are stored. This uses the *substitution factoring* optimization, so that only the bindings of the free variables of the call are stored in the trie. This has the effect of considerably reducing the size of the answer trie. The trie is compiled into WAM-like *trie instructions* which support the unification and backtracking at abstract machine instruction level. The compiled trie code is called directly by the SLG-WAM to return answers from completed tables; this is the completed table optimization. The answer return list is used to traverse the answers for incomplete subgoals. It points to the leaves of the trie, which means that traversing the answers for incomplete tables is done from the leaves to the root, in the inverse order to the one used for completed tables.

For a detailed treatment of tries, its compilation and the tabling data structures in general see [57].

2.3.4 The SLG-WAM Instruction Set

The SLG-WAM introduces new instructions specific to the new tabling operations. The table_try instruction mimics the WAM's try instruction, adding the functionality for the SLG NEW SUBGOAL operation if a new subgoal is encountered and additional

L_1 :	table_try	2	L_3	TR	%	
L_2 :	table_trust	2	L_{13}		%	
L_3 :	allocate	3			%	p(X,Z) :-
L_4 :	getVn	v_3			%	
L_5 :	getpvar	v_1	r_2		%	
L_6 :	putpvar	v_2	r_2		%	p(X,Y
L_7 :	call	3	p/2		%	),
L_8 :	putpval	v_2	r_1		%	p(Y,
L_9 :	putpval	v_1	r_2		%	Z
L_{10}:	call	3	p/2		%	)
L_{12}:	new_answer	2	v_3		%	.
L_{13}:	allocate	2			%	p(X,Z) :-
L_{14}:	getVn	v_2			%	
L_{15}:	getpvar	v_1	r_2		%	
L_{16}:	call	2	e/2		%	e(X,Z),
L_{17}:	putpval	v_1	r_1		%	q(Z
L_{18}:	call	2	q/1		%	)
L_{19}:	new_answer	2	v_2		%	.

Figure 2.18: SLG-WAM code for predicate p/2 of program $P_{2.1}$.

bookkeeping to the tabling data structures. The SLG-WAM table_trust instruction is identical to the WAM's trust instruction except that it sets the failure continuation to check_complete instruction which corresponds to the COMPLETION SLG operation, so that incremental completion can be performed when there are no more answers to this subgoal. There is also a table_try_single instruction for single clause tabled predicates. The tabled predicates need the try-type instructions because a generator choice point must always be created, even for single clause predicates.

The answer_return instruction corresponds to the SLG POSITIVE RETURN operation. This instruction is stored as the failure continuation of consumer choice points and is used to resume the computation and to return answers to the variant subgoal calls suspended waiting for answers from the table.

The new_answer instruction is compiled at the end of every tabled predicate clause and corresponds to the NEW ANSWER tabling operation. The new_answer instruction also performs the functionality of the WAM's deallocate and proceed instructions.

Figure 2.18 shows the compilation into SLG-WAM instructions for predicate p/2 of program $P_{2.1}$ in Figure 2.1. Note the getVn instructions that save a pointer to the generator choice point in a local variable, so that it can later be accessed by the new_answer instruction, even if additional choice points have been pushed meanwhile. Note also that there are no deallocate neither proceed instructions, whose functionality is performed by the new_answer instruction.

The Table_Try Instruction We proceed to explain the table_try instruction for a subgoal S as represented in Figure 2.19. The TIF argument corresponds to the Table Infor-

mation Frame, given in the previous section, which is used to access the call trie, which is traversed in line 1, to obtain the subgoal frame *SF*. If the subgoal frame is not found, a new one is created in line 2.1 and a generator choice point for the evaluation of a new tabled subgoal is created (line 2). A completion stack frame is pushed for that subgoal (line 2.3) and execution branches to the next SLG-WAM instruction so that program clause resolution will be performed.

If the subgoal is already on the table and is completed (lines 3–3.2) execution branches to the answer trie of the completed subgoal. If the subgoal is already on the table but is incomplete (lines 4–4.7), a consumer choice point is created to return the answers. The procedure adjust_levels is called to reflect the dependency of subgoals younger than *S* on *S*. The stacks are frozen to preserve the current computation path, so that the answer_return instruction in this consumer choice point can be executed in the proper environment. Finally execution fails into the consumer choice point, for it to return its first answer.

```
Instruction table_try(Arity, NextClause, TIF)        /* Subgoal is in argument registers */
1   SF ← subgoal_check_insert(Subgoal, TIF)
2   if (SF = NULL )                                   /* Subgoal is new and added */
2.1     SF ← CreateSubgoalFrame(Subgoal);
2.2     GCP ← PushGeneratorChoicePoint(...);
2.3     GCP.FailCont ← NextClause;
2.4     ComplSF ← PushCompletionStackFrame(...);
2.5     ComplSF.SubgFr ← SF ;
2.6      /* Branch to the next SLG-WAM instruction */
3   else if ( (SF.IsCompleted) )                      /* Subgoal is complete */
3.1     Answer_Root ← Subgoal.AnsTrieRoot;
3.2      /* Branch to Answer_Root SLG-WAM instruction */
4   else                                              /* Subgoal is incomplete */
4.1     CCP ← PushConsumerChoicePoint(...);
4.2     adjust_levels(SF);                            /* for scheduling ASCCs: see Figure 2.20 */
4.3     CCP.Prev_CCP ← SF.CCP_Chain;
4.4     SF.CCP_Chain ← CCP;
4.5     CCP.FailCont ← answer_return;
4.6     freeze_stacks();
4.7     Fail;                                          /* to answer_return instruction in CCP */
```

Figure 2.19: The table_try instruction.

Procedure adjust_levels(*S*) updates the completion stack to reflect the dependency of any younger subgoals on *S*. By updating the *DirLink* field of the topmost completion stack frame with the *DirLink* corresponding to *S* or a keeping lower value, it is ensured that *S* is included in the topmost ASCC.

```
Procedure adjust_levels(Subgoal)
    ComplSF_top ← openreg;
    ComplSF ← Subgoal.ComplSF;
        /* Completion stack frames are accessed through the corresponding subgoal frame */
    ComplSF_top.DirLink ← min(ComplSF_top.DirLink, ComplSF.DirLink);
```

Figure 2.20: Updating ASCC information on encountering consumer subgoals.

```
Instruction check_complete
1   SubgCSF ← breg.SubgFr.ComplSF;          /* breg points to the Generator CP of Subgoal */
2   if (is_leader(SubgCSF) )
2.1     tmp_breg ← fixpoint_check(SubgCSF, breg);
2.2     if( tmp_breg ≠ breg )               /* if there are answers to return */
2.2.1       breg ← tmp_breg ;
2.2.2       Fail;
2.3     for each CSF ∈ [SubgCSF..openreg]
2.3.1       CSF.SubgFrame.IsComplete ← true;
2.4     openreg ← SubgCSF − 1
2.5     reclaim_stacks(breg) ;
3   breg ← breg.Breg_Chain;
4   Fail;
```

Figure 2.21: The check_complete instruction for definite programs (Batched scheduling).

2.3.5 Scheduling

In this section we describe how the SLG-WAM implements the two scheduling policies, Batched and Local. For a more detailed description see [34].

Batched Scheduling Batched scheduling inherits the scheduling properties from Prolog, in that when an answer is derived, by the new_answer instruction, it is immediately returned to the calling environment.

Batched scheduling derives its name from the way that answer_return instructions are scheduled within an ASCC. The systems tries to schedule all unreturned answers for the ASCC at each pass, doing multiple passes, until no more answers can be derived.

The Check_Complete Instruction for Batched Scheduling As long as there are PROGRAM CLAUSE RESOLUTION operations to be performed for a generator choice point, they are done as in Prolog. When no more PROGRAM CLAUSE RESOLUTION operations can be performed, execution backtracks to the check_complete instruction. Figure 2.21 shows the actions performed by the completion instruction.

In line 2, it is checked if the subgoal is the leader of the ASCC. If not, execution passes to line 3 where it fails to the previous choice point, until a leader is found.

```
Function fixpoint_check(SubgCSF, sched_chain)
    for each CSF ∈ [SubgCSF..openreg] /* SubgCSF is a pointer to the completion stack frame */
        SubgFr ← CSF.SubgFr;
        sched_chain ← schedule_resumes(SubgFr, sched_chain);
    return sched_chain;
```

Figure 2.22: The fixpoint_check function.

```
Function schedule_resumes(SubgFr, sched_chain)
    CCP ← SubgFr.CCPChain;                /* consumer choice point chain for S */
    while( CCP ≠ NULL )
        if( has_unconsumed_answers(CCP) )
            CCP.Breg_Chain ← sched_chain;
            sched_chain ← CCP;
        CCP ← CCP.PrevCCP;
    return sched_chain;
```

Figure 2.23: The schedule_resumes function.

In line 2.1 a scheduling chain is built out of all the consumer choice points for subgoals of this ASCC that have answers to return by function fixpoint_check. If the chain is not empty (line 2.2) executions fails into it, to return the answers from the environments of the consumer choice points.

From line 2.3 to line 2.5 several actions are taken, to mark the subgoals as completed and reclaim the stack space.

Figure 2.22 shows function fixpoint_check which calls schedule_resumes to build the scheduling chain for the consumer choice points of each subgoal in the ASCC and merges them together.

Figure 2.23 builds the scheduling chain of choice points with answers to return for each subgoal.

Local Scheduling Local scheduling follows the principle given in Section 2.2.3, Definition 2.2.11, which states that tabling operations may only be performed to trees in the independent SCC of the SDG of the evaluation. As such, it evaluates one SCC at a time, not returning any answers outside the SCC until all of its subgoals are completed.

The new_answer instruction is prevented of returning the derived answer to the calling environment by failing to the current generator choice point after deriving the answer.

The Check_Complete Instruction for Local Scheduling The check_complete instruction for Local scheduling has to take care of more details than its equivalent for Batched scheduling. Figure 2.24 shows the actions performed by the completion instruction.

Lines 1 to 3.2.2 take care of the scheduling of answer_return instructions by the leader

```
Instruction check_complete
1   Subgoal ← breg.SubgFr;                    /* breg points to the Generator CP of Subgoal */
2   SubgCSF ← Subgoal.ComplSF;
3   if (is_leader(SubgCSF) )
3.1       tmp_breg ← fixpoint_check(SubgCSF, breg);
3.2       if( tmp_breg ≠ breg )               /* if there are answers to return */
3.2.1         breg ← tmp_breg ;
3.2.2         Fail;
3.3       for each CSF ∈ [SubgCSF..openreg]
3.3.1         CSF.SubgFrame.IsComplete ← true;
3.4       openreg ← SubgCSF − 1 ;
3.5       reclaim_stacks(breg) ;
3.6       if( has_answers(Subgoal) )
3.6.1         /* restore environment from the generator choice point breg */
3.6.2         Answer_Root ← Subgoal.AnsTrieRoot;
3.6.3         /* Branch to Answer_Root SLG-WAM instruction */
4   else                                      /* it is not a leader */
4.1       MakeConsumerFromGenerator(breg);
4.2       breg.PrevCPP ← Subgoal.CCP_Chain;
4.3       Subgoal.CCP_Chain ← breg;
5   breg ← breg.Breg_Chain;
6   Fail;
```

Figure 2.24: The check_complete instruction for definite programs (Local scheduling).

and are the same as the ones for Batched scheduling. When completion takes place, again, lines 3.4 and 3.5 are the same as in Batched scheduling. In line 3.6 it is checked if the leader has any answers. If so, they must be returned outside of the SCC. Remember that by Corollary 2.2.1 there can be only one call into the SCC, and that's the call to the leader. So only the leader has to return answers outside of the SCC, and it does so by backtracking through its answer trie, which is activated in line 3.6.3. If the leader doesn't have any answers, it just fails to the calling SCC through line 6.

Line 4.1 features another interesting aspect of the implementation of Local scheduling in XSB. When a generator choice point has exhausted all PROGRAM CLAUSE RESOLUTION operations and finds that it is not the leader, it transforms itself into a consumer choice point (mainly by changing its *FailCont* field into a answer_return instruction), and it is added to the consumer choice point chain of the subgoal. It will later be scheduled like any other consumer choice point frame, to return its answers into the SCC that it belongs to. This allows for most of the code to be shared among the Local and Batched check_complete instructions.

FailCont	Pointer to negation_resume Instruction
EBreg	Environment Backtrack Point
Hreg	Top of Global Stack (Heap)
TRreg	Top of (Forward) Trail Stack
CPreg	Return Point of Suspended Literal
Ereg	Parent Environment
RSreg	Root Subgoal Choice Point
SubgFr	Frame of Suspended Subgoal
PrevNS	Pointer for Negation Suspension Frame Chain

Figure 2.25: Format of negation suspension frames.

2.4 Support for Negation in the SLG-WAM under Local Scheduling

In this section we give an overview of the features of the SLG-WAM that offer support for negation. We only consider Local scheduling, which allows some features of the implementation to be simplified. For a detailed description of the support for stratified negation under Batched scheduling see [64]. For the implementation of non-stratified negation with delay see [65]. For a detailed description of the data structures that implement delay lists in the table space see [20].

We first consider the basic mechanisms to support stratified negation and then proceed to the mechanisms needed for full support of the WFS.

Data Structures The following registers are added to the SLG-WAM to support negation:

rsreg The *Root Subgoal register* keeps the root of the current SLG tree that is being evaluated.

dreg The *Delay register* keeps the head of the delay list for the subgoal that is being evaluated.

A field for each of these registers is added to all types of choice points, because they have to be restored on backtracking.

Choice Point Stack A new type of choice point, the *negation suspension frame* (see Figure 2.25) is added. A negation suspension is created upon the suspension of the evaluation of a negated subgoal *not S*, and later may be used to switch environments back to *not S*. This is explained in more detail in Section 2.4.2.

Table data structures The field *NS_Chain* is added to the subgoal frame to keep track of any negation suspension frames that depend on a subgoal *S*, in a manner similar to

```
1 tnot(S) :-
2      ( ground(S) →
3          ( subgoal_not_in_system(S), call(S), fail
4          ;  ( is_complete(S) → negate_truth_value(S)
5             ; negation_suspend(S)
6             )
7          )
8      ; error("Flounder:  subgoal ", S, " is not ground")
9      ).
```

Figure 2.26: Implementation of the tabled negation predicate (`tnot/1`)

the way that CCP_Chain keeps track of consumer choice points. The interned delay lists are also supported at the level of the table data structures, Section 2.4.2 gives more details.

2.4.1 Support for Stratified Negation

In this section we focus on the basic mechanisms to support negation for programs that don't need to apply the DELAYING and SIMPLIFICATION operations.

Early Completion When the new_answer instruction derives an answer to a ground subgoal, it is sure to be the only answer, and the subgoal can be completed, as far as the declarative meaning of the program is concerned.

This is done in XSB by setting the failure continuation of the generator choice point to the check_complete instruction, so that no more clauses are explored for that subgoal, and by marking the subgoal as complete. The pointer to the nodes that depend negatively on this subgoal (*NS_Chain*) is set to NULL, effectively trimming the negative paths that were to be explored by the NEGATION SUCCESS operation, in case this subgoal would fail. Thus the NEGATION FAILURE operation is performed.

As it can be seen by the next paragraph, Early Completion is not really needed under Local scheduling. However we find important to mention it, because its in the system, and because it allows to break the fixed left to right computation rule in Batched scheduling, allowing some programs involving stratified negation to be resolved without having to resort to the delay mechanism.

Implementation of `tnot/1` Negation is accessed by the programmer through the `tnot/1` predicate. The high-level implementation of the `tnot/1` predicate can be seen in Figure 2.26.

Remember that in Local scheduling, a subgoal is always completely evaluated before its SCC returns any answers (Corollary 2.2.2). The two cases in which a subgoal may be incomplete when its called within an SCC, is that it is the first time it is called or that it is involved in a loop through this SCC. In Figure 2.26 when *S* is checked in line

4, it has been called before, either in line 3, or before, if it was already present on the system. So the only chance for it to be incomplete is to be involved in a loop through the current SCC, which means that the evaluation has a loop through negation, and is not stratified.

As such, for stratified programs under Local scheduling, `negation_suspend/1` is never called and the truth value is always established by line 4.

2.4.2 Support for Delay and Simplification

To support programs with non-stratified (well-founded) negation the delay and simplification mechanisms must be implemented.

The `negation_suspend/1` predicate in Figure 2.26 suspends execution by creating a *negation suspension* choice point (see Figure 2.25) in the choice point stack and freezing the stacks. The negation choice point failure continuation points to a negation_resume instruction. This is used to resume the computation upon the success of the negated subgoal (performing a NEGATION SUCCESS operation) or the delaying of the subgoal.

Delaying Delaying occurs at the negation_resume instruction: if a negation suspension frame is resumed the negation_resume instruction checks if the subgoal (S) that it depends on has unconditional answers.

If there is an unconditional answer the computation fails; otherwise the literal corresponding to the call to S is delayed negatively, as $notS^S$, and the computation proceeds.

The check_complete instruction (see Figure 2.27) schedules negation suspension frames through the ProcNegSusps function (Figure 2.28) in lines 3.3–3.4.1. This ensures the break of any loop through negation within the SCC. check_complete also triggers the simplification on failure of any subgoals of the SCC, in lines 3.5.3–3.5.3.1.1.

The function ProcNegSusps (Figure 2.28) is similar to fixpoint_check; it schedules any negation suspension frames present in the SCC.

Delay Propagation When a conditional answer is returned, its head is added to the delay list. This operation involves adding a positive literal to the delay list and is done by the answer_return operation. Positive delay elements are annotated with the substitution that takes place in the resolution operation, i.e. if the call to subgoal `p(X)` returns an conditional answer $X = a$ the delay element is annotated as $p(a)^{p(X)}_{p(a)}$.

Simplification There are four cases of simplification:

- Simplify negation success: A subgoal S completes without answers. All the negative delay elements corresponding to S can be removed from its delay lists.
- Simplify negation failure: A unconditional answer is derived for subgoal S. All delay lists with negative delay elements corresponding to S can be removed.

```
Instruction check_complete
1   Subgoal ← breg.SubgFr;                          /* breg points to the Generator CP of Subgoal */
2   SubgCSF ← Subgoal.ComplSF;
3   if (is_leader(SubgCSF) )
3.1       tmp_breg ← fixpoint_check(SubgCSF, breg);
3.2       if( tmp_breg ≠ breg )                     /* if there are answers to return */
3.2.1           breg ← tmp_breg ;
3.2.2           Fail;
3.3       tmp_breg ← ProcessNegSusps(SubgCSF, breg);
3.4       if( tmp_breg ≠ breg )                     /* if there are loops through negation */
3.4.1           breg ← tmp_breg ;
3.4.2           Fail;
3.5       for each CSF in [SubgCSF..openreg]
3.5.1           SF ← CSF.SubgFrame
3.5.2           SF.IsComplete ← true;
3.5.3           if (not has_answers(SF) )
3.5.3.1               for each delay list where SF appears
3.5.3.1.1                 /* Simplify negation success */
3.6       openreg ← SubgCSF − 1 ;
3.7       reclaim_stacks(breg) ;
3.8       if( has_answers(Subgoal) ;
3.8.1           /* restore environment from the generator choice point breg */
3.8.2           Answer_Root ← Subgoal.AnsTrieRoot;
3.8.3           /* Branch to Answer_Root SLG-WAM instruction */
4   else                                            /* it is not a leader */
4.1       MakeConsumerFromGenerator(breg);
4.2       breg.PrevCPP ← Subgoal.CCP_Chain;
4.3       Subgoal.CCP_Chain ← breg;
5   breg ← breg.Breg_Chain;
6   Fail;
```

Figure 2.27: The check_complete instruction for normal programs (local scheduling).

```
Function ProcNegSusps(SubgCSF,sched_chain)
    for each CSF ∈ [SubgCSF..openreg]
        SubgFr ← CSF.SubgFr);
        NSF ← SubgFr.NS_Chain
        while( NSF ≠ NULL )
            NSF.Breg_Chain ← sched_chain ;
            sched_chain ← NSF ;
            NSF ← NSF.PrevNS ;
    return sched_chain ;
```

Figure 2.28: The ProcNegSusps function.

- Simplify positive success: A previously conditional answer A is found to be unconditional for subgoal S. All the positive delay elements corresponding to D_A^S can be removed from its delay lists.

- Simplify positive failure: A previously conditional answer A is found to be false for subgoal S. All delay lists containing a delay element D_A^S can be removed.

The main SLG-WAM instructions that lead to simplification are check_complete, when a subgoal is completed without answers and answer_return, when an unconditional answer is returned. However, more simplification operations may be triggered by other simplification operations, and the data structures that are used to store the delay lists in the table space must support efficiently such cascaded simplification operations.

Data Structures for Delay Lists The delay list for the current computation path is kept on the heap, pointed by *dreg*. When the instruction new_answer derives an conditional answer, its delay list is copied to the table space.

The Delay information is accessed, at first, by the answer trie of the conditional answer. This is connected with, possibly several, delay lists, through a *Delay Info* frame. Other than the delay lists, the Delay Info frame also contains the *PDE list*. The PDE List is a doubly linked list, and each element in this list contains a back pointer (*DE pointer*) to the occurrences of a delay element of the list and a back pointer (*DL pointer*) to the delay list where this element belongs. These back pointers are used for simplification of positive delay elements. Negative delay elements have a similar *NDE* list, however as negative literals must be ground in the SLG-WAM this list is accessed from the subgoal frame.

Each delay list contains several delay elements. Each delay element has four fields:

- **Subgoal_Id** which points to the subgoal frame of the delayed literal.

- **Answer_Subst_Id** which points to the leaf of the answer trie, corresponding to the bindings of the call, if the delay element is positive. If the delay element is negative this field is set to NULL.

- **Delay_Trie** which points to the leaf of the *Delay Trie* where the bindings to that delay element are stored.

- **PNDE** pointer to the PDE element of this delay element (if the delayed literal is positive) or pointer to the NDE element of this delay element (if the delayed literal is negative). Used for simplification operations.

Chapter Summary In this chapter we have introduced tabling and presented SLG, a formal semantics for it. We formalized Local scheduling and shown some of its properties. We presented the SLG-WAM, the abstract machine that implements tabling in the XSB system and discussed the implementation of well founded negation under Local scheduling. This material forms the basis for the development of our work, mainly in Chapters 4 and 5.

3

The Multi-Threaded Engine

In this chapter we describe the multi-threaded XSB engine. We begin by recalling the main aspects of the multi-threading programming model and its mapping to the Prolog language, including the use of tabling. We follow by describing the main design decisions and give the execution and programming model for an XSB process. We conclude with some nitty-gritty details of the implementation.

3.1 Multi-Threaded Programming

Real world applications need support for concurrent programming for a number of reasons:

- Events happen concurrently in the outside world, and sometimes cannot be processed sequentially. For instance a FTP server cannot wait for the transfer of the current file to end before a new transfer is started.
- Some algorithms are inherently concurrent, and are more naturally modeled in a concurrent programming model. For example, some applications might be naturally structured as a pipeline. It would be natural and advantageous to model each of stage of the pipeline as a thread, which would receive its input from the previous stage, and send its output to the following stage.
- Making use of the resources supplied by multi-processor computing systems, by parallel programming, is a way to achieve better results in shorter time periods. For example, chess programs became much stronger with its implementation over multi-processors.

Traditionally, concurrency has been supported by the operating systems through multi-processing, where each process is a single execution environment, protected by the operating system against interference form other processes. This model has the advantage of protecting each process against another process's bugs, or even hostile actions, which is of course essential in a multi-user context. However the actions involved in switching contexts from a process to another and the need for operating system intervention in the communication and synchronization among such processes, lead to an execution overhead, that would not really be necessary, in the case of a single application that consists of multiple processes.

Multi-threading provides a second layer of concurrency, multiple threads within a process, where context switching, synchronization and communication demand less or no intervention from the operating system, leading to a much more efficient execution of concurrent programs.

In the rest of this section we follow the model of *pthreads*, one of the most used models for multi-threaded programming. Pthreads stand for POSIX threads, and are formalized as an ISO standard [40]. For a comprehensive introduction to the multi-threaded programming with pthreads, see [42].

Memory Layout for the Multi-threaded Process As different threads share the process's address space, unlike with different processes, there is no protection of each thread's memory through the operating system's virtual memory mechanism. Global variables and data structures are shared, and can be used for inter-thread communication. However each thread has its own execution stack and local variables, which, even though there is no system protection of this memory area, tend to be viewed as private for each thread. As state has to be maintained among function calls, most implementations also provide a space for thread private variables, which can be used throughout function calls of the same thread as if they were global variables. They can be accessed by other threads, but this is usually not wanted. As all data is shared, communication doesn't involve any overheads due to operating system calls, but hard-to-debug bugs can arise from one thread randomly damaging another's "private" data or from races involving global variables.

Kernel Level Threads vs User Level Threads While the first multi-threading implementations were provided by user level libraries, with little or no interaction with the operating system's kernel, operating systems were gradually upgraded to support multi-threading at the level of the kernel. Support by the operating system incurs in higher overheads but has several advantages:

- More natural integration of multi-threading with the process I/O mechanisms. The user level threads have to resort to elaborated tricks to avoid the whole process to be blocked when one thread is performing I/O.

- Support for parallel execution of different threads on multi-processors. This can only be done with support from the kernel.

Kernel level threads are often referred to as *lightweight processes*. User level threads can provide more control in the commutation of threads, by providing restrictions to thread preemption, and so make it easier for the programmer circumventing concurrency problems. Due to the fact that both approaches have advantages, some implementations offer an integration of both models, where a number of user level threads can be mapped into one kernel thread[1].

Basic Synchronization Mechanisms The *mutex* lock is the mechanism used by threads to guarantee mutual exclusion in the access of variables and serialization of the execution of critical sections of code. The implementation guarantees that only one thread may hold the lock at a time, and other threads are kept waiting for the lock either by busy waiting or by being suspended in a queue.

The *condition variable* is the mechanism used to achieve general synchronization among threads. There is a *cond_wait* primitive that suspends a thread on a condition variable. The *cond_signal* primitive awakes a thread waiting in a condition variable. If the thread has not yet suspended the signal is lost. Because of this, condition variables should be used together with a mutex lock and explicit testing of the conditions involved. The section of code executed atomically under the mutex lock which involves the manipulation of the condition variables, testing of the conditions and other actions, is akin to the monitor construct present in some concurrent programming languages (see for example Chapter 7 of [5] for a general discussion of monitors). Usually the *cond_wait* instruction is repeated in a while cycle which is exited when the condition for resuming execution becomes true.

3.1.1 Multi-Threaded Prolog

Although the original Prolog language doesn't provide constructs for multi-threading, Prolog is still a viable candidate to support multi-threaded programming by considering its procedural interpretation and developing a library of built-in predicates. As Prolog doesn't allow the programmer to manipulate pointers, the main disadvantage of multi-threading, namely buggy interferences in the memory that should be private of other threads, doesn't apply to multi-threaded Prolog programming. This of course, providing that the implementation is correct, so that this burden is placed on the implementors of a multi-threaded Prolog system. In fact several Prolog systems support multi-threading today, and they are discussed in Chapter 6. The basic model for a multi-threaded Prolog system is shown in Figure 3.1, where multiple Prolog executors share a predicate database, consisting of static and possibly dynamic predicates.

[1]More complex mappings from user level threads into lightweight threads are possible – see [42]

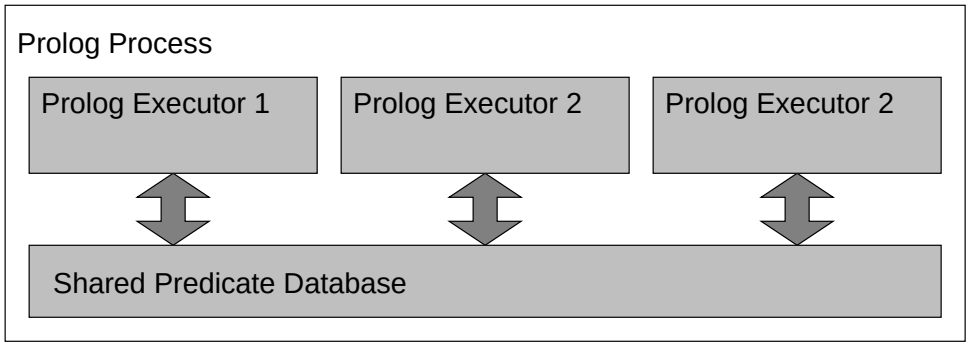

Figure 3.1: Multi-threaded execution model of a Prolog process

Each Prolog executor is a fully self-contained deduction engine, where its logical variables are stored internally. This means that there is no sharing of logical variables among Prolog threads, and other mechanisms must be provided for the Prolog threads to communicate, one logical candidate being the shared predicate database.

Communication and Synchronization Mechanisms The synchronization of threads through condition variables is a somewhat low-level mechanism for a language like Prolog. So higher level mechanisms of synchronization have been provided by multi-threaded Prologs.

`Assert` and `retract` are the two basic mechanisms that Prolog offers for the manipulation of the database, which can be used to implement communication among threads. Some systems, like Ciao (see Chapter 6), provide the synchronization among threads through these primitives.

However, `assert` and `retract` are somewhat complex mechanisms in Prolog, because in general, they must support complex features like indexing and backtracking. Most multi-threaded Prolog systems resort to message passing mechanisms, under various forms, to provide synchronization and communication among threads.

Another desirable mechanism in a multi-threaded Prolog system implementation is the mutex lock, which allows to serialize a series of actions, for example guaranteeing an atomic operation involving changing the shared predicate database through multiple `assert` and `retract` operations.

Even if the shared predicate database is a feature of every multi-threaded Prolog, it can be argued that a private database for dynamic predicates is a most desirable feature to have, as it allows threads to keep its global state through predicate calls without interference with others. More so, private dynamic predicates don't require synchronization support from the implementation which makes the implementation much simpler and more efficient. Issues like space reclamation are much easier to handle for private dynamic predicates than for shared ones. The absence of synchronization also allows private dynamic predicates to give better results for parallel execution.

3.1.2 Multi-Threaded Tabling

The multi-threaded execution of a process that supports tabling can be seen as an extension of the multi-threaded Prolog execution, adding the tabling operations. The main implementation decision is whether to have a table that is shared by the threads or for each thread to have its own private table space. Private tables have several advantages:

- They have a simpler implementation than shared tables, as inter-dependencies among threads don't have to be taken into account to complete or garbage collect tables.
- Features such as negation and subsumption are more readily supported, mainly because the completion operation is simplified by not having to handle SCCs among tables owned by different threads.
- They allow better results for parallel execution on a multi-processor, as the access to the tables doesn't require synchronization.

On the other hand, sharing tables also has advantages:

- It allows the previous computations of other threads to be re-used, thus improving the time efficiency of the system.
- It saves memory, improving the space efficiency of the system.
- Given that we can assign different tables to different threads, so that each thread computes the answers for a set of tables in parallel, shared tables constitute a powerful basis for parallel programming.

As tabled execution has a declarative semantics, we could in principle hide the detail of whether the tables are shared or private from the programmer, and deem it an implementation detail. However, given the very different execution properties of both models we decided that the programmer is allowed to specify for each tabled predicate, whether its tables are shared or private.

3.2 Design of Multi-Threaded XSB

In this section we present the major design decisions taken in the development of the multi-threaded system and give an overview of its execution model. We describe the system's API giving application examples.

Design Considerations In the conception of the multi-threaded system we tried to adhere to the following principles:

1. **Support of the Sequential System** – The sequential system would continue to be supported as is, with minimal interference from the multi-threading extensions.

2. **Minimal overhead** – we want to keep to overhead for the multi-threaded system as low as possible w.r.t. the sequential system.

3. **Incremental implementation** – we want to provide multi-threaded execution for a core functionality and to make it possible to gradually extend it to support the multiple XSB functionality and packages.

4. **Parallelism** – we wish to be able to take advantage of the presence of more CPUs to achieve faster execution when executing a multi-threaded program – provided that the program itself allows multiple threads to be executed in parallel.

5. **Scalability** – we strive to support the execution of many threads with minimal unneeded interactions, so that a multi-processor's potential can be used effectively.

Native Threads Versus Built-in Scheduler One important issue when implementing threads for any language is related to the use of a native threads package, usually integrated with the operating system, or support all the mechanisms of the multi-threaded execution from within our implementation. The latter solution involves explicitly commuting from one thread to another, while the context for each thread is kept in a data structure and managed by a scheduler within the language runtime support (e.g. the Prolog emulator).

- The thread context variables can be kept in global variables, as they were for the sequential system, they're values being swapped when the scheduler commutes between threads. This allows to minimize changes to an existing sequential system, specially in terms of access to global variables.

- Complete control on thread preemption and switching allows easier control of critical sections, by disabling thread preemption at critical points.

- Complete control of the scheduling of threads allows the implementation of sophisticated scheduling policies and more powerful concurrency mechanisms.

- As a significant part of the implementation doesn't rely on the operating system, portability problems are greatly reduced.

However in this way kernel level threads cannot be used, and as was mentioned in the previous section, user level threads have the severe disadvantage of not allowing

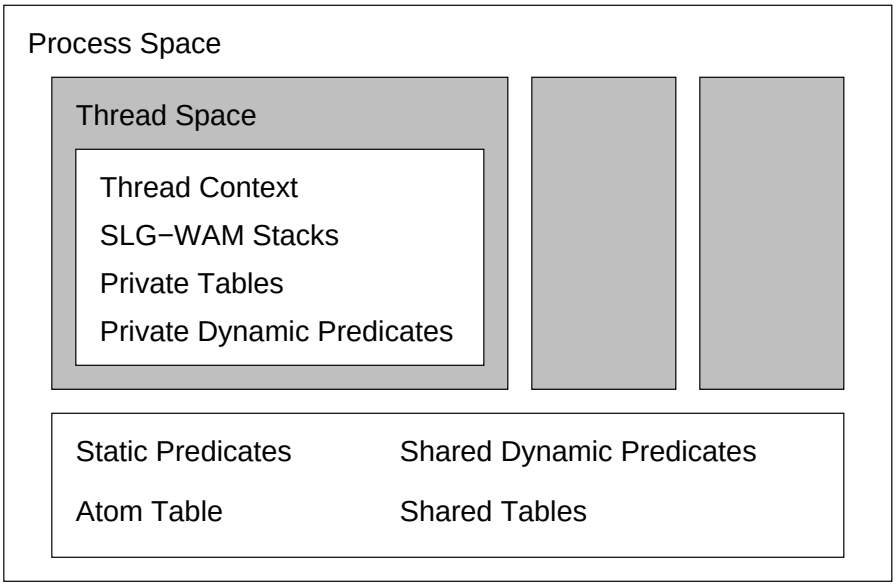

Figure 3.2: Multi-threaded memory layout of a XSB process

different threads to run on different CPUs when using a multi-processor, thus violating our parallelism design goal. We thus decided to base our implementation on the standard POSIX thread interface, which is available for most UNIX systems. POSIX provides a standard programming model for threads and portability for most systems should be good[2].

Execution Model for Multi-Threaded XSB Figure 3.2 shows the basic organization of the multi-threaded XSB process. There are global data structures for the shared dynamic predicates and the shared tables, as well as static code and the atom table. Each thread has its execution context (discussed in Section 3.3.1), stacks, private dynamic code and tables.

This model allows threads to avoid most inter-dependencies if they restrict themselves to private tables and dynamic predicates, while allowing them to share tables and dynamic predicates when it is appropriate.

3.2.1 Multi-Threaded API

The API for XSB[3] follows the current proposal for the ISO standard on the multi-threaded extensions to Prolog [52]. It includes predicates for thread manipulation and thread synchronization and directives for the sharing of predicates. Most predicates on the API have shortcut versions with less arguments, which use default values for

[2]In fact XSB supports threads under Windows (and the .NET platform through Win32 native code) – although the system must be compiled through the Cygwin Linux interface to Windows (`http://cygwin.com`).

[3]The multi-threaded API of XSB suffered an evolution throughout the dissertation process, including during the time of the writing of the thesis. Professor Terrance Swift made significant contributions in the re-design and implementation process.The current API is based on the ISO standard proposal [52].

the missing arguments — we don't give these versions in this text. For a more detailed description consult the XSB programming manual [81].

Thread Management There are some basic predicates to create and destroy threads. The identifier is an integer, which is mapped by the system to a pthread identifier.

- **thread_create(+Goal, -ThreadId, +OptionList)** This predicate creates a new thread to execute the given goal and returns its identifier. The option list may include the initial XSB stack sizes for the thread and if the thread will be created as detached (so its return information automatically released at exit and the thread cannot be joined). It may also include aliases, a mechanism that allows the thread to be known by symbolic names.

- **thread_join(+ThreadIds, -ExitDesignators)** This predicate causes the thread to wait for a thread or a set of threads (ThreadIds can be a list) to exit and returns its exit status in ExitDesignators. Each exit status is a Prolog term, which contains the reason for the termination of the thread. In case the thread terminated via the `thread_exit` predicate the exit status contains the term passed to thread exit. In case the thread terminated via an exception the exit status contains the exception ball.

- **thread_exit(+Term)** This predicate terminates the current thread and allows returning an arbitrary term, which can be retrieved by other threads using the `thread_join` predicate.

- **thread_self(-ThreadId)** This predicate allows the current thread to know its identifier.

- **thread_detach(+ThreadId)** This predicate informs the system that the thread will not be joined, and that its return information can be disposed at thread exit. Any mutexes held by a thread are released when it exits.

- **thread_cancel(+ThreadId)** This predicate kills another thread. The main thread of XSB cannot be cancelled. If any mutex locks are held by the cancelled thread they are released. Note that, in general, cancelling a thread may be dangerous, for example if the thread was manipulating some data structures which might be left incoherent. However there's never any problem from the point of view of XSB's correct execution in the case of thread cancellation. More, as in the case above, when a thread is cancelled all the mutexes it holds are released.

- **thread_yield** This predicate tells the operating system to commute to a new thread.

Thread Synchronization and Communication Predicates These are the predicates used for communication and to achieve synchronization among threads.

- **mutex_create(?Mutex)** This predicate creates a new mutex. If an atom is given, the mutex will be designated by that atom, otherwise a system generated designator will be returned.
- **mutex_destroy(+Mutex)** This predicate destroys a mutex.
- **mutex_lock(+Mutex)** This predicate acquires a lock on a mutex, if necessary suspending the thread until the lock can be held.
- **mutex_try_lock(+Mutex)** This predicate tries to acquire a lock on a mutex, but if unsuccessful it fails instead of suspending the current thread.
- **mutex_unlock(+Mutex)** This predicate releases a lock on a mutex.
- **with_mutex(+Mutex, +Goal)** This predicate executes `Goal` deterministically, while holding the given mutex. Unlike the previous low-level mutex locking primitives, it is guaranteed that if the goal fails or an exception is raised the mutex is unlocked.
- **mutex_unlock_all** This predicate releases all the mutex locks that the current thread holds.
- **mutex_property(?Mutex, ?Property)** This predicate allows to check the current status of a mutex (whether it's locked and by which thread). On failure, it backtracks through all mutexes in the system if the first argument is given as a variable.
- **message_queue_create(-QID, +Options)** This predicate creates a new message queue and returns its identifier. Message queues are buffers used for threads to communicate through message passing. The queue has a fixed size (the number of slots in the queue which can be passed in the Options list) and threads are suspended if they are trying to remove items from an empty queue or add items to a full queue.
- **thread_send_message(+QID, +Message)** This predicate copies a new message into the message queue given. The message can be any valid Prolog term. If the queue is full it blocks, until there is an empty slot for the new term.
- **thread_get_message(+QID, ?Message)** This predicate reads a message from the given message queue. The parameter message is unified with all the messages on the message queue, in FIFO order, until the unification succeeds. The message for which unification has succeeded is then removed from the message queue. If no term in the queue matches the given message, the predicate blocks until such a term is present in the queue.

Directives for Sharing Tables and Predicates The following directives control the sharing of predicates:

- **:- thread_private Predicate/Arity** This directive specifies that the predicate is private to this thread. It means that if the predicate is dynamic, each thread sees it's own clauses, and if the predicate is tabled, private tables are used for its computation.

- **:- thread_shared Predicate/Arity** This directive specifies that the predicate is shared by all threads in the XSB process. It means that if the predicate is dynamic, every thread sees its clauses, and if the predicate is tabled, shared tables are used for its computation.

By default all predicates are private in XSB. This is because the private tables implementation became stable earlier than the shared tables implementation and has been more thoroughly optimized.

The user is responsible that a tabled dynamic shared predicate doesn't get modified while its tables are being computed, which could lead to an incoherency between the answers in the table and the logical meaning of the predicate. Note that a shared dynamic tabled predicate can't use private tables for its computation whereas a private dynamic tabled predicate can't use shared tables. While not allowing the first situation is questionable, the second case is definitely to be prevented.

Examples of Multi-Threaded Programs in XSB Figure 3.3 shows an example of a multi-threaded goal server in XSB, which makes extensive use of XSB's socket library. The server listens for requests from clients and spawns a worker thread to fulfill each request. The worker thread has a connection with the client, from where it reads the request and where it sends the answers, one at a time. Being the case of XSB, the goals executed by the server could be tabled and take advantage of the shared tables implementation. Halting of the server is done by the thread cancellation mechanism, and a thread alias is used to make the server's thread identifier known to the worker threads. The alternative was to read the request in the server and check if it was a request to stop the server, in which case the server would just exit – however this solution would be problematic if the client had to exchange more messages with the server, other than just reading which goal to execute. Note that we create a specific thread for the main server loop, because the main thread cannot be cancelled.

The following example[4], Figure 3.4, uses a multi-threaded execution model to compute a series of prime numbers in parallel. The master thread partitions the work and creates two worker threads. The worker threads each compute its portion of the interval and return their results to the master through a message queue.

[4]This example was inspired on one in from the LogTalk manual [53] which is shown in section 6.1.

```
server :-
   socket(SockFD),
   socket_set_option( SockFD, linger, SOCK_NOLINGER ),
   xsb_port(XSBport),
   socket_bind(SockFD, XSBport),
   socket_listen(SockFD,Q_LENGTH),
   thread_create( server_loop(SockFD), Id, [alias(server)] ),
   thread_join( Id ).

server_loop(SockFD) :-
   socket_accept(SockFD, SockClient),
   thread_create( attend_client(SockClient) ),
   server_loop(SockFD).

attend_client(SockClient) :-
   socket_recv_term(SockClient, Goal),
   (  Goal == stop →
      thread_cancel( server ),
      socket_close( SockClient ),
      thread_exit
   ;  true
   ),
   (  is_valid(Goal) →
      call(Goal),
      socket_send_term(SockClient, Goal),
      fail,
   ;  socket_send_term(SockClient, invalid_goal(Goal))
   ),
   socket_send_term(SockClient, end),
   socket_close(SockClient).
```

Figure 3.3: A multi-threaded goal server in XSB

```
prime(P, I) :- I < sqrt(P),!.
prime(P, I) :- Rem is P mod I, Rem = 0, !, fail.
prime(P, I) :- I1 is I-1, prime(P, I1).

prime(P) :- I is P-1, prime(P, I ).

list_of_primes(I, F, Tail, Tail) :- I > F, !.
list_of_primes(I, F, [I|List], Tail) :-
    prime(I), !,
    I1 is I+1, list_of_primes(I1, F, List, Tail).
list_of_primes(I, F, List, Tail) :-
    I1 is I+1, list_of_primes(I1, F, List, Tail).

partition_space(N, H, H1) :-
    H is N//2, H1 is H+1.

worker( Q, Id, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    thread_send_message( Q, primes(Id,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    message_queue_create(Q),
    thread_create( worker(Q, p1, 1, H, L, L1) ),
    thread_create( worker(Q, p2, H1, N, L1, []) ),

    thread_get_message( Q, primes(p1,L,L1) ),
    thread_get_message( Q, primes(p2,L1,[]) ).
```

Figure 3.4: A multi-threaded program to generate prime numbers in XSB using term queues

Notice how the `primes/2` predicate uses difference lists to avoid the use of the append predicate[5], and while threads don't share variables, the bindings of the terms in the messages are correctly handled, allowing Prolog's unification to assume its full power. Although only two threads are used, the program could easily be extended to use an arbitrary number of threads

In Figure 3.5 we present the changes to the example for the worker threads to use the exit mechanism to communicate with the master. We only give the code for the main predicates, which are the only ones that are different from the last example.

[5]For a description on how to program with difference lists see a Prolog programming text, such as [71].

```
worker( I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    thread_exit( primes(List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    thread_create( worker(1, H, L, L1), W1 ),
    thread_create( worker(H1, N, L1, []), W2 ),

    thread_join( W1, exited(primes(L,L1)) ),
    thread_join( W2, exited(primes(L1,[])) ).
```

Figure 3.5: A multi-threaded program to generate prime numbers in XSB using the exit/join mechanism

3.3 Implementation Details

The development of the multi-threaded system without shared tables took four steps:

1. Supporting threads for pure Prolog – this involved supporting a private WAM instance for each thread.

2. Supporting the Prolog built-ins within multi-threading – this involved dealing with the data structures for a variety of system predicates such as assert and retract of shared predicates, I/O, findall, etc..

3. Supporting tabling without sharing tables – this involved adding the SLG-WAM state for each thread.

4. Support to efficiently access to private dynamic predicates and tables – this involved providing direct entry points to private tables and dynamic predicate clauses[6].

All these steps were validated by extensive testing[7].

In the following sections we describe each of these steps in detail, beginning with the support for Prolog, followed by extensions to support thread private tables. We conclude with the implementation of efficient dynamic private predicates, including tabling.

[6]The mechanism to support efficient access to private tables and dynamic predicates was proposed by Professor David S. Warren, which also did its initial implementation. Later Professor Terrance Swift proposed the use of private structure managers for private tables and did an initial implementation of them.

[7]The test-suites can be found at the XSB repository (`xsb.sourceforge.net`) under the module mttests.

3.3.1 Multi-Threading for Pure Prolog

The support for multi-threading for pure Prolog requires the introduction of the thread context. The thread context contains all the information that is relevant for the execution of the threads, except the stacks, and is used to access all data that is private to the thread. It includes a couple of hundred items. The most important information present is:

- The WAM and SLG-WAM registers (see Sections 2.3.1 and 2.3.2).
- The trie registers (used by the trie instructions when copying terms out of the tables – see Section 2.3.3).
- Data structures to support the copying of terms into the tables (i.e. building the tries). The trie data structures mentioned in Section 2.3.3 require several auxiliary stacks and register sets to be correctly initialized and maintained. Furthermore, table subsumption (which is not discussed in this thesis) requires *time-stamped* tries, which are supported by further auxiliary data structures.
- Data structures used in garbage collection of private dynamic predicates and tables.
- Exception handling data.
- The thread private system flags (e.g. stack reallocation and garbage collection modes, current input and output file descriptors and several other implementation flags).
- Pointers to the threads' private data structures shown in Figure 3.2.
- Pointers to the *structure managers* for tabled data structures – these are the memory management units that provide allocation for the table data structures.
- State variables for Shared Completed Tables (see Chapter 4).
- State variables for Concurrent Completion (see Chapter 5).

The Need for the Thread Context Data Structure The thread context data structure is needed because in the multi-threaded execution model all global variables are shared among all threads, and XSB cannot be made into a single function having only local variables as it state[8]. The option of using thread private data (global data with special access mechanisms), or any other global data structure, for this purpose was discussed but thought to be inefficient because of the need to use a mapping from the thread identifier to access the specific items as opposed to the dereferencing of the pointer that we actually use.

[8]For some time gcc supports thread local variables. However this implementation depends on support from the linker and it isn't clear how portable that is.

The Hypothesis of Changing XSB to C++ As each thread has its own execution context, an ingenious solution involving minimal changes in XSB code would be to change XSB to C++. Each thread would be an object, its state being the thread context and the XSB functions that used the thread context would be redefined as methods of the thread object.

This method was followed with some success at a stage but, alas, only on thanks to a "liberal" compiler, provided by Red Hat Linux 6.0. When we switched to a more conventional compiler it turned out that XSB uses features of C which are not a subset of C++, namely the ability to use void pointers without explicit casts in assignments to other pointer variables. Having to choose between major changes to XSB code and redoing the multi-threaded experience using the already gained knowledge about the thread context, this time in C, we took the latter choice.

Changes to XSB Code to Access the Thread Context In order to give access to the thread context to all functions that need it, we proceeded in the following way:

- An extra parameter is passed to every function that needs to access the thread context. This parameter is a pointer to a structure which contains all the variables in the thread context. This parameter is conditionally compiled in, only in the case of multi-threaded compilation.
- To keep the sequential system intact, all the context variables are defined as fields of a pointer to the context structure. This definition only gets in effect in case of a multi-threaded compilation.
- Conditional compilation ensured that the sequential system was not changed, while keeping a single source tree.

System Issues There were a couple of system details that have to be handled even for the execution of pure Prolog programs. The following list is by no means exhaustive, we give only some of the more fundamental changes:

- **Flags** XSB has an array of status flags that act has parameters for the working of the system. This was split into private and global flags. The main private flags are given in the previous section. The global flags store stuff such as the installation directory, configuration file and command line goal. There is a global flag that represents the number of threads in the system and is used in ensuring that there is only one thread active when performing certain actions.
- **Dynamic loading of predicates** XSB supports dynamic loading of predicates, i.e. when a predicate P of a module M_1 is imported by a module M_2, module M_1 is only loaded when P is called. This would lead to a problem when two threads try to load the module simultaneously. This problem is solved using a lock in order

to make the sequence from detecting a missing predicate to loading its module an atomic sequence.

- **Atom table** The atom table is global to the XSB process and access to it is controlled by a high-level mutex lock. The atom table can only be garbage collected when there is only one active thread.

- **Tagging arrays** When running under some operating systems XSB must use a tag array to map the different addresses returned by the operating system to different tags. Cells in this array are initialized on demand, when the memory allocation functions are called. The initialization of these cells must be an atomic operation as the array must be global to the process as its used to map the address space.

Supporting the Prolog Built-ins Lots of little things had to be changed to support the multi-threaded execution of XSB's private built-in predicates. We give only some of the more important ones that had to be changed:

- **The I/O predicates** The I/O streams are global to the Prolog process. The current I/O stream is stored in a thread private flag, and may be different for each thread. The I/O functions have file stream specific locks, however a global I/O mutex is used to access the file stream table.

- **The shared dynamic predicates** The data structures that store the shared dynamic code are protected for concurrent accesses through a high level mutex. Garbage collection of the retracted shared dynamic predicates is only done when there is only one active thread.

- **Findall** The buffers used to store the solutions of the findall family of predicates have been included in the thread context.

- **Exception handling** Exception handling is done within a thread, although the result of exceptions that are not handled by its thread can be examined by the joining thread (c.f. Section 3.2.1) The exception ball is passed through asserting a dynamic shared predicate, indexed by the thread identifier.

Supporting the Multi-Threading Primitives Some actions had be taken to support the primitives in the multi-threading API:

- **The thread table** – there is a global table with an entry for each thread. Each entry includes the pthread identifier, a pointer to the thread context, and several boolean flags to maintain and control the state of the thread (for example, whether the thread is detached or has exited). The pthread identifier enables to map Prolog thread identifiers into pthread identifiers when there is the need to use pthread primitives. The thread context allows for a thread's internal state to

be accessed from other thread, when the need arises (this happens in situations discussed in Chapters 4 and 5). The entries of the table are kept in two lists, one for used entries other for unused ones. This allows to keep the execution time of reserving an entry for a new thread or releasing an entry that is no longer needed, a constant factor.

- **Creation of a new thread** – creation of a new thread is done through the pthread_create primitive. The predefined "C" function that is used to run the thread creates its stacks and data structures, copies the initial goal into the stacks and accesses a predefined Prolog predicate that executes the initial goal for the thread. This Prolog predicate handles any uncaught exceptions and calls the thread exit primitive, whether the initial goal succeeds, fails or is interrupted by an exception. When thread aliases are specified, they are asserted as Prolog facts.

- **Thread exiting** – when a thread exits all its memory areas and any mutexes held by the thread are released, whether they are mutexes from the Prolog application or internal mutexes to the system. If the thread is detached its thread table entry is released, otherwise it is released only when the thread is joined. The pthread_exit primitive is called to terminate the thread. The exit status that is passed to a following join is asserted as a dynamic Prolog fact. If the thread is detached its aliases are removed.

- **Thread cancellation** – thread cancellation is not implemented via the pthread_cancel primitive. Instead an UNIX signal is send to the thread, and XSB's asynchronous signal handling mechanism is used to terminate the thread gracefully, via the thread exiting mechanism discussed in the previous item.

- **Term queues** – At the heart of the term queue implementation is the bounded buffer model, sometimes also called producer-consumer model. Its concurrent solution using pthreads is described in [42]. The terms are compiled into the queue slots using code initially designed for the `assert` built-in. Access to terms using non-strictly FIFO order (by unification with a particular term on the queue) is handled by a higher-level layer of Prolog code.

3.3.2 Support for Thread Private Tables

Support for tabling involved the continuation of the work described in the previous section, extending the thread context to include the SLG-WAM registers and stacks, including the support for delay and simplification for well-founded negation and the additional data structures used by tabled subsumption.

We took some effort in guaranteeing that we would minimize the need for locking with thread private tables; at this point we'd like to mention a difference with respect

to the work reported in [62] – for thread private tables, the call tries are not shared by threads so we don't implement locking at the trie level. The same is true for the answer tries. The data structures that support the conditional answer information delay lists are also private to the thread. This conditional answer information includes delay lists, delay elements, answer substitution information, back-pointer lists and delay tries.

The *structure managers*, XSB's basic mechanism to allocate memory for the tabled data structures, have a private version, for the allocation of private tables. This allows to avoid locks in the allocation of space for new tabled data structures (except for the locks managed by the "C" runtime system, at the few times that malloc must be called).

Garbage collection of abolished private tables follows the same scheduling strategy as garbage collection of abolished tables in the sequential engine; however for shared tables it is only done when there is only one thread active. Likewise, garbage collection of retracted predicates, is done on a regular basis for thread private predicates, but for shared predicates it's only done when there is only one thread active.

The Table_Try and Check_Complete Instructions The table_try and check_complete instructions are the same for private and shared tables. In the case of the check_complete this is necessary to correctly support the evaluation of SCCs that contain both shared and private table; the reasons for this will be clear after reading the following chapters, specially Sections 4.1 and 5.1. In the case of the table_try instruction we could have provided an optimized instruction for private tables, but as this instruction's code is very complex (much more complex then the regular WAM instructions) the code bloating and other complications involved were judged to be more significant than avoiding a few memory accesses and if statements.

Support for Efficient Access to Thread Private Tables Support for efficient access to the thread private tables' table information frame (TIF), which represents a tabled predicate (cf. Section 2.3.3) is done through a *Thread Dispatch Block* which is an array of pointers to the real table information frames for each tabled predicate. At the beginning of the table_try instruction, the thread dispatch block is used to find the thread's table information frame for that predicate. For this to be possible, the thread dispatch block mimics the table information frame's in its first fields, and is passed to table_try as if it were a regular table information frame. The table_try instruction finds that the predicate is being evaluated through thread dispatch block and follows the indirection in the array to the real table information frame. Figure 3.6 shows the data structures for the private tables.

The thread dispatch blocks are chained in a list, so that when a thread exits it may reclaim the space for its private tables. The thread dispatch blocks are initialized when loading a program. The table information frames are created on demand, when a tabled call to a predicate happens.

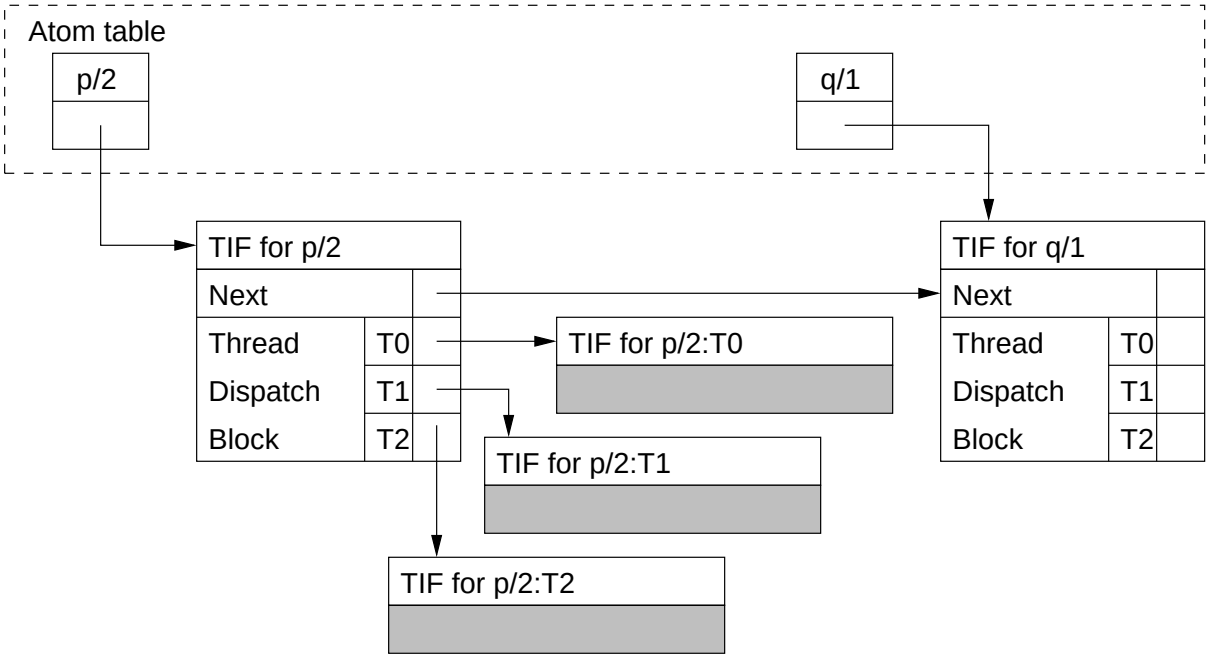

Figure 3.6: Data structures for thread private tabled predicates

Support for Private Dynamic Predicates In XSB dynamic predicates are accessed through a *prref* (predicate reference block), which contains a WAM instruction to branch into the code of the predicate. In the case of private dynamic predicates this instruction is a switch_on_thread WAM-like instruction, which uses the current thread identifier as an index in the thread dispatch block to branch into the WAM code that corresponds to the clauses that belong to this thread.

When a thread exits, it follows the thread dispatch block chain to reclaim its clauses. The thread dispatch block for a thread private predicate is created by the first thread that finds that that predicate is dynamic and thread private.

Thread Private Dynamic Tabled Predicates In XSB, assert of tabled predicates is done through a program transformation, that implements a tabled predicated with multiple clauses as a tabled predicate with a single clause that calls the correspondent non-tabled predicate with multiple clauses. So, the entry point for a thread private dynamic tabled predicate has a table_try_single instruction for each thread. This instruction includes a pointer to the table information frame for that predicate. It is important to note that, in this case, although the tables are private, the table information frame doesn't correspond to a thread dispatch block but to a regular table information frame. The data structures for thread private dynamic tabled predicates are illustrated in Figure 3.7.

The table information frame is created when the thread first finds that the predicate is dynamic and tabled.

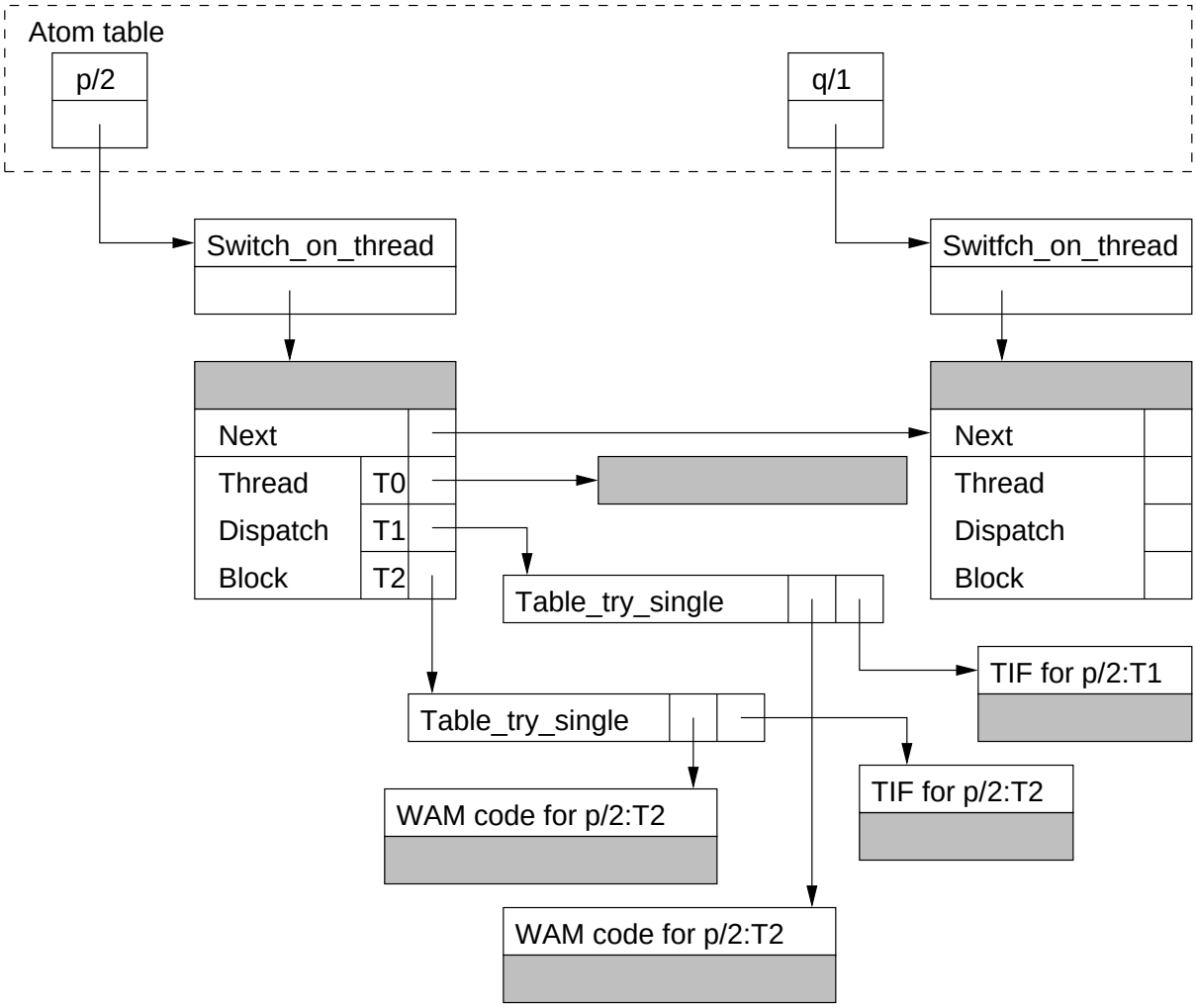

Figure 3.7: Data structures for thread private tabled predicates

Chapter Summary in this chapter we introduced the fundamental concepts for multi-threaded programming, including its interpretation for the Prolog language. We presented the model of execution and programming of multi-threaded XSB which is largely based on the current ISO proposal for Prolog threads but also includes tabling. We discussed the design and implementation decisions, and presented some details about the actual implementation, including the efficient support for private dynamic predicates and tables.

4

Shared Completed Tables

In this chapter we describe a simple way of sharing tables among threads. We call this the Shared Completed Tables as it's based on only sharing tables that have already been completed.

Every table is owned by a thread that computes all its answers. Only when the table is completed may other threads read the answers from the table. When calling a tabled subgoal that is owned by another thread and which is not completed, the thread suspends until that subgoal is completed.

```
:- table p/1, q/1.
:- thread_shared p/1, q/1.

p(X) :- q(X).

q(a).
q(b).
q(c).
```

Figure 4.1: Program $P_{4.1}$

See Figure 4.1. Assume that thread *T0* calls query `?- p(X), fail` and thread *T1* calls query `?- q(X), fail`.

Scenario 1 Assume that the calls are done at the same time, or that *T1* calls the subgoal `q(X)` before *T0*. In this case *T0* will own the table for the subgoal `p(X)` and *T1* will own the table for the subgoal `q(X)`. Only after *T1* computes all the answers to `q(X)`, will the call to `q(X)` in *T0* return its first answer.

Scenario 2 Assume that *T0* calls the subgoal `q(X)` before *T1*. In this case *T0* will own both tables, for `p(X)` and `q(X)`. The call to `q(X)` in *T1* will only return its first answer after *T0* computes all the answers to `q(X)`

This simple example shows that it doesn't matter which thread owns a tabled predicate, the result of the queries is the same to all ordering of events, only the order in which the threads return their results changes.

However the idea of suspending the thread until the table owned by another completes has a catch. If there's a cycle in the SDG of the tabled program and tables are owned by different threads a deadlock situation will occur.

```
:- table p/1, q/1.
:- thread_shared p/1, q/1.

p(X) :- q(X).
p(a).

q(X) :- p(X).
q(b).
```

Figure 4.2: Program $P_{4.2}$

See Figure 4.2. Assume that thread *T0* calls query `?- p(X)` and thread *T1* calls query `?- q(X)`. Assume that both calls are simultaneous i.e. *T0* calls subgoal `p(X)` at the same time that *T1* calls subgoal `q(X)`.

When *T0* goes ahead and calls `q(X)` it will suspend as `q(X)` is owned by *T1* and its not complete. When *T1* calls `p(X)` it will also suspend because `p(X)` is owned by *T0* and its not complete. A deadlock situation arises.

To solve that problem, before a thread suspends on a subgoal it checks if this will create a cycle so that a deadlock would arise. If this happens a deadlock breaking algorithm is invoked. The thread that detected the cycle (*T1* in the above case) is called the *leader* and its responsible for breaking the deadlock.

In the above case the deadlock would be resolved by resetting *T0* to the call `q(X)` and giving ownership of the table of subgoal `q(X)` to the leader (thread *T1*). The table for `q(X)` would be emptied, and thread *T1* would recompute all the solutions for `q(X)`. *T0* would only be restarted when the re-computation of all the answers of `q(X)` (which involve computing the solution for `p(X)` too) would be complete.

4.1 Changing the SLG-WAM to support Shared Completed Tables

In this section we describe the changes to the SLG-WAM to allow the sharing of completed tables.

Some data structures of the concurrent SLG-WAM have to be extended to include some new fields to support the deadlock breaking algorithm (see Section 4.1.1).

The changes to the SLG-WAM instruction set to implement the Shared Completed Tables are located in the table_try instruction (see section 4.1.2), which is executed when calling a subgoal. The instruction has to check if the called subgoal is owned by the current thread. If not, it has to suspend until that subgoal is completed.

Our algorithm detects this deadlock situation and breaks the deadlock, by given ownership of all the tables in the cycle to a leader thread, which will re-compute any tables that were already being computed by other threads. The other threads will be reset to the computation of the subgoal which is the parent of the reallocated subgoal. This is only possible under local scheduling, under which the answers are only returned to the parent subgoal after completion.

To find that there's a cycle is a relatively simple matter. We keep the graph dependencies among currently suspended threads (which has only one edge going out of each node) and, whenever a thread suspends, check for a loop (see Section 4.1.3). We call the subgoal that detects the deadlock the *leader*.

To implement the breaking of the deadlock, the table_try instruction has to be restarted when a deadlock involving the suspended thread is found. This is done through the condition variable *completing_cond*. The table_try instruction checks if it was restarted by the reset boolean field of the thread data structure. If this is set the table_try instruction is aborted and the next SLG-WAM instruction is executed. This is set by the leader thread through the procedure reset_other_threads (see Section 4.1.4).

4.1.1 Data Structures

We use two global variables to control concurrency:

completing_mut A mutex that ensures mutual exclusion for the algorithm, so that only one thread may be executing it at a time.

completing_cond A condition variable where the threads wait for a subgoal to complete.

Thread Context We extend the thread context (as given in Section 3.3.1) with the following fields:

waiting_for_subgoal Subgoal frame that the thread is waiting for to complete. Instantiated when a thread makes a call to a subgoal that belongs to other thread.

waiting_for_thread context of the thread on which this thread depends on. This could be omitted and the *tid* field of the subgoal frame given by *waiting_for_subgoal* be used to compute it.

reset This boolean field indicates to the algorithm that the thread has been reset and that control flow should proceed from the following SLG-WAM instruction.

is_leader This boolean field is set to true when a thread becomes the leader in the deadlock breaking algorithm. It's used so that subgoals that have been grabbed can be called by this thread.

We assume that all functions have the local parameter *th* so that they can access the thread context (cf. Section 3.3.1).

Subgoal Frame The Subgoal Frame, defined in Section 2.3.3 is extended with the following fields:

grabbed This boolean field, if set to true means that this subgoal frame is not being used to generate answers, but is reserved for the leader thread who will later recompute this subgoal.

tid An integer which denotes the thread to which this subgoal belongs.

Generator Choice Point the Generator Choice Point, defined in Section 2.3.2 is extended with the field:

reset_pcreg which is the address of the table_try instruction that created this generator choice point. It is set in the table_try instruction and it is used as the forward continuation when a thread that was reset is restarted.

4.1.2 The Table_Try Instruction

The table_try instruction (see Figure 4.3) is the SLG-WAM instruction responsible for calling a subgoal. The main change to this instruction for Shared Completed Tables is the introduction of the first if (line 1 in Figure 4.3). This code can only be executed by a thread at a time, i.e. under mutual exclusion. Although this imposes a big restriction on parallel execution of a tabled program, it must be done be cause of the deadlock detection (see procedure would_deadlock in the next section). The lock completing_mut ensures that condition.

If the subgoal is not a new one, the while (line 1.1) loop is executed. We note that this loop is a programming device; the same result could have been obtained by different nestings of if statements, however a while would always be needed somewhere.

This loop may end for a number of conditions. The first is if the subgoal is completed; in that case the thread may go ahead and share the table of the subgoal – this is the basis of Shared Completed Tables.

Another condition (line 1.1.1) for breaking this loop is if the thread is the leader in the breaking of a loop and the subgoal has been grabbed for re-computation. We say

```
Instruction table_try(Arity, NextClause, TIF)          /* Subgoal is in argument registers */
    shared ← table_is_shared(TIF.Shared);
    if( shared ) lock(completing_mut);
    grabbed ← false ;
    SF ← subgoal_check_insert(Subgoal, TIF);
1   if( shared and SF ≠ NULL )                          /* if the Subgoal is already in the table */
1.1     while( not SF.is_completed )
         /* if is leader and subgoal is marked to be computed by leader */
1.1.1       if( th.is_leader and SF.grabbed )
                SF.tid ← th.tid ;
                SF.grabbed ← false ;
                grabbed ← true ;
                break ;
            table_tid ← SF.tid ;
1.1.2       if (table_tid = th.tid)
                break;                                  /* if the thread owns the table, proceed */
            waiting_for_thread = context(table_tid) ;
1.1.3       if( would_deadlock( waiting_for_thread, th ) )
                reset_other_threads( waiting_for_thread, SF );
                th.is_leader ← true ;
                continue ;
            th.waiting_for_thread ← waiting_for_thread ;
            th.waiting_for_subgoal ← SF ;
            cond_wait(completing_cond,completing_mut) ;
1.1.4       if( th.reset )                              /* if it was reset by leader */
                th.reset ← false;
                unlock(completing_mut) ;
                 /* Branch to next SLG-WAM instruction */
        th.waiting_for_thread ← NULL ;
        th.waiting_for_subgoal ← NULL ;
        unlock(completing_mut);
2   if ( SF = NULL )                                    /* the subgoal is not in the table */
        SF ← CreateSubgoalFrame(Subgoal);
        SF.tid ← th.tid;
        SF.grabbed ← false;
        if( shared ) unlock( completing_mut );
        GCP ← PushGeneratorChoicePoint(...);
        ...     /* actions in lines 2.3–2.6 of Figure 2.19 */
3   else if ( SF.IsCompleted )
        Answer_Root ← Subgoal.AnsTrieRoot;
         /* Branch to Answer_Root SLG-WAM instruction */
4   else if ( grabbed )  /* SF already exists and the subgoal will be restarted */
        GCP ← PushGeneratorChoicePoint(...);
        ...     /* actions in lines 2.3–2.6 of Figure 2.19 */
5   else
        CCP ← PushConsumerChoicePoint(...);
        ...     /* actions in lines 4.2–4.7 of Figure 2.19 */
```

Figure 4.3: The table_try instruction.

```
would_deadlock( subg_thread, current_thread )
    t ← subg_thread
    while( t ≠ NULL )
        if( t = current_thread )
            return true ;
        else
            t ← t.waiting_for_thread;
    return false ;
```

Figure 4.4: The would_deadlock function.

the subgoal has been *grabbed* when the thread that was computing it got involved in a deadlock and the subgoal must be recomputed to break the deadlock. In this case this thread will recompute the table for this subgoal.

Then it follows (line 1.1.2) one obvious condition to break the loop, which is if the subgoal is owned by this thread. In that case the thread will normally compute the subgoal.

After checking for breaking the loop, and following with a "normal" table_try instruction, there is the deadlock check (line 1.1.3). This is done by the function would_deadlock discussed below.

In the case that a deadlock would arise from waiting for the current subgoal the thread becomes the *leader* for breaking this deadlock and resets the computations of other threads which were blocked while computing subgoals involved in the deadlock. This is done by the procedure reset_other_threads which is discussed below. After resetting the other threads they are awaken by a cond_broadcast. In this case the loop is re-entered so that the leader can continue by recomputing one of the subgoals of the reset threads.

If blocking the thread would not create a deadlock, the thread sets its fields *waiting_for_thread* and *waiting_for_subgoal* so that new deadlocks can be detected and blocks itself on the condition variable.

When it awakes (line 1.1.4) it checks to see if it has been reset; if yes it skips the rest of the table_try instruction, continuing to execute SLG-WAM instructions in the *pcreg* location to where it was reset to in reset_other_threads.

When a new subgoal is created (line 2) the owner thread is set to be the current thread and the *grabbed* field is initialized to false.

4.1.3 Detecting a Deadlock

As it was said above, detecting a deadlock is not especially difficult. We keep a field *waiting_for_thread* in each thread that is blocked waiting for the completion of subgoal being computed by another thread.

The thread data structures augmented with that field form a graph, but one in

which each node only has at most one outgoing edge. We just follow the path in the graph starting in the current thread and find if it forms a cycle. This is shown in Figure 4.4.

The problem with this code is that it and the maintenance of the graph, given on the previous and next sections, access the graph that keeps the dependencies among threads and have to be executed under mutual exclusion. If not, the following situations might arise:

- Two (or more) threads might try to suspend on a subgoal that would proudce a deadlock and would "think" they were the leader, leading to a situation in which there would be more than one leader for the same SCC.

- Two threads (or more) might try to block on two (or more) subgoals that would produce a cycle. In that case it might happen that none would detect the deadlock, with obvious consequences.

4.1.4 Resetting the other Threads in the Deadlock

```
reset_other_threads( ctxt, sgf )
    reset_thread( ctxt, sgf, reset_sgf );
    while( ctxt ≠ NULL )
        next ← ctxt.waiting_for_thread;
        ctxt.is_leader ← false ;
        ctxt.waiting_for_subgoal ← reset_sgf ;
        ctxt.waiting_for_thread ← th ;
        if( next ≠ th )
            reset_thread( next, ctxt.waiting_for_subgoal, reset_sgf );
        ctxt ← next ;
```

Figure 4.5: The reset_other_threads procedure.

The procedure reset_other_threads (see Figure 4.5) follows the cycle in the thread waiting graph and just calls the procedure reset_thread for each thread in the cycle, but for the leader thread. reset_thread is called with the subgoal which is depended on that thread, and which the thread will be reset to.

Procedure reset_thread (see Figure 4.6) resets a thread so that a subgoal, and all subgoals called directly or indirectly by it can be recomputed by another thread.

It starts with a check to see if this subgoal was already given to another thread - this might by needed in case of multiple deadlock detections, for example in a case where the number of threads involved in a deadlock is growing and an already reset thread is still blocked,

Next the bottom_leader (Figure 4.8) function is used to find out to which subgoal the thread will really be reset to. It happens that threads can only be reset under local

```
reset_thread( ctxt, sgf, reset_sgf )
 /* if the subgoal has not yet been computed, the thread should not be reset */
    if( sgf.grabbed )
        sgf.tid ← th.tid ;
        return ;
    ctxt.reset ← true ;
    sgf ← bottom_leader(ctxt, sgf) ;
    reset_sgf ← sgf ;
    ReclaimDSAndMarkReset(ctxt, sgf, th.tid);
    th ← ctxt ;                                  /* trick to use other thread's context */
    RestoreChoicePoint(generator_cp(sgf)) ;      /* this effectively resets the stacks */
    pcreg ← breg.reset_pcreg ;
    breg ← breg.prevbreg) ;                      /* delete the generator cp */
```

Figure 4.6: The reset_thread procedure.

scheduling, because each subgoal has only one caller outside its SCC. It follows that when resetting a subgoal that is not the leader of an SCC we have to reset everything up to the leader. This function just follows the completion stack downwards until it finds the leader of the current SCC.

The ReclaimDSandMarkReset procedure (Figure 4.7) reclaims the table data structures and marks the subgoal frames as grabbed for re-computation. It follows the completion stack from its top until the subgoal we are resetting the thread to.

Next the generator choice point for the subgoal we are resetting the thread to is restored (a delicate operation for which we give no detail) and the *pcreg* is restored to the start of that subgoal. The choice point is deleted and the thread will restart by suspending until that subgoal completes.

```
ReclaimDSandMarkReset(ctxt, to, leader)
    csf ← ctxt.openreg ;
    for(;;)
        csf.subg_ptr.grabbed ← true ;
        csf.subg_ptr.tid ← leader ;
        abolish_tables_but_keep_sgf(csf.subg_ptr) ;
        if( csf.subg_ptr = to )
            break ;
        csf − −;
```

Figure 4.7: The ReclaimDSandMarkReset procedure.

4.1.5 The Check_Complete Instruction

The check_complete instruction must wake the threads waiting for the subgoals to complete in the completing_cond condition variable. Marking the subgoals as completed is

```
bottom_leader(to_sgf)
    csf ← to_sgf.compl_stack_ptr ;
    while( not is_scc_leader(csf) )
        csf − − ;
    return csf.subg_ptr ;
```

Figure 4.8: The bottom_leader function.

```
Instruction check_complete
    Subgoal ← breg.SubgFr;                    /* breg points to the Generator CP of Subgoal */
    SubgCSF ← Subgoal.ComplSF;
    if (is_leader(SubgCSF) )
        . . .
        for each CSF ∈ [SubgCSF..openreg]
            CSF.SubgFrame.IsComplete ← true;
            if (not has_answers(SF) )
                for each delay list where SF appears
                    /* Simplify negation success */
        cond_broadcast(completing_cond) ;
        . . .
    else                                      /* it is not a leader */
        MakeConsumerFromGenerator(breg);
    breg ← breg.Breg_Chain;
    Fail;
```

Figure 4.9: The check_complete instruction.

done under mutual exclusion with the deadlock breaking algorithm.

4.2 A Semantics for Shared Completed Tables

4.2.1 SLG_{sct}: an Operational Semantics for Shared Completed Tables

In this section we modify SLG for Shared Completed Tables. We re-use most of the definitions and use the terminology and assumptions introduced in section 2.2.

Namely we still use Definition 2.2.1 for the SLG_{sct} trees, which are now however marked with a thread identifier.

Definition 4.2.1 (TID) *SLG_{sct} trees are marked with a Thread Identifier. This may be an integer which denotes the thread that is evaluating the tree or the token* complete. *We denote the marker for tree T as $TID(T)$. For each node N in tree T we use $TID(N)$ as an alias for $TID(T)$. We graphically represent this markers by appending $[TID(T)]$ to the root node of T.*

Every subgoal that is not completed is said to belong to a *thread* according to its TID

Definition 4.2.2 (Thread) *We define a thread state as the subset of trees marked with the same TID in an SLG_{sct} forest. We define a thread as the sequence of thread state sub-forests in a SLG_{sct} evaluation. We say a thread is active in a forest if its thread state for that forest is not empty.*

Definition 4.2.3 (Thread Compatible) *Let N be a node and T a tree for subgoal S. N is thread compatible with S if $TID(T) =$* completed *or $TID(T) = TID(N)$.*

Definition 4.2.4 (Deadlock) *A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is in* deadlock *if:*

1. *For each $S \in \mathcal{S}$ there are no applicable* NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN, DELAYING, *or* NEGATIVE RETURN *operations (Definition 4.2.7) for N.*

2. *$\exists S_1, S_2 \in \mathcal{S}$ such that S_1 has a leaf node N that is SLG resolvable with an answer A of S_2.*

A set of subgoals is completely evaluated when it can produce no more answers. Formally,

Definition 4.2.5 (Completely Evaluated) *A set $\mathcal{S}$ of subgoals in a forest $\mathcal{F}$ is* completely evaluated *if at least one of the conditions holds for each $S \in \mathcal{S}$*

1. *The tree for S contains an answer S :- |; or*

2. *For each node N in the tree for S:*

   (a) *The selected literal L_S of N is completed or in $\mathcal{S}$; or*

(b) *There are no applicable* NEW SUBGOAL, PROGRAM CLAUSE RESOLUTION, POSITIVE RETURN, DELAYING, NEGATIVE RETURN *or* USURPATION *operations (Definition 4.2.7) for N.*

Once a set of subgoals is determined to be completely evaluated, the COMPLETION operation marks the trees for each subgoal (Definition 2.2.1).

Definition 4.2.6 (Multi-threaded Tabled Evaluation) *Given a program P, a set of atomic queries* $Q = \{Q_1 \ldots Q_n\}$ *and a set of tabling operations (from Definition 4.2.7), a multi-threaded tabled evaluation* $\mathcal{E}$ *is a sequence of SLG forests* $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_\beta$*, such that:*

- $\mathcal{F}_0$ *is the forest containing the set of trees* $\{Q_1 \text{ :- } |Q_1[1] \ldots Q_n \text{ :- } |Q_n[N]\}$
- *For each successor ordinal,* $n+1 \leq \beta$*,* $\mathcal{F}_{n_1}$ *is obtained from* $\mathcal{F}_n$ *by an application of a tabling operation.*

If no operation is applicable to $\mathcal{F}_\alpha$*,* $\mathcal{F}_\alpha$ *is called a* final forest *of* $\mathcal{E}$*. If* $\mathcal{F}_\beta$ *contains a leaf node with a non-ground selected negative literal, it is* floundered.

We call Q_i *the root subgoal for thread i.*

These operations are as follows.

Definition 4.2.7 (*SLG*$_{sct}$ Operations) *Given a forest* $\mathcal{F}_n$ *of a SLG evaluation of program P and query Q, where n is a non-limit ordinal,* $\mathcal{F}_{n+1}$ *may be produced by one of the following operations.*

1. NEW SUBGOAL*: Let* $\mathcal{F}_n$ *contain a non-root node*

$$N = Ans \text{ :- } DelaySet|G, Goal\_List$$

where G is the selected literal S or not *S. Assume* $\mathcal{F}_n$ *contain no tree with root subgoal S. Then add the tree* $S \text{ :- } |S[TID(N)]$ *to* $\mathcal{F}_n$*.*

2. PROGRAM CLAUSE RESOLUTION*: Let* $\mathcal{F}_n$ *contain a root node* $N = S \text{ :- } |S$ *and C be a program clause* $Head \text{ :- } Body$ *such that* $Head$ *unifies with S with mgu* θ*. Assume that in* $\mathcal{F}_n$*, N does not have a child* $N_{child} = (S \text{ :- } |Body)\theta$*. Then add* N_{child} *as a child of N.*

3. POSITIVE RETURN*: Let* $\mathcal{F}_n$ *contain a non-root node N whose selected literal S is positive. Let* Ans *be an answer node for S in* $\mathcal{F}_n$ *such that N is thread compatible (Definition 4.2.3) with S and* N_{child} *be the SLG resolvent of N and* Ans *on S. Assume that in* $\mathcal{F}_n$*, N does not have a child* N_{child}*. Then add* N_{child} *as a child of N.*

4. NEGATIVE RETURN*: Let* $\mathcal{F}_n$ *contain a leaf node*

$$N = Ans \text{ :- } DelaySet|not\ S, Goal\_List.$$

whose selected literal not *S is ground such that N is thread compatible (Definition 4.2.3) with S.*

(a) NEGATION SUCCESS: *If S is failed in $\mathcal{F}$, then create a child for N of the form: Ans :- DelaySet|Goal_List.*

(b) NEGATION FAILURE: *If S succeeds in $\mathcal{F}$, then create a child for N of the form* fail.

5. DELAYING: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|not\ S, Goal\_List$, such that S is ground, in $\mathcal{F}_n$, but S is neither successful nor failed in $\mathcal{F}_n$. Then create a child for N of the form Ans :- $DelaySet, not\ S|Goal\_List$.*

6. SIMPLIFICATION: *Let $\mathcal{F}_n$ contain a leaf node $N = Ans$:- $DelaySet|$, and let $L \in DelaySet$*

   (a) *If L is failed in $\mathcal{F}$ then create a child* fail *for N.*

   (b) *If L is successful in $\mathcal{F}$, then create a child Ans :- $DelaySet'|$ for N, where $Delay\_Set' = Delay\_Set - L$.*

7. COMPLETION: *Given a completely evaluated set $\mathcal{S}$ of subgoals (Definition 4.2.5), mark the trees for all subgoals in S as completed.*

8. ANSWER COMPLETION: *Given a set of unsupported answers $\mathcal{UA}$, create a failure node as a child for each answer $Ans \in \mathcal{UA}$.*

9. USURPATION: *Given a set of subgoals $\mathcal{S}$ in deadlock (Definition 4.2.4) mark all trees of $\mathcal{S}$ with $TID(S), S \in \mathcal{S}$*

An interpretation induced by a forest (Definition 2.2.7) has its counterpart for SLG, (Definition 5.2 of [13]). Using these concepts, we can relate SLG to SLG_{sct} evaluations.

Theorem 4.2.1 (Correctness of SLG_{sct}) *Let P be a finite program and $\mathcal{Q}$ a set of atomic queries. Then a finite SLG_{sct} evaluation of $\mathcal{Q}$ against P exists with final state $\mathcal{F}^{sct}$, if (completeness) and only if (soundness), for every $Q_i \in \mathcal{Q}$ there exists a finite SLG evaluation of Q_i against P with final state $\mathcal{F}^i$ and $I_{\mathcal{F}^{sct}} = (\bigcup I_{\mathcal{F}^i})$.*

Definition 4.2.8 (depends) *Let $\mathcal{F}$ be a forest. We say that a subgoal S1* directly depends *on S2 if and only if S2 is the selected literal on a tree with root S1. The transitive closure of the* directly depends on relation *is the* depends on *relation. This is a slight modification to the* depends on *relation of Definition 2.2.9*

Proof:

Soundness: Let $P' = P \cup \{Q_0 \text{ :- } Q_1 \ldots Q_0 \text{ :- } Qn\}$ with Q_0, an atom such that $Q_0 \notin H_P$. Assuming that $\forall Q_i$ against P there is a finite SLG evaluation then there is a finite SLG evaluation $\mathcal{E}'$ of Q_0 against P' such that the final SLG Forest $\mathcal{F}'$ will have an interpretation $I_{\mathcal{F}'}$.

For each Q_i there will be an evaluation $\mathcal{E}'|Q_i$ with final forest $\mathcal{F}'|Q_i$ which is a subsequence of $\mathcal{E}'$, constructed considering only the operations which affect trees for the subgoals that Q_i *depends on* (Definition 4.2.8) given the final forest $\mathcal{F}'$. Note that there

is an SLG evaluation $\mathcal{E}^i$ of Q_i against P with final forest $\mathcal{F}^i$ having exactly the same sequence of $E'|Q_i$ where the forests are restricted to the trees of the subgoals that Q_i *depends on* in $\mathcal{F}'$ since at each stage any operation of $E'|Q_i$ is an applicable operation for that forest. Clearly $\mathcal{E}'|Q_i$ is finite and $I_{\mathcal{F}^i} = I_{\mathcal{F}'|Q_i}$.

Let T_0 be the SLG tree with root Q_0 :- $|Q_0$ in $\mathcal{F}'$. From the previous paragraph and Theorem 2.2.1 $\bigcup I_{\mathcal{F}^i} = I_{\mathcal{F}'} \setminus I_{\{T_0\}}$.

We prove that for every SLG_{sct} evaluation $\mathcal{E}_{SCT}$ of $\bigcup^N \{Q_i\}$ against P with a final forest $\mathcal{F}_{SCT}$, there is an SLG evaluation $\mathcal{E}'$ of Q_0 against P' that produces a final forest $\mathcal{F}'$ such that $\mathcal{F}_{SCT} = \mathcal{F}' \setminus \{T_0\}$ disregarding the TID tags which are meaningless for interpretations.

Induction base: The forest $\mathcal{F}_0 = \{T_0[N], Q_1 \text{ :- } |Q_1[1] \ldots Q_N \text{ :- } |Q_N[N]\}$ is obtained from the forest $\mathcal{F}'_0 = \{Q_0 \text{ :- } |Q_0\}$ by applying PROGRAM CLAUSE RESOLUTION N times and by then applying NEW SUBGOAL N times.

Induction hypothesis: For $\mathcal{F}^0_{SCT} \ldots \mathcal{F}^k_{SCT}$ there is an SLG evaluation $\mathcal{E}' = \mathcal{F}'_0 \ldots \mathcal{F}'_i$ where the trees are arbitrarily marked with the TID.

Induction thesis: For $\mathcal{F}^0_{SCT} \ldots \mathcal{F}^{k+1}_{SCT}$ there is an SLG evaluation $\mathcal{E}' = \mathcal{F}'_0 \ldots \mathcal{F}'_j$ where the trees are arbitrarily marked with the TID.

From the $\mathcal{F}^k_{SCT}$ which is a forest $\mathcal{F}'_i$ of $\mathcal{E}'$ (induction hypothesis) marked with TID we obtain $\mathcal{F}^{k+1}_{SCT}$ by applying one of the tabling operations of Definition 4.2.7. We show that $\mathcal{F}^{k+1}_{SCT}$ corresponds either to $\mathcal{F}'_i$ or $\mathcal{F}'_{i+1}$ either of which corresponds to $\mathcal{F}'_j$.

1. NEW SUBGOAL: Similar to NEW SUBGOAL of Definition 2.2.8 except that it marks the new tree with a TID.

2. PROGRAM CLAUSE RESOLUTION: Similar to PROGRAM CLAUSE RESOLUTION of Definition 2.2.8.

3. POSITIVE RETURN: Similar to POSITIVE RETURN of Definition 2.2.8 if it can be applied.

4. NEGATIVE RETURN: Similar to NEGATIVE RETURN of Definition 2.2.8 if it can be applied.

5. DELAYING:Similar to DELAYING of Definition 2.2.8.

6. SIMPLIFICATION: Similar to SIMPLIFICATION of Definition 2.2.8.

7. COMPLETION: Similar to COMPLETION of Definition 2.2.8. $\mathcal{F}^{k+1}_{SCT}$ corresponds to $\mathcal{F}'_{i+1}$, as in all of the above cases.

8. ANSWER COMPLETION: Similar to ANSWER COMPLETION of Definition 2.2.8.

9. USURPATION: This only changes the TID. $\mathcal{F}^{k+1}_{SCT}$ corresponds to $\mathcal{F}'_i$.

Completeness: Given the SLG evaluations $\mathcal{E}_1 \ldots \mathcal{E}_N$ of $Q_1 \ldots Q_N$ against P with final forest $\mathcal{F}_1 \ldots \mathcal{F}_N$, we construct a SLG_{sct} evaluation of $\{Q_1 \ldots Q_N\}$ against P as follows:

1. The initial forest consists of $\{Q_1 \text{ :- } |Q_1 \ldots Q_N \text{ :- } |Q_N\}$.

2. The initial operations consist in the operations of $\mathcal{E}_1$. If any POSITIVE RETURN or NEGATIVE RETURN operation involves the tree for $Q_2 \ldots Q_N$ an USURPATION operation is used before it to mark that tree with $TID = 1$. At the end all the trees in $\mathcal{F}_1$ are completely evaluated.

3. Repeat the operations for $\mathcal{E}_i$. If any POSITIVE RETURN or NEGATIVE RETURN operation involves the tree for $Q_{i+1} \ldots Q_N$ an USURPATION operation is used before it to mark that tree with $TID = i$. If any operation involves changing a tree in $\mathcal{F}_1 \ldots \mathcal{F}_{i-1}$ (note that any tree preset in $\mathcal{E}_i$ will be present in $\mathcal{F}_i$) then that tree is already completely evaluated and the operation is omitted. If any POSITIVE RETURN or NEGATIVE RETURN operation involves a tree in $\mathcal{F}_1 \ldots \mathcal{F}_{i-1}$ then there's no problem as that tree is completed so the nodes are *thread compatible*. At the end all the trees in $\mathcal{F}_1 \ldots \mathcal{F}_i$ are completely evaluated.

4. The final forest $\mathcal{F}_{SCT} = \bigcup^N \mathcal{F}_i$, where the TID markers are all instantiated to *complete*. So $I_{\mathcal{F}_{SCT}} = \bigcup^N I_{\mathcal{F}_N}$.

The above shows that for any SLG evaluations of $Q_1 \ldots Q_N$ against P is always possible to construct a SLG_{sct} evaluation of $\{Q_1 \ldots Q_N\}$ against P with the same final interpretation.

∎

4.2.2 Local Multi-Threaded Evaluations

In SLG_{sct} the SDG (Definition 2.2.9) can be partitioned in disjoint sub-graphs for each thread state of a forest. We recall Definition 2.2.10 but in a local SLG_{sct} forest there will be one independent SCC for each thread state's sub-graph.

Definition 4.2.9 (Thread Subgoal Dependency Graph) *For each thread state, the* Thread Subgoal Dependency Graph *(Thread SDG) for a forest consists of the sub-graph of the SDG which contains all the nodes reachable from the node for the thread root subgoal.*

Definition 4.2.10 (Local SLG_{sct} evaluation) *We say that $\mathcal{E}$ is a local SLG_{sct} evaluation if $\mathcal{E}$ satisfies the locality property (Definition 2.2.11) for each Thread SDG and each* USURPATION *operation is applied an entire SCC of an active thread's sub-graph.*

Theorem 4.2.2 *If an SLG_{sct} evaluation $\mathcal{E} = \mathcal{F}_0 \ldots \mathcal{F}_n$ with m threads satisfies the locality property than $\forall 0 \leq k \leq n : \text{SDG}(\mathcal{F}_k)$ has one and only one independent SCC for each active Thread SDG in $\mathcal{F}_k$.*

Proof: The proof is similar to the one for Theorem 2.2.2 with the addition, in the inductive step, of a case to the USURPATION operation. The USURPATION operation's proof is trivial for the usurping thread (the independent SCC gets extended) and for the usurped thread it is similar to the case of the COMPLETION operation (the independent SCC for the thread's sub-graph gets deleted like in the COMPLETION operation). ∎

We define the Thread Dependency Graph (TDG) has an alteration of the SDG for a SLG_{sct} forest.

Definition 4.2.11 (Thread Dependency Graph) *Let $T1$ and $T2$ be two active thread states in a SLG_{sct} forest $\mathcal{F}$. We say that $T1$ directly depends on $T2$ if there exist a subgoal in $T1$ that directly depends on a subgoal in $T2$ (according to Definition 2.2.9).*

The Thread Dependency Graph TDG($\mathcal{F}$) = (V,E) *of $\mathcal{F}$ is a directed graph where V is the set of active threads in $\mathcal{F}$ and $(t_i, t_j) \in E$ iff thread state t_i directly depends on thread state t_j.*

4.2.3 Complexity of SLG_{sct}

Theorem 4.2.3 *In the TDG for a SLG_{sct} local forest there's at most one outgoing edge from each node.*

Proof:

Once a edge outgoing from a thread state of $T0$ into the thread state of $T1$ appears in a local SLG_{sct} evaluation, the independent SCC of the Thread SDG of $T0$ will belong to the thread state of $T1$. From now on there will be no new edges added beginning in the thread state of $T0$, until this independent SCC is either completed or usurped. In the first case the outgoing edge will be deleted. In the second case the independent SCC will be incorporated in the thread state of $T0$. In no other way may an edge be added from one thread to another in the TDG.

∎

Theorem 4.2.4 (Complexity) *For each subgoal in the execution of a multi-threaded program at most one* USURPATION *operation happens during the Multi-threaded evaluation of a program.*

Proof:

When a USURPATION operation happens the subgoals in the usurped thread are grabbed by the leader thread, and the usurped threads are kept suspended and marked as non-leaders so that they will never be able to execute that subgoals again. New USURPATION operations will only be executed as the SCC grows to include more subgoals.

∎

4.3 Example of Execution under Shared Completed Tables

```
:- table y/1, x/1, a/1, c/1.
:- thread_shared y/1, x/1, a/1, c/1.

y(X) :- x(X).
y(y).

x(X) :- a(X).
x(X) :- y(X).
x(x).

a(X) :- c(X).
a(X) :- x(X).
a(a).

c(X) :- a(X).
c(c).

t1 :- x(X).
t2 :- a(X).
t3 :- y(X).
```

Figure 4.10: Program $P_{4.3}$

For program $P_{4.3}$ (see Figure 4.10) let there be three threads with ids 1,2 and 3, executing the initial queries `:- t1`, `:- t2`, `:- t3`, respectively.

Initial State to State 1 Let's assume that thread 1 starts and calls `x(X)`, which calls `a(X)`. Meanwhile thread 2 calls `c(X)`. Now thread 1 calls `c(X)` and is suspended because `c(X)` belongs to thread 2 and there is no deadlock yet. Now thread 2 calls `a(X)`. This point in the computation is shown in Figure 4.11.

State 1 to State 2 Thread 2 now checks for deadlock and finds it. The subgoal `a(X)` is usurped from thread 1 which is reset to the call to `a(X)` by `x(X)`. This is shown in Figure 4.12.

State 2 to State 3 Thread 1 is still suspended and depends on thread 2 because of the dependency of `x(X)` on `a(X)` and `a(X)` is recomputed by thread 2. This means that `a(X)` again calls `c(x)` which is already in table for thread 2. Then `a(X)` calls `x(X)`. This moment is shown in Figure 4.13.

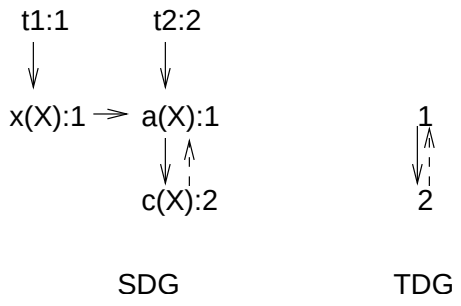

Figure 4.11: Concurrent execution of $P_{4.3}$: State 1

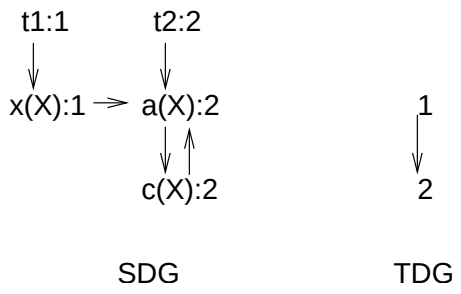

Figure 4.12: Concurrent execution of $P_{4.3}$: State 2

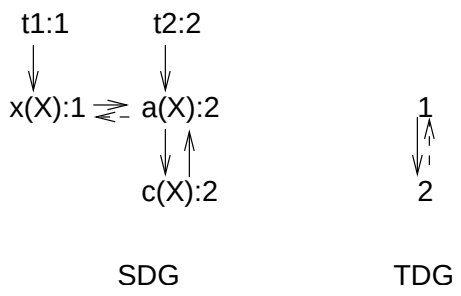

Figure 4.13: Concurrent execution of $P_{4.3}$: State 3

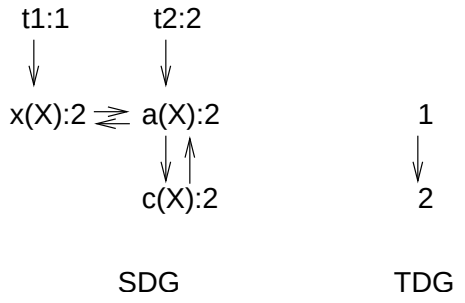

Figure 4.14: Concurrent execution of $P_{4.3}$: State 4

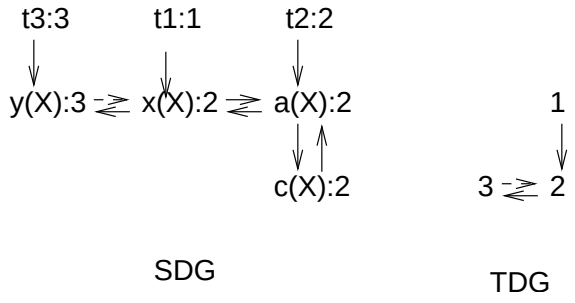

Figure 4.15: Concurrent execution of $P_{4.3}$: State 5

State 3 to State 4 Thread 2 again checks for a deadlock and succeeds. The subgoal `x(X)` is usurped from thread 1 which is reset to the call to `a(X)` by `t1`. This is shown on Figure 4.14.

State 4 to State 5 Thread 1 is still suspended and depends on thread 2 because of the dependency from `t1` to `x(X)`, Now let thread 3 begin. `t3` calls `y(X)`. Let thread 2 proceed. `x(X)` calls `a(X)` which is already on the table for thread 2. Then `x(X)` calls `y(X)` which belongs to thread 3 and as there is no deadlock, thread 2 suspends. Let finally thread 3 call `x(X)`. This situation is shown on Figure 4.15.

State 5 to State 6 Now thread 3 detects a deadlock and usurps the SCC that was being computed by thread 2. Thread 3 starts computing `x(X)` while the other computations are lost. However all subgoals are marked as grabbed and belonging to thread 3, which is the only one allowed to recompute them. Threads 1 and 2 continue suspended. This is shown in Figure 4.16

State 6 to State 7 Thread 2 still depends on thread 3, although `a(X)` is not on the stacks of thread 3 but it is marked as grabbed and belongs to thread 3. Thread 3 stacks grow to eventually encompass all the the subgoals which is show in Figure 4.16.

State 7 to Final State Eventually all answers are returned and the SCC is completed. Only then threads 1 and 2 are resumed and answers are returned to them (which they

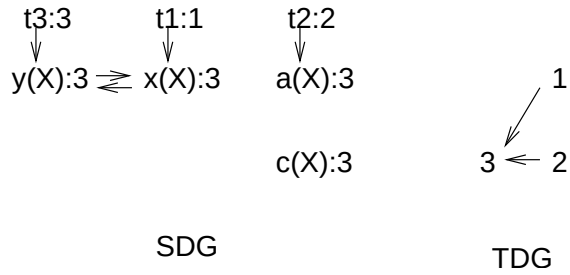

Figure 4.16: Concurrent execution of $P_{4.3}$: State 6

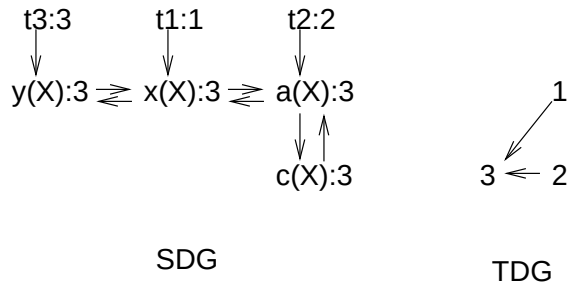

Figure 4.17: Concurrent execution of $P_{4.3}$: State 7

ignore in this example).

4.4 Correctness of Shared Completed Tables

In this section we argue that the algorithms presented in Section 4.1 implement the local SLG_{sct} semantics defined in Section 4.2 and make an argument for the complexity of Shared Completed Tables.

The table_try instruction implements the NEW SUBGOAL and USURPATION operations. The first if block (line 1 in Figure 4.3) implements the USURPATION operation while the second if block (2 in Figure 4.3) implements the NEW SUBGOAL operation. As the second block is essentially the same as in the original SLG-WAM table_try instruction (see Figure 2.19) we only argue about the concurrency issues and the correctness of the USURPATION operation.

4.4.1 Correctness of the Implementation of the USURPATION Operation

The execution of the USURPATION operation can be divided in three parts:

- Deadlock detection.
- Resetting the usurped threads.
- Awaking the suspended threads

Deadlock Detection The function would_deadlock (Figure 4.4) is used to detect deadlocks by detecting cycles in the TDG (Definition 4.2.11). The implementation of the TDG and the detection of cycles is simplified by the fact that each node has at most one outgoing edge (Theorem 4.2.3). This allows for a simple while loop to detect the deadlock.

Resetting the Usurped Threads Resetting the usurped threads in the SLG-WAM under local scheduling is possible thanks to Corollary 2.2.1 that states that each SCC in a local evaluation has only one incoming edge. It works by finding the oldest subgoal in the SCC (the one with the incoming edge) and removing the entire SCC (or set of entire SCCs) from the completion stack. Function reset_other_threads (Figure 4.5) was discussed in Section 4.1.4. In our implementation of the USURPATION operation the subgoals are re-computed by the usurper thread after the stacks of the usurped threads are reset. The usurper thread just jumps to point 3 in Figure 4.3 to (re)compute the grabbed subgoal. The correctness of the evaluation is not affected by the re-computation, as the tabled program is assumed to be static.

Awaking the Suspended Threads Clearly if a thread waits for the completion of a subgoal when there is no deadlock, the subgoal will eventually be completed and the thread awakened. If there occurs a deadlock and the corresponding USURPATION operation, in a finite evaluation, the SCC will eventually be completed and the suspended threads awaken by the completion instruction.

4.4.2 Complexity of Shared Completed Tables

Theorem 4.2.4 which states that there's at most one USURPATION operation for each subgoal, holds true for the implementation as well.

As for each USURPATION operation the SCC has to be recomputed, Theorem 4.2.4 implies that in the worse case Shared Completed Tables is quadratic for the Well Founded Semantics. However we believe that in practice deadlocks and the corresponding USURPATION operations hardly ever occur, making the best case complexity the overwhelmingly most common one, in which the complexity for Shared Completed Tables is the same as for the SLG-WAM which is linear for positive programs.

Chapter summary In this chapter we introduced Shared Completed Tables, an optimistic method for implementing shared tables, which only allows other threads to read from a table after it is completed. The main point of this method is to allow deadlocks to occur and then to break them – although this is only possible under Local Scheduling. We presented the changes to the SLG-WAM's data structures and instruction set to support Shared Completed Tables – most changes are in the table_try instruction.

We formalized Shared Completed Tables, modifying standard SLG to support multiple resolution threads and extend it with the USURPATION operation. We named this variation of SLG SLG_{sct}. We gave an example of execution of Shared Completed Tables including the occurrence of deadlocks. Finally we discussed the correcteness of the implementation with respect to SLG_{sct}.

5

Concurrent Completion

Concurrent Completion allows the returning of answers concurrently among non-completed tabled subgoals owned by different threads. This means that when answers are derived they may be returned immediately to other threads, modeling more closely a concurrent producer-consumer model. This means that a one-answer query involving tables owned by different threads may be computed faster than with Shared Completed Tables as it doesn't need for the involved tables to be completed before it returns. This also opens more opportunities for table parallelism as inter-dependent subgoals in different threads may be using each others answers in parallel in a computation.

The main burden of this method is a more complex completion algorithm, as implemented by the SLG-WAM check_complete instruction. This generalizes the completion method for the sequential SLG considering the completion of sets of subgoals scattered among the stacks of different threads. The Concurrent Completion algorithm is a generalization of the SLG-WAM completion algorithm and its main data structures are kept, although with some detailed changes to model the dependencies among subgoals owned by different threads.

It was decided to use Batched scheduling for the implementation of Concurrent Completion, as the speed of returning answers is stressed over the speed of completing SCCs as in Local scheduling. However, and unlike Shared Completed Tables that require Local scheduling, Concurrent Completion can in principle be used with either Local or Batched scheduling.

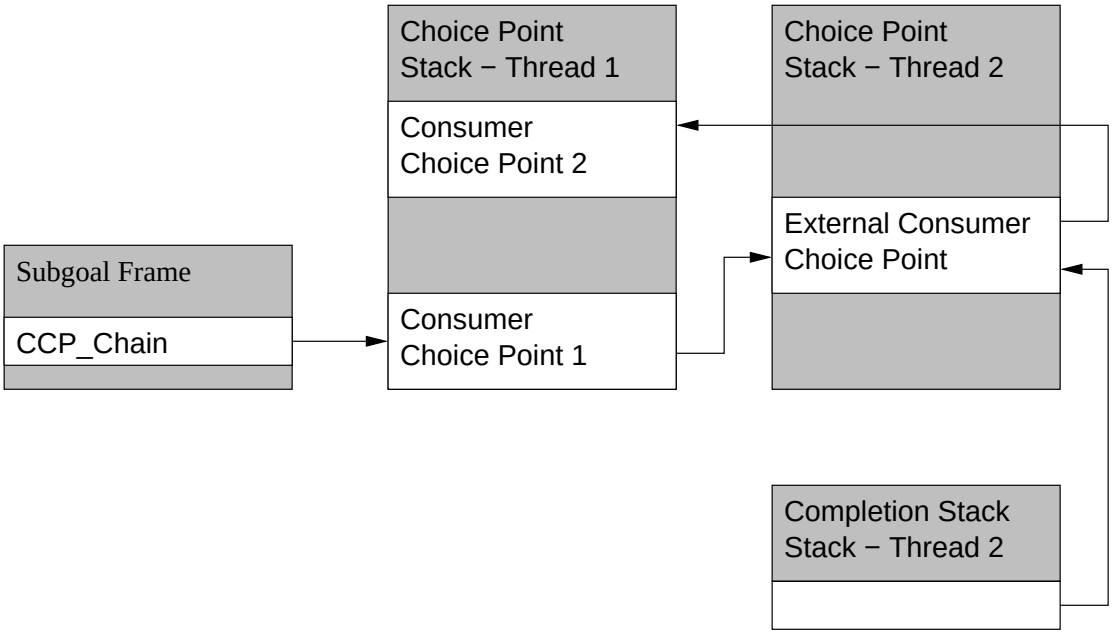

Figure 5.1: The external consumer choice point

5.1 Changing the SLG-WAM to Support Concurrent Completion

Concurrent Completion involves more elaborate changes to he SLG-WAM data structures than Shared Completed Tables. The most significant change is the new external consumer choice point, a special consumer choice point used to return answers from the generator thread to threads that use the answers from the table. In the thread's completion stack there's an associated completion stack frame used to enable the scheduling of external answers. Figure 5.1 shows the new relationships among the subgoal frame, the threads choice point stacks and the threads completion stacks, in the presence of external consumer choice points.

On the thread context data structures, the principal addition is the one of the *Thread Dependency List* aka *TDL* to the thread context. This is used by the completion algorithm to keep track of the dependencies among subgoals in different threads so that it may compute the multi-threaded ASCCs.

At the level of SLG-WAM instructions the check_complete instruction is extensively modified to handle the completion of multi-threaded ASCCs. The table_try instruction also has to be changed to handle the external consumer choice points.

5.1.1 Data Structures

We use a global mutex to control concurrency:

- **completing_mut** This is used to ensure that the check_complete instruction is atomic.

Thread Context We extend the thread context (given in Section 3.3.1) with the following fields:

- **Completing** A boolean that is true when the thread is waiting for the ASCC (cf. Section 2.3.2) to complete in the completion algorithm.

- **Completed** A boolean that signals that the multi-threaded ASCC is completed in the completion algorithm.

- **round** The round number is an integer which is used to keep track of the last round performed in the completion process. It is incremented every time answers are scheduled and is used by other threads to check if this thread may have generated more answers.

- **local_leader** The leader for the top ASCC of this thread.

- **cond_var** A condition variable used to suspend the thread when it should wait for other threads to perform its tasks in the completion algorithm.

- **TDL** The Thread Dependency List. A list of triplets $< tid, subgoal\,frame, round >$[1] used to keep track of the topology of the dependency graph and to find the completing ASCC when its goals are distributed among threads. The list is indexed by thread, as only the deepest dependency on any thread is maintained – the *subgoal frame* present in the *TDL* is always the one which has the oldest Completion Stack Frame for every dependency on that thread that has been found so far.

We assume that all functions have the local parameter *th* so that they can access the thread context (cf. Section 3.3.1).

Subgoal Frame The subgoal frame, defined in Section 2.3.3, is extended with the following field:

- **tid** An integer which denotes the thread identifier of the thread which generates solutions for this subgoal. As terminology we refer to such a thread as the *owner* of the subgoal.

[1] In the implementation, as the *subgoal frame* record contains the *tid* the *TDL* only contains the *subgoal frame* and the *round*. The *tid* is given in this description of the algorithm to simplify the *TDL* handling procedures.

Generator Choice Point the generator choice point, defined in Section 2.3.2, is extended with the field:

- **ComplSF** This is needed to access the Completion Stack frames that correspond to this Generator Choice point in the case of Generator Choice points associated with external consumers[2].

Consumer Choice Point The consumer choice point, defined in Section 2.3.2, is extended with the following field:

- **tid** An integer which denotes the thread in which stacks this consumer choice point resides.

Completion Stack Frame The completion stack frame, defined in Section 2.3.2, is extended with the following field:

- **ExtCons** This is null for regular SLG-WAM completion stack frames. For completion stack frames related to external consumers it points to the external consumer choice point.

5.1.2 Changes to the Table_Try Instruction

The main changes to the table_try instruction concern the call to an already existing subgoal (lines 4–8 of Figure 5.2).

The *tid* field stored in the consumer choice point can be compared with the *tid* field of the subgoal frame to check if we are dealing with an external consumer choice point, in which case these fields have different values.

In line 4.1 a generator choice point is created below the consumer choice point in case there are no generator choice points below in the choice point stack, in which case the completion stack is empty. This is needed because there must be a generator choice point before all consumer choice points with the check_complete instruction as the failure continuation to schedule the answers returned by external consumer choice points [3].

The creation of the consumer choice point proper in line 5 is the same whether it is external or not. The *tid* field is used to distinguish between them (if it's different from the current thread it's an external consumer).

In line 6.1 a new completion stack frame is pushed, only in the case of an external consumer choice point, with the *ExtCons* field set to the new external consumer choice

[2]Generator Choice Points have a pointer to the Subgoal Frame which itself points to the Completion Stack Frame. This doesn't work for "dummy" Generator Choice Points as they must point to the Completion Stack frame in the Completion Stack of their thread, not the one in the thread that generates solutions for this subgoal

[3]With internal consumer choice points there is always the guarantee that the generator choice point for that subgoal is below in the choice point stack of the thread.

```
Instruction table_try(Arity, NextClause, TIF)          /* Subgoal is in argument registers */
    shared ← table_is_shared(TIF);
    if ( shared ) lock( completing_mut );
    SF ← subgoal_check_insert(Subgoal, Subgoal_Trie_Root)
1   if (SF = NULL )                                    /* Subgoal is new and added */
        SF ← NewSubgoalFrame(Subgoal);
        if ( shared ) unlock( completing_mut );
        SF.tid ← th.tid;
        GCP ← PushGeneratorChoicePoint(...);
        ...     /* actions in lines 2.3–2.6 of Figure 2.19 */
2   else if ( (SF.IsCompleted) )                       /* Subgoal is complete */
        if ( shared ) unlock( completing_mut );
        Answer_Root ← Subgoal.AnsTrieRoot;
         /* Branch to Answer_Root SLG-WAM instruction */
3   else                                               /* Subgoal is incomplete */
        if ( shared ) unlock( completing_mut );
4       if(SF.tid ≠ th.tid and empty(completion_stack) )
4.1         gcp ← PushGeneratorChoicePoint(...);
            gcp.FailCont ← check_complete_instruction;
5       CCP ← PushConsumerChoicePoint(...);
        CCP.tid ← th.tid;
6       if(SF.tid ≠ th.tid )
6.1         csf ← PushCompletionStackFrame(...);
            csf.ExtCons ← CCP;
7       if(SF.tid = th.tid )
            adjust_levels(subgoal);
8       ...     /* actions in lines 4.3–4.7 of Figure 2.19 */
```

Figure 5.2: The table_try instruction.

point. The completion stack frame is needed to schedule the answers returned by other threads in the check_complete instruction.

In line 7, in the case that the consumer choice point is not external , the adjust_levels procedure (c.f. Figure 2.20 in Section 2.3.4) which adjusts the DFNs in the completion stack to extend the current SCC is not called. This is not done for external consumer choice points because the generator choice point for the subgoal belongs to the stacks of another thread and so the SCC for this thread must not be extended.

5.1.3 The New Check_Complete Instruction

The first thing to notice about the new check_complete instruction is that it is atomic i.e. only one thread can execute check_complete at a time and it will not be interrupted.

Lines 1 to 3 2 of Figure 5.3 are the same as in the sequential check_complete instruction. However the fixpoint_check procedure (Figure 5.4) is changed to check for completion stack frames that correspond to external consumer choice points and to build the initial *TDL* with the direct dependencies of this thread to other threads. If answers are found, the round number is incremented and the instruction fails to the chain of consumer choice points that return the answers.

If no unreturned answers are found for any subgoal in the ASCC, procedure UpdateDeps (described in Section 5.1.5) is called. Procedure UpdateDeps extends the *TDL* with the indirect dependencies of this thread and, if it finds a back dependency into this thread, it will set *NewLeader* to the completion stack frame of this thread which is depended on. The flag *busy* is set by UpdateDeps if any of the threads we depend on is not in the *Completing* state, i.e. it is not blocked waiting for completion, but instead running and generating solutions.

The condition in line 6 checks if the *NewLeader* is deeper in the completion stack than the current leader of this thread. This would mean that the ASCC is larger than what fixpoint_check considered and the levels in the completion stack must be adjusted to extend the thread's ASCC up to the new leader. The check_complete instruction fails and execution backtracks to the new leader so that it will handle the completion process.

Then if the *busy* flag is found to be true all other threads we depend on are awaken (line 7.3.1). This is because new answers may be being returned and the other threads will have to check for them.

If the *busy* false is false the MayHaveAnswers function checks the round number of each thread with the last seen round number of the other thread. If these don't match, it means some answers have not been returned and MayHaveAnswers wakes those threads.

If *busy* is false and all the round numbers match, it topology of the dependency graph (the *TDL*) is checked by function CheckForSCC to see if it forms a proper SCC (if all the edges have been already added). If yes, all others threads are signaled to

```
Instruction check_complete
    csf ← breg.completion_frame;
    lock(completing_mut);
    TDL ← empty ;
    for(;;)
1       if( not is_leader(csf) )
            breg ← breg.Breg_Chain;
            break;
2       tmp_breg ←fixpoint_check(csf, breg, th.TDL );
3       if( tmp_breg ≠ breg )                    /* if there are answers to return */
            th.round++ ;
            breg ← tmp_breg;
            break ;
4       NewLeader ← csf ;
5       UpdateDeps(th.TDL, busy, NewLeader)
6       if( NewLeader < csf )
6.1         adjust_levels(csf, NewLeader);
            breg ← breg.Breg_Chain;
            break;
7       if( not busy )
7.1         if( MayHaveAnswers(th.TDL) )
7.1.1           ;
7.2         else if( CheckForSCC(th.TDL) )
7.2.1           CompleteOtherThreads( th.TDL ) ;
7.2.2           CompleteTop(th,csf) ;
                break ;
7.3         else
7.3.1           WakeOtherThreads( th.TDL ) ;
        th.Completed ← false ;
        th.Completing ← true ;
        th.local_leader ← csf ;
7.4     cond_wait(th.cond_var,completing_mut);
        th.Completing ← false ;
7.5     if( th.Completed )
            break ;
    unlock(completing_mut)
    Fail ;
```

Figure 5.3: The check_complete instruction.

complete by the CompleteOtherThreads procedure. CompleteOtherThreads also runs CompleteTop for the other threads.

The CompleteTop procedure is similar to the sequential completion procedure in that it marks the goals as completed, frees the stack space and resets the freeze registers. Unlike in the sequential case it also wakes dependent threads that might be waiting for these subgoals to complete.

If completion wasn't successful the *Completing* flag is set to true and the thread is suspended in its condition variable. When it awakes it checks to see if successful completion was achieved by any other thread and if so, it fails to the next instruction. If not it loops and again performs the leader check, scheduling of instructions, etc.

5.1.4 Scheduling Answers

```
Procedure fixpoint_check(SubgCSF, sched_chain, TDL)
                    /* SubgCSF is a pointer to the completion stack frame */
  while (SubgCSF ≤ top_of_completion_stack)
    SubgFr ← CSF_SubgFr(SubgCSF);
    if (SubgCSF.ExtCons ≠ NULL)
      if (has_unconsumed_answers(SubgCSF.ExtCons))
                    /* schedule the external consumer */
        SubgCSF.ExtCons.Breg_Chain ← sched_chain
        sched_chain ← SubgCSF.ExtCons ;
      if (not SubgFr.is_completed)
        if (not < SubgFr.tid, _, _ > in TDL )
          insert < SubgFr.tid, SubgFr, round(SugFr.tid) > in TDL
        else if (< SubgFr.tid, S1, _ > in TDL and S1.ComplSF < SubgFr.ComplSF)
          replace S1 by SubgFr in TDL
    else
      sched_chain ← schedule_resumes(SubgFr,sched_chain);
        /* as in sequential batched fixpoint_check (Figure 2.22) */
    SubgCSF++;
  return sched_chain
```

Figure 5.4: The fixpoint_check Procedure.

As it was said before, the only extension for the sequential case to procedure fixpoint_check (Figure 5.4) happens when a completion stack frame corresponding to an external consumer is found. In that case a dependency is inserted in the *TDL* if there is no previous element corresponding to that thread in the *TDL*. If there was a previous element for that thread in the *TDL*, the one that refers to a subgoal corresponding to a deeper reference in the completion stack frame for that thread is kept.

```
UpdateDeps(TDL, busy, leader)
    busy ← false
    NewTDL ← empty
1   do
        new_deps ← false
        for each < tid, sgf, round > in TDL
            if (not sgf.is_completed)
                if Completing(tid)
                    round1 ← round(tid)
                else
                    round1 ← 0
2               insert < tid, sgf, round1 > in NewTDL
3               if Completing(tid)
4                   for each < ntid, nsgf, nround > in TDL(tid)
5                       if (ntid = th.tid)
                            if( nsgf.ComplSF < leader )
6                               leader ← ngsf.ComplSF
                            else if( < ntid, sgf1, round1 > in NewTDL )
                                if( sgf.ComplSF < sgf1.ComplSF )
                                    replace sgf1 by sgf in NewTDL
                                    new_deps ← true
                            else
                                insert sgf in NewTDL
                                new_deps ← true
                else
7                   busy ← true ;
8       TDL ← NewTDL ;
9   while( new_deps ) ;
```

Figure 5.5: The UpdateDeps procedure.

5.1.5 TDL Handling Procedures

In this section we present the various relatively low-level *TDL* handling routines. From here on the notation field(*tid*) is used to denote the field of the thread context *th* to which the thread identifier *tid* corresponds to. We use the particular field *th* to mean the thread context, i.e. th(*tid*) means the context for thread *tid*.

The UpdateDeps procedure (Figure 5.5) expands the initial *TDL* (created by procedure fixpoint_check) to include all the threads which the current thread indirectly depends on. It uses a breadth first algorithm: for each iteration of the do loop in line 1 it adds all the dependencies in the *TDL* to the *NewTDL*; in the cycle in lines 4–7 all the new direct dependencies from the current ones are checked; in part 6 the *NewTDL* is copied back to the *TDL*; if no new dependencies have been added a fixpoint has been reached and the loop finishes (part 7). If at any moment a thread is found to be computing (not in the *Completing* status) *busy* is set to true (condition in line 3). The condition in line 5 checks for back dependencies into the current thread, which are not

```
CheckForSCC(TDL))
1     for each < tid, sgf, round >∈ TDL
1.1       if( sgf.ComplSF < local_leader(tid) )
              return false;
1.2       if (not th.tid ∈ TDL(tid))
              return false;
1.3       for each < ntid, nsgf, nround >∈ TDL(tid)
1.3.1         if( ntid ≠ th.tid ) and not ntid ∈ TDL
                  return false ;
1.4       for each < ntid, nsgf, nround > in TDL
1.4.1         if( ntid ≠ tid ) and not ntid ∈ TDL(tid)
                  return false ;
      return true;
```

Figure 5.6: The CheckForSCC function.

```
MayHaveAnswers(TDL)
      rc ← false ;
      for each < tid, sgf, round >∈ TDL
          for each < ntid, nsgf, nround >∈ TDL(tid)
              if( nround < round(ntid) )
                  rc ← true ;
                  cond_signal(cond_var(tid))
      return false;
```

Figure 5.7: The MayHaveAnswers function.

added to the TDL but instead used to update the *leader* parameter.

The CheckForSCC procedure (Figure 5.6) simply checks that every thread on the TDL depends on every other thread. It also checks if the deepest dependency on to a thread corresponds to its local leader. This means that all dependencies have been added and completion may take place.

The MayHaveAnswers procedure (Figure 5.7) checks if there is one thread, t in the TDL whose last seen round from other thread, nt, is stale, i.e. thread nt has already performed more rounds after thread t has performed the completion instruction. This means that thread t has to check for more answers that may not yet have been returned to its consumers.

```
CompleteOtherThreads(TDL)
      for each < tid, sgf, round >∈ TDL
          CompleteTop(th(tid),local_leader(tid));
          Completed(tid) ← true;
          cond_signal(cond_var(tid))
```

Figure 5.8: The CompleteOtherThreads procedure.

```
WakeOtherThreads(TDL)
    for each < tid, sgf, round > ∈ TDL
        cond_signal(cond_var(tid))
```

Figure 5.9: The WakeOtherThreads procedure.

```
CompleteTop(th, leader)
    csf ← leader;
    while (csf ≤ openreg)
        subgoal ← csf.SubgFr;
        if( not subgoal.is_completed )
            subgoal.is_completed ← true;
            WakeDependentThreads(th, subgoal);
        csf++;
    openreg ← leader − 1;
    reclaim_stacks(breg);
    breg ← breg.Breg_Chain;
```

Figure 5.10: The CompleteTop procedure.

The CompleteOtherThreads procedure (Figure 5.8) completes the top ASCC for all threads in the multi-threaded ASCC and signals all threads that they have been completed.

The WakeOtherThreads procedure (Figure 5.9) simply wakes all other threads when completion is unsuccessful because a global fixed point has yet been reached.

5.1.6 Completion of a Single Thread

In this section we describe the steps taken to finalize the completion for a single thread.

The CompleteTop procedure (Figure 5.10) shows that the main change to the sequential procedure to reclaim the stacks consists of the call to the WakeDependentThreads procedure. This checks for all threads that have External Consumer Choice Points in their stacks that return answers from this thread and signals them to restart their check_complete instruction. This is because completion of a thread may be blocked waiting for completion of an ASCC that doesn't depend on subgoals in the top ASCC for that thread, and such threads must be awaken.

Procedure WakeDependentThreads (Figure 5.11) runs the list of consumers for a subgoal, and in case they are External Consumers, i.e. they don't reside in the same thread as the generator of the subgoal, the thread where they reside is signaled, so it may unblock when waiting for a dependent subgoal to complete.

```
WakeDependentThreads(subgoal)
    Cons ← subgoal.CCP_Chain ;
    while(Cons ≠ NULL)
        if(Cons.tid ≠ th.tid)
            cond_signal(cond_var(tid)) ;
        Cons ← Cons.PrevCCP ;
```

Figure 5.11: The WakeDependentThreads procedure.

```
:- table a/1, b/1, c/1.
:- thread_shared a/1, b/1, c/1.

a(X) :- b(X).
a(a).
b(X) :- c(X).
b(b).
c(X) :- a(X).
c(c).

t1 :- a(X), fail.
t2 :- b(X), fail.
t3 :- c(X), fail.
```

Figure 5.12: Program $P_{5.1}$

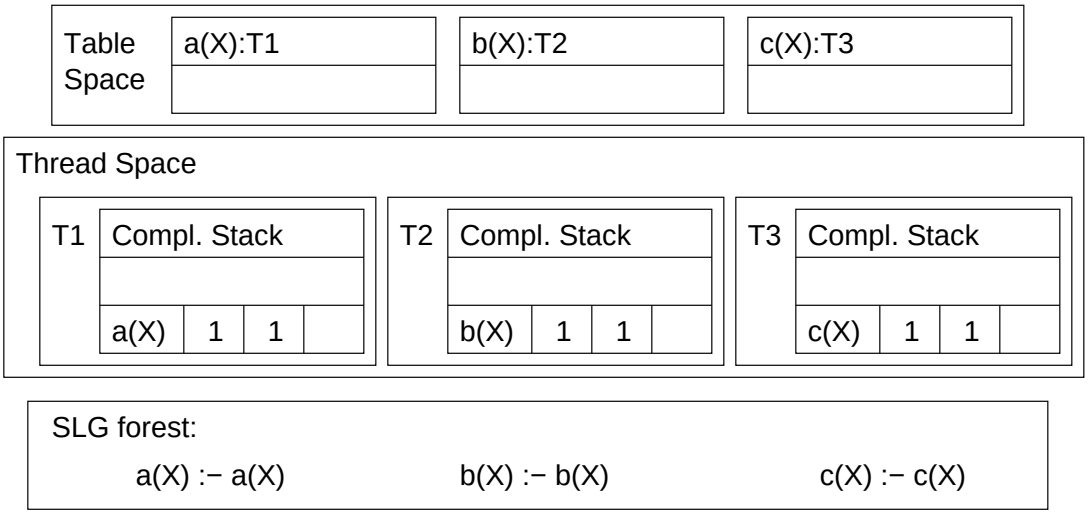

Figure 5.13: Concurrent execution of $P_{5.1}$: State 1

5.2 Example of Concurrent Completion

To illustrate the aspects of the Concurrent Completion algorithm in detail we consider the program $P_{5.1}$ in Figure 5.12 and 3 threads $T1$, $T2$ and $T3$ which execute concurrently the goals `t1`, `t2` and `t3` respectively. For simplicity of the example we assume that only one thread may be running at a given time.

Initial State to State 1 $T1$ calls `a(X)`. A table_try instruction is executed (Figure 5.2) which creates a table for `a(X)` owned by $T1$, a generator choice point for `a(X)` on $T1$'s choice point stack and pushes a completion frame onto $T1$'s completion stack. We assume that at this point $T2$ calls `b(X)`, then $T3$ calls `c(X)`. In executing table_try, both of these threads perform actions similar to $T1$, leading to a state of the computation as shown in Figure 5.13.

State 1 to State 2 Now assume that $T1$ executes and calls `b(X)`. As `b(X)` is in the table a consumer frame is created on $T1$ Choice Point Stack by instruction table_try. As the table is owned by another thread an external consumer choice point is created. A completion stack frame is pushed for `b(X)` which points to the external consumer choice point (this is denoted in the corresponding figure with an "e"). Note that adjust_levels is not called. Assume that at this point $T2$ calls `c(X)` and then to $T3$ calls `a(X)`. In executing table_try, both of these threads perform similar actions, leading to the state of the computation shown in Figure 5.14.

State 2 to State 3 Let's consider that thread $T1$ executes again. As `b(X)` has no answers it backtracks to the failure continuation of the generator choice point of `a(X)`. The continuation in the generator is a table_trust instruction which sets the failure continuation of the generator choice point to a check_complete instruction and leads to

T1	Compl. Stack			
	b(X)	2	2	e
	a(X)	1	1	

T2	Compl. Stack			
	c(X)	2	2	e
	b(X)	1	1	

T3	Compl. Stack			
	a(X)	2	2	e
	c(X)	1	1	

SLG forest:

```
a(X) :- a(X)          b(X) :- b(X)          c(X) :- c(X)
     |                     |                     |
a(X) :- b(X)          b(X) :- c(X)          c(X) :- a(X)
```

Figure 5.14: Concurrent execution of $P_{5.1}$: State 2

the derivation of answer `a(a)` by program clause resolution, which is stored in the table for `a(X)` by a new_answer SLG-WAM instruction. Execution backtracks to the check_complete instruction in the generator choice point for `a(X)`. As a(X) is the thread's local leader it runs the fixpoint_check procedure which finds that there are no answers to be returned and initializes the *TDL* with <$T2$,`b(X)`,0>. The UpdateDeps procedure finds that $T2$ is not in the completing state, sets the local variable *busy* to true and $T1$ suspends. Let's assume that now $T2$ takes control. A similar course of events takes place involving the generator choice point for `b(X)` and $T2$ also suspends when execution the check_complete instruction after having derived the answer `b(b)` leading to the state shown in Figure 5.15.

State 3 to State 4 With $T1$ and $T2$ suspended only $T3$ can execute. In a similar manner to $T1$ and $T2$ in the above paragraph, $T3$ fails to the generator choice point of `c(X)` and derives `c(c)` by program clause resolution, failing back to the check_complete instruction in the generator choice point for `a(X)`. Procedure fixpoint_check finds that there is one answer to return. *round* is incremented to 1, execution fails to the external consumer choice point of `a(X)` and answer `a(a)` is returned by the SLG-WAM instruction answer_return. Answer `c(a)` is derived and entered into the table for subgoal `c(X)`. Execution fails into the check_complete instruction. Procedure fixpoint_check finds that there are no answers to return and initializes the *TDL* with <$T1$, `a(X)`, 0>. When procedure UpdateDeps is called <$T2$, `b(X)`, 0> is added to the *TDL*. A back dependency is found from $T1$ to `c(X)` on $T3$ but as `c(X)` is already the leader for $T3$ execution proceeds. Function MayHaveAnswers finds out that $T2$ has last seen round 0 for $T3$ whereas $T3$'s round is now 1 and returns true. $T2$ is awakened. $T3$ suspends leading to the state of the computation shown in Figure 5.16.

State 4 to State 5 With $T1$ and $T3$ suspended, only $T2$ may run. $T2$ loops in the check_complete instruction. Procedure fixpoint_check now finds that there is one answer to return. *round* is incremented to 1 and execution fails to the external consumer

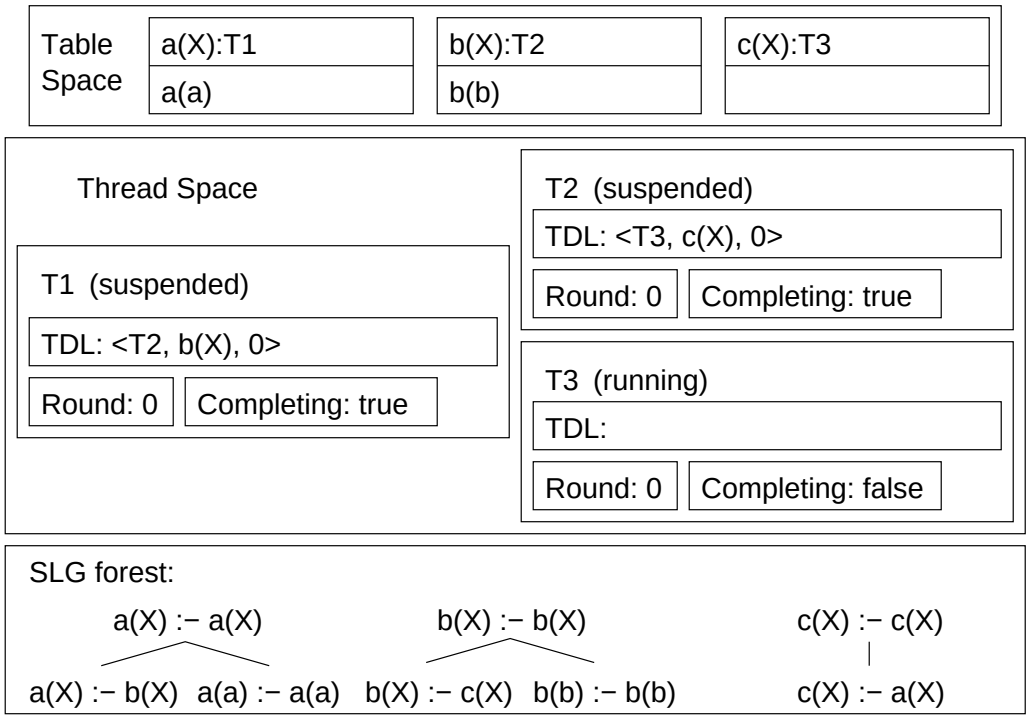

Figure 5.15: Concurrent execution of $P_{5.1}$: State 3

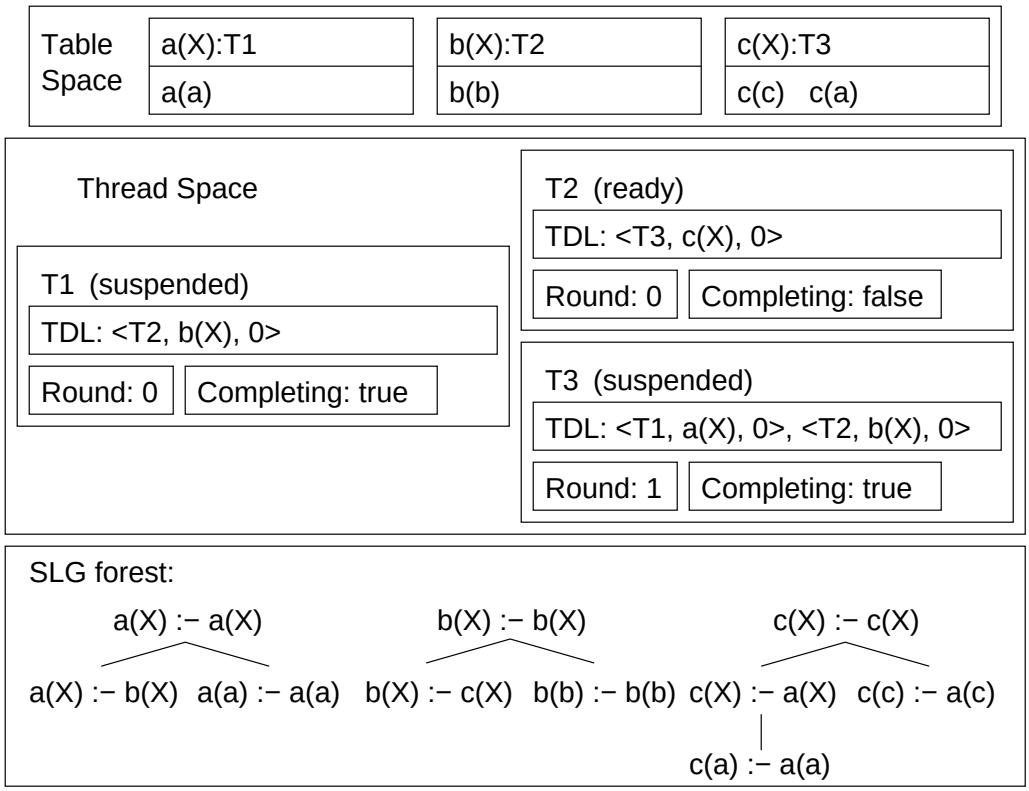

Figure 5.16: Concurrent execution of $P_{5.1}$: State 4

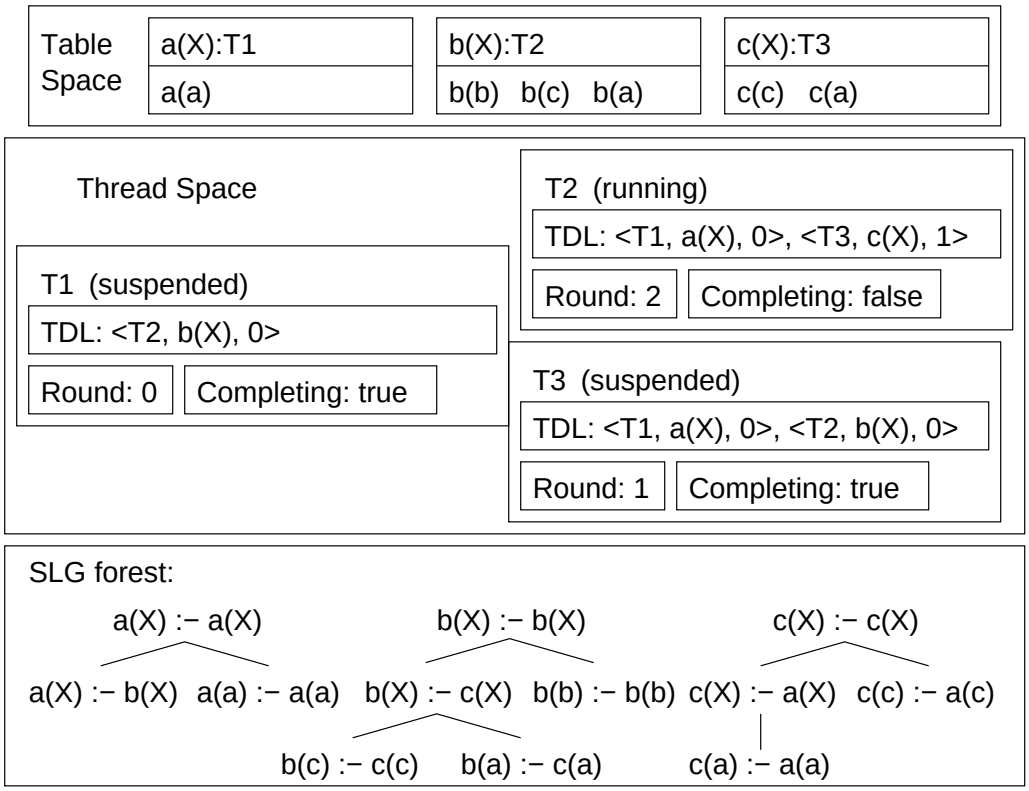

Figure 5.17: Concurrent execution of $P_{5.1}$: State 5

choice point of `c(X)`. The answer `c(c)` is returned and the answer `b(c)` is derived and stored in the table. Execution backtracks to the check_complete instruction. Procedure check_fixpoint finds that there is one answer to return. *round* is incremented to 2. Answer `c(a)` is returned and answer `b(a)` is derived and stored in the table. Execution fails to the check_complete instruction. Procedure check_fixpoint finds that there are no answers to return. Procedure UpdateDeps is called, leading to the state of the computation shown in Figure 5.17.

State 5 to State 6 Function MayHaveAnswers finds out that threads $T1$ and $T3$ have 0 for the last round of $T2$ while it is now 2, so it returns true. Threads $T1$ and $T3$ are awaken while thread $T2$ is suspended. Assume that now $T1$ executes. $T1$ loops in the check_complete instruction and find_fixpoint finds that there are answers to return. *round* is incremented to 1. Answer `b(b)` is returned by the external consumer choice point and answer `a(b)` is derived and stored in the table. Execution backtracks to the check_complete instruction. Procedure fixpoint_check finds that there is one answer to return. *round* is incremented to 2. Answer `b(c)` is returned and answer `a(c)` is derived and stored in the table. Execution backtracks to the check_complete instruction. Procedure fixpoint_check finds that there are answers to return. *round* is incremented to 3. Answer `b(a)` is returned and answer a(a) is derived but not added to the table as it is already there. Again execution fails into the check_complete instruction. This time fixpoint_check doesn't find any answers to return. After UpdateDeps the state of the

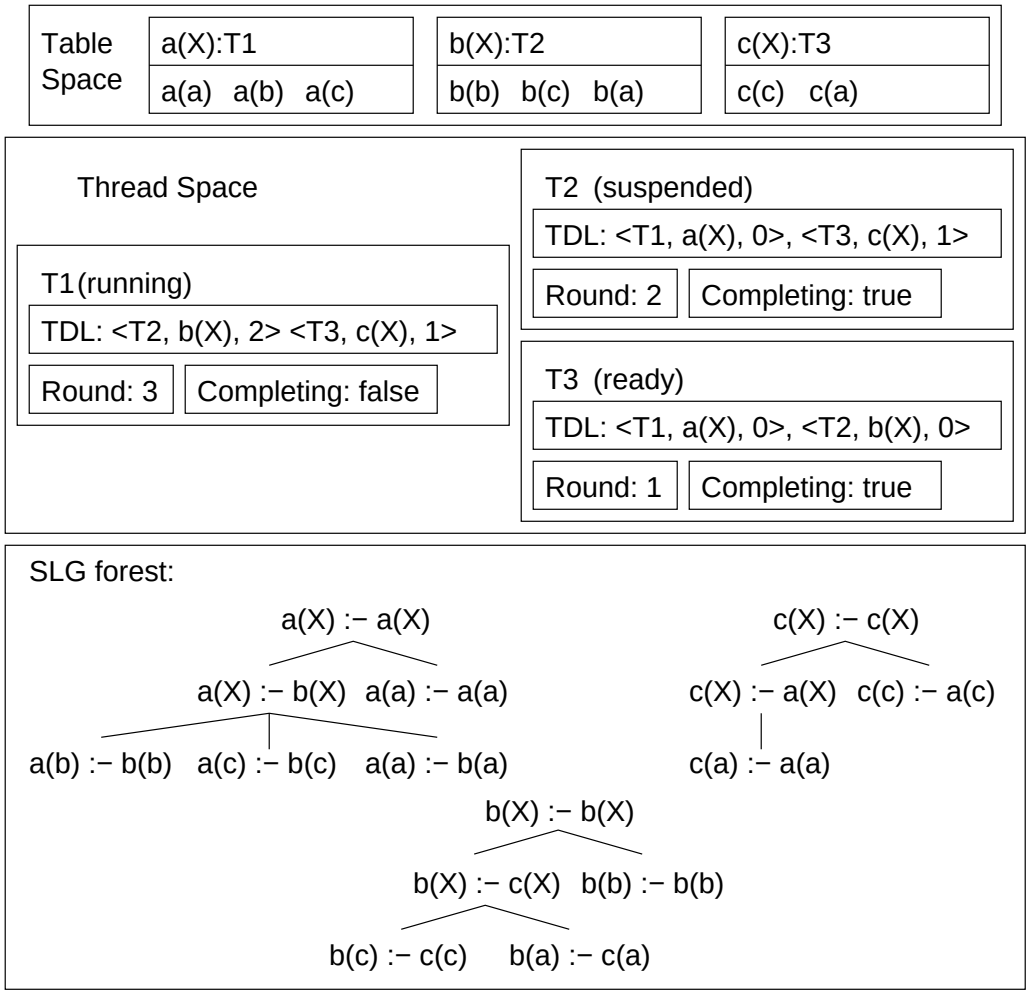

Figure 5.18: Concurrent execution of $P_{5.1}$: State 6

Concurrent Completion engine looks like Figure 5.18.

State 6 to State 7 Function MayHaveAnswers finds that $T2$ and $T3$ have 0 as the value for $T1$'s round while it is already 3, so it returns true. $T2$ and $T3$ are awakened while $T1$ is suspended. Assume that now $T3$ runs. The check_complete instruction loops. Procedure check_fixpoint finds that there is one answer to return. *round* is incremented to 2. Answer `a(b)` is returned. Answer c(b) is derived and stored in the table. Execution backtracks to the check_complete instruction. Procedure check_fixpoint finds that there is one answer to return. *round* is incremented to 3. Answer `a(c)` is returned. Answer `c(c)` is derived but it's not added to the table because it's already there. Execution backtracks to the check_complete instruction. Procedure fixpoint_check finds that there are no answers to return. After procedure UpdateDeps returns the state of the computation engine is shown in Figure 5.19.

State 7 to State 8 In this case function MayHaveAnswers finds that $T1$ and $T2$ have last seen round 1 of $T3$ but $T3$ is already on round 3 and returns true. This means that $T1$ and $T2$ are awaken and $T3$ suspends. Let's assume that now $T2$ executes. $T2$ loops in

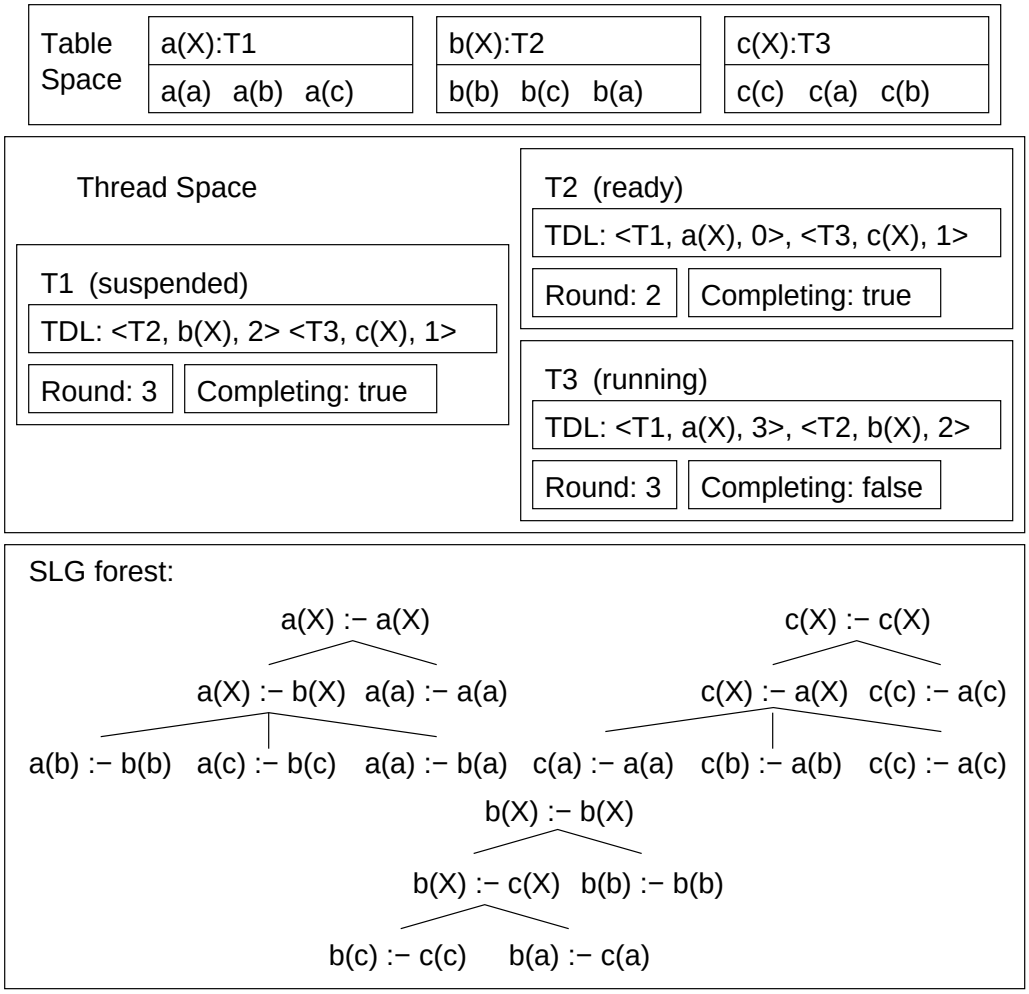

Figure 5.19: Concurrent execution of $P_{5.1}$: State 7

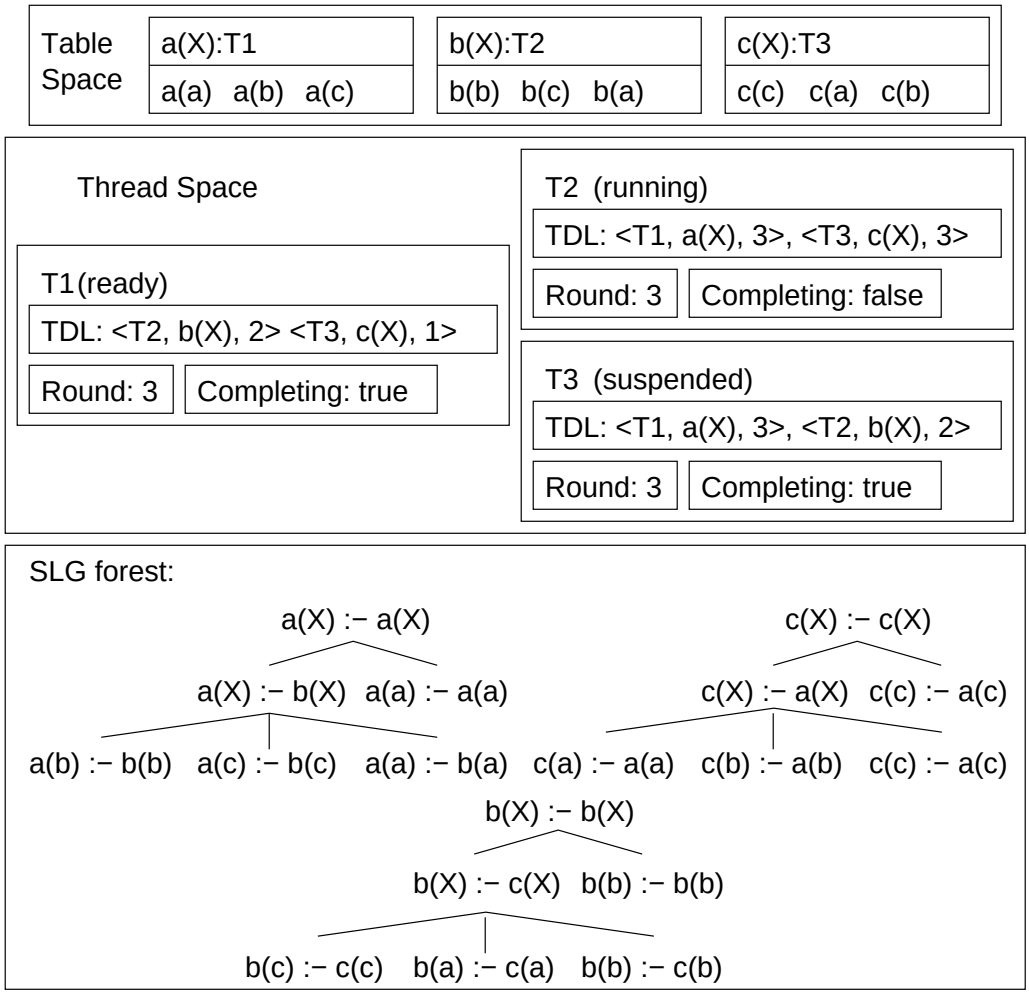

Figure 5.20: Concurrent execution of $P_{5.1}$: State 8

the check_complete instruction. Procedure fixpoint_check finds that there is one answer to return. *round* is incremented to 3 and execution backtracks to the consumer choice point which returns answer `c(b)`. Answer `b(b)` is derived but not entered into the table because it's already there. Execution backtracks to the check_complete instruction. This time fixpoint_check finds no answers. After procedure UpdateDeps returns the state of the computation is the one shown in Figure 5.20.

State 8 to State 9 Function MayHaveAnswers finds that $T1$ and $T3$ have their views about the last round of $T2$ stale, so it returns true. $T2$ suspends while $T1$ and $T3$ are awakened. Let's assume that now $T1$ runs. $T1$ loops in the check_complete instruction. Procedure fixpoint_check finds that there are no answers to return. The state of the computation after procedure UpdateDeps returns is shown in Figure 5.21.

State 9 to State 10 Function MayHaveAnswers finds out that $T3$ has a obsolete view of the round number for $T2$, so it returns true. This means that $T1$ is suspended and $T3$ awaken. As $T1$ and $T2$ are suspended only $T3$ may run. It loops in instruction check_complete. Procedure fixpoint_check is called. It finds that there are no answers to

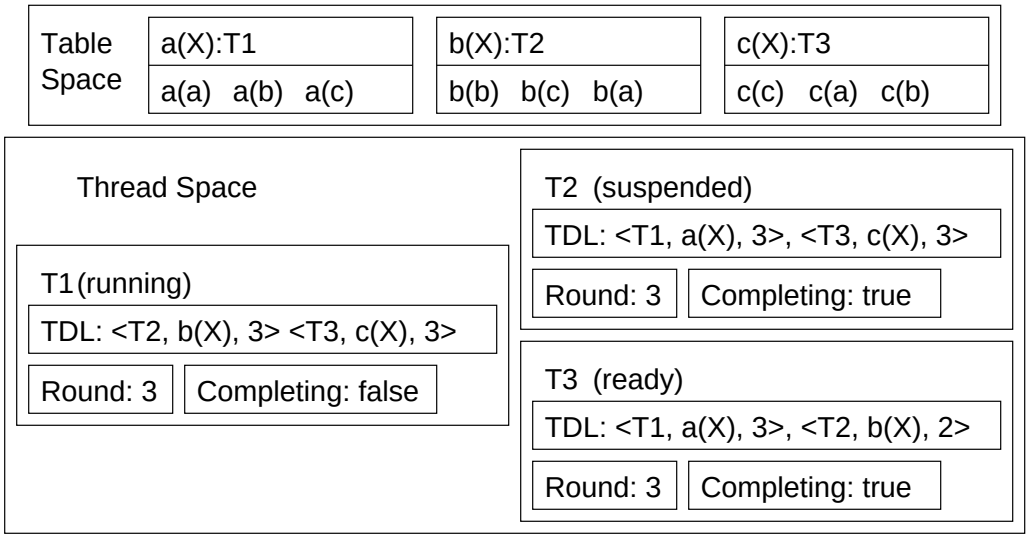

Figure 5.21: Concurrent execution of $P_{5.1}$: State 9

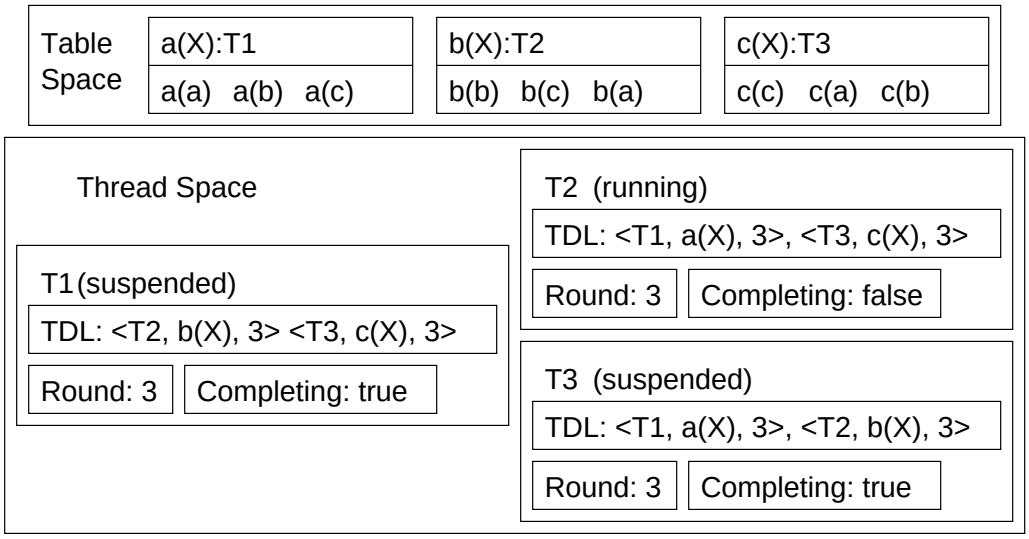

Figure 5.22: Concurrent execution of $P_{5.1}$: State 10

return. Let's assume that now $T2$ runs. Procedure fixpoint_check finds no answers. After procedure UpdateDeps returns we get the state of the computation shown in Figure 5.22.

State 10 to Final State Function MayHaveAnswers returns false because the threads last views of the rounds match with the actual round numbers. Procedure CheckForSCC returns true as every thread depends on each other and the leaders are correctly set. Procedure CompleteTop is called by $T3$ for $T1$ and $T2$, effectively completing the top tabled subgoals of these threads. Its completion stacks are emptied. The field *completed* is set to true in $T1$ and $T2$. Procedure CompleteTop is called to empty the stacks of thread $T3$. $T1$ and $T2$ are awaken. Figure 5.23 shows the state of the computation.

Query `t3` fails. Query `t1` fails. Query `t2` fails.

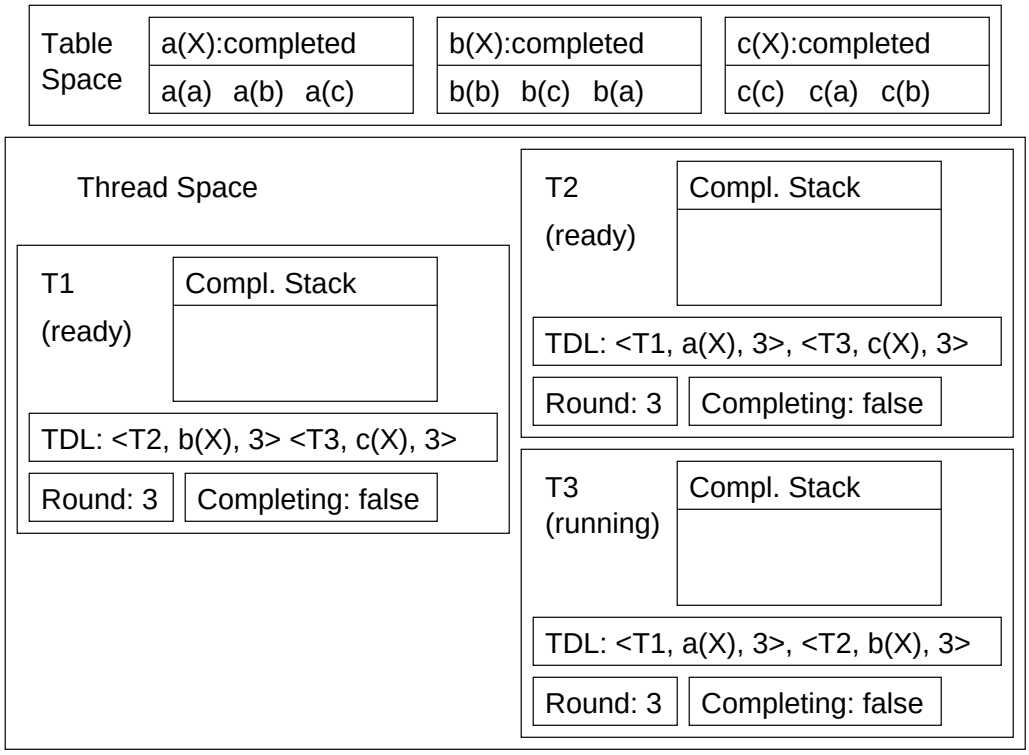

Figure 5.23: Concurrent execution of $P_{5.1}$: Final State

5.3 Correctness of Concurrent Completion

In this section we show that the Concurrent Completion algorithm derives all the answers of a program (Section 5.3.1), i.e. that the check_complete instruction correctly determines that a set of subgoals has been completely evaluated when it finishes and that this point will always be reached (Section 5.3.2). The soundness proof for the derivation of answers is similar to the one for sequential engine (see [73]), with the trivial addition of the external Consumer Choice Points when answers are returned from other threads. We don't develop that proof here. At the end of this section we discuss how complexity of Concurrent Completion relates to the sequential engine.

5.3.1 Completeness

In this section we prove that the check_complete instruction is correct, in the sense that if it terminates, all answers to a query have been derived.

Theorem 5.3.1 (Completeness) *Given the check_complete instruction for the generator choice point of subgoal S in thread T, if execution reaches line 7.2.1 of Figure 5.3 then S and all subgoals that S depends on have been completely evaluated.*

Proof: Recall that in an SLG evaluation of a definite program P, an independent SCC can be completed as long as all new subgoal, program clause resolution, and answer

return operations have been performed. At a high level, in the sequential SLG-WAM the batched scheduling strategy performs this as follows (see [73] for details).

In order to keep track of mutually dependent subgoals, the SLG-WAM uses the notion of an Independent Approximate SCC (ASCC) – a set of SCCs whose subgoals depend on no subgoals outside of that set of SCCs. The SLG-WAM keeps track of the ASCC as follows. When a new tabled subgoal is encountered, it is given a (global) depth first number DFN(S) along with a *DirLink* field, initialized to DFN(S) and updated by adjust_levels, when a call to a subgoal with an older completion stack frame, $S1$ is encountered for a clause of S. Then all for all completion stack frames between S and $S1$, *DirLink* is set to $DirLink(S1)$. The leader of an independent ASCC is defined as that subgoal S_{leader} such that for all subgoals S' that are on the completion stack and equal to or younger than S_{leader}, DirLink(S') = DFN(S_{leader}). Once the ASCC has been identified, completion can be assured by using the completion stack to traverse each subgoal S in the ASCC, and then ensuring that, for each consumer choice point for CCP_S, if the root subgoal for CCP_S is in the ASCC, then all answers for S have been returned to CCP_S. Since the completion operation is performed only when all program clause resolution and new subgoal operations have been performed, the final check that all answers have been returned to subgoals in the ASCC ensures that the subgoals will have been completely evaluated before they are completed.

The Concurrent Completion algorithm generalizes this sequential algorithm so that ASCCs may be maintained and completion checked when subgoals are owned by different threads and scattered among different completion stacks.

Definition 5.3.1 (Local ASCC) *Let $T1$ be a thread. Then* Local_Leader(T1) *is the subgoal S associated with the youngest completion stack frame C in $T1$ such that $DFN(S) = MinLink(S)$ on the completion stack. Let $Local\_ASCC(T)$ denote the partial ASCC containing its local leader (S) and all subgoals whose completion stack frames are younger than the one of S.*

Consider the situation when a thread $T1$ reaches line 4 of check_complete in Figure 5.3. Then Property 5.3.1 holds:

Property 5.3.1 *All answers that have been derived at this point have been returned to all internal Consumer Choice Points for subgoals in $Local\_ASCC(T1)$ that are owned by $T1$, and to all external Consumer Choice Points owned by $T1$ for subgoals in $Local\_ASCC(T1)$ not owned by $T1$.*

This is ensured by the fixpoint_check of condition 2 of check_complete in a similar manner to the sequential algorithm.

Note that the round number for the blocked threads on which this subgoal depends can be used to keep track of the last point at which the answers were returned to the External Consumer Choice Points.

Property 5.3.1 is weak. We still don't know what subgoals need to be added to $Local\_ASCC(T1)$ to make it a true ASCC, nor do we even have any assurance that answers are not being added to external subgoals in $Local\_ASCC(T1)$ by other threads.

Definition 5.3.2 (LocalThreads) *Let $LocalThreads(T1)$ be the minimal set of threads such that any goal in $ASCC(T1)$ is owned by one of the threads in $LocalThreads(T1)$.*

We define $Threads(T1)$ as the fixed point of $LocalThreads(T1)$, i.e $T \in Threads(T1)$ iff:

$$T \in LocalThreads(T1)$$

or

$$\exists T2 : T \in LocalThreads(T2) \text{ and } T2 \in Threads(T1)$$

Note that fixpoint_check adds an entry for $T2$ whenever there is an external Consumer Choice Point in $Local\_ASCC(T1)$ owned by $T2$. Thus, by the end of part 3, $TDL(T1)$'s threads correspond to $LocalThreads(T1)$. The subgoal kept for each thread is the oldest that belong to that thread, which is an optimization, as multiple dependencies an thread could in principle be kept.

UpdateDeps is a simple breadth first algorithm that adds all threads which are reachable from its initial set, which as we saw correspond to LocalThreads($T1$). As the TDLs for every thread, which constitute the universe of dependencies, were generated by check_fixpoint and UpdateDeps we have at the end of part 4:

$$Thread\_Set = \{T | T \text{ is a thread} \in TDL(T1)\} \cup \{T1\} \subseteq Threads(T1)$$

Thus all threads in $Thread\_Set$ will belong to the multi-threaded ASCC. Its also clear that *busy* will be true if any thread in $Threads(T1)$ is not blocked waiting for completion (in line 6.4).

Also note that *NewLeader* will be set only if there is an S owned by $T1$ and belonging to the Local_ASCC(T'), where T' belongs to $Threads(T1)$, such that the completion stack frame of S is younger than that of $LocalLeader(T1)$.

If the local leader changes, the completion process cannot proceed, as the check_complete instruction no longer corresponds to the local leader which should be backtracked to.

The checks that ensure that the union of $Local\_ASCC(T)$, $T \in Thread\_Set(T)$ is in fact an independent multi-threaded ASCC are conditions 6, 6.1 and 6.2:

1. Condition 6 ensures that all threads in $Thread\_Set(T1)$ are awaiting completion, which implies that they don't have any unreturned internal answers and they are quiescent.

2. Condition 6.1, the call to MayHaveAnswers ensures that no thread in Threads($T1$) has unreturned external answers.

This is because each thread, in procedure UpdateDeps records the round number for every other thread it depends on on the TDL before it suspends for completion. Let $R = LastRound(T, T')$ be the *round* field of the entry for T' in the TDL of T. For MayHaveAnswers to fail, it means:

$$\forall T, T' \in Thread\_Set(T1), round(T') = LastRound(T, T')$$

This means that $round(T')$ wasn't incremented since the moment that T suspended for completion. As T did run fixpoint_check at that moment and there were no more answers to return (remember that check_complete is atomic), it means that thread T has returned all answers that T' generated for the subgoals in $Local\_ASCC(T)$. (See also the discussion of Property 5.3.1).

3. Condition 6.2, the call to CheckForSCC ensures that:

$$\forall T \in Thread\_Set(T1), Thread\_Set(T) = Threads(T)$$

This ensures that every thread in the multi-threaded ASCC is included in the TDL.

It also ensures (because of the condition in line 1.1 of Figure 5.6) that:

$$\forall T, T' \in ThreadSet(T1) : \nexists S \in Local\_ASCC(T') : Local\_Leader(T) \textit{ is younger than } S$$

This ensures that no subgoal in the multi-threaded ASCC depends on another on thread T that is younger than the local leader of T.

From 1,2 and 3 above it follows that when execution reaches 7.2.1 of Figure 5.3 an independent multi-threaded ASCC has been detected and all answers of its subgoals have been derived. ∎

In the next section we prove that execution will always reach 7.2.1 of Figure 5.3 for some thread.

5.3.2 Liveness

In this section we show that if the query being evaluated has a finite number of answers then Concurrent Completion always terminates.

Theorem 5.3.2 (Liveness) *Given a program P and a set of queries $Q_1, Q_2 \ldots Q_n$ evaluated by a set of threads $T_1, T_2 \ldots T_n$, given that:*

- *Every query Q_i requires the computation of all answers to it.*

- *For every query Q_i its evaluation against program P results in a finite SLG evaluation.*
- *The underlying operation system scheduling of threads is fair.*

Then all answers for $Q_1, Q_2 \ldots Q_n$ against P will be computed by Concurrent Completion

Proof:

Terminology We begin the proof by introducing some terminology. Note that for a given thread T performing tabled computation, the ASCC for T, denoted $true\_ASCC(T)$ and potentially distributed among threads, can be defined as the least set such that:

$$Local\_ASCC(T) \subseteq true\_ASCC(T) \text{ and } S' \in true\_ASCC(T) \text{ iff every direct dependency (see Definition 2.2.9) of } S' \text{ is in } true\_ASCC(T).$$

For a set of subgoals $\mathcal{S}$, we define $threads(\mathcal{S})$ as the least set of threads such that every subgoal in $\mathcal{S}$ is owned by some thread in $threads(\mathcal{S})$.

Although we have not modeled Concurrent Completion formally, we make use of an informal notion of a Concurrent Completion computation as a sequence (with cardinality ω or less) of SLG-WAM instructions executed by various threads, and the state of such a computation as a sub-sequence of that sequence. Accordingly, consider a state of computation in which a thread T is computing subgoals involved in $true\_ASCC(T)$. Note that in the process of completing $true\_ASCC(T)$, any suspended threads that have non-completed subgoals with dependencies on subgoals in $true\_ASCC(T)$ will eventually be restarted by the invocation of procedure WakeDependentThreads by procedure CompleteTop (see Figure 5.10) so that liveness will be preserved for threads with goals suspended on true_ASCC(T) but that are not contained in $true\_ASCC(T)$. We thus may restrict our attention to states involved in the computation of $true\_ASCC(T)$ itself.

In addition, note that if $\mathcal{T}$ consists of a single thread T_0, it will complete as in the SLG-WAM, so that we can restrict our attention to cases in which the cardinality of $threads(true\_ASCC(T))$ is greater than 1 – i.e. when local_ASCC(T) is involved in a multi-threaded computation.

We next define the notion of the known ASCC for a thread T, denoted known_ASCC(T). For a subgoal S in a thread T, the *completion segment for S in T* contains S together with all subgoals in the completion stack of T that are younger than S. Thus the known ASCC for a thread T consists of the union of the subgoals in the Local_ASCC(T) together with the completion segment for S' in T' for each <T′,S′,round> in the TDL of T. As S' is set as the older subgoal for T' that $Local\_ASCC(T)$ depends on, by check_fixpoint and UpdateDeps, we have $known\_ASCC(T) \subseteq true\_ASCC(T))$.

Using these notions, we define the notion of computational progress for true_ASCC(T) as the number of answers returned to subgoals in $\mathcal{A}$ combined with sum of |known_ASCC(T′)| for $T' \in threads(true\_ASCC(T))$. Note that for a finite program the number of answers returned and subgoals in true_ASCC(T) will be

finite, and since $known\_ASCC(T) \subseteq true\_ASCC(T)$, the sum of $|known\_ASCC(T)|$ for $T' \in threads(true\_ASCC(T))$! will also be finite.

Attaining a Synchronization Point As described previously, the algorithm for concurrent completion works as follows for an ASCC. The threads evaluate the goals in the ASCC by generating and consuming answers, and by adjusting their local leaders to reflect the cross-thread dependencies in the ASCC. The use of a *completing_mut* mutex ensures that the check_complete instruction is atomic. We refer to a *synchronization point for a set of subgoals* $\mathcal{S}$ as a computation state in which a thread owning some subgoal in $\mathcal{S}$ executes a check_complete instruction and reaches line 7.1 in Figure 5.3.

Accordingly, let *state* be a state of a tabled computation in which some thread T is evaluating subgoals in $known\_ASCC(T)$. We show that there will be a synchronization point for $known\_ASCC(T)$ after *state*. Since synchronization points are executed by check_complete instructions we begin by showing that execution for any thread T' will always backtrack into a check_complete instruction for at least 1 subgoal S' in $local\_ASCC(T)$. This occurs because a completion instruction is set as the failure continuation of a generator choice point by the table_trust or table_try_single instruction or is set up directly in the case of the generator choice point created by part 4 of Figure 5.3. Note also that each thread will always have a generator choice point as the initial choice point of a tabled computation – see notes to line 5 of Figure 5.3.

Next, consider the various actions taken for the check_complete instruction for S' in a thread T' described above. The first possible action is that the check_complete instruction fails. There are several cases in Figure 5.3 to consider:

1. Condition in line 1: S is not the local leader for the current thread i.e. in its completion stack frame $DFN \neq MinLink$. Execution will eventually backtrack to the completion instruction corresponding to the local leader deeper in the completion stack because:

   (a) when the completion stack frame for S' is created it is initialized with $DFN(S') = MinLink(S')$

   (b) when adjust_levels is called for S' the completion stack frame for S' is unchanged and only younger completion stack frames *DirLink* fields are set to $DFN(S')$.

2. Condition in line 3: There are answers to be returned to the evaluation of S or other subgoals in $local\_ASCC(T)$. These answers will be returned, and by the argument made above, the execution will eventually backtrack to the completion instruction corresponding to the local leader deeper in the completion stack.

3. Condition in line 6: A new local leader for the thread has been found (cf. line 6 of Figure 5.5) adjust_levels will change the completion stack and the situation

will be similar to case 1. In this case new dependencies have been added to the completion stack

Note that for a finite program, failure of a check_complete instruction will not lead directly to a synchronization point, but to further computation within a thread. Also note that in a finite program, each of these conditions can be executed only a finite number of times by a given thread within a synchronization point, since the first and third cases depend on the number of subgoals in $local\_ASCC(T)$, which is finite, and the second case depends on the number of answers to subgoals in $local\_ASCC(T)$ which is again finite.

So far, this argument shows that condition 6 of Figure 5.3 will be reached for each thread in $threads(true\_ASCC(T))$. Next, consider the actions for condition 6 of Figure 5.3.

1. The condition in line 7 fails: *busy* is true, which means that a thread $T' \in threads(known\_ASCC(T))$ may be deriving or returning answers, so that T' will suspend.

2. The condition in line 7 succeeds: in which case a synchronization point has been reached.

Note since $threads(known\_ASCC(T))$ is finite, and since case 2 must occur if T′ is the only non-suspended thread in $known\_ASCC(T)$, a synchronization point must be reached for $known\_ASCC(T)$ from *state*.

Actions at a Synchronization Point Note that by the argument above, each thread will perform any available computational work – either by returning answers or adjusting its local leader – before reaching Condition 6 of Figure 5.3 and suspending or entering a synchronization point.

More formally, let $State_{S1}$ be the state of a synchronization point for $true\_ASCC(T)$, and $State_{S2}$ be the next synchronization point. Also let a T' be a thread in $threads(true\_ASCC(T))$. Then if at $State_{S1}$, the local leader of T' is not equal to the oldest subgoal owned by T' in $known\_ASCC(T')$, or if there are unreturned answers for T', the local leader will be adjusted for T' and the answers returned before $State_{S2}$.

Thus, between $State_{S1}$ and $State_{S2}$ either some thread T in $threads(true\_ASCC(T))$ performs one of the steps in the previous paragraph (i.e. makes computational progress), or no such thread performed any work. Note that there can be only a finite number of synchronization points in which some thread makes computational progress.

Next, consider a case in which no computational progress has been made between two synchronization points. There can be three cases:

- The condition in line 7.1 of Figure 5.3 succeeds. In this case there are T', T'' such that $LastRound(T', T'') < round(T'')$. Thread T' will be awakened in line 6.1.1, and in some future synchronization point $state_{S2}$, $LastRound(T', T'') = round(T'')$ because thread T'' will execute at some point (because there is a fair scheduling policy) and procedure UpdateDeps will update $LastRound(T, T')$. Since there is no computational progress, $round(T'')$ will not be incremented and there will be a finite number of steps between $state_{S1}$ and $state_{S2}$.

- The condition in line 7.2 of Figure 5.3 fails. In this case there are $T', T'' \in Threads(T))$ such that $Thread\_Set(T') \neq Thread\_Set(T'')$. Threads T' and T'' will be awakened by line 7.3.1 and in some future synchronization point there will be $Thread\_Set(T') = Thread\_Set(T'')$ because both T' and T'' will execute at some point (because there is a fair scheduling policy) and procedure UpdateDeps will eventually update the TDLs for either T' or T'' since both T' and T'' belong to $threads(true\_ASCC(T))$ (because we are not considering the case that $true\_ASCC(T) \not\subseteq true\_ASCC(T') = true\_ASCC(T'')$ in which case $true\_ASCC(T)$ wouldn't be independent.).

- The condition in line 7.1 fails and the condition in line 7.2 succeeds. We have reached the conditions of Theorem 5.3.1 and completion may take place

In cases 1 and 2, even if no computational progress is done, the information contained in the TDLs increases monotonically.

To summarize, we have shown that at any state *state* of a tabled computation in which a thread T is computing subgoals in a non-completed $true\_ASCC(T)$ a future synchronization point is attainable for $true\_ASCC(T)$ (not necessarily executed by T). We next showed that there can be only a finite number of synchronization points for $true\_ASCC(T)$ that either make computational progress or increase the information in the TDLs. Thus, step 6.2.1 and 6.2.2 of Figure5.3 will be reached and liveness shown.

∎

5.3.3 Note on Complexity for Concurrent Completion

We note that Concurrent Completion doesn't change the complexity of the SLG-WAM for single threaded programs. In such cases the *TDL* will be empty and all the functionality added by the Concurrent Completion will result in constant time computations, except for the WakeDependentThreads in the CompleteTop procedure (see Figure5.10). But that won't add to the complexity of the execution as procedure fixpoint_check which has the same complexity will also be called.

5.4 Open Aspects of Concurrent Completion

In this section we wish to note some points about the Concurrent Completion algorithm which have note been completely solved. The first point is that, in a program which has a consumer subgoal $S1$ in one thread that depends on a producer subgoal $S2$ in other thread, if the producer thread at some point is slower than the consumer thread, a parallel execution will degenerate in a sort of Shared Completed Tables, i.e. the consumer will wait for the producer to complete. This is because the WakeDependentThreads procedure is only called at the CompleteTop procedure which is called at completion time. An obvious solution would be to call WakeDependentThreads before the call to fixpoint_check, therefore waking all the threads that might have new answers computed by the previous round of computation. Another one would be to wake up the dependent threads when a new answer is derived. We experimented a little with this alternative solutions, but no obvious gain was achieved, so we left it in its original form. Further experiments are needed to choose the appropriate solution to exploit parallelism.

The second point is support for negation. We tried to implement a shortcut to support negation for programs without inter thread dependencies by checking if the TDL is empty. However the XSB early completion mechanism interfered with the concurrent completion algorithm, and we had to turn it off, for the Concurrent Completion engine. As early completion is deeply ingrained in the XSB implementation of negation, it turns out that some programs with negation are no longer correctly evaluated by this engine.

A third issue has to do with the consumer choice point list, which includes external consumer choice points and traverses threads, thus preventing the automatic growing of the choice point stacks (because there are pointers from other threads onto it). This list has to be changed to more robust data structure, that allows the growing of the choice point stacks. One possible change would be having separate lists for internal and external consumer choice points — while the first would be kept as an ordinary list, the second would be changed to something different.

Chapter Summary In this chapter we presented Concurrent Completion, a method for sharing tables which allows a thread to generate answers for a table while others are reading it. We present the modifications to the SLG-WAM data structures and instruction set to support Concurrent Completion. This changes include the introduction of the external consumer choice points, which allow a thread to consume answers from tables generated by others, and a new check_complete instruction which allows completion of SCCs scattered among different threads. The completion instructions makes use of the thread dependency lists (TDL) to keep track of the inter-thread dependencies. We give an example of execution of Concurrent Completion and discuss

the correctness of the implementation. We finish with some details of Concurrent Completion which deserve further study.

6

Related Work

In this chapter we analyze some approaches that have been taken to the different approaches to integrate concurrent and parallel execution models with logic programming, with the goals of, either speeding up execution through parallel execution on multi-processors, or allowing concurrent execution to enhance responsiveness by allowing several processes or threads in a single program.

The material can be divided into systems that allow the programmer to specify which actions of the program can be executed in parallel (the explicit parallelism approach) and systems that transparently to the programmer, allow the parallel of logic programs. While the first approach can be used to either to support concurrency within a logic program, for instance to increase responsiveness within a distributed system or to take advantage of parallel hardware, the second has as it's only goal to take advantage of parallel hardware to speedup execution.

In Section 6.1 we cover the explicit parallelism approach, taking a particular focus on multi-threaded Prolog systems. Multi-threaded Prolog systems have been around for a long time and we review some of them, owing to their historical or practical importance. We finalize the section with a table summarizing the features of those systems and ours (Table 6.1).

In Section 6.2 we review the implicit parallelism approach, focusing on tabling systems. Although this approach is rather different than the one taken by our systems, the problems of completing a set of tables in parallel and accessing the shared table space are very similar. One of the proposals discussed, OPTYap, takes the form of a real system, publicly available and yielding good speedups, which we try to analyze in more detail.

In Section 6.3 we examine some proposals for executing tabling programs where

the tables are distributed among different memory spaces. The problem of computing tabling in a distributed memory environment is much harder than the one of computing tabling in parallel in a shared memory computer, and accordingly this work is more experimental than the ones mentioned in the previous sections. However we feel that this is an interesting problem and analyze two systems that were developed to compute distributed tabling.

6.1 Multi-Threaded Prolog

In the early days of concurrent logic programming, the researchers were trying to integrate concurrency in the semantics of logic programming by designing new languages that changed the semantics of Prolog. One approach, the so called committed choice languages, like Parlog [15], Concurrent Prolog [68] and FGHC [82] imposed restrictions to backtracking, some going all the way to eliminate it completely and substituting Prolog's don't know determinism with don't care determinism – where only one of the branches of the search tree would be explored, without caring which.

Another approach was taken by Delta Prolog [21], which allowed multiple Prolog processes to communicate through message passing, where the message exchanging events, instead of committing execution to the exchange of that message, allowed, through backtracking, for the communication event to be retried. This feature was called distributed backtracking and allowed a Delta-Prolog program to have a declarative semantics, based on the theory of Distributed Logic [50]. It also implied that execution of a Delta-Prolog program could involve backtracking through multiple communication events, incurring in high complexity for the time of execution.

At some point there started to appear more pragmatic systems that relied on a multi-process Prolog model, using message passing to communicate, but where the message passing was deterministic and treated as a side-effect to the Prolog process execution, but allowing the Prolog's semantics for each process. Two examples examples of such systems were CS-Prolog[1] [32] and PMS-Prolog [87]. With the advent of multi-threaded programming this approach was adapted, so that instead of multiple isolated Prolog executors, each with its own clause database, there were multiple Prolog executors sharing the clause database and other process specific resources. These were the multi-thread Prolog systems, which we examine in this section, following a roughly chronological order. In the end we examine Logtalk, which while not being a Prolog system, allows some very useful concurrency constructs over multi-threaded Prolog systems. For each system we give the implementation of the multi-threaded program to generate prime numbers from Section 3.2[2]. We only show the `master` and `worker` predicates, the rest of the program being regular Prolog and essentially the

[1]CS-Prolog also allowed the non-deterministic semantics for communication of Delta-Prolog, but strongly advised programmers to use the deterministic communication model.

[2]This is example was inspired in one from the Logtalk manual [53].

```
worker( Port, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    write_pipe( Port, primes(List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    pipe(In1, Out1),
    pipe(In2, Out2),
    t_fork( worker(Out1, 1, H, L, L1) ),
    t_fork( worker(Out2, H1, N, L1, []) ),

    read_pipe( In1, primes(L,L1) ),
    read_pipe( In2, primes(L1,[]) ).
```

Figure 6.1: A multi-threaded program to generate prime numbers in IC-Prolog II

same as the XSB example.

IC-Prolog II The first of multi-threaded implementation to be reported was IC-Prolog II [14]. IC-Prolog II provided the primitive `t_fork/1` to create a thread to execute a new subgoal. Within the same process threads communicate through *pipes*. These are roughly similar to Unix pipes, but are message (term) oriented instead of byte oriented. The primitive `pipe/2` creates a new pipe and returns both its endpoints, the input port and the output port. The primitives `read_pipe/2` and `write_pipe/2` allow reading and writing to the pipe.

Threads in different processes can use TCP/IP primitives for communication. However IC-Prolog II also supplies a transparent communication mechanism, in the form of *mailboxes*, whose location is transparent to the processes that intervene in the communication, and which can be used for any thread in any process/machine to communicate with any other. Mailboxes may be bound to a global name, and the sending and receiving of messages is done in strictly FIFO order, not being possible to retrieve a message that is not the first, but unifies with a given term.

IC-Prolog II didn't rely on native threads (the pthreads standard was only issued later) and its threads were implemented at user level with a built-in scheduler, which supported preemption. Figure 6.1 shows the multi-threaded program to generate prime numbers in IC-Prolog II.

PVM-Prolog The author participated in the development of PVM-Prolog [22, 45, 46] the extension of a Prolog core system produced at UNL with multi-threading and an interface with the PVM System [72]. While multi-threaded programming allows the Prolog process to react concurrently to external events, the PVM System allows multiple Prolog processes to be spawned and communicate among themselves. Threads

```
worker( Qi, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    q_put( Qi, primes(List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    q_create( q1, 2, 65536, Q1 ),
    q_create( q2, 2, 65536, Q2 ),
    t_create( worker(Out1, 1, H, L, L1) ),
    t_create( worker(Out2, H1, N, L1, []) ),

    q_get( Q1, primes(L,L1) ),
    q_get( Q2, primes(L1,[]) ),
    q_destroy( Q1 ),
    q_destroy( Q2 ).
```

Figure 6.2: A multi-threaded program to generate prime numbers in PVM-Prolog

communicate through deterministic FIFO term queues. PVM-Prolog's multi-threading is done by an internal scheduler, at the WAM level, and doesn't support true parallel execution among threads of the same process. The system was used as the implementation base for some research projects like in CAP [67] and GroupLog [4]. Figure 6.2 shows the multi-threaded program to generate prime numbers in PVM-Prolog. In PVM-Prolog there was a memory limit for each term queue; if a queue would reach its maximum memory size, the following `q_put` operations would block. This was though to be essential so term queues wouldn't exhaust the process's memory. There's also the need to create two queues because terms can only be retrieved in FIFO order (it isn't possible to use unification to match an arbitrary term in the queue).

SICStus MT A multi-threaded version of the commercial system SICStus Prolog has been reported [31]. Threads are implemented at user level, by a built-in scheduler, with preemption. For communication, each thread is associated with a term queue from which it can consume terms. An arbitrary term can be retrieved from a queue using unification. The sender thread sends terms directly to the receiver's thread term queue[3]. In Figure 6.3 we show the prime number example in SICStus MT, showing the use of the private term queues.

BinProlog BinProlog[4] is a multi-threaded commercial Prolog which builds on the research efforts of Paul Tarau, implemented on Windows and UNIX platforms using native threads. BinProlog has a `synchronize/1` primitive that allows a set of goals to

[3]Those were later called in the proposed ISO standard "private term queues" as the term queue is owned by the receiver thread.

[4]http://www.binnetcorp.com/BinProlog

```
worker( Master, Id, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    send( Master, primes(Id,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    self( Master ),
    spawn( worker(Master, Id, 1, H, L, L1), _ ),
    spawn( worker(Master, Id, H1, N, L1, []), _ ),

    receive( primes(Id,L,L1) ),
    receive( primes(Id,L1,[]) ).
```

Figure 6.3: A multi-threaded program to generate prime numbers in SICStus MT

```
worker( Id, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    local_out( primes(Id,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    bg( worker(p1, 1, H, L, L1) ),
    bg( worker(p2, H1, N, L1, []) ),

    local_in( primes(p1,L,L1) ),
    local_in( primes(p2,L1,[]) ).
```

Figure 6.4: A multi-threaded program to generate prime numbers in BinProlog

be run atomically. This is essentially the same functionality that other multi-threaded Prolog systems support through mutexes. BinProlog uses the Linda [9] tuple space model intensively, both among threads in a process, a local tuple space, and among threads scattered through processes over the network, a global tuple space. It also supports an high-level interface to sockets, inspired on Java's, for high performance communication. BinProlog allows the *migration* of threads [79] among processes; i.e. a thread may be interrupted, have its state sent over the network, and be continued on another process.

Jinni [80], a multi-threaded light-weight Prolog engine implemented in Java, can run as a sibling process to the BinProlog process, using the same communication primitives, and allowing for interoperability with other Java code. Figure 6.4 shows the multi-threaded prime numbers generation program in BinProlog. The threads are using the tuple space local to the process to communicate.

```
:- concurrent primes_q/3.

worker( Id, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    assert( primes_q(Id,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    launch_goal( worker(p1, 1, H, L, []) ),
    launch_goal( worker(p2, H1, N, L1, []) ),

    retract( primes_q(p1,L,L1) ),
    retract( primes_q(p2,L1,[]) ).
```

Figure 6.5: A multi-threaded program to generate prime numbers in Ciao

Ciao Prolog Ciao Prolog [7] supports a primitive inter-thread synchronization and communication mechanism through the so called *concurrent* predicates [10]. Concurrent predicates are declared by using the `concurrent` directive. In a producer-consumer model, the writer (producer) thread uses `assertz/1` to write a fact at the end of the buffer and the reader (consumer) thread uses a standard predicate call to read a fact or `retract/1` if it wants to consume the fact. The predicate call to a concurrent predicate blocks if there is no fact to read. The writer process may use the `close_predicate/1` to signal the consumer that it won't be adding any more facts, which will cause the reader to fail when trying to read the next fact. There are also non blocking versions, `call_nb/1` and `retract_nb/1` of the reading predicates that never block. Figure 6.5 shows the multi-threaded program to generate prime numbers in Ciao. This example doesn't really show the true potential of concurrent predicates as only two pre-determined facts are passed among threads. Essentially the same program using standard shared `assert` and `retract` could be done, provided that the master thread would wait for the worker threads to finish (with `join_goal/1` in Ciao). Ciao's multi-threaded implementation uses native threads from the operating system and is capable of parallel execution.

Qu-Prolog Qu-Prolog was designed as a language to support interactive theorem provers. It supports a multi-threaded programming model [16] based on user level threads, implementing a built-in scheduler with preemption. The primitives `thread_forbid` and `thread_resume` can be used to disable and later enable preemption, allowing the programming of atomic actions without the use of mutex locks. Threads are identified by a triple $< Thread-id, Process-id, Machine-id >$ on the distributed system and communication is done by explicitly naming the sender and receiver threads. Between each pair of threads a communication channel is established

```
worker( I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    m_primes(List,Tail) ->> creator.

master( N, L ) :-
    partition_space( N, H, H1),

    thread_fork( Worker1, worker(1, H, L, L1) ),
    thread_fork( Worker2, worker(H1, N, L1, []) ),

    m_primes(L,L1) <<= Worker1,
    m_primes(L1,[]) <<= Worker2.
```

Figure 6.6: A multi-threaded program to generate prime numbers in Qu-Prolog

from which the receiver thread can read terms in FIFO order, with the `ipc_recv/4` primitive, or used the `ipc_peek/5` to try to find a message that unifies with the argument. In both cases there is an option, `timeout`, which can be set to `poll`, `block` or an integer N, to specify the communication to be respectively non-blocking, blocking or blocking with a timeout of N seconds. An option to remember variable names is also provided, allowing for multiple messages to share variables among themselves. The messages are associated with the sender address and a reply-to address. Unification can be used either in the message itself or on the sender's address to fetch a particular message from the queue. This essentially corresponds to a sophisticated form of private message queues, which can be used transparently over the distributed system. The communication primitives are used to implement higher level communication predicates for exchanging messages, which included guarded communication. Qu-Prolog also allows for the shared predicate database of a process to be used for synchronization through a special meta-call `thread_wait` that allows the call to `retract/1` or `clause/1` to suspend. A Linda tuple space is implemented using these primitives.

In Figure 6.6 the multi-threaded program that generates prime numbers is shown, using the high-level communication mechanism of Qu-Prolog. Notice the ability to name the sender in the message receiving predicate.

SWI-Prolog SWI-Prolog [85] is a popular open source Prolog system, that supports native threads under UNIX and Windows [86]. SWI-Prolog's API for thread management, communication and synchronization is very similar to the one for XSB, presented in Section 3.2. This is the interface defined by the proposed ISO standard for the multi-thread extensions to Prolog [52]. SWI-Prolog was the original implementation of this interface. It also has a `thread_signal/2` predicate, which allows a thread to interrupt another. The interrupted thread is given a goal to execute – this might be to throw an exception and interrupt the regular flow of control of the thread. `thread_signal` is also on the proposed ISO multi-threading extension. SWI-Prolog also supports thread

```
worker( Q, Id, I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail),
    thread_send_message( Q, primes(Id,List,Tail) ).

master( N, L ) :-
    partition_space( N, H, H1),

    message_queue_create(Q),
    thread_create( worker(Q, p1, 1, H, L, L1), _, [detached(true)]),
    thread_create( worker(Q, p2, H1, N, L1, []), _, [detached(true)]),

    thread_get_message( Q, primes(p1,L,L1) ),
    thread_get_message( Q, primes(p2,L1,[]) ),
    message_queue_destroy( Q ).
```

Figure 6.7: A multi-threaded program to generate prime numbers in SWI-Prolog

private dynamic predicates, however, unlike XSB, dynamic predicates are by default shared among threads. Another Prolog that supports threads following the proposed ISO multi-threading extension is Yap [19].

SWI-Prolog implements garbage a collector for the atom space suspending all the threads and then running the global garbage collector and it allows garbage collection of retracted shared predicates. In Figure 6.7 we present the prime number generation example in SWI-Prolog.

P# P# [17] is a Prolog compiler for the .NET framework [49]. It translates Prolog code into C#, which is to be later translated to MSIL, the .NET intermediate language, and allows interoperability with C#. P# allows a thread to be created through the `fork/2` primitive that has a goal to execute in the first argument and return an handle to the C# object that is created to evaluate that goal in the second argument. P# differs from the other multi-threaded Prolog systems presented in this section, in that variables may be shared among the newly created thread and the creator thread. One of the threads may use the `wait_for/1` predicate to wait for the variable to be bound. Repeated uses of the `wait_for/1` primitive will return successive bindings of the variable, done through backtracking. It also supports atomic execution through the `lock/1` and `unlock/1` primitives. The primitive `backtrackable_lock/1` sets a lock which is unlocked by failure over the primitive. The regular Prolog database predicates (assert et al) refer to a thread private database. To access shared predicates, the global table predicates must be used. We couldn't find anything about remote communication in P#'s documentation, so we assume that the interoperability with C# must be used to that end.

We should notice that, whereas there are variables which are syntactically shared among threads, there is really a hidden copy of the variable value from one thread's

```
worker( I, F, DiffList) :-
    list_of_primes( I, F, List, Tail),
    DiffList = List - Tail.

master( N, L ) :-
    partition_space( N, H, H1),

    fork( worker(1, H, D1) ),
    fork( worker(H1, N, D2) ),

    wait_for( D1 ),
    wait_for( D2 ),

    L - L1 = D1,
    L1 - [] = D2.
```

Figure 6.8: A multi-threaded program to generate prime numbers in P#

stacks to another, at the time of the binding of the variable, and which triggers success of the `wait_for/2` predicate. This is done using the `global_assertz` primitive. In Figure 6.8 we show the prime number generation example programmed in P#. It shows that for complex data structures the shared variables aren't a clear improvement on the message passing used by other multi-threaded Prolog systems.

Logtalk Logtalk [51,53] is an object oriented extension for Prolog that has been implemented over most popular Prolog systems, including XSB[5]. Logtalk provides an object oriented interface to multi-threading which is similar to Java, including `threaded` objects, `synchronized` methods and the `notify` and `wait` primitives.

Logtalk allows the starting of a goal in a thread by the `threaded_call/1` predicate. The results may be retrieved through a call to the `threaded_exit/1` predicate, which allows to retrieve the bindings of the variables resulting from the execution of the goal.

Logtalk also supports a form of And-parallelism, so that a conjunction of goals is executed in parallel, in different Prolog threads. This is illustrated in Figure 6.9 for the prime number generation example. Another mechanism supported by Logtalk is `threaded_race` which evaluates in parallel a number of goals and allows the user to retrieve the answer returned for the first one to finish. This may be seen as a form of or parallelism. And parallelism and Or parallelism are discussed in the next section.

Summary In Table 6.1 we present a summary of some of the most important features offered by the systems discussed in this section, together with our XSB implementation. Native thread support is important because it allows parallelism to be exploited. Thread private dynamic predicates are a most useful features, as Prolog programmers

[5]At the time of the writing Logtalk still didn't support threads in XSB.

```
worker( I, F, List, Tail) :-
    list_of_primes( I, F, List, Tail).

master( N, L ) :-
    partition_space( N, H, H1),

    threaded( (
          worker(1,H,L,L1),
          worker(H1,N,L1,[]) ) ).
```

Figure 6.9: A multi-threaded program to generate prime numbers in Logtalk

like to use assert and retract to keep state information, and allowing state to the computation of a thread is an important feature. Remote communication mechanisms are important in implementing distributed applications. Higher level layers to basic multi-threading functionality, like the one from Logtalk, are appearing, to make the use of multi-threading more transparent to the programmer.

6.2 Parallel Tabling over Shared Memory Environments

In this section we address proposals for using implicit parallelism with tabling in shared memory systems. Basically Logic Programs provide two main sources of parallelism – **Or parallelism** which allows the parallel execution of different clauses of a predicate and **And parallelism** where the literals of a clause are evaluated in parallel. And-parallelism can be classified in dependent and independent subclasses, dependening on whether the goals that are executed in parallel are able to share logical variables or not. The first class implies communication among parallel goals, and is thus more complex to implement.

Aurora [44] is an example of an early Or-parallel implementation of Prolog. Today, Or-Parallel Prolog implementations are available for popular Prolog systems, like SICStus (Muse) [2] and Yap (YapOr) [60]. An early implementation of an independent And-Parallel system was &-Prolog [38]. The Andorra I system is an example of an implementation of dependent And parallelism[6] [18]. For a survey of the various dimensions involved in the implementation of and and or parallelism, see, for example [37].

We consider a new source of parallelism is present in tabled programs, by considering the evaluation of each tabled subgoal as a parallel process. This is referred to as table parallelism.

[6]The Andorra model not only provides dependent And parallelism, but also extends Prolog clauses with flat guards, effectively subsuming FGHC [82] which itself belongs to the group of committed choice languages mentioned in Section 6.1; The Andorra model also combines And parallelism with Or parallelism.

	Native Threads Support	Primitive Inter-Thread Communication Model	Thread Private Dynamic Predicates	Remote Communic. Model
IC-Prolog II	No	Pipes	No	TCP/IP Mailboxes
PVM-Prolog	No	Global Term Queues	No	PVM
SICStus MT	No	Private Term Queues	No	Sockets
BinProlog	Windows Linux Solaris	Linda	No	Java Like Sockets Linda
Ciao	Posix	Concurrent Predicates	No	Sockets
Qu-Prolog	No	Private Term Queues Shared Predicates	No	Private Term Queues
SWI-Prolog Yap	Posix Windows	Private and Global Term Queues Signals	Yes	Sockets
P#	.NET	Shared Variables	Yes	Relies on C#
XSB	Posix Windows	Global Term Queues	Yes	Sockets

Table 6.1: Summary of the features supported by several Multi-Threaded Prolog systems

6.2.1 Proposals for Parallel Evaluation of Tabling

Several proposals have been made concerning the parallel execution of tabling; however to our knowledge only the OPT model has been implemented.

Linear Tabling Proposal In [35] an implementation of tabling is proposed that doesn't involve the suspend and resume mechanism. The calls to table subgoals are saved in the table, and when a variant call is encountered the environment is copied to the new call and the computation of the tabled subgoal proceeds. Termination is detected when after trying all tabled calls, no more solutions are found. This mechanism of implementation involves less changes in the WAM (e.g. no freezing) but may involve the re-computation of Prolog goals, called by the tabled predicates.

Because of the closeness to the WAM, the authors argue that this implementation should be easier to parallelise by using standard parallel Prolog technology. However we are not aware of such parallel implementation being available so far.

Table Parallelism Table Parallelism [33] considers a source of parallelism for tabled programs consisting of evaluating different tabled subgoals in parallel. The paper proposes an implementation framework, where each process computes a different tabled subgoal. The global table information is shared and the completion stack is replaced by a completion table and a subgoal dependency list which are shared by all processes. The shared subgoal dependency list is used to compute the leader of an SCC. The completion table has several items for each incomplete subgoal, including a color field, used by the parallel completion algorithm that detects completion. This algorithm is a variation of Dijkstra's [29] parallel termination algorithm that contemplates the changing topology of the SDG is used to detect the completion of an SCC. Our Concurrent Completion engine can be used as a start to implement Table Parallelism. Whether using the more sophisticated parallel termination algorithm, instead of our simple Concurrent Completion algorithm, pays off in gains for the parallel execution of tabled programs is an open question.

6.2.2 OPTYap - a Parallel Tabling System

YapTab [59] is an implementation of tabling over the popular Yap [19] Prolog system, similar in many aspects to the SLG-WAM. The main difference is that YapTab doesn't have a completion stack. Instead it has a *dependency frame* area. Each dependency frame corresponds to a consumer choice point, and as such, the dependency frames could have been kept on the choice point stack, but with parallelism in view, they opted to keep them in a separate area, which would be shared in a parallel implementation. The completion algorithm traverses the dependency frames to check if they have unreturned answers, but unlike in XSB it doesn't chain the dependency frames

that have unreturned answers, returning answers only to the first consumer node. Local scheduling is implemented by associating a dependency frame with a generator choice point, so that all the alternatives in the generator choice point are exhausted before the dependency frame starts to return the answers to the parent environments. Unlike XSB, YapTab doesn't support negation for tabled subgoals.

YapOr [60] is a Or-Parallel Prolog implementation based on the Yap system. On Or-Parallel Prolog systems each Or-branch of the search tree is explored by a *worker* which corresponds to a process or a thread, executing in parallel. Or-branches stem from Or-nodes which correspond to the point of execution in which multiple clauses can be executed. Or-Parallel Prologs have two important dimensions: how to differentiate bindings of variables that are common to several workers (conditional bindings), and how to schedule the work among different workers.

The two approaches to support conditional bindings are called environment sharing and environment copying. With environment sharing the stacks are shared among all workers and separate bindings are kept for each worker (for example through *binding arrays*). With environment copying a different copy of the stacks is kept for each worker. These copies are made on demand, using a technique called *Incremental Copying* which minimizes the portions of the stacks to be copied. As the classic Or-Parallel systems are concerned, Aurora uses environment sharing while Muse uses enviroment copying.

The *scheduler* is a significant part of the Or-Parallel implementation, which manages the distribution of work among idle workers and chooses which alternatives to explore. As the search tree includes or nodes within Or-branches and as such there must be a strategy to allocate Or-branches to workers. The scheduler must incur in minimum overhead, maximize the efficiency of the search and preserve the Prolog semantics and as can be easily seen is critical for the success of an Or-Parallel system. YapOr uses the environment copying approach and its scheduler is based on the Muse scheduler.

OPTYap [58, 61, 63] is a combination of YapOr and YapTab, to implement the OPT model. OPT stands for Or Parallelism within Tabling, and allows workers to explore the different alternatives in an Or-Node within a tabled computation.. Whenever a worker doesn't have more answers to the consumer nodes of the subgoal in its SCC and determines that it depends on branches that are outside this SCC, instead of suspending and waiting for the other branches to return answers, it suspends the current SCC, saving the stacks, and becomes ready to explore other alternatives. The saved SCCs can be reloaded later, for example by workers working on nodes which depend on such SCCs or by idle workers, to which the scheduler attributes this SCC. Each suspended SCC is linked to the Or-Frame in which it arose so that the worker knows which SCCs are suspended on its branch. Completion is only possible when (a) there are no answers to return to this SCC, (b) there are no other workers active or unexplored branches in any Or-Nodes of this SCC and (c) there are no saved SCCs to which

answers can returned. This approach is very different from our completion schemes, in that we leave the control of the execution of the threads to the programmer, while the OPT-Yap scheduler can decide whether workers will or will not explore alternatives in the current SCC. OPTYap's work takes in consideration the layout of the shared data area (ensuring that different pages will be used to access data structures that may be accessed by different workers, to avoid memory contention problems) which can be compared with our structure managers, although OPTYap has done much better in practice. OPTYap also explores different policies for locking the shared table space (the tries) at various levels – this is further explored in [62] — in our work we followed the much simpler approach of either locking the entire call trie or avoid locking the answer tries at all. As YapTab, which is its base, OPTYap doesn't support negation of tabled subgoals. OPTYap gives very good speedups for tabled programs in multi-processors, as reported in [62], which is even more impressive when one considers that Yap is already a very fast system.

6.3 Tabling over Distributed Memory Environments

Computing tabling in a distributed memory environment, where there is no single view of the global state, is also a much harder problem then supporting concurrency in a shared memory environment. Both the determination of the topology of the SDG and the detection of quiescence is a significantly harder problem, which solution involves additional costs in performance. However, other than for eventual gains in performance, such approaches will pay off in case of the need to maintain tables for subgoals in different geographic locations, where the tabling system must reason about geographically distributed information to achieve a global result. Another advantage of distributed tabling may be the possibility of the use of the huge memory space that is available in distributed computers, and which might allow some problems which require very large tables to be solved.

Distributed XSB In [39] XSB was extended to allow the distributed evaluation of tabled subgoals. The dependencies among subgoals are encoded in a number that is attributed to each subgoal and the dependency graph is kept distributed, each process maintaining its subgoal dependencies through the subgoal numbers. For instance if subgoal G_1 with number 1.2 called subgoal G_2, G_2's number would be 1.2.1. Subgoal G_1 would depend all subgoals that have its number as a prefix, subgoals 1.1, 1.2 and 1.2.1. As in the tabled evaluation there may be cross dependencies, and additional dependency list has to be kept for each subgoal. Termination detection is based on the Dijkstra-Scholten algorithm message counting algorithm [30]. The complexity associated with maintaining the distributed dependency and detecting completion information is N^3 in messages. The dependencies (subgoal numbers) must be passed among

processes, and as such the messages can get arbitrarily big.

For the implementation of this work, the XSB process was extended to allow concurrent computations, by using the freeze mechanism. In this way different threads shared the same SLG-WAM and space reclamation in the stacks was impossible. This allowed for the implementation of a running prototype, but wasn't extended to support the execution of real applications.

Distributed Tabling Architecture In [23] a distributed tabling system is proposed. The implementation uses classes of processes: the goal manager, the table manager, the table storage clients and the table storage clients. The goal manager provides an interface between the distributed tabling system and the outside world. The table manager decides on the location of the tables among the several table storage clients, guaranteeing the uniqueness of the tables. The table storage clients store the tables of tabled subgoals and provide for the returning of answers to table prover clients. The prover clients are the inference engines that prove the subgoals.

Dependency information is centralized on the table manager, and as such, the N^3 message complexity of the previous system is overcome. To detect completion of the SCCs the credit recovery algorithm [48] for detecting distributed termination is used. This is based on distributing a number of credits to the processes, which return them when they are done. When all credits are returned, the computation has terminated. This has N message complexity. A clever representation of the number of credits attributed to each process allows to keep a precise account of the credits used by each process using messages of constant size.

The implementation of the system was done through a Prolog meta-interpreter using PVM-Prolog. The different processes react to concurrent external events using the multi-threaded features of PVM-Prolog and communicate via the PVM interface. It is described in [3].

Chapter Summary In this chapter we presented several multi-threaded Prolog systems, showing the evolution of the multi-threaded Prolog programming model until the current ISO standard proposal. We discuss the Prolog implicit parallelism approach applied to tabling, discussing an Or-Parallel Tabling system, OPT-Yap. In the end we presented some approaches taken to the problem of distributed memory computation of tabling.

7

Performance Analysis

In this Chapter we analyze the performance of the multi-threaded XSB system.

We measure the performance of two engines:

- The engine using Shared Completed Tables as the concurrent tabling algorithm and Local as the scheduling policy
- The engine using Concurrent Completion as the concurrent tabling algorithm and Batched as the scheduling policy

These engines are available in XSB's cvs repository [1][2]. The benchmark programs themselves are available in the repository[3].

We measure the overhead of each of these engines vs. the sequential engine, and tested their scalability both on a dual core processor system and on a multi-processor. We also used a single processor system to show some of the properties of the Shared Completed Tables. The times taken were the CPU times for the overheads section and the elapsed time for the other two sections.

The measurements were taken from the following systems:

- A 2 GHz Intel Pentium M Laptop with 1 GB of memory, running Fedora Core 6 Linux.
- A 2.13 GHz Intel Core 2 Duo Desktop with 2 GB of memory, running Fedora Core 6 Linux.

[1] `xsb.sourceforge.org`

[2] The version used in the tests in this chapter is tagged `mt_thesis` and will soon be released as version 3.1.

[3] The benchmarks programs are in the module `mttests` tagged as `mt_thesis`.

- A 2 GHz Intel Core 2 Duo Laptop with 2 GB of memory, running Mac OS X version 10.4.8.

- A 16 processor SPARC[4] running Solaris 6.

We'll refer to the first system as the "mono-processor system", to the second system as the "Linux system" or as the "Dual Core system", to the third system as the "Mac OS X system" and the fourth system as the "Solaris system" or as the "Multi-processor system".

In all measurements the engines were compiled using maximum optimization allowed by the system compiler. All times were taken as the best of three runs.

All the benchmarks were run a number of iterations, which tried to make different benchmarks take times of approximately the same magnitude, except for the random graph benchmarks used in Section 7.3 which run the programs only once.

7.1 Overhead for Execution of One Thread

In this section we analyze the overhead of supporting multi-threading in the engine for the execution of a single thread. This is important to show that the multi-threaded system can efficiently run single threaded programs.

As we saw in Chapter 3 there are two main factors that introduce overheads: locking shared data structures (e.g. the clause database) and using the thread context local variable to access SLG-WAM variables (e.g. the WAM EREG register). The use of local variables also implies the existence of an extra parameter in function calls to pass the context data among different functions.

For programs with tabling there are some more specific overheads, relating to the different SLG-WAM instructions for concurrent tabling (as seen in chapters 4 and 5) and the use of the *dispatch block* to select the table information frame for the actual thread (as seen in Chapter 3).

7.1.1 Prolog Execution

Here we examine the execution of Prolog programs. Deriv, nrev, qsort,serialise and query are the original WAM benchmarks[5], which are run a number of times for each test. Tak is a benchmark that runs the Ackermann function[6] . Compiler is the XSB compiler which is used to compile a file.

Query stresses choice point creation and use, particularly with regard to shallow backtracking; nrev (naive reverse) stresses trail-recursive list traversal that does not require choice point creation; deriv (computation of the derivative of a polynomial) tests

[4]For site policy reasons we were only able to use 8 processors.
[5]Designed by D. H. D: Warren
[6]Designed by C.R. Ramakrishnan

Lock	symbol	string	load undef	dynamic	io	gentag
Benchmark						
deriv	9	50	0	2	0	0
nrev	9	50	0	2	0	0
qsort	9	50	0	2	0	0
serialise	9	51	0	2	1	0
query	9	50	0	0	0	0
tak	9	50	0	0	0	2
compiler	71416	15323	27	1554	4	0

Table 7.1: Lock usage for Prolog benchmarks

Benchmark	Sequential engine	Multi-threaded engine	Overhead
deriv	1.11s	1.21s	9%
nrev	2.12s	2.06s	-2%
qsort	1.21s	1.23s	2%
serialise	1.31s	1.47s	12%
query	1.47s	1.48s	1%
tak	0.82s	0.78s	-3%
compiler	0.33s	0.34s	2%

Table 7.2: Overheads of the multi-threaded engine for Prolog benchmarks for one thread in Linux

trailing and untrailing of variables; tak, the non-primitive recursive Ackermann function, tests determinacy detection within clause selection. Qsort, serialise, and compiler test different combinations of Prolog functionality.

Table 7.1 shows the lock usage for the Prolog benchmarks. The *symbol lock* is the lock on the symbol table, used to store the Prolog functors; the *string lock* is the lock on the atom table; the *dynamic lock* is used for dynamic code; the *load undef lock* is used for dynamic loading of static predicates; the *io lock* is used to synchronize diverse I/O operations; and the *gentag lock* is used for the tag table that must be used in Linux. See Chapter 3 for details on these locks.

This table shows that the WAM benchmarks use few locks, and that the compiler benchmark intensively uses the locks on the string table and on the functor (symbol) data structures.

In Table 7.2 the execution overheads for the Linux system are shown. The results show overheads from 12 to -3%. There is no known relationship between the overhead and known factors of the multi-threaded implementation, and this and other tables that follow suggest that the overhead is somewhat random, based on the patterns of storage of the code in memory, like the alignment of instructions and data in virtual memory pages and cache blocks. These results show a small overhead for some cases,

Benchmark	Sequential engine	Multi-threaded engine	Overhead
deriv	1.74s	1.74s	0%
nrev	3.71s	3.49s	-5%
qsort	1.86s	1.70s	-8%
serialise	0.61s	0.53s	-12%
query	1.75s	1.64s	-5%
tak	1.19s	1.11s	-2%
compiler	0.47s	0.45s	-3%

Table 7.3: Overheads of the multi-threaded engine for Prolog benchmarks for one thread in the Mac OS X system

```
:- table ancestor/2.
ancestor(X,Y) :- move(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- move(X,Y).
```

Figure 7.1: Ancestor with right recursion

while in most cases the overhead is negligible for Prolog.

We were surprised that in the Mac OS X system (see Table 7.3) the multi-threaded system is actually a little faster (0 to 12%) than the sequential engine in most of the benchmarks. However it's noticeable that the times are somewhat slower on the Mac OS X System, which suggests less compiler optimization of the XSB system, since the processor clock is only slightly slower than the one of the Linux system, and the processor is the same (although the mobile version.)

Given the range of overheads for the two platforms tested, it appears that the cost for introducing multi-threading in Prolog is less than random factors accounted by small changes in the execution environment, such as the patterns of code generated by the compiler, the operating system overheads, etc.

7.1.2 Transitive Closure Benchmarks

In this section we examine some simple programs that make intensive use of tabling.

The ancestor relation with right recursion is show on Figure 7.1 and was run over a `move/2` relation configured as a cycle and as chain with 1024 nodes each. It creates a new table for each invocation of the predicate, so our tests create and abolish 1024 tables. Note that running this predicate over a cycle would loop in Prolog.

The ancestor relation with left recursion is shown on Figure 7.2. It was also run

```
:- table ancestor/2.
ancestor(X,Y) :- ancestor(X,Z), move(Z,Y).
ancestor(X,Y) :- move(X,Y).
```

Figure 7.2: Ancestor with left recursion

```
:- table win/1.
win(X) :- move(X,Y), tnot(win(Y)).
```

Figure 7.3: "Win not win" program

over a cycle and a chain with 1024 nodes each. This predicate tables only one subgoal, no matter how many nodes the move relation has. Note that this program relies on the ancestor predicate being tabled, as it would loop in Prolog.

The "win not win" program is shown on Figure 7.3. `tnot/1` is the well founded negation operator for tabled predicates. The program was run over a cycle and a chain of 2048 nodes each. The "win not win" program over a cycle is an example of non stratified negation.

We compare execution of these programs on the sequential engine and on the multi-threaded engine where they were run both with shared tables and private tables.

With tabling the sources of overhead are the same as in Prolog (locks, use of the context data structure and passing of an extra parameter) plus the use of dispatch blocks for private tables.

Shared Completed Tables using Local Scheduling

In this section we analyze the performance of the engine that uses Shared Completed Tables as the concurrent tabling algorithm, and Local as the scheduling policy. As there's only one thread the concurrent tabling algorithm is not the main object of test, but nevertheless it also adds some overhead.

Table 7.4 analysis the usage of the different locks that protect critical sections of code in the multi-threaded engine by the transitive closure benchmarks. The *call trie lock* is used to protect the search/insert operation of a new subgoal in the call trie — it's not really one lock, but a set of locks, one for each predicate's call trie; the *delay lock* is used to protect access to shared delay lists for well founded negation; the *struct manager lock* protects the structure managers (see Chapter 3) – again it's not really one lock, but a set of them, one for each structure manager. The *compl* (completion) *lock* protects the relevant sections of the check_complete and table_try instructions and allows for integration of shared and private tables. For more details on these locks see Section 4.1. The lines referring to the *ancestor* benchmarks show the total lock accesses for both right and left recursion benchmarks.

This table shows that for private tables only the symbol and completion locks get to be used. For shared tables the structure manager lock is very intensively used. For non stratified negation programs (win not win) the delay lock which protects the delay lists of shared tables is also intensively used.

In Linux (see Table 7.5) the overheads for the multi-threading system are somewhat larger (1 to 22% for private tables) than for the Prolog benchmarks. The shared tables show a much larger overhead than the private tables (from 25 up to 120%) probably

Lock	call trie	symbol	delay	string	struct manager	compl
Benchmark						
ancestor chain private	0	9226	0	200	0	7096
ancestor chain shared	10096	9226	0	200	15590492	10096
ancestor cycle private	0	9220	0	201	0	3002
ancestor cycle shared	8050	9220	0	201	15579088	6004
win chain private	0	212	0	187	0	819200
win chain shared	819200	212	0	187	4096800	1638000
win cycle private	0	212	0	186	0	150
win cycle shared	307200	212	307200	186	2150700	307350

Table 7.4: Lock usage for Transitive Closure benchmarks with Local scheduling in the Shared Completed Tables engine.

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
Left rec. chain	1.58s	1.88s	19%	2.44s	55%
Left rec. cycle	1.60s	1.94s	22%	2.48s	54%
Right rec. chain	0.40s	0.44s	12%	0.87s	120%
Right rec. cycle	0.48s	0.55s	13%	0.95s	98%
Win chain	1.38s	1.53s	11%	2.04s	48%
Win cycle	1.00s	1.02s	1%	1.26s	25%

Table 7.5: Times for the Transitive Closure benchmarks for one thread using Local scheduling in the Shared Completed Tables engine in Linux

Benchmark	Private Tables	Without Locks	Lock Impact	Shared Tables	Without Locks	Lock Impact
Left rec. chain	1.88s	1.83s	2%	2.44s	1.86s	31%
Left rec. cycle	1.94s	1.91s	1%	2.48s	1.87s	32%
Right rec. chain	0.44s	0.44s	0%	0.87s	0.45s	93%
Right rec. cycle	0.55s	0.52s	5%	0.95s	0.53s	79%
Win chain	1.53s	1.43s	6%	2.04s	1.54s	32%
Win cycle	1.02s	1.06s	-3%	1.26s	1.10s	14%

Table 7.6: Gain for taking out the locks in the multi-threaded engine with Local scheduling in the Shared Completed Tables engine for Transitive Closure benchmarks on Linux

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
Left rec. chain	2.75s	2.57s	-5%	2.80s	2%
Left rec. cycle	2.90s	2.64s	-8%	2.86s	0%
Right rec. chain	0.63s	0.62s	0%	0.78s	26%
Right rec. cycle	0.72s	0.74s	3%	0.90s	24%
Win chain	2.18s	2.28s	5%	2.48s	14%
Win cycle	1.65s	1.67s	1%	1.78s	8%

Table 7.7: Times for the Transitive Closure benchmarks for one thread using Local scheduling under the Shared Completed tables engine in Mac OS X

due to the structure manager lock.

In Table 7.6 we analyze the result of taking out the locks for single threaded execution of the Transitive Closure benchmarks. These results show very small gains for taking out the locks with private tables and a big gain (almost as big as the overhead for executing the multi-threaded engine) for taking out the locks on shared tables. This table proofs that the big overhead for shared tables comes from the use of locks.

In the Mac OS X system (see Table 7.7) with private tables the multi-threaded system we see a more random pattern of overheads. In some cases the multi-threaded system is faster in others it's slower and in one case the times are the same. This confirms the results of Prolog that show that in the Mac OS X System the multi-threaded system is sometimes faster. With shared tables the multi-threaded system is a little slower, which again we account for the use of locks to access the shared structure managers.

Concurrent Completion using Batched Scheduling

In this section we analyze the performance for the engine that uses Concurrent Completion as the concurrent tabling algorithm and Batched as the scheduling policy. The overhead of the concurrent tabling algorithm is not the main factor being stressed because there's only one thread.

Lock	call trie	cons list	symbol	string	struct manager	compl
Benchmark						
ancestor chain private	0	0	9226	201	0	10096
ancestor chain shared	10096	3000	9226	201	15583400	10096
ancestor cycle private	0	0	9220	201	0	6002
ancestor cycle shared	8050	3002	9220	201	15574040	6002

Table 7.8: Lock usage for Transitive Closure benchmarks using Batched scheduling in the Concurrent Completion engine.

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
Left rec. chain	1.71s	1.96s	15%	2.46s	44%
Left rec. cycle	1.77s	2.03s	14%	2.52s	42%
Right rec. chain	1.30s	1.31s	1%	1.59s	22%
Right rec. cycle	1.20s	1.05s	-12%	1.66s	38%

Table 7.9: Times for the Transitive Closure benchmarks for one thread using Batched scheduling in the Concurrent Completion engine in Linux

With batched scheduling we didn't consider the negation benchmarks, as negation is not completely stable for batched scheduling in the Multi-threaded engine.

Table 7.8 analyzes the usage of the different locks that protect critical sections of code in the multi-threaded engine by the Transitive Closure benchmarks. The *cons* (consumer) *list lock* is the lock that ensures correct management of the consumer lists, including external consumers in the concurrent completion. The *compl* (completion) *lock* forces the check_complete instruction to execute in mutual exclusion – note that this is not really the same completion lock discussed on the last section, although it is similar. See Section 5.1 for details on these locks. This table shows that for private tables only the symbol and completion locks are really used. For shared tables the structure manager lock is very intensively used.

In the Linux machine (see Table 7.9) one of the benchmarks is faster in the multi-threaded engine for private tables and these results seem a lot more random than the ones for shared completed tables which is difficult to explain. For shared tables the results are a little more consistent, and some overhead shows, which we attribute to the structure manager locks.

In Table 7.10 we analyze the changes for compiling the multi-threaded engine without locks for the Transitive Closure benchmarks in Linux. It shows very small to no gains at all for private tables. For shared tables it shows a significant gain, as for Shared

Benchmark	Private Tables	Without Locks	Lock Impact	Shared Tables	Without Locks	Lock Impact
Left rec. chain	1.96s	1.93s	1%	2.46s	1.85s	32%
Left rec. cycle	2.03s	1.97s	3%	2.52s	1.90s	32%
Right rec. chain	1.31s	1.31s	0%	1.59s	1.24s	28%
Right rec. cycle	1.05s	1.05s	0%	1.66s	0.91s	82%

Table 7.10: Gain for taking out the locks in the multi-threaded engine with Batched scheduling in the Concurrent Completion engine in Linux

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
Left rec. chain	2.77s	2.65s	-3%	2.91s	5%
Left rec. cycle	2.95s	2.73s	-7%	2.99s	1%
Right rec. chain	1.80s	1.83s	2%	1.83s	2%
Right rec. cycle	1.46s	1.48s	1%	1.65s	12%

Table 7.11: Times for the Transitive Closure benchmarks for one thread using Batched scheduling in the Concurrent Completion engine in Mac OS X

Completed tables, derived from the elimination of the structure manager lock.

Table 7.11 shows the execution times for the Transitive Closure benchmarks in the Mac OS X system. The private tables show somewhat random overheads (three cases have gains and one is equal) confirming that in the Mac OS X System the multi-threaded engine sometimes performs better than the sequential engine. The overheads for the shared tables are also somewhat random, being in some cases better than the private tables.

These Transitive Closure benchmarks show overheads which are much more unstable. In most cases they are much larger than in Prolog, but in some cases the times are actually much better for the multi-threaded engine. The Concurrent Completion implementation also shows more performance instability than the Shared Completed Tables implementation.

7.1.3 Abstract Interpretation Benchmarks

In this section we examine the behavior of some programs generated by program analysis that make use of tabling mixed with Prolog. These programs were used to measure the timings for the CHAT implementation described in [27]. These programs are more representative of the SLG-WAM instruction mix for 'practical' programs than those of the previous section. The programs are run a number of times and the tables are abolished between runs.

Lock	symbol	call trie	string	struct manager	compl
Benchmark					
cs_o private	378	0	742	0	15200
cs_o shared	581	19800	882	151600	28000
cs_r private	378	0	742	0	7600
cs_r shared	581	17300	882	81000	21200
disj private	376	0	740	0	29600
disj shared	579	53600	880	220800	70800
gabriel private	378	0	742	0	16520
gabriel shared	581	31640	882	157640	42560
kalah private	380	0	744	0	29400
kalah shared	583	61800	884	329700	78000
peep private	378	0	742	0	4600
peep shared	581	13300	882	82400	15700
pg private	377	0	741	0	19200
pg shared	580	26800	881	119200	42000

Table 7.12: Lock usage for Abstract Interpretation programs for Local scheduling in the Shared Completed Tables engine

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
cs_o	1.77s	1.73s	-1%	1.74s	-1%
cs_r	1.70s	1.67s	-1%	1.68s	-1%
disj	2.18s	2.02s	-6%	2.04s	-5%
gabriel	1.70s	1.77s	4%	1.71s	1%
kalah	1.71s	1.72s	1%	1.76s	3%
peep	1.74s	1.68s	-2%	1.70s	-1%
pg	1.71s	1.67s	-1%	1.71s	0%

Table 7.13: Times for the Abstract Interpretation programs for one thread using Local scheduling in the Shared Completed tables engine in Linux

Shared Completed Tables using Local Scheduling

In this section we analyze the performance for the engine that uses Shared Completed Tables as the concurrent tabling algorithm and Local as the scheduling policy. Again the concurrent tabling algorithm's overhead not the main object of test as there's only one thread.

Table 7.12 shows the usage of locks for the Abstract Interpretation programs in the multi-threaded engine. It shows that the structure manager lock is only used with shared tables. Otherwise the completion lock is also used significantly.

In Table 7.13 we see the times of execution of the Abstract Interpretation programs for the Linux machine. It shows a very small, some times even negative, overhead for private tables (-6 to 4%). These are effectively less than the overheads for Prolog. The

Benchmark	Private Tables	Without Locks	Lock Impact	Shared Tables	Without Locks	Lock Impact
cs_o	1.73s	1.72s	0%	1.74s	1.76s	-1%
cs_r	1.67s	1.70s	-1%	1.68s	1.67s	0%
disj	2.02s	2.04s	0%	2.04s	2.06s	0%
gabriel	1.77s	1.70s	4%	1.71s	1.70s	0%
kalah	1.72s	1.76s	-2%	1.76s	1.74s	1%
peep	1.68s	1.72s	-2%	1.70s	1.73s	-1%
pg	1.67s	1.70s	-1%	1.71s	1.70s	0%

Table 7.14: Gain for taking out the locks in the multi-threaded engine with Local scheduling in the Shared Completed Tables engine in Linux

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
cs_o	3.01s	2.39s	-20%	2.41s	-19%
cs_r	2.92s	2.31s	-20%	2.33s	-19%
disj	3.43s	2.71s	-20%	2.74s	-19%
gabriel	2.48s	2.31s	-6%	2.32s	-6%
kalah	2.51s	2.34s	-6%	2.38s	-4%
peep	2.54s	2.35s	-7%	2.35s	-7%
pg	2.63s	2.28s	-12%	2.31s	-11%

Table 7.15: Times for the Abstract Interpretation programs for one thread using Local scheduling with the Shared Completed Tables engine in Mac OS X

shared tables show times that are about the same as the private tables ones, showing insignificant extra overhead.

Table 7.14 compares the times for the multi-threaded engine compiled with and without locks on Linux. The times without locks sometimes happen to be worse than with locks. We think this shows the relatively small usage of locks by these programs (cf. Table 7.12) and the fact that the times are better is due to the random location of instructions in the code pages of the differently compiled programs of the different engines.

Table 7.15 shows the overheads of Abstract Interpretation programs for the Mac OS X system. Like in previous benchmarks, we see significant gains for the multi-threaded engine, specially in the cs_o, cs_r and disj benchmarks.

Concurrent Completion using Batched Scheduling

In this section we analyze the performance of the engine that uses Concurrent Completion as the concurrent tabling algorithm and Batched as the scheduling policy. Again the concurrent tabling algorithm's overhead is not the main object of test as there's only one thread.

Table 7.16 shows the usage of locks for the Abstract Interpretation programs in the

Lock	symbol	call trie	string	struct manager	compl	cons list
Benchmark						
cs_o private	378	0	742	0	13600	0
cs_o shared	581	19800	882	146400	13600	3000
cs_r private	378	0	742	0	6700	0
cs_r shared	581	17300	882	77900	6700	3200
disj private	376	0	740	0	26400	0
disj shared	579	53600	880	209200	26400	20400
gabriel private	378	0	742	0	13720	0
gabriel shared	581	31640	882	152040	13720	12600
kalah private	380	0	744	0	24600	0
kalah shared	583	61800	884	316500	24600	13200
peep private	378	0	742	0	1700	0
peep shared	581	13300	882	80200	1700	9200
pg private	377	0	741	0	3200	0
pg shared	580	26800	881	115200	3200	19200

Table 7.16: Lock usage for Abstract Interpretation programs for Batched scheduling in the Concurrent Completion engine

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
cs_o	1.77s	1.77s	0%	1.80s	2%
cs_r	1.69s	1.68s	0%	1.69s	0%
disj	2.08s	2.04s	-1%	2.06s	0%
gabriel	1.69s	1.71s	1%	1.73s	2%
kalah	1.72s	1.77s	3%	1.78s	4%
peep	1.76s	1.74s	0%	1.74s	0%
pg	1.74s	1.76s	1%	1.74s	0%

Table 7.17: Times for the Abstract Interpretation programs for one thread using Batched scheduling for the Concurrent Completion engine in Linux

multi-threaded engine. It shows that only the completion lock is significantly used with private tables. The structure manager lock is intensively used with shared tables.

Table 7.17 shows the times for Abstract Interpretation benchmarks in the Linux machine. Again we see a very small overhead (from -1 to 3%) for private tables. The overheads for shared tables are also very small (from 0 to 4%) and not significantly different.

In Table 7.18 we compare the multi-threaded system compiled with and without locks in the Linux machine. The results are somewhat similar that the ones for Local scheduling and Shared Completed Tables (cf. Table 7.14). The private tables show a very small impact of the locks on execution time (from 1 to 4%) and the shared tables also show very small impact of the locks on execution time (1 to 4%).

Table 7.19 shows the overheads of Abstract Interpretation programs for the Mac OS

Benchmark	Private Tables	Without Locks	Lock Impact	Shared Tables	Without Locks	Lock Impact
cs_o	1.77s	1.72s	2%	1.80s	1.74s	3%
cs_r	1.68s	1.66s	1%	1.69s	1.66s	1%
disj	2.04s	1.98s	3%	2.06s	2.01s	2%
gabriel	1.71s	1.64s	4%	1.73s	1.66s	4%
kalah	1.77s	1.69s	4%	1.78s	1.70s	4%
peep	1.74s	1.68s	3%	1.74s	1.68s	3%
pg	1.76s	1.70s	3%	1.74s	1.70s	2%

Table 7.18: Gain for taking out the locks in the multi-threaded engine with Batched scheduling using the Concurrent Completion engine in Linux

Benchmark	Sequential Engine	Private Tables	Overhead	Shared Tables	Overhead
cs_o	3.01s	2.39s	-20%	2.41s	-19%
cs_r	2.93s	2.30s	-21%	2.33s	-19%
disj	3.41s	2.71s	-19%	2.75s	-18%
gabriel	2.47s	2.30s	-6%	2.32s	-5%
kalah	2.52s	2.35s	-5%	2.39s	-4%
peep	2.57s	2.37s	-7%	2.38s	-6%
pg	2.64s	2.31s	-12%	2.32s	-11%

Table 7.19: Times for the Abstract Interpretation programs for one thread using Batched Scheduling in the Concurrent Completion engine in Mac OS

X system. Like in previous benchmarks, we see significant gains for the multi-threaded engine, specially in the cs_o, cs_r and disj benchmarks.

The Abstract Interpretation benchmarks show small overheads for Linux and significant gains for the Mac OS X, similar to the Prolog overheads. The results are very regular for the same machine, in contrast with the Transitive Closure benchmarks. As in the other benchmarks the Linux machine presents very small overheads and the Mac OS X System sometimes even presents gains of performance. This results are encouraging for using the multi-threaded engine as the main XSB engine.

7.2 Scalability of Private Tables on a Multi-Processor

In this section we show the results of running the benchmarks discussed on the previous section, simultaneously, in a number of threads, over a multi-processor. For this and the next section we measure the elapsed time, instead of the CPU time, as this is the parameter that really shows that the parallel programs are faster.

As there are N threads doing the same thing as one thread, we calculate the projected speedup in the following way;

$$ProjectedSpeedup = \frac{N.T_1}{T_N}$$

Where T_1 is the time to run one thread, N is the number of threads and T_N is the time taken to run N threads.

As every thread is running an independent program, which has absolutely no logical dependence on the programs that are being run by other threads, and uses private tables, in an ideal system the projected speedup should always be equal to the number of threads (given that the number of processors is greater or equal than the number of threads).

The biggest obstacles to this goal are memory contention which degrades program performance due to hardware problems and lock contention, that forces sections of the code to execute sequentially. We have tried very much to overcome such obstacles for private tables. We provide private structure managers for each thread and use very few locks (cf Tables 7.4, 7.8, 7.12 and 7.16). However, there are still some algorithms and data structures in the underlying runtime system that require locks and that exhibit some memory contention: foremost among these are memory allocation calls. Although we strove to limit the number of such calls, they remain when execution stacks need to be reallocated, when structure managers require the allocation of new blocks, or when trie hash tables are resized.

Scalability for Prolog In Table 7.20 we show the times for executing the Prolog benchmarks in parallel.

N. threads	1	2	Projected Speedup	4	Projected Speedup	8	Projected Speedup
Benchmark							
deriv	0.66	0.64	2.06	0.65	4.06	0.73	7.23
nrev	1.31	1.32	1.98	1.34	3.91	1.40	7.48
qsort	0.56	0.60	1.86	0.66	3.39	0.77	5.81
query	0.64	0.64	2.00	0.66	3.87	0.69	7.42
serialise	0.74	0.74	2.00	0.76	3.89	0.78	7.58
tak	13.06	12.88	2.02	14.13	3.69	18.66	5.59

Table 7.20: Times for scalability of Prolog benchmarks for Solaris

N. threads	1	2	Projected Speedup	4	Projected Speedup	8	Projected Speedup
Benchmark							
Left rec. chain	0.96	1.33	1.44	1.44	2.66	2.09	3.67
Left rec. cycle	0.97	1.34	1.44	1.44	2.69	2.03	3.82
Right rec. chain	0.51	1.00	1.02	1.49	1.36	2.69	1.51
Right rec. cycle	1.46	2.35	1.24	3.56	1.64	6.42	1.81
Win chain	0.85	1.09	1.55	1.46	2.32	4.65	1.46
Win cycle	0.53	0.57	1.85	0.68	3.11	0.76	5.57

Table 7.21: Times for scalability of Transitive Closure benchmarks for private tables using Local scheduling for Solaris

The projected speedups are mostly very close to the ideal ones. Only in qsort and tak they are show speedups which are clearly less than linear. As they are the most complex and demanding programs in the benchmarks that is not completely surprising.

The Transitive Closure Benchmarks The results for running the Transitive Closure benchmarks in parallel, using private tables, are shown in Tables 7.21 and 7.22.

In Table 7.21 the times are shown for the engine using private tables using Local scheduling under the Shared Completed Tables engine. The speedups are much more modest than for Prolog. We attribute this to the need to use the "C" library function *realloc* to grow the node hash tables for the answer tries in left recursion, and call tries in right recursion, as well as to the use of the completion mutex lock (to make integration with shared tables possible).

In Table 7.22 the times are shown for private tables using Batched scheduling under the Concurrent Completion engine. These speedups are even more modest than the previous ones. We attribute this to the longer time by which the completion lock has to be held in the check_complete instruction.

N. threads	1	2	Projected Speedup	4	Projected Speedup	8	Projected Speedup
Benchmark							
Left rec. chain	0.71	1.11	1.27	1.16	2.44	2.09	2.71
Left rec. cycle	0.72	1.12	1.28	1.25	2.30	2.07	2.78
Right rec. chain	0.53	1.09	0.97	2.02	1.04	3.74	1.13
Right rec. cycle	1.34	2.63	1.01	4.76	1.12	8.99	1.19
Win chain	0.75	0.98	1.53	1.57	1.91	4.70	1.27
Win cycle	0.42	0.52	1.61	0.97	1.73	0.96	3.50

Table 7.22: Times for scalability of Transitive Closure benchmarks for private tabled using Batched scheduling for Solaris

N. threads	1	2	Projected Speedup	4	Projected Speedup	8	Projected Speedup
Benchmark							
cs_o	0.80	0.80	2.00	0.83	3.85	1.07	5.98
cs_r	0.72	0.74	1.94	0.78	3.69	0.88	6.54
disj	0.81	0.84	1.92	0.87	3.72	0.98	6.61
gabriel	0.73	0.76	1.92	0.79	3.69	0.89	6.56
kalah	0.66	0.71	1.85	0.72	3.66	0.82	6.43
peep	0.71	0.73	1.94	0.76	3.73	0.85	6.68
pg	0.71	0.74	1.91	0.77	3.68	0.87	6.52

Table 7.23: Times for scalability of Abstract Interpretation benchmarks for private tables using Local scheduling for Solaris

Abstract Interpretation Benchmarks In Tables 7.23 and 7.24 we show the results for running the Abstract Interpretation programs in parallel, using private tables. These benchmarks are, in a sense, more representative of the typical tabled program's performance than the previous ones, because they use real programs that use a more standard mix of Prolog and different features of tabling.

In Table 7.23 we show the results for private tables using Local scheduling under the Shared Completed Tables engine. Although not as good as the results from Prolog, they give a speedup that is nearly linear, up to eight processors, and much better than the ones from the Transitive Closure benchmarks.

In Table 7.24 we show the results for scalability using private tables using Batched scheduling under the Concurrent Completion engine. A little surprisingly, these are even better than the ones for Local scheduling.

We conclude that although a very intensive use of tabling may lead to some performance losses with the current engine, the results for parallelizing real applications with private tables should be, as far as the system is concerned, good, and overall dependent of the parallelism that the programmer can extract from the application.

N. threads	1	2	Projected Speedup	4	Projected Speedup	8	Projected Speedup
Benchmark							
cs_o	0.70	0.70	2.00	0.71	3.94	0.75	7.46
cs_r	0.64	0.65	1.96	0.67	3.82	0.70	7.31
disj	0.72	0.74	1.94	0.76	3.78	0.82	7.02
gabriel	0.64	0.66	1.93	0.66	3.87	0.70	7.31
kalah	0.59	0.60	1.96	0.62	3.80	0.66	7.15
peep	0.63	0.65	1.93	0.67	3.76	0.70	7.20
pg	0.64	0.66	1.93	0.67	3.82	0.84	6.09

Table 7.24: Times for scalability of Abstract Interpretation benchmarks for private tables using Batched scheduling for Solaris

7.3 Experiments with Shared Tables

In this section we analyze the behavior of shared tables and compare it to private tables. To analyze share tables we use a new set of benchmarks, which we describe in the next paragraph. We analyze memory usage, elapsed time and we try to answer the question of how frequent deadlocks are in Shared Completed Tables.

The Random Graph Tests We used a program to generate random directed graphs, that takes a number of nodes and the number of edges that fan out of each node. We designate each graph by these two numbers, as *NxE* where *N* is the total number of nodes and *E* is the number of edges fanning out of each node. Thus, the graph 2048x2 has 2048 nodes, each of which has exactly 2 outgoing edges, for and 4096 edges overall.

We use four graph models, instances of which are randomly generated for every run of the tests:

256x128 A very dense (almost complete) graph with few nodes.

512x8 Although less dense this graph's nodes are still densely connected and tend to form a single component.

2048x2 This is the toughest graph for the programs to handle. Although less dense than the previous it tends to have a single component, and with a fairly large number of nodes, the number of paths among nodes tends to be very big.

8192x1 Although this is the graph with more nodes, as the fan out is small it tends to have more than one component, and so the total number of paths among nodes in the graph tends to be smaller than the previous.

The graphs are stored in a predicate `edge/2` which is indexed by tries. Basically we run two programs over these graphs:

Transitive Closure with Left Recursion The graph is partitioned among the threads. Each thread computes all nodes that are reachable from all the nodes in its partition, using left recursion. So, in the 2048x2 graph, if one thread is used it performs the transitive closure for all the 2048 nodes in the graph, if two threads are used each performs the transitive closure for 1024 nodes of the graph, and so on. This process creates a tabled subgoal for every node in the partition, so each thread computes a separate set of tabled subgoals.

Although not terribly useful in practice, this is an ideal application to parallelize (by partitioning the graph), that uses the resources of the tabling in an optimal way.

Transitive Closure with Right Recursion In this case the graph is also partitioned among the threads, but this time the reachable nodes are computed using right recursion instead of left recursion. This means that for each node, there is the need to create a tabled subgoal for every node that is connected to it, thus recomputing the tables for all the nodes in the component form by every node in the partition. As these graphs tend to have few components, tables are usually computed for most nodes in the graph.

Right recursion is not a recommendable way to program with tabling (due to the creation of many tabled subgoals as opposed to one with left recursion), however there may be cases where it is the natural way to program (e.g. `same_generation/2`, [76]). We use it mostly to test limit situations of performance, such as forcing the occurrence of deadlocks in Shared Completed Tables, and forcing the creation of an artificially high number of tables. We also expected Concurrent Completion to yield some parallelism in evaluating tables with right recursion, as we are computing a large set of interdependent tables.

The programs can write all the reachable nodes to the output, but we disabled that, so that the benchmarks presented are all CPU bound.

7.3.1 Memory Usage

We run the Right Recursion program with a number of threads and measured the peak memory usage, as obtained by the malloc function. We used the `memusage` program, from the Linux package `glibc tools` to get these values. We only count the memory allocated by malloc, and don't take into account the threads "C" stacks and static code.

Table 7.25 shows the memory usage for the Right Recursion program using Local scheduling. We see that the graph that is more demanding in computational power, the 2048x2 graph, got up to 1GB of malloc'ed memory. In fact that was the reason we stopped measuring at 16 threads[7]. The fact that we are using a 32 bit system, in which

[7]The machine has 2GB of memory, and indeed it was possible to run the benchmark with 32 threads,

N. Threads		1	2	4	8	16
Tables	Graph					
Shared	256x128	11M	13M	16M	24M	38M
Shared	512x8	12M	14M	18M	25M	40M
Shared	2048x2	90M	92M	91M	98M	112M
Shared	8192x1	26M	28M	28M	35M	47M
Private	256x128	11M	17M	29M	49M	92M
Private	512x8	12M	21M	39M	51M	128M
Private	2048x2	90M	166M	197M	343M	981M
Private	8192x1	26M	38M	56M	80M	64M

Table 7.25: Peak memory usage for Right Recursion benchmarks with Local scheduling under the Shared Completed Tables engine in Linux

N. Threads		1	2	4	8	16
Tables	Graph					
Shared	256x128	9M	10M	14M	21M	36M
Shared	512x8	12M	14M	18M	25M	40M
Shared	2048x2	96M	97M	97M	102M	115M
Shared	8192x1	32M	34M	34M	41M	54M
Private	256x128	9M	12M	20M	35M	64M
Private	512x8	12M	21M	26M	63M	112M
Private	2048x2	90M	165M	181M	478M	1058M
Private	8192x1	32M	38M	39M	51M	103M

Table 7.26: Peak memory usage for the Right Recursion program with Batched scheduling under the Concurrent Completion engine in Linux

Linux can only use 3GB of virtual memory, shows how close we are to the system's limitations.

Table 7.26 shows a similar behavior for batched scheduling, although here there is even more demand on memory, as the batched evaluation method tends to keep the subgoals in the stacks longer.

These values show that shared tables can be a lot better when it comes to memory usage, and this might make the difference for an application to able to run in a system or not. Although in 64 bit systems these problems might be mitigated, there will always be a limit on physical memory, and abundant use of private tables in multiple threads may easily lead to the exhaustion of memory.

7.3.2 Occurrence of Deadlocks

We counted the deadlocks occurring for Shared Completed Tables. It turns out that deadlocks can only occur with the Right Recursion program, as with Left Recursion

yielding about 2.2GB of malloc'ed memory, but the virtual memory system trashed and we opted not to include those values in the tables.

N. Threads		2	4	8	16	32	64	128	256	512
Graph	Cores									
256x128	1	0	0	0	0	0	0	0	0	0
512x8	1	0	0	0	0	0	0	0	0	0
2048x2	1	0	0	0	0	0	0	4	0	0
8192x1	1	0	1	0	2	2	2	0	0	2
256x128	2	0	0	0	0	1	9	5	0	0
512x8	2	0	0	2	1	0	22	1	1	1
2048x2	2	0	0	2	4	1	22	5	2	19
8192x1	2	1	2	1	2	0	1	4	0	0

Table 7.27: Number of deadlocks for the Right Recursion program with Local scheduling under the Shared Completed Tables engine in both Linux Systems

the computation of each transitive closure subgoal doesn't depend on any other tabled subgoal. We used both the Dual Core machine and the single core machine running Linux for this measurements.

Table 7.27 shows the number of deadlocks that occurred. It confirms our hypothesis from Chapter 4, that deadlocks will rarely happen in single processor systems. In systems with more processors/cores they will happen more, as it is shown by the table. It would be interesting to count the deadlocks on a system with more processors, but we didn't have such an opportunity.

7.3.3 Performance in a Dual Core System

In this section we measure the timings of the random graph tests on the Dual Core system and compare them with the results for private tables. This enables us to see if the system is amortizing tabled calls by re-using shared tables and also to observe the speedup that can be gained by having a dual core system.

Lock usage First we count the locks used by the random graph tests when running one thread. In the tables that follow we added both the locks used by both left and right recursion to save space.

Table 7.28 shows the lock usage for Local scheduling. The locks used were introduced in Section 7.1. We can see that private tables use relatively few locks, mainly the dynamic lock to access the graph data. The shared tables make a very intensive use of the structure manager lock. The completion and call trie locks are also moderately used.

Table 7.29 shows the lock usage for Concurrent Completion over Batched scheduling. These values are much like the ones in the previous table, although now the consumer list lock is also moderately used by shared tables.

Lock	dynamic	call trie	symbol	string	compl	struct manager
Benchmark						
256x128 Private	1566	0	765	155	257	0
256x128 Shared	1566	33536	765	155	33281	265732
512x8 Private	3102	0	1277	155	513	0
512x8 Shared	3102	5632	1277	155	5121	1055748
2048x2 Private	12318	0	4349	155	2424	0
2048x2 Shared	12320	10240	4349	155	8568	13734876
8192x1 Private	49184	0	16637	155	16297	0
8192x1 Shared	49182	32768	16637	155	32681	2519862

Table 7.28: Lock usage for random graph benchmarks using Local scheduling under the Shared Completed Tables engine

Lock	dynamic	call trie	symbol	string	compl	struct manager	cons list
Benchmark							
256x128 Private	1566	0	765	155	515	0	0
256x128 Shared	1566	33536	765	155	515	265220	32769
512x8 Private	3102	0	1277	155	1028	0	0
512x8 Shared	3102	5632	1277	155	1028	1054724	4097
2048x2 Private	12318	0	4349	155	4476	0	0
2048x2 Shared	12318	10240	4349	155	4476	13730780	3722
8192x1 Private	49182	0	16637	155	24494	0	0
8192x1 Shared	49182	32768	16637	155	24494	2503478	8197

Table 7.29: Lock usage for the Random Graph benchmarks using Batched scheduling under the Concurrent Completion engine

N. Threads		1	2	Speed Up	4	8	16
Tables	Graph						
Private	256x128	1.44	1.20	1.20	0.93	1.12	0.84
Private	512x8	0.53	0.39	1.35	0.37	0.26	0.26
Private	2048x2	2.88	1.70	1.69	1.69	1.61	1.69
Private	8192x1	0.56	0.31	1.80	0.32	0.29	0.31
Shared	256x128	1.90	1.11	1.71	0.84	1.09	0.93
Shared	512x8	0.50	0.42	1.19	0.41	0.32	0.33
Shared	2048x2	3.44	2.69	1.27	2.31	2.34	2.35
Shared	8192x1	0.61	0.45	1.35	0.45	0.47	0.47

Table 7.30: Times for the Left Recursion program on the Dual Core system using Local scheduling under the Shared Completed Tables engine

N. Threads		1	2	Speed Up	4	8	16
Tables	Graph						
Private	256x128	1.43	0.98	1.45	0.87	0.87	0.88
Private	512x8	0.44	0.29	1.51	0.28	0.25	0.26
Private	2048x2	2.76	1.63	1.69	1.54	1.59	1.58
Private	8192x1	0.50	0.28	1.78	0.29	0.33	0.44
Shared	256x128	1.84	0.84	2.19	0.84	0.84	0.86
Shared	512x8	0.48	0.32	1.50	0.31	0.33	0.32
Shared	2048x2	3.32	2.68	1.23	2.67	2.38	2.71
Shared	8192x1	0.58	0.44	1.31	0.46	0.47	0.51

Table 7.31: Times for the Left Recursion program on the Dual Core system using Batched scheduling under the Concurrent Completion engine

Parallelism We run the Left Recursion program on the Dual Core system. The results are shown in Tables 7.30 and 7.31.

Table 7.30 shows the speedups for Shared Completed Tables under Local scheduling. Although they are not ideal, the times for the private tables are in coherent with the results for the transitive closure benchmarks in Section 7.2. The speedups for shared tables are, in average, not worse than the ones for private tables. We can see the fact that private tables show better speedups for less dense graphs, while shared tables show the inverse behavior.

We believe, as far as shared tables are concerned, that it has to do with the more intensive use of locks in the bigger graphs (see Table 7.28), which causes more contention problems. As far as the private tables are concerned, we believe, that the denser graphs, with relatively more answers for each subgoal, will have the answer trie hash table expanded more often, causing problems with malloc.

We can see that, for more than two threads, the times remain approximately constant, as expected for this application, running on a dual core machine.

Table 7.31 shows the speedups for Concurrent Completion over Batched scheduling

N. Threads		1	2	4	8	16
Tables	Graph					
Private	256x128	1.35	2.02	3.70	6.94	13.94
Private	512x8	0.51	0.47	1.16	2.37	4.52
Private	2048x2	1.61	1.86	4.18	7.92	18.66
Private	8192x1	0.26	0.23	0.33	0.48	0.70
Shared	256x128	1.38	1.39	1.85	1.40	1.42
Shared	512x8	0.56	0.46	0.55	0.43	0.42
Shared	2048x2	1.98	1.87	2.03	1.89	1.85
Shared	8192x1	0.33	0.32	0.31	0.31	0.32

Table 7.32: Times for the Right Recursion program on the Dual Core system using Local scheduling under the Shared Completed Tables engine

in the Dual Core system. These numbers show approximately the same properties as the ones in the previous table. The speedups for shared tables seem a little better (there's even a super linear speedup for the 256x128 graph), but we don't attribute this to any specific property of Batched scheduling, as the properties of the application should be about the same for both cases.

Amortization We measured the times to compute the Right Recursion program on the Dual Core system. Again, we repeat that right recursion is not an efficient way to compute transitive closure with tabling, but in this case it shows a large number of tables being re-used when possible.

In Table 7.32 we show the times for computing the Right Recursion program with Shared Completed Tables over Local scheduling. It shows that the shared tables are being effectively re-used by all the threads, as the computation time doesn't increase with the number of threads. It may even happen that deadlocks are occurring (cf. Section 7.3.2) but the time to compute more threads doesn't increase, as it always implies computing the same tables, whatever number of threads are being run. On the other hand, with private tables, we see that each thread has to compute a new tabled subgoal for every reachable node from the nodes in its partition, degrading performance when the number of threads increases. This is a little less evident for the sparser graphs, but completely evident for the denser ones.

In some cases we can see some speedups for two threads; while Shared Completed Tables weren't designed for parallelism, they may yield some parallelism, while computing a set of interdependent tables, depending on the SDG's topology.

In Table 7.33 we show the times for the Right Recursion program for Concurrent Completion over Batched scheduling. The results are essentially the same as the ones for the previous table, where amortization is concerned, as the characteristics of the application are not dependent on scheduling.

Where table parallelism is concerned the results are slightly disappointing, there

N. Threads		1	2	4	8	16
Tables	Graph					
Private	256x128	1.57	1.92	4.56	7.65	15.80
Private	512x8	0.72	0.75	1.91	3.40	6.84
Private	2048x2	2.13	3.09	5.10	12.55	27.61
Private	8192x1	0.25	0.24	0.32	0.47	0.88
Shared	256x128	1.57	1.57	1.71	1.01	1.51
Shared	512x8	0.72	0.58	0.74	0.59	0.60
Shared	2048x2	2.36	2.39	2.24	2.18	2.19
Shared	8192x1	0.31	0.31	0.31	0.32	0.40

Table 7.33: Times for the Right Recursion program on the Dual Core system using Batched scheduling under the Concurrent Completion engine

N. threads		1	2	Speedup	4	Speedup	8	Speedup
Tables	Graph							
Private	256x128	5.29	4.94	1.07	4.45	1.18	4.38	1.20
Private	512x8	1.76	1.36	1.29	1.16	1.51	1.12	1.57
Private	2048x2	11.02	7.30	1.50	4.84	2.27	3.98	2.76
Private	8192x1	2.36	1.52	1.55	0.90	2.62	0.61	3.86
Shared	256x128	5.24	4.88	1.07	4.46	1.17	4.35	1.20
Shared	512x8	1.79	1.68	1.06	1.27	1.40	1.28	1.39
Shared	2048x2	11.29	12.23	0.92	9.61	1.17	22.88	0.49
Shared	8192x1	2.36	2.73	0.86	2.26	1.04	5.45	0.43

Table 7.34: Times for the Left Recursion program on the Multi-processor system using Local scheduling under the Shared Completed Tables engine

being only one case of a very small speedup. We do understand that right recursion imposes a big strain on the system, and that this may not be the ideal test case to test parallelism with concurrent completion.

7.3.4 Scalability of Shared Tables on a Multi-Processor

We measured the scalability of the shared tables in a true multi-processor (as opposed to a multi-core system) and present the results in this section.

In Table 7.34 we see the speedups for the Left Recursion program, using shared and private tables using Shared Completed Tables on the Solaris Multi-processor. While the times for private tables are in line with the ones reported on Section 7.2, the ones for shared tables are just terrible. With 8 processors the performance, instead of improving, degrades down to 43%. We deem these to the intensive use of the shared structure managers, including structure manager lock. In the case of a true multi-processor, frequent write accesses of the same memory block, such as locking and unlocking a mutex, by different threads, will cause the cache entries for that memory block to be

N. threads		1	2	Speedup	4	Speedup	8	Speedup
Tables	Graph							
Private	256x128	3.49	3.72	0.93	3.51	0.99	3.51	0.99
Private	512x8	1.21	0.93	1.30	0.89	1.35	0.90	1.34
Private	2048x2	8.13	5.83	1.39	4.17	1.94	3.71	2.19
Private	8192x1	1.14	0.87	1.31	0.51	2.23	0.42	2.71
Shared	256x128	3.63	3.43	1.05	3.53	1.02	3.55	1.02
Shared	512x8	1.26	1.18	1.06	1.03	1.22	1.66	0.76
Shared	2048x2	8.75	10.25	0.85	11.44	0.76	25.71	0.34
Shared	8192x1	1.23	1.56	0.78	1.89	0.65	3.82	0.32

Table 7.35: Times for the Left Recursion program on the Multi-processor system using Batched scheduling under the Concurrent Completion engine

invalidated in the remaining processors, leading to a big degradation in memory access performance. As such, the fine-grained and heavily used nature of the shared structure managers, including its locks, and perhaps also some other variables, causes a memory contention problem. In traditional multi-processor systems, where the cache blocks that contain the same data for other processors must be invalidated with each write access this causes the system performance to degrade, and it will only degrade more with the addition of more processors to the application. The good results shown for the Dual Core system, in Section 7.3.3, although only for 2 cores, just show that different hardware technology can have a major impact in a parallel program's performance.

In Table 7.35 we see the times for the Left Recursion program using shared and private tables under Concurrent Completion and Batched scheduling. If anything, the slowdowns for shared tables are even worse than the ones from the previous table.

In this Section we have shown that for some cases, where re-using tabled computations is possible, shared tables just perform much better whether in memory usage or speed, than private tables. For more than one processor, the shared tables seem to perform OK on multi-core systems, but have to be seriously re-worked for traditional multi-processors, due to cache problems, caused at least by the fine grained use of the shared structure managers.

Chapter Summary In this chapter we presented performance results for the multi-threaded XSB engine. The overheads of the system are very variable, being negative in some systems, while being considerable on others, specially for shared tables. In a multi-processor the private tables have scaled well up to eight processors. We show that the shared tables may allow big savings of memory over private tables and that the deadlock situation of Shared Completed Tables occurs rarely. Shared tables also allow time gains by the re-use of previous computations. However they scale badly for parallel execution in a traditional multi-processor, but seem to be doing OK in multi-

core systems.

8

Conclusions

Working in the ever changing environment of the XSB system has not always been easy. The demand to maintain an usable system and the interactions from conflicting implementation lines are challenging issues. However we are rewarded by the synergy of several different research directions, contributing to a powerful and efficient system that can support interesting real world applications and which has a large community of users.

8.1 Contributions of this Work

We believe we have achieved contributions to both the XSB user community and the knowledge on the implementation of tabling with the work that is described in this dissertation.

A Multi-threaded Tabling Engine The Multi-threaded Tabling Engine is the basic foundation of this dissertation. The multi-threading of a real Prolog system, while not an original achievement, is still a demanding effort. We support native threads, which makes the concurrency issues harder to handle but gives the power of native thread support. A large part of the effort of the dissertation was spent in testing it and catching obscure global data access conflicts. The multi-threaded system now passes the sequential test suite of 461 test files (most of which contain multiple tests) as well as a multi-threaded test suite of 232 files[1]. Performance experiments show around 20% of worst case performance for one thread vs the sequential engine. However most of

[1]Both these test suites are publicly available in the XSB repository under the modules `tests` and `mttests`.

the times the overheads are much lower, while for some systems and compilers it is significantly faster than the sequential engine.

We highlight as positive features of the Multi-threaded engine:

- the engine is built over POSIX threads and supports their semantics at the Prolog level.
- explicit parallelism, where the programmer can specify parallel actions
- support for thread private tables, including well-founded negation
- a real system for real applications
- comparable times to the sequential engine
- private tables scale well on a multi-processor.

Shared Completed Tables The implementation of Shared Completed Tables has achieved a mature status. In fact Shared Completed Tables are very easy to implement and provide a way for multiple threads to reuse each others tabled results without re-computation. The fact that it doesn't allow the parallel computing of interdependent goals doesn't prevent it from being useful in the re-utilization of computations and achieving good throughput in single processor systems. We devolved SLG_{SCT}, a formal semantics that models the concurrent evaluation of shared tables, providing the USURPATION operation to simulate the breaking of deadlocks. We proved the correctness of SLG_{SCT} w.r.t. SLG. We showed informally that the implementation is correct w.r.t. SLG_{SCT}. We believe the deadlock events that force the re-computation of tables are difficult to achieve in real execution of programs, specially in a system with a single or few processors (c.f. Table 7.27), and in the worst case its complexity doesn't change that of the sequential SLG-WAM for programs with negation.

We highlight as positive points of our implementation of Shared Completed Tables:

- an effective and simple way to implement shared tables
- supported by a formal semantics, including the USURPATION operation
- a method to restart a SCC spanning multiple threads within Local Scheduling
- support for well-founded negation with shared tables
- worst case complexity happens rarely in benchmarks but is still acceptable
- although it wasn't designed for parallelism, it can yield some, depending on the topology of the SDG.

Concurrent Completion Although implementation of the Concurrent Completion algorithm is not completely mature it provides a way for several threads to compute inter-dependent tabled subgoals in parallel. The Concurrent Completion doesn't alter the properties of execution of the sequential SLG-WAM, and the performance results indicate that it doesn't incur in significant performance overheads over private tables. We proved the completeness and liveness of the design, providing a solid case for the correctness of the implementation.

We highlight as positive features of Concurrent Completion:

- a generalization of the SLG-WAM completion mechanism for definite programs, including the completion stacks
- proved completeness and liveness
- basic support for parallelism has been built into the system.

With these three dimensions, a fully multi-threaded system that can exploit the benefits of both explicit parallelism and shared tables has been built, and we expect it to be useful to the research community.

8.2 Future Work

While we believe to have built a stable system for users, some of the results reported on this dissertation are still open to short term improvements. On the other hand, new research directions are opened by providing a sound platform for further experimentation in the parallel and distributed field.

ISO Prolog threads compatibility An effort is being done to publish a ISO standard for multi-thread Prolog [52]. At the time of the writing, XSB already supports much of the proposed standard. Some features are still to be supported, for example the signal mechanism. We expect to support it, as well as most of the standard, very soon.

Parallel Execution on Multi-Processors As was seen in Chapter 7, while parallel computation of shared tables in a dual core system has been achieved successfully, it remains to provide efficient parallel execution on traditional multi-processors. To this end the fine grained locking of the structure managers has to be refined, to avoid the heavy memory contention that the system is actually exhibiting.

Concurrent Completion Concurrent Completion can be somewhat more polished, and support for Local Scheduling and Negation should be added in the near future. The XSB early completion feature that kept us from supporting negation shouldn't prove a long term obstacle. The points made in Section 5.4 about waking up consumer

threads, and the consumer choice point lists, should be addressed. Testing concurrent completion to evaluate in parallel a set of tables is another pressing issue.

However one very though challenge remains – to allow threads to stop in the middle of a tabled computation while allowing the other threads on the SCC to complete, under batched scheduling. The only way to do this seems to be, in some way, to migrate or restart the tables that are incomplete to the remaining threads. Such progress would very much raise the interest of using Batch Scheduling for multi-threaded computations. It is possible that a modification of OPTYap's mechanism of saving and restoring SCCs would support this in an efficient manner.

Other Shared Table Paradigms We did not in any way exhaust the models which can be used to compute shared tables in a multi-threading environment. At this stage two models are readily apparent:

Exporting Private Tables Under this model the tables would be computed privately, and made known to others when completed.

Sharing the Table Space Under this model each thread would have its own private stacks, but the global table space would be shared. There would be some complications, as there are pointers from the Table Space into the stacks (the consumer choice point list in the subgoal frame comes to mind) but it might be doable.

While these two approaches would not be best suited to parallelism (as a subgoal table could be being computed by a number of threads in a replicated way before it was completed) they would solve the problem of incomplete tables: when a thread didn't complete the table, the others would just not know about it.

Abolishing Shared Tables and Retracting Dynamic Predicates Currently, we can reclaim space for the shared data structures corresponding to retracted shared dynamic predicates and abolished shared tables, only when there is a single active thread. Multi-threaded garbage collection of retracted/abolished shared data is a topic to explore in the future.

Table Parallelism As we mentioned before, Concurrent Completion can be used as a starting point for the implementation of table parallelism as defined in [33]. Further research is needed to achieve an implementation of Table Parallelism. It can be argued that the current Concurrent Completion algorithm its not parallel enough and that an adapted version of the algorithm described in [33] would be better, although more complex.

Distributed Tabling and Applications As was seen Section 6.3 a multi-threaded engine is a step forward into the development of a distributed tabling system that makes effective use of the available resources. None of the systems mentioned in that section has been used to support real applications, something which is still to be achieved. Further research using the current system to support Distributed Tabling is desirable. The multi-threaded system also makes it easier to build distributed tabling applications, where internal concurrency is essential for each engine, in a distributed multi-engine environment.

Bibliography

[1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.

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

[3] M. Alves. Arquitectura de tabulação distribuida com detecção de terminação. Master's thesis, Universidade Nova de Lisboa, 2004. In portuguese.

[4] F. Barbosa and J. Cunha. A coordination language for collective agent based systems: GroupLog. In *Proceedings of SAC'00*, Como, Italy, March 2000. ACM.

[5] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006.

[6] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[7] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. *The Ciao Prolog system, Reference Manual*. Available from http://www.clip.dia.fi.upm.es/.

[8] M. Calejo. InterProlog: a declarative Java-Prolog interface. In *Procs. Logic Programming for Artificial Intelligence and Information Systems*, Porto, Portugal, December 2001.

[9] N. Carriero and D. Gelernter. Linda in context. *Communications of ACM*, 32(4):444–458, April 1989.

[10] M. Carro and M. Hermenegildo. Concurrency in Prolog using threads and a shared database. In Dany De Schreye, editor, *Proceedings of the 19th International Conference on Logic Programming, ICLP'1999*, pages 320–334, Las Cruces, NM, USA, September 1999. MIT Press.

[11] L.F. Castro, T. Swift, and D.S. Warren. Suspending and resuming computations in engines for SLG evaluation. In *Practical Applications of Declarative Languages*, volume 2257 of *LNCS*. Springer-Verlag, 2002.

[12] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *J. Logic Programming*, 15(3):187–230, 1993.

[13] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[14] D. Chu and K. L. Clark. IC-Prolog II: A multi-threaded Prolog system. In *ICLP-Workshop on Concurrent, Distributed & Parallel Implementations of Logic Programming Systems*, pages 115–141, Budapest, Hungary, 1993.

[15] K. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Trans. on Programming Languages and Systems*, 8:1–49, 1986.

[16] Keith L. Clark, Peter J. Robinson, and Richard Hagen. Multi-threading and message communication in Qu-Prolog. *TPLP*, 1(3):283–301, 2001.

[17] Jonathan J. Cook. P#: a concurrent Prolog for the .NET framework. *Software, Practice and Experience*, 34(9):815–845, 2004.

[18] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: a parallel Prolog system that transparently explots both and- and or- parallelism. *SIGPLAN Notices*, 26(7):83–93, April 1991.

[19] Vítor Santos Costa, Luís Damas, Rogério Reis, and Rúben Azevedo. *YAP Prolog User's Manual*, 2006. Available online from `http://www.ncc.up.pt/~vsc/Yap`.

[20] B. Cui, T. Swift, and D. S. Warren. From tabling to transformation: Implementing non-ground residual programs. In *International Workshop on Implementations of Declarative Languages*, 1999.

[21] J. C. Cunha, M. B. Carvalhosa, P. Medeiros, and L. M. Pereira. Delta Prolog: A distributed logic programming language and it's implementation. In P. Kacksuk and Michael Wise, editors, *Implementations of Distributed Prolog*, chapter 1. John Wiley & Sons, 1992.

[22] J. C. Cunha and R. Marques. Distributed algorithm development with PVM-Prolog. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 211–215. IEEE Computer Society Press, 1997.

[23] C. Damásio. A distributed tabling system. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000*, pages 65–75, Vigo, Spain, September 2000.

[24] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analisys of workflows. In *ACM Principles on Database Systems*, pages 25–33, 1998.

[25] B. Demoen and K. Sagonas. CAT: A copying approach to tabling. In *PLILP*, pages 21–35, 1998.

[26] B. Demoen and K. Sagonas. Chat is θ(slg-wam). In *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, pages 337–357, 1999.

[27] B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. In *Practical Aspects of Declarative Languages: First International Workshop*, 1999.

[28] K. Devlin. *Fundamentals of Contemporary Set Theory*. Springer-Verlag, Berlin Germany, 1980.

[29] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computations. In *Information Processing Letters*, pages 217–219, 1983.

[30] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. In *Information Processing Letters*, 1980. 11-1:1-4.

[31] J. Eskilson and M. Carlson. *SICStus MT – A Multithreaded Execution Environment for SICStus Prolog*, pages 36–53. LNCS. Springer-Verlag, 1998.

[32] S. Ferenczi and I. Futo. CS-Prolog: A communicating sequential Prolog. In P. Kacsuk and M. Wise, editors, *Implementations of Distributed Prolog*, pages 357–378. John Wiley & Sons, New York, 1992.

[33] J. Freire, R. Hu, T. Swift, and D. S. Warren. Parallelizing tabled evaluation. In *7th International PLILP Symposium*, pages 115–132. Springer-Verlag, 1995.

[34] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998(3), 1998.

[35] H. Guo and G. Gupta. A simple scheme for implementing tabling based on dynamic reordering of alternatives. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000*, pages 141–154, Vigo, Spain, September 2000.

[36] H. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternates. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2001.

[37] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: a survey. *Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[38] M. Hermenegildo and K. Greene. The &-Prolog system: Exploiting independent and parallelism. *New Generation Computing*, 9(3,4):233–257, 991.

[39] R. Hu. *Distributed Tabled Evaluation*. PhD thesis, SUNY at Stony Brook, 1997.

[40] IEEE. *Portable Operating System Interface (POSIX) — Part 1: System Application: Program Interface*, 1995. Part 1003.1c - Threads Extension [C Language].

[41] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, July 1995.

[42] B. Lewis and D. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, 1998.

[43] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin Germany, 1987.

[44] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2-3):243–271, 1990.

[45] R. Marques. Um modelo de programação paralela e distribuída para o prolog. Master's thesis, Faculdade de Ciêcias e Tecnologia, Universidade Nova de Lisboa, 1996. In portuguese.

[46] R. Marques and J. Cunha. PVM-Prolog: Parallel logic programming in the PVM system. In *Proceedings of the PVM Users' group meeting, Pittsburgh, Pa*, 1995.

[47] R. Marques, J. Cunha, and T. Swift. An architecture for a multi-threaded tabling engine. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000*, pages 51–63, Vigo, Spain, September 2000.

[48] Friedemann Mattern. Global Quiescence Detection Based on Credit Distribution and Recovery. In *Information Processing Letters 30*, pages 195–200, 1989.

[49] Microsoft Inc. *The Microsoft developer .NET home page*. Available online at `http://msdn.microsoft.com/net`.

[50] L. Monteiro. A proposal for distributed programming in logic. Technical report, Universidade Nova de Lisboa, Departamento de Informática, 1986.

[51] P. Moura. *LogTalk: Design of an Object-Oriented Logic Programming Language*. PhD thesis, Universidade da Beira Interior, 2003.

[52] P. Moura. *Prolog multi-threading predicates*. ISO/IEC, 5 2007. DTR 13211.

[53] Paulo Moura. *LogTalk User Manual*. Available online from `http://logtalk.org`.

[54] L. R. Pokorny and C. R. Ramakrishnan. Modeling and verification of distributed autonomous agents usign logic programmming. In *Second International Workshop on Declarative Agent Languages and Techologies*, pages 148–165, 2004.

[55] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings on the Conference on Automated Verification*, pages 143–154, 1997.

[56] C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka Y. Dong, X. Du, A Roychoudbury, and V. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Proceedings on the Conference on Automated Verification*, pages 576–580, 2000.

[57] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–55, January 1999.

[58] R. Rocha. *On applying Or-parallelism and tabling to logic programs*. PhD thesis, Universidade do Porto, 2001.

[59] R. Rocha, F. Silva, and V. S. Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of the 2nd Workshop on Tabulation in Parsing and Deduction, TAPD'2000*, pages 77–87, Vigo, Spain, September 2000.

[60] R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In P. Barahona and J. Alferes, editors, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence, EPIA'1999*, number 1695 in LNAI, pages 178–192, Évora, Portugal, September 1999. Springer-Verlag.

[61] R. Rocha, F. Silva, and V. Santos Costa. Achieving Scalability in Parallel Tabled Logic Programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS'2002*, Fort Lauderdale, Florida, USA, April 2002. IEEE Computer Society.

[62] R. Rocha, F. Silva, and V. Santos Costa. Concurrent Table Accesses in Parallel Tabled Logic Programs. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference, Euro-Par 2004*, number 3149 in LNCS, pages 662–670, Pisa, Italy, August/September 2004. Springer-Verlag.

[63] R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Journal of Theory and Practice of Logic Programming*, 5(1 & 2):161–205, 2005.

[64] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS*, 20(3):586 – 635, May 1998.

[65] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for efficiently computing queries to well-founded models. *Journal of Logic Programming*, 45(1-3):1–41, 2000.

[66] K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 254(1-2):465–499, 2000.

[67] M. Schroeder, R. Marques, G. Wagner, and J. Cunha. CAP - concurrent action and planning: Using PVM-Prolog to implement vivid agents. In *Proceedings of PAP'97, 5th International Conference and Exhibition on the Practical Application of Prolog*, London, UK, April 1997.

[68] E. Shapiro. A subset of concurrent Prolog and its interpreter. In E. Shapiro, editor, *Concurrent Prolog Collected Papers*. MIT Press, 1987.

[69] C. Silva. On Applying Program Transformation to Implement Tabled Evaluation in Prolog. MSc Thesis, University of Porto, Portugal, March 2007.

[70] Zoltan Somogyi and Konstantinos F. Sagonas. Tabling in mercury: Design and implementation. In *PADL*, pages 150–167, 2006.

[71] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[72] V. Sunderam, A. Geist, J. Dongarra, and R. Manchek. Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 1994.

[73] T. Swift. *Efficient Evaluation of Normal Logic Programs*. PhD thesis, SUNY at Stony Brook, 1994.

[74] T. Swift. A new formulation of tabled resolution with delay. In *Recent Advances in Artifiial Intelligence*, volume 1695 of *LNAI*, pages 163–177. Springer-Verlag, 1999. Available at http://www.cs.sunysb.edu/~tswift.

[75] T. Swift. A note on trailing in the SLG-WAM. Available from `http://www.cs.sunysb.edu/~tswift`, 2001.

[76] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the Symposium on Logic Programming*, pages 219–238, 1994.

[77] T. Swift and D. S. Warren. Coherent description framework. Available via `xsb.sourceforge.net`, 2005.

[78] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986.

[79] P. Tarau and V. Dahl. Mobile threads through first order continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruña, Spain, 1998.

[80] Paul Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.

[81] *The XSB Programmer's Manual: version 3.0, vols. 1 and 2*, 2006. `http://xsb.sourceforge.net`.

[82] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986.

[83] A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[84] P. van Roy. 1983-1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[85] Jan Wielemaker. *SWI-Prolog Reference Manual*. Available online from `http://www.swi-prolog.org`.

[86] Jan Wielemaker. Native preemptive threads in SWI-Prolog. In Catuscia Palamidessi, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, December 2003. Springer Verlag. LNCS 2916.

[87] Michael J. Wise. Experience with PMS-Prolog: A distributed, coarse-grain-parallel Prolog with processes, modules and streams. *Software: Practice and Experience*, 23(2):151–175, 1993.

[88] N. F. Zhou, Y. D. Shen, L. Y. Yuan, and J. H. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.

[89] N.F. Zhou, Y.D Shen, and T. Sato. Semi-naive evaluation in linear tabling. In *ACM-SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 90–97, 2004.