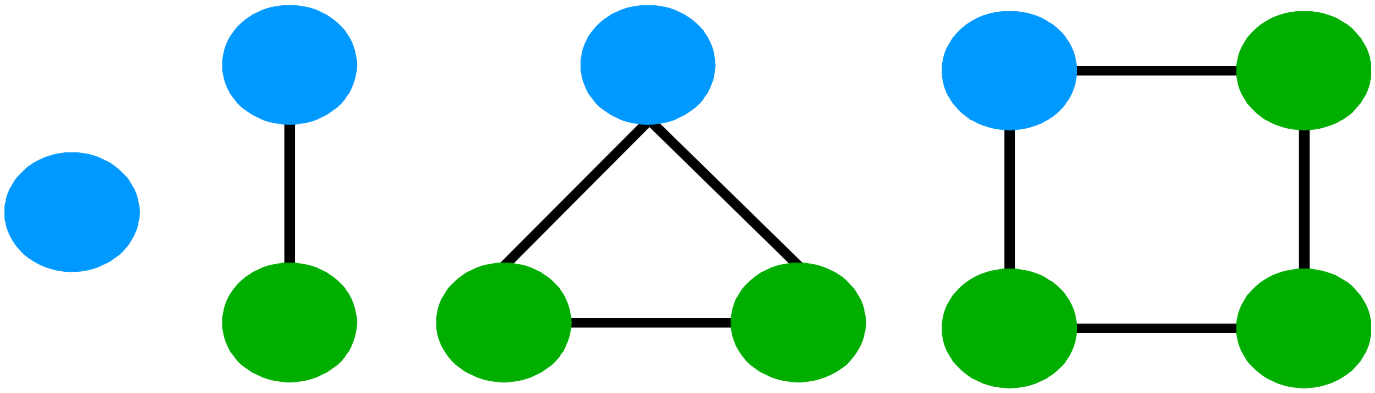

Multithreaded Tabling for Logic Programming

Miguel Areias
Programa Doutoral em Ciência de Computadores
Departamento de Ciência de Computadores
2015

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

Dedicated to

Rita, Bia, Margarida and João

Acknowledgments

I am deeply indebted to my advisor Prof. Ricardo Rocha for his fundamental role in my doctoral work. Prof. Ricardo provided me with every bit of guidance, assistance, and expertise that I needed during my first years; then, when I felt ready to venture into research on my own and branch out into new research areas, Prof. Ricardo gave me the *(lock-)*freedom to do whatever I wanted, at the same time continuing to contribute valuable feedback, advice, and encouragement. In addition to our academic collaboration, I valued deeply the close personal rapport and friendship that we have beget over the years. I simply cannot imagine a better adviser.

Besides my advisor, I would like to express my deepest gratitude to the following persons. A really special acknowledgment to Prof. Vítor Costa, for his crucial contribution to this work. His complete availability towards this research since the first minute and his immense knowledge about the topics involved were simply out of this world. I would like to express another special acknowledgment to Prof. Fernando Silva for his huge support and extremely keen comments. Finally, to Prof. Inês Dutra and Prof. Luís Lopes, whose work in parallel and distributed computing is really inspiring.

I am thankful to the Fundação para a Ciência e Tecnologia (FCT) for the research grant SFRH/BD/69673/2010, which supported this work for 48 months, and to the Center for Research and Advanced Computing Systems (CRACS) for the travel funding through projects HORUS (PTDC/EIA-EIA/100897/2008), LEAP (PTDC/EIA-CCO/112158/2009) and SIBILA (NORTE-07-0124-FEDER-000059).

To all my fellow colleagues from DCC-FCUP, specially João Santos, Flávio Cruz, and Joana Côrte-Real, I wish you much success for your professional and personal future.

Finally, I would like to thank my family and friends for all the support during the last years, specially to Rita, Bia and my parents, Margarida and João, that believed, trusted and supported me even more than I could ever imagined. I do not have words to express my deepest gratitude. I am also specially grateful to my grandparents, Margarida and Fernando, whose way to life is really inspiring, and to the Pires and Martins families. A piece of each one of you is present in this work. Thank you all.

Miguel Areias

Resumo

As linguagens de programação em lógica, como o Prolog, são baseadas em clausulas de Horn e fornecem um mecanismo de inferência bem reconhecido. Apesar do Prolog ser uma linguagem popular e bem sucedida, o seu potencial é limitado pelo seu método de resolução que é baseado na resolução SLD.

A tabulação é uma técnica poderosa e reconhecida que melhora a declaratividade e expressividade, dos sistemas tradicionais de Prolog, em programas com recursão e computações redundantes. Muito resumidamente, a tabulação consiste em armazenar respostas intermédias para subgolos de forma a que estas as respostas possam ser reutilizadas quando um subgolo similar aparece. A técnica de tabulação pode, assim, ser vista como uma ferramenta natural para a resolução de problemas de programação dinâmica, onde uma estratégia recursiva geral divide um problema em sub-problemas mais simples que, muitas vezes, são os mesmos.

Multithreading é uma técnica que permite aos computadores a execução de um programa usando em simultâneo de vários caminhos de execução num único processo, não necessitando portanto de ter uma cópia completa do programa em cada caminho de execução. Quando a tabulação é combinada com multithreading, temos o melhor dos dois mundos, uma vez que podemos explorar a combinação de uma semântica mais declarativa com um maior controle processual. No entanto, apesar da disponibilidade de ambos a tabulação e multithreading em alguns sistemas Prolog, a implementação dessas duas técnicas em conjunto implica laços complexos ao nível do mecanismo de suporte subjacente inerente ao sistema Prolog.

Nesta tese, propomos uma nova abordagem para a combinação de tabulação com multithreading, onde cada caminho de execução vê suas tabelas como privadas, mas, para ao nível do mecanismo de suporte subjacente ao Prolog, teremos um espaço de tabela comum onde as tabelas são compartilhadas entre todos os caminhos de execução. Nós apresentaremos três arquiteturas para a nossa abordagem de espaço de tabelas

comum: No Sharing (NS), Subgoal Sharing (SS) e Full Sharing (FS), e mostramos como explorar suas vantagens. Além disso, apresentaremos um novo alocador de memória e dois tipos de estruturas de dados lock-free que são destinadas especialmente para ambientes com as características do nosso ambiente de trabalho. Os resultados obtidos nesta tese são muito promissores e abrem várias direções de pesquisa para trabalhos futuros.

Abstract

Logic programming languages, such as Prolog, are derived from Horn Clause Logic and provide a well understood resolution based inference mechanism. Although Prolog is a popular and successful language, its potential is limited by the SLD resolution method on which it is based.

Tabling is a recognized and powerful technique that improves the declarativeness and expressiveness of traditional Prolog systems in dealing with recursion and redundant computations. In a nutshell, tabling consists of storing intermediate answers for subgoals so that they can be reused when a similar subgoal appears. The tabling technique can thus be viewed as a natural tool to implement dynamic programming problems, where a general recursive strategy divides a problem in simple sub-problems that, often, are the same.

Multithreading is a technique that enables computers to support multiple concurrent paths of execution within a single process without the need of having an entire copy of the program. When tabling is combined with multithreading, we have the best of both worlds, since we can exploit the combination of higher declarative semantics with higher procedural control. However, despite the availability of both tabling and multithreading in some Prolog systems, the implementation of these two features implies complex ties to each other and to the underlying engine.

In this thesis, we propose a new approach for multithreaded tabling where each thread views its tables as private but, at the engine level, we will use common table space where tables are shared among all threads. We present three designs for our common table space approach: No Sharing (NS), Subgoal Sharing (SS) and Full Sharing (FS), and show how to exploit their advantages. Additionally, we introduce a novel memory allocator and two lock-free trie data structures that are specially aimed for environments with the characteristics of our framework. The results obtained with this thesis are very promising and open several research directions for future work.

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

API Application Programming Interface

AT Answer Trie

BAE Bucket Array of Entries

CAS Compare-And-Swap

CHM Concurrent Hash Map

CN Consumer Node

CSL Concurrent Skip List Map

CTries Concurrent Hash Tries

DCAS Double-Compare-And-Swap

DRA Dynamic Reordering of Alternatives

DRS Dynamic Reordering of Solutions

FS Full Sharing

GCAS Generation-Compare-And-Swap

GN Generator Node

IN Interior Node

LCS Longest Common Subsequence

LF$_1$ Lock-Free Trie

LF$_2$ Lock-Free Hash Trie

LL/SC Load-Linked and Store-Conditional

LP Linearization Point

NS No Sharing

OLD Ordered Linear Deduction

OLDT Ordered Linear Deduction with Tabling

OS Operating System

PCC Private Consumer Chaining

PDL Push-Down List

POSIX Portable Operating System Interface

SCC Strongly Connected Component

SE Subgoal Entry

SF Subgoal Frame

SLD Selected Linear Deduction

SLDT Selected Linear Deduction with Tabling

SLG Selected Linear Goal-oriented

SS Subgoal Sharing

ST Subgoal Trie

TCC Table Completion Check

TE Table Entry

TLEL Table Lock at Entry Level

TLNL Table Lock at Node Level

TLWL Table Lock at Write Level

TLWL-ABC Table Lock at Write Level - Allocate Before Check

TMU Total Memory Usage

TNA Table New Answer

TSC Table Subgoal Call

UMA User-level Memory Allocator

WAM Warren's Abstract Machine

Chapter 1

Introduction

The main goal of a programming language is to enable the communication between humans and machines in order to define problems and their general means to obtain solutions. The first programing languages were machine languages. To communicate, the programmer had to learn how to express problems in machine-oriented terms. Higher-level languages, developed from machine languages, through the provision of facilities, for the expression of problems, in terms closer to the problem's conceptualization. It is believed that higher-level languages are particularly helpful in developing succinct and correct programs that are easy to write and understand. Logic programming languages, together with functional programming languages, form a major class of languages, called *declarative languages*, and because they are based on the predicate calculus, they have a strong mathematical basis [4]. Arguably, Prolog is the most popular and powerful logic programming language. Prolog gained its popularity mostly because of the success of the sophisticated compilation technique and abstract machine known as the WAM (Warren's Abstract Machine), presented by David H. D. Warren in 1983 [133].

The operational semantics of Prolog is given by SLD resolution [65], an evaluation strategy particularly simple that matches current stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. Tabling is a recognized and powerful implementation technique that overcomes the limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion and that can considerably reduce the search space, avoid looping and have better termination properties than pure SLD resolution [27].

Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [110], the most successful engine of XSB. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Currently, the tabling technique is widely available in systems like XSB Prolog [125], Yap Prolog [118], B-Prolog [138], ALS Prolog [48], Mercury [120], Ciao Prolog [28] and more recently in Picat [140].

Multithreading is a type of execution model that allows multiple threads to coexist within the context of a process such that they execute independently but share the process resources. The increasing availability of computing systems with multiple cores sharing the main memory is already a standardized, high-performance and viable alternative to the traditional (and often expensive) shared memory architectures. The number of cores per processor is expected to continue to increase, further expanding the potential for taking advantage of multithreading support. As consequence, multithreading has become an increasingly popular way to implement dynamic, highly asynchronous, concurrent programs. Multiple examples of frameworks exist that exploit the modern multicore architectures currently available. For example, for imperative programming languages, the Cilk [21] and Intel Threading Building Blocks [102] frameworks provide runtime systems for multithreaded parallel programming, providing programmers with the means to create, synchronize, and schedule threads in an efficient fashion. For functional programming languages, the Eden [75] and HDC [59] Haskell based frameworks allow the users to express their programs using polymorphic higher-order functions. For object-oriented programming languages, the MALLBA [2] and DPSKEL [89] frameworks also showed relevant speedups in the parallel evaluation of combinatorial optimization benchmarks.

In the specific case of Prolog, when multithreading is combined with tabling, one can have the best of both worlds, since one can exploit the combination of higher procedural control with higher declarative semantics. In a multithreaded tabling system, tables may be either *private* or *shared* between threads. While thread-private tables are easier to implement, shared tables have all the associated issues of locking, synchronization and potential deadlocks. Here, the problem is even more complex because we need to ensure the correctness and completeness of the answers found and stored in the shared tables. Thus, despite the availability of both threads and tabling in Prolog compilers such as XSB, Yap, and Ciao, the implementation of these two features such that they

work together seamlessly implies complex ties to one another and to the underlying engine.

1.1 Thesis Purpose

One of the great advantages of Prolog is its potential for the implicit exploitation of parallelism. Many references to parallel Prolog systems exist in the literature [51], being the most common proposals those that exploit *Or-Parallelism* and/or *And-Parallelism*. Or-parallelism corresponds to the simultaneous execution of the body of different clauses, thus it is used when more than one clause unifies with the current call. And-Parallelism corresponds to the simultaneous execution of the subgoals contained in a clause's body, thus it is used when more than one subgoal occurs in the body of the clause.

Given the advantages of tabling evaluation, the question that arises is if a tabling mechanism has the potential for the exploitation of parallelism/concurrency. On one hand, tabling still exploits a search space as traditional Prolog, but on the other hand, the parallel/concurrent model of tabling is necessarily far more complex than the traditional models of parallelism, once it also introduces concurrency on the access to tables. Currently, only the Yap [105, 109] and the XSB [76] systems combine tabling with some form of parallelism/concurrency. Yap combines the tabling-based SLG-WAM [110] execution model with Or-Parallelism using shared memory processes to exploit the advantages of shared memory architectures. XSB also extends the tabling execution model based on the SLG-WAM to support concurrency, using threads instead of processes. In XSB, the SLG-WAM was extended with a concurrent shared tables model that ensures the correct execution of concurrent sub-computations. However, the practical results of using this model are still limited [12].

In this thesis, we follow XSB's approach and we exploit the potential of Prolog for explicit parallelism/concurrency using threads. The current version of Yap (which is our base Prolog system) includes support for multithreading, but its implementation was not compatible with the current tabling engine. This thesis aims to create a robust and efficient solution that allows the integration of both mechanisms not only for Yap, but also for systems based on similar tabling and multithreaded mechanisms. Together with the expected increase in the number of cores per processor in the next generation architectures, this thesis is a contribute for making Prolog an even more powerful programming language since, at least, the Yap system offers the advantages

of concurrent tabling.

Accordingly, our journey begun with the study and implementation of three concurrent designs to support different levels of concurrency within the table spaces. These designs represent alternative trade-offs between concurrency and memory usage. In the first design, we avoid concurrency by allowing threads to consume all memory in a private fashion. In the second design, threads share part of the table space in a concurrent fashion, while in the third design, threads fully share the table space. In order to understand the advantages and limitations of each design, next we selected and adopted a set of benchmarks that could expose the vulnerabilities of each design. These benchmarks create worst case scenarios, where an arbitrary number of threads is used to evaluate exactly the same sub-computations [8]. One of the initial main limitations found, that was common to all designs, was the memory management. Thus we studied and implemented a new state-of-the-art memory allocator that minimizes the performance degradation that the designs suffered when exposed to simultaneous memory allocation requests [6].

Arguably, one of the greatest challenges that we have faced during the development of the thesis, was the question of whether we could or not improve the performance of Yap when handling concurrent data structures. This question led us to a hot topic in the parallel community which was *lock-free data structures*. With lock-free, we were able to reduce granularity of the synchronization, using low-level tools such as atomic compare-and-swap operations. A deep study led us to the development of two new state-of-the-art lock-free trie data structures, specially aimed from environments with the characteristics of our system [13, 10, 11, 14].

It is somehow painful to recognize that, currently, Prolog systems are not in the same position as they were in the 80's. Other programming paradigms, languages and frameworks have positioned themselves as stronger alternatives. A major purpose of this thesis is them to get back part of the glamour of Prolog systems, specially in the parallel programming community. To do so, we will show how to take advantage of our multithreaded tabling framework to scale well-known dynamic programming problems [12]. With all these contributions and the ones that we discuss at the end, as further work, we hope to pinch Prolog and our specific implementation towards an efficient and novel parallel framework that can be useful to everyone interested in parallel programming.

1.2 Thesis Outline

This thesis is structured into eight chapters that reflect the work developed. Next, follows a summary of the main topics presented and discussed in each chapter.

Chapter 1: Introduction. The current chapter.

Chapter 2: Prolog and Multithreaded Tabling. Introduces the background terminology related with logic programming, the Prolog language and tabling. Focus is then given to the combination of tabling with multithreading, which is the core topic of this thesis.

Chapter 3: Concurrent Table Space Designs. Presents the three new concurrent designs for supporting multithreaded tabling at the table space level and their implementation in Yap Prolog, named as the YapTab-Mt framework.

Chapter 4: Concurrent Memory Allocation. Presents an efficient and scalable user-level memory allocator specially aimed for environments with the characteristics of our multithreaded tabling framework.

Chapter 5: Lock-Free Data Structures. Presents two proposals for lock-free data structures that address concurrency within table space data structures. For each proposal, it discusses the implementation of the concurrent search and insert operations, the correctness of the proposal and its efficiency in the context of the YapTab-Mt framework.

Chapter 6: Batched Scheduling on Concurrent Table Spaces. Describes key implementation details necessary to extend the system to support concurrent batched scheduling and presents a performance analysis comparison between local and batched scheduling. As we will observe, the default implementation supports local scheduling.

Chapter 7: Subgoal-Sharing with Shared Answers. Discusses how to scale the execution of concurrent tabled programs using a subgoal-sharing design.

Chapter 8: Conclusions. Discusses the research and contributions of the thesis to the state-of-the-art in multithreaded tabling systems and suggests directions for future work.

Chapter 2

Prolog and Multithreaded Tabling

This chapter introduces the background needed for the following chapters. First, it describes the background terminology related with Logic Programming, the Prolog language and tabling. Next, it introduces multithreading in the context of logic programs and gives an overview about the state-of-the-art systems on the combination of tabling with multithreading, which is the core topic of this thesis.

2.1 Logic Programming

Logic Programming roots started mostly with Robinson in 1965, when he began the research for an automated theorem proving tool, on his work about the *Resolution Principle* [104]. The resolution principle is based on the induction principle "*if the implication $A \Rightarrow B$ is true, then to prove B, it is sufficient to prove A*". The expression *Logic Programming* was introduced afterwards by Kowalski, to designate the use of logic as the theoretical base for computer programming languages [64]. Kowalski showed how Selected Linear Deduction (SLD) resolution treats implications as deduction procedures. Kowalski and Kuehner argued that SLD resolution was the best inference system for first order logic, because it fills the following criteria [65]:

- Admits few redundant *deductions* and limits those which are irrelevant to a *proof*;
- Admits simple proofs;
- Determines a *search space* which is amenable to a variety of methods for heuristic

search.

The completeness of SLD resolution ensures that, by applying SLD resolution to a theory (or computer program) and a query, is it possible to use the theory to search for all solutions that satisfy the query [29, 3, 74].

Logic Programming is based on *predicate calculus*. An algorithm is seen as a set of two disjoint elements: *logic* and *control*. The logic component corresponds to the definition of the problem to be solved, while the control component, defines how the solution can be reached. The programmer needs only to *specify* the logic component of the algorithm, which is the problem to be solved, and leave the control of execution to the Logic Programming system.

According to Karlsson [62], Prolog as become the most popular Logic Programming language due to its efficient implementations. Prolog has the following major features:

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

Common literature, also recognizes that Prolog has the following advantages [25]:

- **Simple declarative semantics**. A logic program is simply a collection of predicate logic clauses.
- **Simple procedural semantics**. A logic program can be read as a collection of recursive procedures. Clauses are tried in the order they are written and goals within a clause are executed from left to right.
- **High expressive power**. Pure Prolog is based on a subset of first-order predicate logic and Horn clauses, thus it is Turing complete. Logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.

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

These advantages lead to a more flexible programming style, in the sense that programs are more easy to understand, transform and/or expand.

The basic data structures for logic programs are called **terms**. Terms can be constants, variables or functors (functional terms). A **functor** can be identified by name and arity (number of arguments). For example, **f/n** denotes the functional term $f(t_1, ..., t_n)$, where t_1 to t_n are themselves terms and called the **arguments** of **f**. **Constants** can be considered functors with arity zero, **atoms** represent symbolic constants syntactically similar to terms and **literals** are similar to terms, except that literals form individual goals to which a truth value can be assigned.

A **substitution** (or unification) is an operation that replaces some variables occurring in a formula with terms.

A logic program is a finite set of **clauses**. Each clause has the logic form:

$$\forall_{\vec{X}}(C \Leftarrow A_1 \wedge A_2 \wedge ... \wedge A_n)$$

where, C is called the **head**, $A_1 \wedge A_2 \wedge ... \wedge A_n$ is called the **body**, individual A_i's are called **atoms** and $\vec{X}$ represents the vector of variables present in the clause. If $n = 0$, the clause is called a **fact**. A set of clauses with the same functor name in the head and arity define a **predicate**.

A **computation** (or evaluation) of a logic program corresponds to the act of solving a query, by searching for a proof (a sequence of logical deductions from the clauses and facts) for all goals present in the body of the query and substitutions for all variables present in the goals. **Goals**, have the following logic form:

$$\forall_{\vec{X}}(\Leftarrow A_1 \wedge A_2 \wedge ... \wedge A_n)$$

where $\vec{X}$ denotes the vector (possibly empty) of variables present on the query.

2.2 The Prolog Language

One of the most popular Logic Programming languages is the Prolog language. Prolog has its origins in a software tool implemented by Colmerauer in 1972 at the *Université*

de Aix-Marseille, that was named *PROgramation en LOGic* [30].

Prolog is based on Horn clauses, which are defined as,

$$c(\vec{X}) :- g_1(\vec{X_1}),\ g_2(\vec{X_2}),\ ...,\ g_n(\vec{X_n}).$$

for clauses with head and body ($n \geq 1$), and

$$c(\vec{X}).$$

for fact clauses ($n = 0$). The symbol :- represents the implication $\Leftarrow$, the comma (,) represents the conjunction symbol $\wedge$. The $\vec{X}$, $\vec{X_1}$, $\vec{X_2}$ and $\vec{X_3}$, represent the vectors of the arguments on each goal.

Pure and sequential evaluations in Prolog systems consist then in traversing a search tree in a *depth-first left-to-right* form. Next, using Figure 2.1 we introduce some key terminology about the Prolog evaluation.

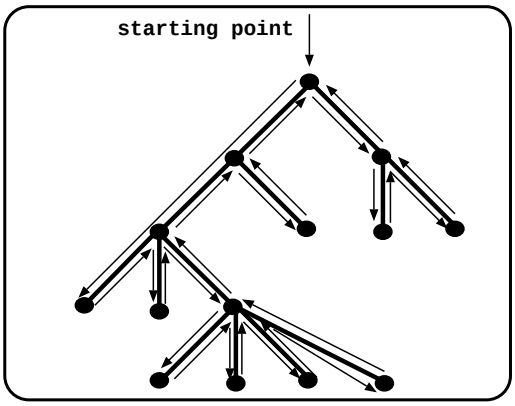

Figure 2.1: Depth-first search with backtracking in Prolog

Non-leaf nodes of the search tree represent stages of computation (*choice points*) where alternative branches (clauses) can be explored, to satisfy program's query, while leaf nodes represent solution or fail nodes. When the computation reaches a non-leaf node and can not advance any further, Prolog starts the *backtracking* mechanism, which consists in restoring the computation up to the previous node and schedule an alternative unexplored branch. A programmer can optimize the default search procedure by pruning the search tree through the use of the *cut* operator (!). Cut allows programs to use less memory and to be faster, because it reduces the allocation of backtracking nodes and thus the search space [131].

Some major characteristics of Prolog systems can be resumed as follows:

- It is a system oriented for symbolic processing;
- Presents a declarative semantic inherent to logic;
- Supports iterative and recursive programs;
- Represents programs and data with the same formalism;
- Allows different answers for the same query.

When comparing with imperative languages, we can see Prolog's evaluation as a natural generalization of the execution of imperative languages, that can be summarized as [4]:

$$\text{Prolog} = \text{imperative language} + \text{unification} + \text{backtracking}$$

As in imperative languages, the execution flow is left to right within a clause. The goals in the body of a clause are called like procedures. When a goal is called, the program clauses which match with it, are chosen in the top-bottom textual order. Figure 2.2 resumes our view about the relation between concepts used in Prolog and in imperative programming languages.

In general, the Prolog performance in the execution time and memory used is lower than imperative languages, due to the extra control and structures required by the unification and backtracking procedures, but the trade-offs are considered to be good enough for a logical and efficient programming style to be possible [87].

Next, we present an example for the evaluation of a small program in a standard Prolog system. To do so, we will use a *well-known* program which is the *path problem* with a small graph. The path problem is typically defined by two predicates, a first predicate that defines the transition of a graph and a second predicate that defines how the graph is connected. Consider next the $path/2$ and $edge/2$ definitions that illustrate in the Prolog language, the first and second predicates respectively.

$$path_1 = \begin{cases} path(X, Z) :- edge(X, Y), path(Y, Z). \\ path(X, Z) :- edge(X, Z). \end{cases}$$

$$edge_1 = \begin{cases} edge(1, 2). \\ edge(2, 3). \end{cases}$$

Predicate $path/2$ has two clauses, the first defines the transitivity property of a graph, which states that exists a path between two nodes X and Z, if exists an edge between

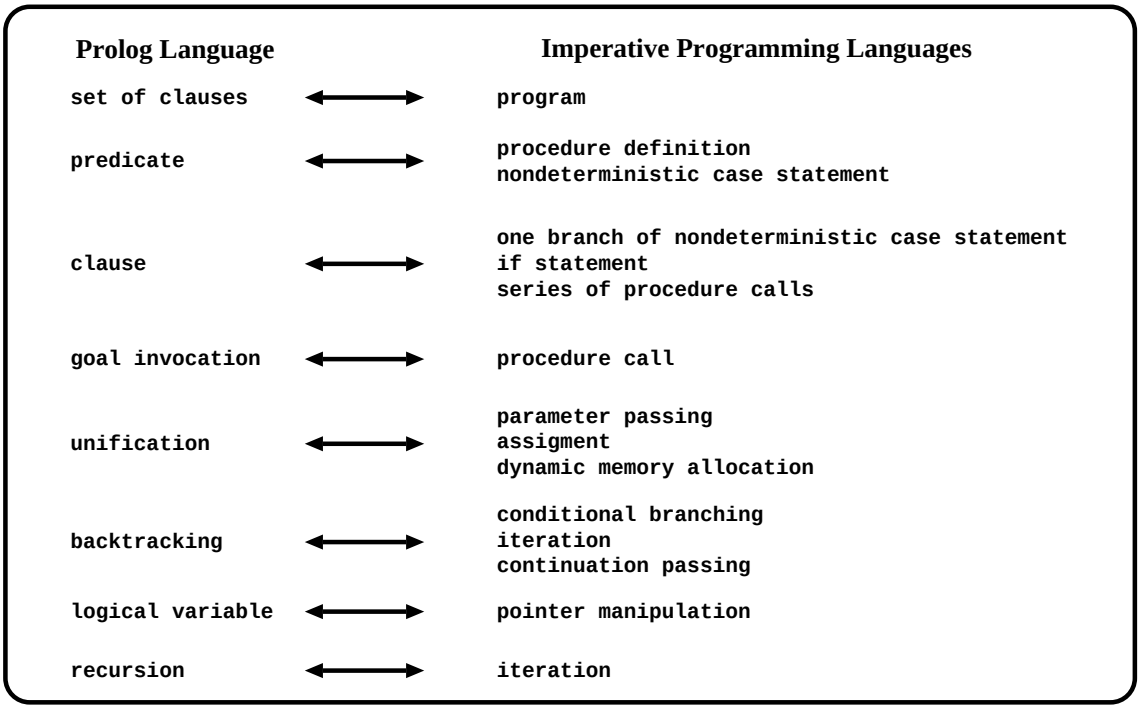

Figure 2.2: Correspondence between concepts used in the Prolog language and in imperative programming languages

the node X and a node Y and exists a path between the node Y and the node Z. The second clause defines that exists a path between nodes X and Z, if exists an edge between both nodes. Predicate $edge/2$ has also two clauses that define a direct acyclic graph with three nodes, where the node 1 is adjacent to the node 2, and 2 is adjacent to the node 3. For later reference in this chapter, we will name these definitions of the $path/2$ and $edge/2$ as $path_1$ and $edge_1$, respectively.

Figure 2.3 uses this definition of the path problem with the clauses numbered with $c1$, $c2$, $c3$ and $c4$ to show the evaluation tree for the top query call $path(1, Z)$. The evaluation tree sequence is numbered in steps and the top query begins with step 0. The aim of the top query is to find all nodes that can be reachable from node 1, thus the solution set is $\{\{Z = 2\}, \{Z = 3\}\}$.

Concerning the evaluation, the Prolog system begins at step 0, by using the first clause from the path definition that matches the top query (clause $c1$) and unifies the variable X to 1, calling in the continuation $edge(1, Y)$ (step 1). At this step, the Prolog system uses the clause $c3$ to unify the variable Y to 2 and calls in continuation the subgoal $path(2, Z)$. For illustration purposes, we are showing each different call to the $path/2$ predicate as an independent sub-tree. Continuing with the depth-first left-to-right

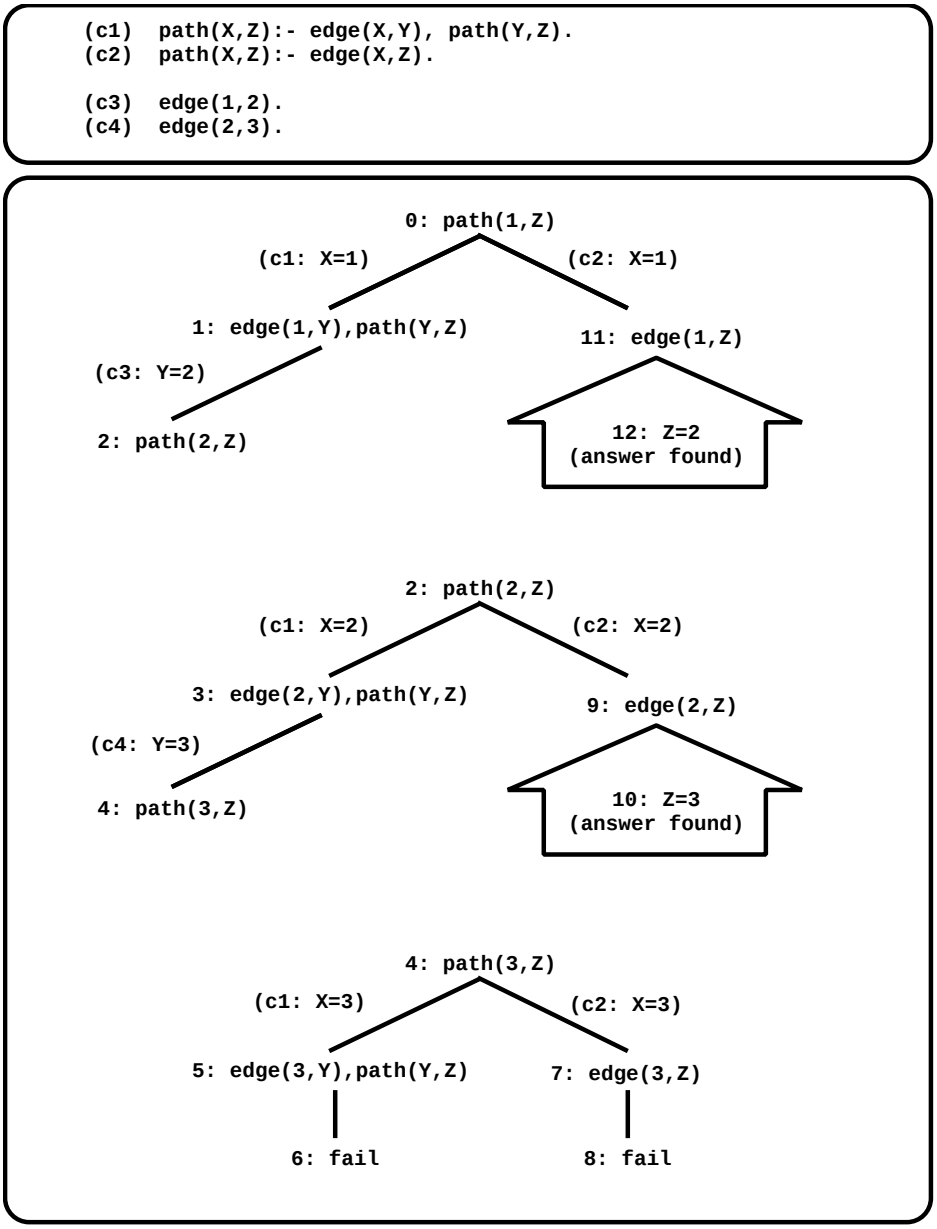

Figure 2.3: Evaluation of $path_1$ with $edge_1$

strategy, the evaluation continues using the clauses $c1$ with $X = 2$ and $c4$ with $Y = 3$ and the subgoal $path(3, Z)$ is reached at step 4. At this step the evaluation proceeds using the clause $c1$ with $X = 3$ and calls $edge(3, Y)$, but Y has no unification in the edge predicate, so the Prolog system fails (step 6) and backtracks to the subgoal $path(3, Z)$ and evaluates then the second clause that matches the subgoal $path(3, Z)$ (clause $c2$ with $X = 3$), leading again to $edge(3, Z)$ where the evaluation fails again (step 8). The Prolog system backtracks again, evaluates the second matching call for the subgoal $path(2, Z)$ and this leads to the answer $Z = 3$ at the step 10. Finally, the

Prolog system backtracks to the top unexploited clause matching subgoal $path(1, Z)$ (clause $c2$ with $X = 1$) and the evaluation reaches to the answer $Z = 2$ (step 12). As there are no further unexploited clauses, all the answers were found and the Prolog system finishes the evaluation. The result of the evaluation was correct since it equals the solution set.

2.2.1 The Warren's Abstract Machine

Most of the currently available Prolog systems are based on a sophisticated compilation technique and abstract machine known as the *Warren's Abstract Machine* (or simply WAM). The WAM was originally proposed by David H. D. Warren [133, 134] and its compiler correctness was later formally verified by Pusch in the work [96].

The tutorial book on the WAM [1], describes the WAM as a sequence of engines that incrementally support the different functionalities of a pure Prolog system. This division in incremental engines, benefits the presentation and comprehension of all the small tasks involved in the complex problem which is the implementation of a Prolog system. The minimal engine is the abstract machine M_0, which is only capable of determining whether a goal unifies with a given term. The abstract machine M_1 extends M_0, by allowing programs with more than one fact but with at most one fact per predicate name. The machine M_2, which is the next stage, is capable of compiling Prolog with conjunction rules (that is, with the form $a_0 :- a_1, ..., a_n$). The machine M_3, allows disjunctive definitions (more that one rule for each predicate), by adding the *backtracking* mechanism. Finally, the complete Prolog system is reached, by adding support for cuts, constants, lists and anonymous variables. Different Prolog systems employ also various design optimizations, such as swapping final instructions and/or avoiding the allocation of environments in special cases. The main goal behind all these optimizations is to reduce the computation's execution time and/or use as less memory as possible.

At the implementation level, the WAM is defined by a set of data structures, a set of registers and a set of low-level instructions.

Regarding the memory organization of the WAM, it is divided in seven logical data structures: one stack for data objects (the global stack), one stack for execution control (the local stack), one stack to support the interaction between the unification and the backtracking mechanism (the trail), one stack to support unification (the *Push Down List*, or PDL), one stack for the code area, one stack for the table of symbols and one

array to store argument registers. In more detail:

- **Global stack** (or **heap**). It is an array of data cells used to represent compound data terms, such as lists and structures.
- **Local stack**. It holds *environments* and *choice points*. Environments (also known as *local frames*) store the permanent variables for the current alternative clause, i.e., the variables that appear in more than a body subgoal and the continuation pointer. Choice points are used to store the current state of the computation. This means that, whenever a predicate starts execution, a choice point is allocated, with information of execution's state up to that moment, and with information about unexploited alternatives to be explored via the backtracking mechanism.
- **Trail**. It is used to store the addresses of the variables which must be unbound when backtracking occurs.
- **Push-Down List (PDL)**. This stack is used by the unification process when handling nested compound terms.
- **Code area**. This area contains the WAM compiled code of the programs loaded.
- **Symbol table**. Used to store information about the symbols, such as atoms or structures. An example is the mapping between the internal representation of a term and it's printing name.
- **Arguments array**. Used to store the arguments of the calls made during the evaluation.

The registers used to control WAM's flow of execution are described in Figure 2.4. The purpose of most registers is straightforward, but some can be not so obvious. For example, the HB register caches the value of H stored in the most recent choice point and is used to store the backtracking point in the heap structure. The S register is used during unification of compound terms (terms with arguments) and points to the argument being unified. The arguments are accessed one by one by successively incrementing this register. Some instructions have different behaviors during read and write mode unification, and the mode flag is used to distinguish between both situations.

Figure 2.5 shows the correspondence between registers and stacks. It also shows the information stored by choice points, environments and data terms. The choice points

P	Program Counter
CP	Continuation Pointer (top of return stack)
E	Current Environment Pointer (in local stack)
B	Most Recent Backtrack Point (in local stack)
A	Top of local stack (max between E and B)
TR	Top of Trail
H	Top of Heap
HB	Heap Backtrack Point (in heap)
S	Structure Pointer (in heap)
Mode	Mode Flag (read or write)
A1,A2,...	Arguments
Y1,Y2,...	Permanent Variables

Figure 2.4: WAM's registers

store all key registers needed to restore the computation and launch the alternative clauses, which includes the continuation registers for the code area (BCP), the environment (BCE) and program counter (BP), additionally with the H, TR, B registers and the arguments of the present call ($A1$, ...,An). The environments store the previous (or continuation) environment (CE), the continuation pointer (CP) of the choice to which the environment is associated and the permanent variables ($Y1$, ...,Yn).

Regarding the low-level instruction set of the WAM, it can be divided into four major groups. The most relevant instructions per group are:

- **Choice point instructions**. They allow the allocation/deallocation of choice points and the recovery of the computation state stored on those choice points.
  - **try_me_else** L: creates a choice point and sets the label L as the next alternative for the choice point.
  - **retry_me_else** L: recovers the computation's state stored on the top-most choice point and updates the next alternative for the choice point to be L.
  - **trust_me**: recovers the computation's state stored on the top-most choice point and removes the top-most choice point from the local stack.
  - **try** L: creates a choice point, sets the next instruction as the next alternative for the choice point and moves the execution to L.
  - **retry** L: recovers the computation's state stored on the top-most choice point, updates the next alternative for the choice point to be the next instruction and moves the execution to L.

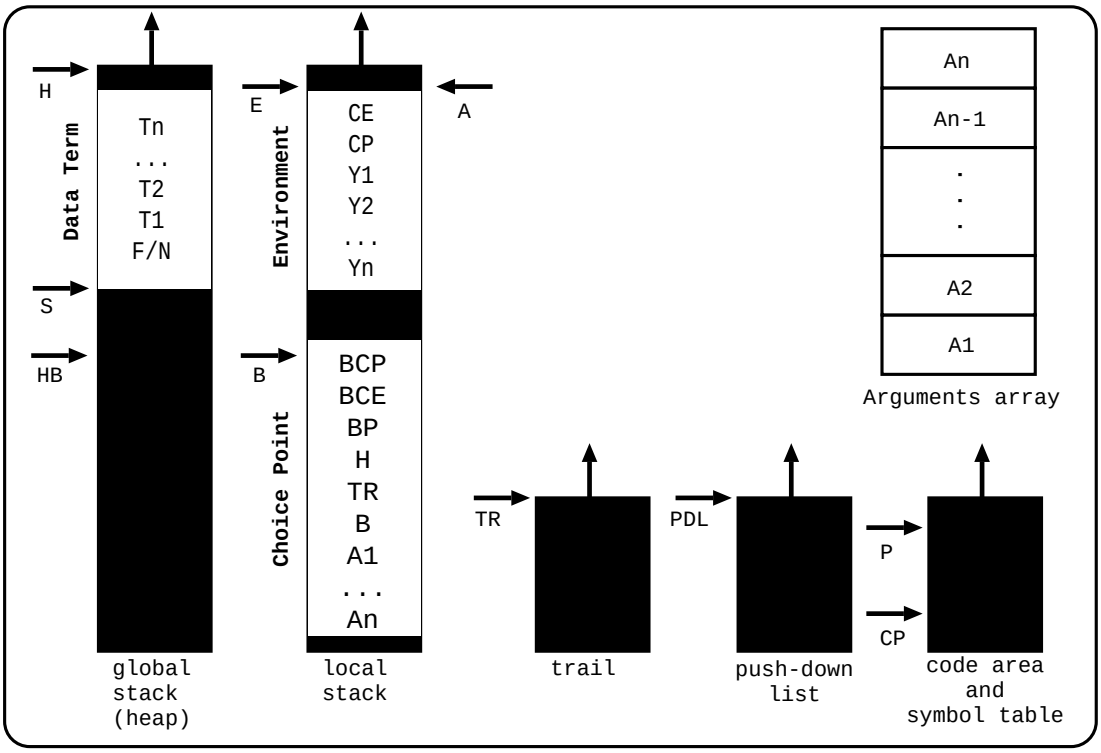

Figure 2.5: WAM's memory organization and registers

  - **trust** L: recovers the computation's state stored on the top-most choice point, removes the top-most choice point from the local stack and moves the execution to L.

- **Control instructions**. Used to allocate/remove environments and manage the call/return sequence of subgoals.

  - **allocate/deallocate**: used to create and remove environments, respectively.
  - **call** *pred, N*: calls the predicate *pred* and trims the current environment size to N (N represents the number of permanent variables that should be kept).

- **Unification instructions**. These instructions implement specialized versions of the unification algorithm according to the position and type of the arguments.

  - The **get_** instructions are used for head unification with registers. Some examples are *get_variable*, *get_structure* or *get_constant*.

  - The **unify_** instructions are used for head unification with structure arguments. Some examples are *unify_variable* or *unify_value*.
  - The **put_** and **set_** instructions are used for loading argument registers just before a call. Some examples are *put_structure*, *put_value* and *set_value*.

- **Indexing instructions**. These type of instructions accelerate the process of determining which clauses unify with a given predicate. Depending on the first argument of the call, they jump to specialized code that can directly index the unifying clauses.
  - The **switch_on_term** instruction is used to jump to specialized code accordingly to the type of term (being a variable, a constant, a list or a structure).
  - The **switch_on_constant** instruction indexes the clauses which match with a constant term.
  - The **switch_on_structure** instruction indexes the clauses which match with a structure term.

2.2.2 Infinite Loops

The Prolog language is based on the combination of the SLD resolution mechanism with linear top-down exploration of clauses defined in a program. This combination can be incomplete for certain types of programs. The cause for this incompleteness is the presence of recursive predicates during the evaluation of a program, which can lead to the infinite exploration of the same search space.

We use again the path problem example to show a situation where the usage of the SLD resolution is incomplete, but, for that, we introduce two new predicates, the $path_2$ and the $edge_2$. The predicate $path_2$ is logically equivalent to the previous $path_1$ definition and $edge_2$ defines a direct cyclic graph with two nodes.

$$path_2 = \begin{cases} path(X,Z) :- path(X,Y), edge(Y,Z). \\ path(X,Z) :- edge(X,Z). \end{cases}$$

$$edge_2 = \begin{cases} edge(1,2). \\ edge(2,1). \end{cases}$$

Consider now that we would like to use both definitions of the path predicate ($path_1$ and $path_2$), to compute the nodes that we can reach on both graph definitions ($edge_1$

and $edge_2$), starting from node 1 (i.e., we will use again the top query call $path(1, Z)$). Since both path predicates are logically equivalent, we would expect to get successful equal solutions. The solution set for $edge_1$ and $edge_2$ is $\{\{Z = 2\}, \{Z = 3\}\}$ and $\{\{Z = 1\}, \{Z = 2\}\}$, respectively. Next, we show the evaluation for the combination of each path predicate against each edge graph definition.

Figure 2.6 shows the evaluation tree for $path_1$ with $edge_2$. The evaluation begins with the first matching clause (the clause $c1$ with $X = 1$) for the subgoal $path(1, Z)$ and continues with the clause $c3$ with $Y = 2$, leading to a call to subgoal $path(2, Z)$, which then leads again to a call to subgoal $path(1, Z)$ (steps 0-4). This recursive call to $path(1, Z)$, defines a positive loop, but the Prolog system can not detect it. Since it is using SLD resolution, it will repeatedly begin another evaluation of $path(1, Z)$ and thus not finding any solution for this problem. The outcome of this evaluation would then be an infinite evaluation of the same query call and the right branches of the evaluation (marked with dots in Figure 2.6) would never be evaluated.

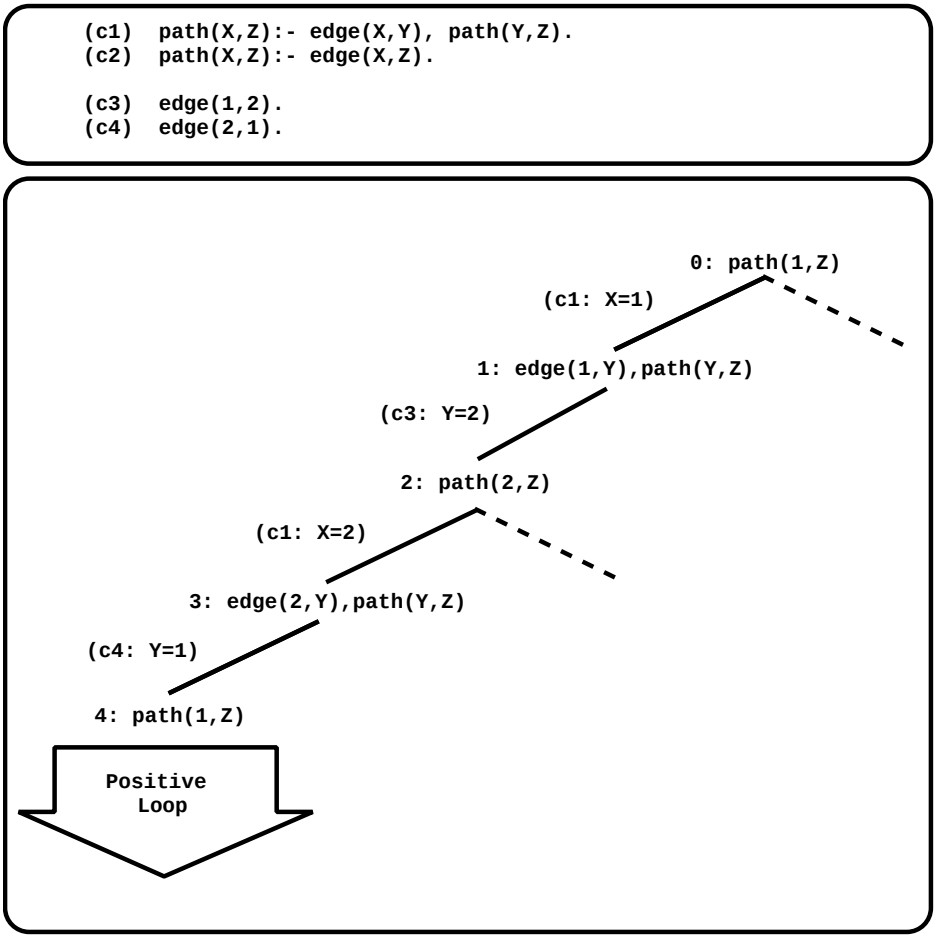

Figure 2.6: Evaluation of $path_1$ with $edge_2$

Figure 2.7 shows the evaluation tree of $path_2$ with $edge_2$. The Prolog system begins the evaluation by using the first clause that matches the call $path(1, Z)$ (step 1), leading to a call to $path(1, Y)$, which is a similar call (also known as a variant[1] call) to the top query call $path(1, Z)$. This then leads to the infinite repetition of the call, so the Prolog system would enter in to a positive loop and would not find any solution for the problem. One is able to observe that this behavior is independent of the edge definition and that the same outcome would occur for any edge definition.

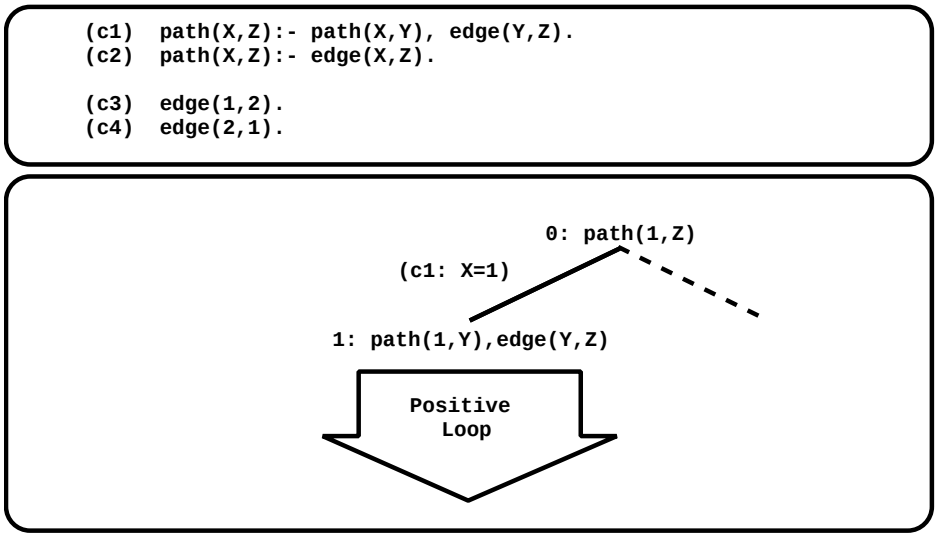

Figure 2.7: Evaluation of $path_2$ with $edge_2$

Therefore, we have seen that, when a Prolog system does not find positive loops during the evaluation, it returns the correct solutions, but when it finds a positive loop, it evaluates the same sub-computation infinitely without reaching to any solution.

This raises some disadvantages for standard Prolog systems. Logically correct problems, such as the path problem, can not be evaluated correctly. The declarative advantage of logic programs became dependent on the programmer's capability of designing his programs with clauses in the correct order and/or adding extra control predicates to avoid programs entering in to infinite loops. Important applications such as Datalog, which is a query language for deductive databases, can not be used because the evaluation does not terminate [111].

The operational incompleteness of Prolog is a well known problem and several proposals to improve Prolog's declarativeness exist. Next we will discuss one of such proposals, generically known as *tabling* (also known as *tabulation* or *memoing*) [84]. Tabling

[1]Variant calls of a subgoal are calls which differ only on variable renaming.

is a kind of *dynamic programming technique* contextualized in a Prolog environment, that has proved its viability for application areas such as deductive databases [111], inductive logic programming [106], knowledge based systems [137], model checking [97], parsing [61], program analysis [32], reasoning in the semantic Web [146], among others. Currently, the tabling technique is widely available in systems like XSB Prolog [125], Yap Prolog [118], B-Prolog [138], ALS Prolog [48], Mercury [120], Ciao Prolog [28] and Picat [140].

2.3 Tabling in Prolog

The key idea of tabling is to use an auxiliary data space, the *table space*, to keep track of the subgoal calls in evaluation and store, for each subgoal, the set of *answers* which are found during program's evaluation. Whenever a similar subgoal call appears, the subgoal is resolved by consuming answers from table space instead of executing the program clauses. This process is called *answer resolution*. In the meantime, as new answers are found, they are added to their tables and later returned to all repeated calls. By using answer resolution in this manner instead of program resolution as usual, tabling based systems can avoid looping and redundant sub-computations reducing the search space and ensuring termination for a wider group of programs [27].

The Ordered Linear Deduction with Tabling (OLDT) [126] was one of the first approaches used to supply the incompleteness of standard Prolog systems. It was presented by Tamaki and Sato, and combines the use of Ordered Linear Deduction (OLD) resolution with a tabling technique. The Selected Linear Goal-oriented (SLG) resolution [27], is another tabling mechanism that has been gaining popularity, since its implementation on the XSB Prolog system [100, 111].

The XSB Prolog design uses an adapted version of the standard WAM, called SLG-WAM [110, 123], that extends SLD resolution with new tabling related structures. The SLG-WAM defines nodes in a different way from the WAM. A node is defined as Generator Node (GN) if it corresponds to first call of tabled subgoal (used to generate answers for the tabled call), Consumer Node (CN) if it corresponds to a similar call to tabled subgoal (used to consume the answers of the tabled call) and Interior Node (IN) if it corresponds to non-tabled subgoals.

The YapTab system extends the Yap with a tabling engine similar to the XSB Prolog engine [105]. In order to give a clear perspective about how YapTab works, we next show on Figure 2.8, the tabled evaluation of the $path_1$ definition with the $edge_2$

transition graph. Recall that the traditional Prolog would immediately enter an infinite loop because the SLD resolution would lead the evaluation to a repeated call to $path(1, Z)$. In contrast, if tabling is applied then termination is ensured. The declaration $:-$ *table* $path/2$ in the program code indicates that predicate $path/2$ should be tabled. Figure 2.8 illustrates the evaluation sequence when using tabling.

```
     :-table path/2.

(c1)  path(X,Z):- edge(X,Y), path(Y,Z).
(c2)  path(X,Z):- edge(X,Z).

(c3)  edge(1,2).
(c4)  edge(2,1).
```

Subgoal Call	Answers
0: path(1,Z)	7: Z = 1 9: Z = 2 13: complete
2: path(2,Z)	6: Z = 1 11: Z = 2 13: complete

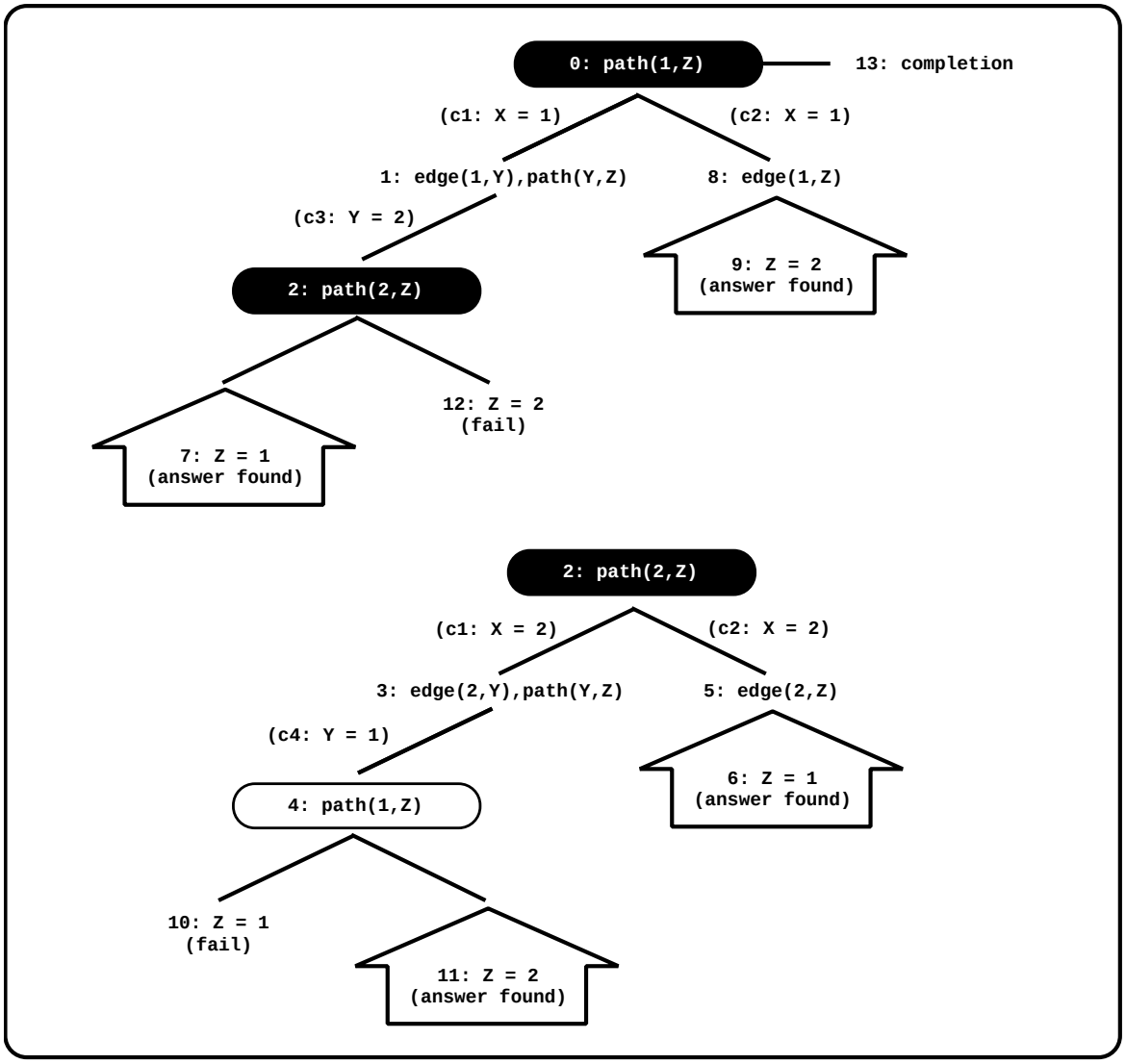

Figure 2.8: Tabled evaluation of $path_1$ with $edge_2$

At the top, the figure illustrates the program code and the state of the table space

at the end of the evaluation. The main sub-figure shows the forest of SLG trees for the original query. The top-most tree represents the original invocation of the tabled subgoal $path(1, Z)$. It thus computes all nodes reachable from node 1. As before, computing all nodes reachable from 1 requires computing all nodes reachable from node 2. The bottom-most tree represents the SLG tree $path(2, Z)$, that is, it computes all nodes reachable from node 2. Next, we describe in detail the evaluation sequence presented in the figure. The numbering within the SLG tree denotes the evaluation steps.

Whenever a tabled subgoal is first called, a new tree is added to the forest of trees and a new entry is added to the table space and a new GN is created for the call (nodes depicted by black oval boxes). In this case, execution starts with GN 0. The evaluation thus begins by creating a new tree rooted by $path(1, Z)$ and by inserting a new entry in the table space for it. The second step is to resolve $path(1, Z)$ against the first clause for $path/2$ (clause $c1$ with $X = 1$), leading in the continuation to the first call of $path(2, Z)$ (clause $c3$ with $Y = 2$). Since this is the first call to $path(2, Z)$, we must create a new tree rooted by $path(2, Z)$ (step 2) with a new GN, insert a new entry in the table space for it, and proceed with the evaluation of $path(2, Z)$, as shown in the bottom-most tree.

The evaluation proceeds using the clause $c1$ with $X = 2$, leading in the continuation (step 4) to a repeated call of $path(1, Z)$. This creates a Strongly Connected Component (SCC) [127], since both subgoal calls are now mutually dependent. We will represent a SCC through its leader node. More precisely, the youngest GN node which does not depend on older generators is called the *leader node*. A leader node is also the oldest node for its SCC, and defines the next completion point. Consequently, in our evaluation, the leader subgoal of the SCC is now $path(1, Z)$[2].

Continuing with the evaluation, the repeated call to $path(1, Z)$ is resolved using CN 4 (node depicted by a white oval box). At this point, the table does not have answers for the call $path(1, Z)$, therefore CN 4 must *suspend* execution. The evaluation then backtracks to GN 2, which uses clause $c2$ with $X = 2$ to call $edge(2, Z)$ and the answer $\{Z = 1\}$ is found for $path(2, Z)$ and stored in the table space (step 6). Then, we backtrack again to GN 2, but since $path(2, Z)$ is not a leader node, the node propagates its answers to the SCC, originating a new answer $\{Z = 1\}$ for $path(1, Z)$ (step 7). Next, we backtrack to GN 1, evaluate the clause $c2$ with $X = 1$, and in the continuation, in step 9, we find the answer $\{Z = 2\}$ for $path(1, Z)$.

[2]The reader can observe that table space remains unchanged. This happens because the subgoal was already inserted in the first call (step 0).

We backtrack again to GN 1, but the node cannot complete because it has a consumer below (CN 4) with unconsumed answers. We thus try to complete by sending answers to CN 4. The first consumed answer $\{Z = 1\}$ leads to a repeated answer for $path(2, Z)$ (step 10). SLG resolution does not store duplicate answers in the table, instead, repeated answers fail. This is how it avoids looping and even unnecessary computations in some cases. The second consumed answer $\{Z = 2\}$ leads to a new answer for $path(2, Z)$, which is stored in the table space (step 11) and later propagated by GN 2 leading to a repeated answer to $path(1, Z)$ (step 12).

Finally, since we have fully evaluated all clauses and fully consumed all answers, we can complete the evaluation of both calls within the table space (step 13). From this point and on, if any of these calls happens to be called again in other evaluations, they would be resolved using only a CN, which would work as a traditional choice point but, instead of using program clauses, it would consume the answers that were stored on the table space one by one through backtracking.

2.3.1 Compilation and Instruction Set

Concerning the compilation of tabled logic predicates, when a tabled predicate is loaded in a Prolog system supporting tabling, the parsing phase will search for *table p/n* declarations. These declarations indicate that calls to predicate *p/n* are to be executed using tabled evaluation instead of the pure SLD resolution. For each one of these declared predicates, the Prolog system creates a table entry structure in the table space. If more than one predicate is declared as tabled, these table entry structures are chained in a linked list thus that they can be accessed during the evaluation of the tabled predicates. Besides the table entry structure on the table space, these predicates are compiled with specific tabling instructions that will allow the tabling component of the system to have extra control over the program's flow of execution. The most important tabling instructions are:

- **Table Subgoal Call (TSC)**: checks if a call is the first call for a tabled subgoal. If so, it allocates a generator node and adds a new entry to the table space for the subgoal at hand. Otherwise, the subgoal is already in the table space, meaning that it is not the first call, so this instruction allocates a consumer node and resolves the subgoal by consuming the available answers. In the previous example shown in Figure 2.8, this instruction would be called during the steps 0, 2, and 4.

- **Table New Answer (TNA)**: checks if an answer found for a particular subgoal is new or repeated. If the answer is new, it is inserted in the table space and the evaluation proceeds accordingly with the scheduling strategy (this will be discussed in more detail in the Chapter 3). Otherwise, if the answer is repeated, the evaluation simply fails. In the example shown in Figure 2.8, this instruction would be called during the steps 6, 7, 9, 10, 11, and 12.

- **Table Completion Check (TCC)**: determines whether a completion point (also known as fix-point) was reached. A completion point is reached when no unconsumed answers are available for consumers and generators have explored all the available alternatives. When this is the case, all subgoals can be marked as completed. As an optimization subgoals, completion detection is only performed at leader nodes. In the example shown in Figure 2.8, this instruction would be called three times, the first between steps 6 and 7, the second between steps 9 and 10 and the third in the step 13. The first time was called to mark the GN as non leader, the second time was to schedule the SCC for a new re-evaluation and the third was to complete the SCC since all answers were found.

The TSC instructions are an extension of the original WAM choice point instructions, while the TNA and TCC instructions were created exclusively for tabling support. Using this terminology, Figure 2.9 shows a generic transformation at the tabling engine level of the original $path_1$ program into a program using tabled evaluation.

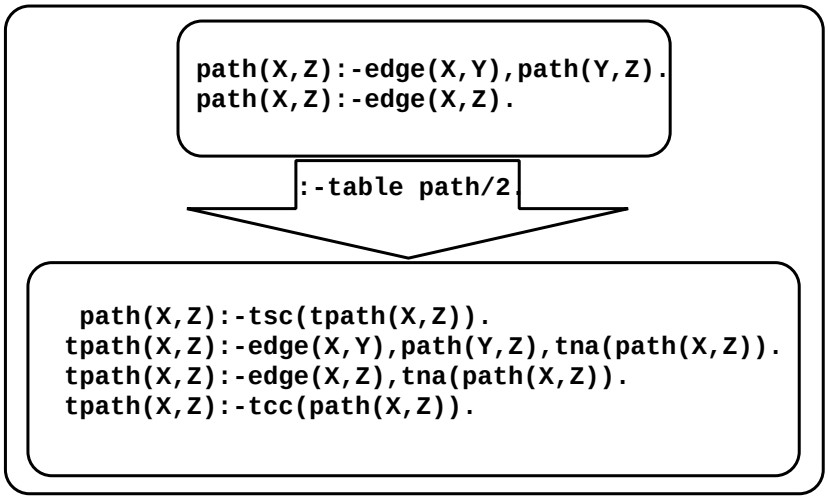

Figure 2.9: Generic tabled transformation for the $path_1$ program

After transformation, the $path/2$ predicate remains only on one clause, which works as an entry point to the new auxiliary predicate $tpath/2$ representing the transformed

predicate. This new predicate includes three clauses, the first two are the extension of the original *path*/2 clauses with TNA instruction at the end (this will allow the detection of all answers found on each clause) and the third clause implements the completion check procedure.

2.3.2 Mode-Directed Tabling

In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling is very good for problems that require storing all answers. However, often the goal of programs that use tabling is to dynamically calculate optimal or selective answers as new results arrive. Writing tabled logic programs can thus be a difficult task without further support.

Mode-directed tabling [49] is an extension to the tabling technique that supports the definition of *modes* for specifying how answers are inserted into the table space. Within mode-directed tabling, tabled predicates are declared using statements of the form *table* $p(m_1, ..., m_n)$, where the m_i's are *mode operators* for the arguments. The idea is to define the arguments to be considered for variant checking (the index arguments) and how variant answers should be tabled regarding the remaining arguments (the output arguments). Implementations of mode-directed tabling are available in ALS-Prolog [50], B-Prolog [142] and Yap Prolog [114], and a restricted form of mode-directed tabling can also be reproduced in XSB Prolog by using *answer subsumption* [124]. Mode-directed tabling has be used recently in the *BPSolver program*, to apply a dynamic programming approach to the *Sokoban* problem [139], and in application areas such as Machine Learning [142], Justification [90], Preferences [50], Answer Subsumption [115], among others. YapTab implements mode-directed tabling through argument indexing and modes. The index arguments are represented with mode *index*, while arguments with modes *first*, *last*, *min*, *max*, *sum* and *all* represent output arguments. After an answer is generated, the system tables the answer only if it is *preferable*, accordingly to the meaning of the output arguments, than some existing variant answer [116].

We use the shortest path problem to show how mode-directed tabling works. We begin by defining the *path*/3 and *edge*/3 predicates, where both of them have a third argument which is the cost C of the transition between two nodes inside the graph.

$$path_3 = \begin{cases} path(X,Z,C) :- path(X,Y,C1), edge(Y,Z,C2), C \ is \ C1 + C2. \\ path(X,Z,C) :- edge(X,Z,C). \end{cases}$$

$$edge_3 = \begin{cases} edge(1,2,1). \\ edge(2,3,1). \\ edge(2,4,4). \\ edge(3,4,1). \\ edge(4,3,1). \end{cases}$$

The $path/3$ predicate still has two clauses, the first defines the transitivity property of a graph, which states that exists a path between two nodes X and Z with a cost C, if exists a path between the node X and a node Y with a cost $C1$ and exists an edge between the nodes Y and Z with a cost $C2$, such that the cost C is given by the sum of both costs $C1$ and $C2$. The second clause defines that exists a path between nodes X and Z, if exists an edge between both nodes with a cost C. Predicate $edge/3$ has five clauses that define a direct weighted graph with four nodes.

To give a clear perspective about how mode-directed tabling works, we next show on Figure 2.10, the evaluation of the $path_3$ with $edge_3$. The top left corner of the figure shows the program code, where the declaration *:- table path(index,index,min).* indicates to the Prolog system that the predicate $path/3$ is a mode-directed tabled predicate and the first two arguments are indexed and the third argument is the *minimum* mode. The bottom-left corner of figure shows the evaluation tree for the query goal *path(1,Z,C)* and the right part of the figure shows the table space. The solution set for the query goal *path(1,Z,C)* is $\{\{Z = 2, C = 1\}, \{Z = 3, C = 2\}, \{Z = 4, C = 3\}\}$.

Figure 2.10 shows that the execution tree follows the normal evaluation of a tabled program and that the answers are stored as they are found, which happens at steps 3, 5 and 6. The most interesting part happens at steps 8 and 10. At step 8, a new answer $\{Z = 4, C = 3\}$ is found. This new answer is a variant of the answer $\{Z = 4, C = 5\}$ found at step 6 but since it is minimal for the third argument, it replaces the previous variant answer (scratched answer in the table space). Finally, at step 10 the answer $\{Z = 3, C = 4\}$ is found, which is a variant of the answer $\{Z = 3, C = 2\}$ found at step 5, but since it is not minimal it does not replace the variant answer in the table space.

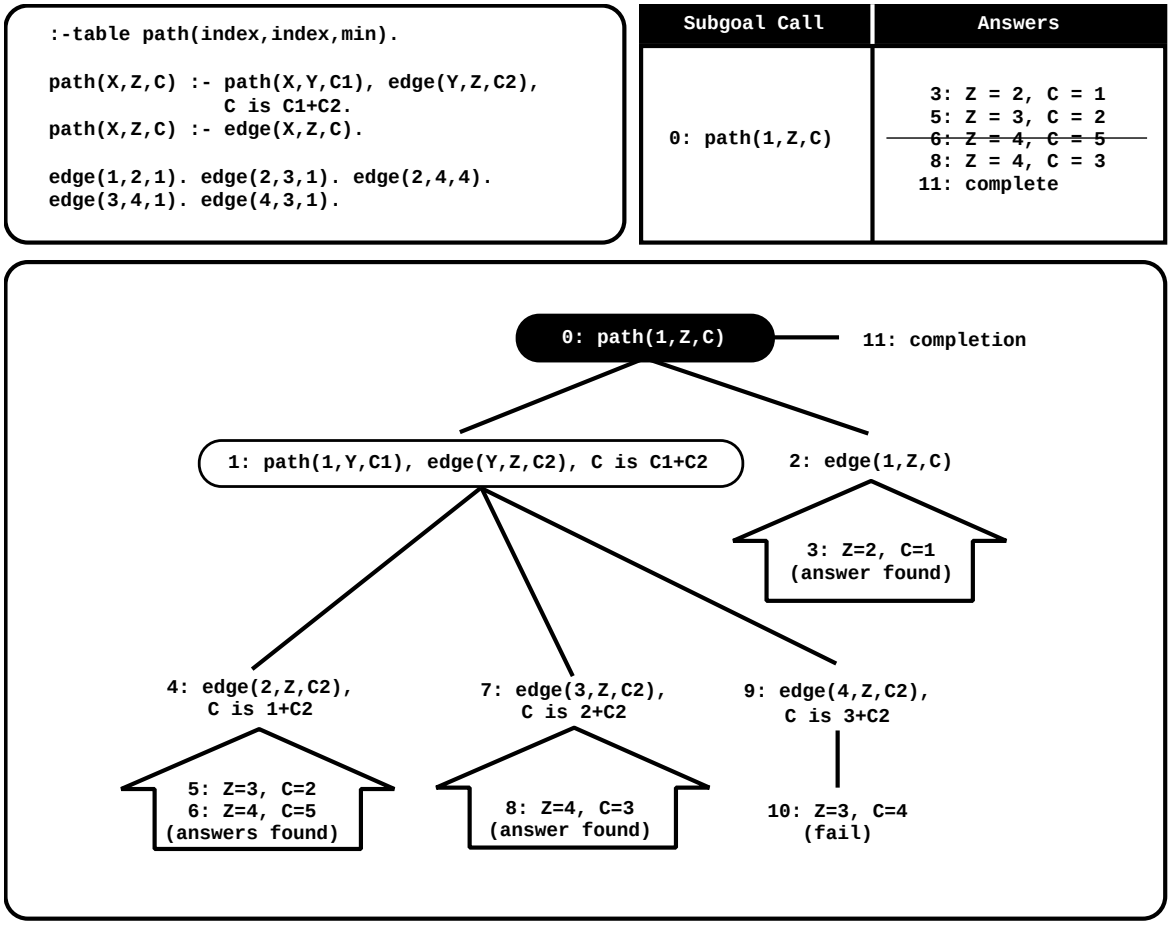

Figure 2.10: Mode-directed tabled evaluation of the $path_3$ with $edge_3$

2.3.3 Scheduling Strategies

The decision about the evaluation flow is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *batched scheduling* and *local scheduling* [42].

Batched scheduling schedules the evaluation of a program in a depth-first manner as does the WAM. It favors the forward execution first instead of backtracking, leaving the consumption of answers and completion for last. It thus tries to delay the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the execution continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC [110] and delaying the completion point to an older generator node.

On the other hand, the local scheduling strategy schedules the evaluation of a program in a breadth-first manner. It favors the backtracking first with completion instead of the forward execution, leaving the consumption of answers for last. Thus, it only allows a cluster of subgoals to return answers only after the completion point has been reached [42]. In other words, the local scheduling tries to keep SCCs as minimal as possible. When new answers are found, they are added to the table space and the computation fails as consequence, tabled subgoals inside an SCC propagate their answers to outside the SCC only after its completion point is found. Local scheduling causes a sooner completion of subgoals, which creates less complex dependencies between them.

The implementation of tabling engines on Prolog systems is actually based in two major paradigms: *linear-based* tabling and *suspension-based* tabling. Linear tabling mechanisms use iterative computations of tabled subgoals to compute completion points. The basic idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. Two different optimization proposals are the Selected Linear Deduction with Tabling (SLDT) strategy of Zhou et al. [144], as originally implemented in B-Prolog, and the Dynamic Reordering of Alternatives (DRA) strategy of Guo and Gupta [48], as originally implemented in ALS Prolog. The key idea of the SLDT strategy is to let repeated calls execute from the backtracking point of the former call. The repeated call is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a completion point is reached. Following versions of B-Prolog implemented an optimized variant of this strategy which tries to avoid re-evaluation of looping subgoals [143]. The DRA strategy is based on dynamic reordering of alternatives with repeated calls. This strategy tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls (*looping alternatives*). It then uses the looping alternatives to repeatedly recompute them until reaching a completion point. A more recent proposal, name Dynamic Reordering of Solutions (DRS), was implemented on top of the Yap system [5]. Yap also supports the combination of the first two strategies with batched scheduling and all the three strategies with local scheduling [5, 7, 9].

On the other hand, the suspension-based tabling mechanisms, which are the focus of this work, suspend the execution stacks of the sub-computations corresponding to consumer nodes, in order to resume them as new answers are found for the tabled subgoals involved on those sub-computations. Since this mechanism avoids the re-evaluation steps required to put the computation on the same state where those

sub-computations were suspended (as it can directly restore the suspended stacks), it has the advantage of reducing the execution time of a program. Since the first implementation of a suspension based mechanism [110], different approaches of tabling were implemented. The Mercury implementation [120] and two alternative XSB-based models, the CAT [33] and the CHAT [34] models, copy the execution stacks to a separate storage place. Two more recent approaches, implemented in Yap [107] and in Ciao Prolog [28], feature a higher-level implementation of suspension-based tabling. They apply source level transformations to a tabled program and then use external tabling primitives to provide direct control over the search strategy.

2.3.4 Tabled Evaluations and the Table Space

The table space is a key component of a tabling engine. The overall performance of a tabling system can be strongly affected if the basic operations that manipulate the table space are not implemented efficiently. Typically, the table space can be accessed to lookup for tabled subgoals, to lookup for tabled answers, to insert new tabled subgoals and answers, and to consume answers present on each tabled subgoal. Currently, there are two major implementations: the B-Prolog system uses *hash tables* [144], and the YapTab and XSB Prolog systems use tries [105] based on the proposal made by I. V. Ramakrishnan et al. [98, 99]. The *hash tables* are expected to be slower than tries for complex terms, since tries provide a complete discrimination of terms, permitting the lookup and possibly insertion to be performed in a single pass through a term, but were shown to be more efficient in ground terms [141, 144].

Let us now analyze in more detail, how the tabling engine interacts with the table space. When a tabled call is made, the first operation is to ground the call. This grounding of the call makes it possible to distinguish between first calls and following calls to the same predicate. Figure 2.11 shows some grounding examples for a $p/3$ predicate. The non-variable terms present on the predicate remain unchanged, but the variables are abstracted and numbered by order of appearance. Thus, the first call $p(X, X, X)$ has only one variable, which is abstracted with VAR_0 during the grounding process[3], the second call and third calls have two variable each, thus both variables are grounded with VAR_0 and VAR_1, and finally the fourth call does not have any variable, thus after the grounding process, the call is exactly the same as before the process.

Then, the next step is to integrate the grounded call on the table space. The inte-

[3]The notion of grounding suggests the usage of terms without variables. However we are using *VAR* to show how the variables are represented internally by the tabling engine.

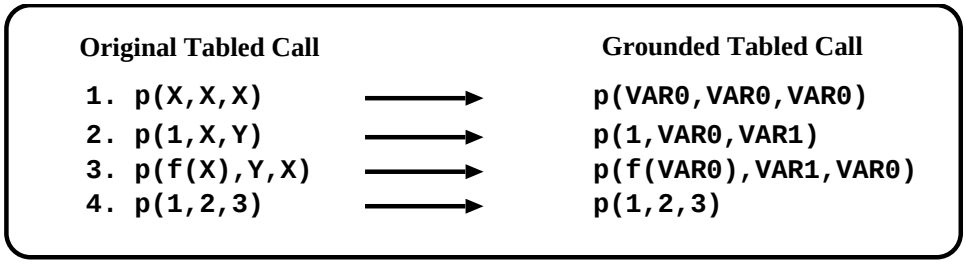

Figure 2.11: Grounding examples for a p/3 predicate

gration depends on whether the call is made via the TSC instruction or via the TNA instruction. For the TSC instruction, the tabling engine performs a search over the calls already in table space in order to check if the call is already there. If it is a first call then a new entry is created. Otherwise, it is a similar call, so the call is scheduled for answer consumption. For the TNA instruction, the tabling engine searches the answers in the table space for the corresponding tabled call and if it is a new answer, it is added to table space.

For table space implementation, XSB's and YapTab's original organization are based on tries, which are known to be efficient, because they allow for a compact representation of subgoal calls and answers to be represented through a unique path within the structure. A *trie* is a tree structure where each different path through the trie data units, the trie nodes, corresponds to a term. Each root-to-leaf path represents a term described by the tokens labeling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing token. Internally, the trie nodes are 4-field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Each node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers. Figure 2.12 shows an example for the table space organization starting from the tabled calls to predicate $p/3$ represented in Figure 2.11.

At the entry point we have the Table Entry (TE) data structure. This structure stores generic information about the predicates and is allocated when a tabled predicate is being compiled, so that a pointer to the table entry can be included in the compiled code. This guarantees that further calls to the predicate will access the table space starting from the same point. The TE data structure connects to a Subgoal Trie (ST), that is used to store tabled subgoal calls. On the ST, each different path corresponds then to a term described by the tokens labeling the nodes traversed. For example,

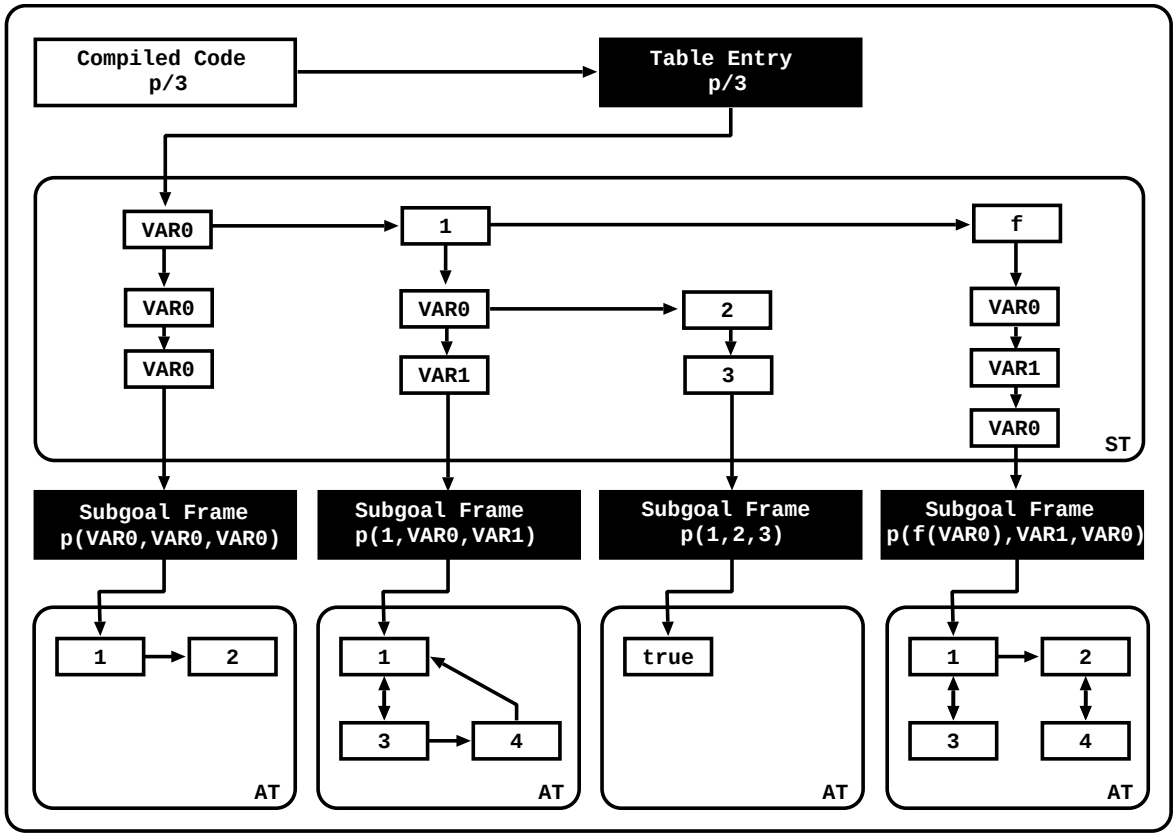

Figure 2.12: Table space representation using tries on the tabled predicate $p/3$

the tokenized form of the term $p(1, X, Y)$ is the sequence of 4 tokens $p/3$, 1, VAR_0 and VAR_1, where each variable is represented as a distinct VAR_i constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $p(1, 2, 3)$. Since the main functor and the first argument, tokens $p/3$ and 1, are common to both terms, only two additional nodes will be required to fully represent this second term in the trie. Thus, each different tabled subgoal call to the predicate at hand corresponds to a unique top-down path through the ST structure, always starting from the table entry, passing by several subgoal trie nodes until the leaf node is reached. The leaf nodes within the ST point to the Subgoal Frame (SF) structure. The SF stores additional information about the execution of subgoal calls and acts like an entry point to the Answer Trie (AT) structure.

The behavior of the AT structure is similar to the behavior of the ST structure when the trie structure is accessed in a top-down fashion. The difference is that each path corresponds now to a different answer to the tabled subgoal call. Moreover, ATs only store the substitution factor, i.e., the answers to the variables (VAR_i) in the corresponding call. Figure 2.12 shows some answers for the calls. For example, the

subgoal call $p(VAR0, VAR0, VAR0)$ has the answers 1 and 2, meaning that $p(1,1,1)$ and $p(2,2,2)$ are answers for this subgoal call. For the subgoal call $p(1, VAR0, VAR1)$, the answers shown are $p(1,1,3)$ and $p(1,1,4)$. For the subgoal call $p(1,2,3)$ the answer is *true* [4] and for the subgoal call $p(f(VAR0), VAR1, VAR0)$ the answers shown are $p(f(1),3,1)$ and $p(f(2),4,2)$. An important difference for the ST structure is that the AT structure can also be accessed in a bottom-up fashion. The bottom-up accesses occur when the trie structure is accessed by a CN.

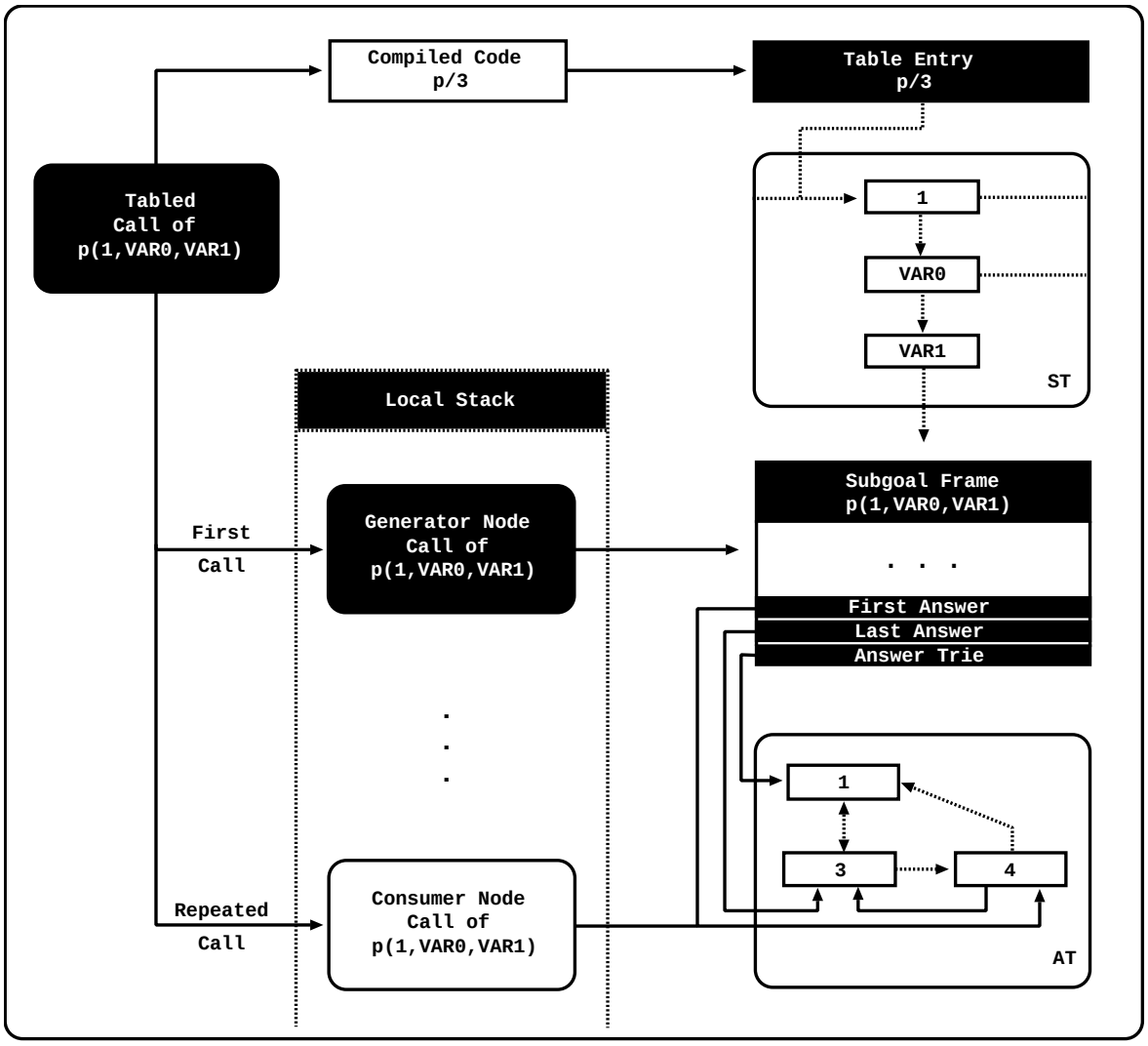

Figure 2.13: Accessing the table space

Figure 2.13 goes one step beneath and describes different types of accesses to the

[4] In subgoal calls that do not have variables, the answer *true* is stored in the AT structure if the subgoal has proven to the true during its evaluation, otherwise the *false* answer is stores in the AT structure.

table space by giving some low-level details about how the SF structure is used to provide bottom-up access to the AT structure. To do so, we will use the subgoal call $p(1, VAR0, VAR1)$ from the example in Figure 2.12.

Figure 2.13 shows then that a tabled call to the subgoal $p(1, VAR0, VAR1)$ (left-most black oval box) follows the pointer to the compiled code to access the corresponding TE, and from there access the ST data structure. Next, it traverses the ST structure in a top-down fashion. For the first call, the nodes are created and inserted in the path as it traverses the ST structure, leading then to the insertion of the SF structure after the leaf node. During the first call to a tabled predicate, a GN node is allocated in the local stack with a pointer to the SF structure of $p(1, VAR0, VAR1)$, allowing the GN to access directly the AT structure in order to easily check/insert for new found answers. Concerning the connections to the AT, the SF structure has three pointers:

- **Answer Trie**: points to the top of the AT structure, thus that it can be used by the GN as the entry point to the top-down path that allows to check if a answer is new or repeated.
- **First Answer**: points to the first answer found for the subgoal during the evaluation. In the figure the first answer is $(1, 4)$.
- **Last Answer**: points to the last answer found for the subgoal during the evaluation and it is used to chain the new answers with the answers already in the AT. In the figure the last answer is $(1, 3)$.

For a repeated call to $p(1, VAR0, VAR1)$ the tabled subgoal call operation allocates a CN in the local stack. A CN accesses its AT in a bottom-up fashion. The leaf nodes within the AT structure are chained so that the answers can be consumed by the CN node. A CN uses the first answer pointer to mark the beginning of the chain of answers to be consumed and the last answer to test on each round of evaluation if it has consumed or not all answers in the AT. For each answer, the consumer node begins by the leaf node and traverses the AT structure bottom-up (using the upper arrows as in Figure 2.13 until the top of the trie is reached).

2.4 Multithreaded Tabling in Prolog Systems

The ISO Prolog multithreading standardization proposal [85] is currently implemented in several Prolog systems including XSB [112], Yap [23], Ciao [23] and SWI-Prolog[136],

providing a highly portable solution given the number of operating systems supported by these systems. Yap implements a SWI-Prolog compatible multithreading library, where each Prolog thread is an Operating System (OS) native thread running a Prolog engine consisting in a set of stacks and the required state to accommodate the engine. After being started from a *goal*, a thread proves this goal just like a normal Prolog implementation [135]. Like in SWI-Prolog, Yap's thread library is based on the Portable Operating System Interface (POSIX) thread standard [24], where each thread has its own execution stacks and only share the code area where predicates, records, flags and other global non-backtrackable data are stored. Yap's thread specific predicates can be found in [117].

When multithreading is combined with tabling, one can have the best of both worlds, since one can exploit the combination of higher procedural control with higher declarative semantics. In a multithreaded tabling system, tables may be either private or shared between threads. While thread-private tables are easier to implement, shared tables have all the associated issues of locking, synchronization and potential deadlocks. Here, the problem is even more complex because we need to ensure the correctness and completeness of the answers found and stored in the shared tables. Thus, despite the availability of both threads and tabling in Prolog compilers such as XSB, Yap, and Ciao, the implementation of these two features such that they work together seamlessly implies complex ties to one another and to the underlying engine. To the best of our knowledge until this moment, the only systems that were able to support the combination of tabling with multithreading are XSB and Yap.

XSB was the first system to combine tabling with multithreading. It support two types of models for the combination: *private tables* and *shared tables* [76, 125]. On the private tables model, each thread keeps its own copy of the table space. On one hand, this avoids concurrency over the tables but, on the other hand, the same table can be computed by several threads, thus increasing the memory usage necessary to represent the table space.

For shared tables, the running threads store only once the same table, even if multiple threads use it. This model can be viewed as a variation of the *table-parallelism* proposal [41], where a tabled computation can be decomposed into a set of smaller sub-computations, each being performed by a different thread. Each tabled subgoal is computed independently by the first thread calling it, the *generator thread*, and each generator is the sole responsible for fully exploiting and obtaining the complete set of answers for the subgoal. Similar calls by other threads are resolved by consuming the answers stored by the generator thread.

Based on these two strategies, XSB supports two types of concurrent scheduling strategies: *concurrent local evaluation* and *concurrent batched evaluation*. In the concurrent local evaluation, similar calls by other threads are resolved by consuming the answers stored by the generator thread, but a consumer thread suspends execution until the table is completed. In the concurrent batched evaluation, new answers are consumed as they are found, leading to more complex dependencies between threads. In both scheduling strategies, when a set of subgoals computed by different threads is mutually dependent, then a *usurpation operation* [77] synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads.

From our point of view, the usurpation operation restricts the potential of concurrency to non-mutually dependent sub-computations. In problems that create mutual dependent sub-computations, which can be executed in different threads, XSB is actually unable to execute them in a concurrent fashion due to this operation. By other words, even if we launch an arbitrary large number of threads on those programs, the system would tend to use only one thread at the end to evaluate most of the computations.

In this thesis, we present an alternative view to XSB's approach, implemented on top of the Yap Prolog system, where each thread views its tables as private but, at the engine level, we use a *common table space*, i.e., from the thread point of view, the tables are private but, from the implementation point of view, the tables are shared among all threads. We propose three designs for the common table space: NS, SS and FS. The NS is similar to the XSB's private tables model and the SS and the FS designs are two new models that are aimed to be time and space efficient [8]. This will be discussed in detail in the next chapter.

2.5 Chapter Summary

This chapter introduced several key concepts about Logic Programming and the implementation of Prolog systems. It discussed some well-known and important limitations of Prolog systems in order to motivate for the appearance of tabling mechanisms. In the continuation, we have described the key concepts in the implementation of a tabling mechanism. The chapter concluded with the start-of-the-art about multithreading in the context of tabling mechanisms.

Chapter 3

Concurrent Table Space Designs

This chapter presents three new concurrent designs for the support of multithreaded tabling at the table space level and their implementation in the YapTab-Mt system. The last part of the chapter is dedicated to the presentation and discussion of experimental results using such designs.

3.1 General Idea

Multithreading in Prolog is the ability to concurrently perform multiple computations, in which each computation runs independently but shares the database (clauses). Yap is based on the POSIX thread standard [24], which defines a thread as an OS native thread running inside a process. Native threads use the operating system's native ability to manage multithreaded processes. In particular, they use the pthread library, which leaves to the OS kernel the ability to schedule and manage the various threads that make up the process. Using native threads, the OS is able to switch between threads preemptively, switching control from a running thread to a non-running thread at any time. On multi-CPU machines, native threads can run more than one thread simultaneously by assigning different threads to different CPUs. A Yap Prolog thread consists then in a combination of an OS native thread with a set of stacks and structures that are the required to accommodate the state of a Prolog engine. The key idea is that after a thread starts from a *goal*, it proves this goal just like a normal Prolog engine.

The current version of the Yap system incorporates by default the YapTab subcomponent which provides to the user a complete and transparent Application Programming

Interface (API) that gives the tools for the user to control as much as possible the tabling engine. The user is able to define multiple features about the tabling engine, such as defining the tabling scheduling strategy (batched or local), defining the table space model to be used (local tries or global trie), defining the load answer mechanism, or taking internal statistics about the execution. The YapTab-Mt system is the multithreaded version of YapTab. The YapTab-Mt system is a framework that is specially aimed to support the simultaneous usage of tabling and multithreading. The user defines the problem and declares the predicates that have to be solved using tabling and then uses a set of Prolog thread specific predicates to explicitly control how the problem will be concurrently evaluated.

Figure 3.1 shows a small example of a Prolog program for the concurrent evaluation of the path problem. The program has three parts. The first part is the path problem using the previous $path_1$ and $edge_2$ definitions. The second part is a naive thread scheduler. The scheduler is quite simple, it begins with the predicate $go\_scheduler(N)$, where N stands for the total number of threads to be launched by the scheduler. The $go\_scheduler(N)$ predicate uses then the $go\_threads(N, TidList)$ predicate to launch the concurrent evaluation of the path problem. The $TidList$ argument receives the list with the identifiers of the threads that were created, this is used afterwards by the $join\_threads(TidList)$ predicate to recursively join all threads. The $go\_threads(N, TidList)$ predicate is then defined by a recursion over the number N of threads to be launched, using the current N during the recursion to create a thread that will evaluate the $go\_path(N)$ predicate. The $go\_path(N)$ predicate simply calls the path predicate with the N bounded in the first argument and fails at the end, thus ensuring that all the nodes from the graph can be visited using the backtracking mechanism provided by the Prolog engine. The third and last part is used to launch the naive scheduler, in the example, with 2 threads.

The reader is encourage to compare the Prolog code of the single threaded version of *path problem* described the Figure 2.8 with the multithreaded version of *path problem* described in Figure 3.1. One will notice that the Prolog code of both *path problems* is exactly the same. It is also important to notice that the multithreaded tabling support is provided to the user in a completely transparent fashion. The high level Prolog instruction *table path/2* is actually abstracting the multithreaded framework that is beneath the YapTab-Mt engine. On the next sections, will go step by step into the engine level and make an exhaustive description about the infra-structure that was implemented to pass from the YapTab engine to the YapTab-Mt engine.

```
% tabling declaration
:- table path/2.
% the path problem
path(X, Z) :- edge(X, Y), path(Y, Z).
path(X, Z) :- edge(X, Z).
%graph configuration
edge(1, 2).
edge(2, 1).
% naive thread scheduler
go_scheduler(N) :-
    go_threads(N, TidList),
    join_threads(TidList).
go_threads(0, []).
go_threads(N, [Tid|TidList]) :-
    thread_create(go_path(N), Tid),
    N1 is N - 1,
    go_threads(N1, TidList).
go_path(N) :- path(N, _), fail.
join_threads([]).
join_threads([Tid|TidList]) :-
    join_threads(TidList),
    thread_join(Tid, _).
% launching a naive thread scheduler with 2 threads to
% concurrently evaluate the path problem with 2 nodes
:- go_scheduler(2).
```

Figure 3.1: An example of a concurrent evaluation of the *path* problem

3.2 Concurrent Table Space Designs

YapTab-Mt's key idea for multithreaded tabling is still based on the idea that each computational thread runs independently. This means that each tabled evaluation depends only on the computations being performed by the thread itself, i.e., there isn't the notion of being a consumer thread since, from each thread point of view, a thread is always the generator for all of its subgoal calls. We propose three different designs to accomplish this, the *No Sharing (NS)*, the *Subgoal Sharing (SS)* and the *Full Sharing (FS)* designs. We begin by introducing and analyzing the original YapTab's

sequential design and them we present the new concurrent designs by comparing them with YapTab's original design.

3.2.1 YapTab's Memory Usage Analysis

We show now a detailed analysis of the memory used by the YapTab system on the table space. This will be useful for the memory analysis of the multithreaded designs that we will be presenting next. We begin by remembering the structure of YapTab's table space with multiple tabled predicates. Figure 3.2 shows the general representation of a non multithreaded table space in YapTab.

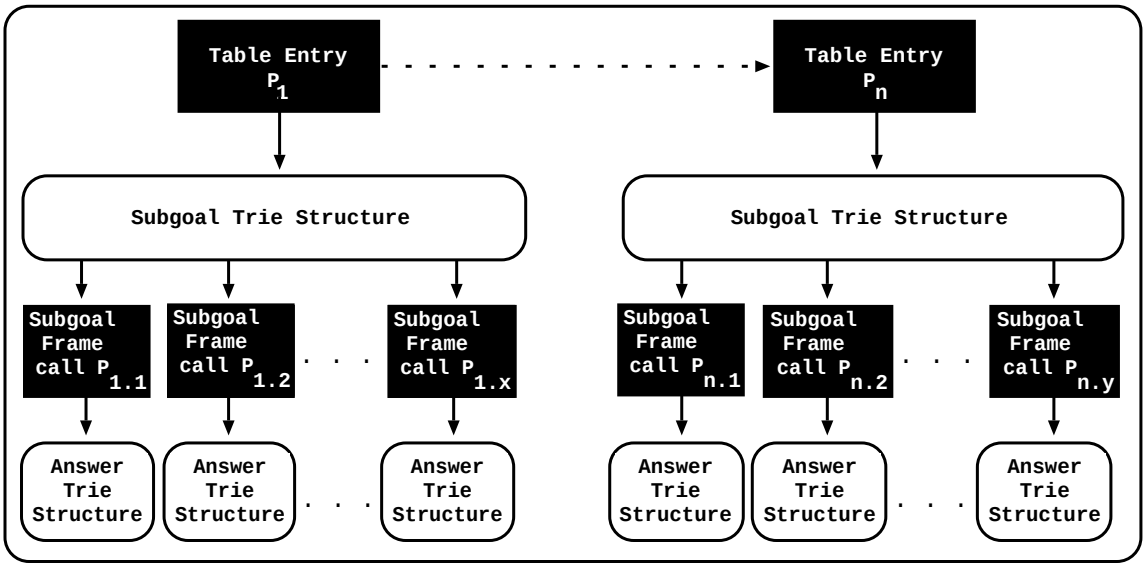

Figure 3.2: YapTab's original table space organization

At the entry point we have the TE structures for each tabled predicate P_i. Remember that, this structure is allocated when a tabled predicate is being compiled, so that a pointer to the table entry can be included in the compiled code for predicate P_i. The TE structure for each predicate is inside a chain, which in our example begins with predicate P_1 and ends with predicate P_n. Underneath each TE structure we have the subgoal trie and subgoal frame data structures for each call $P_{i.j}$ made to the predicate. In the example, predicate P_1 has x calls and predicate P_n has y calls. Finally, underneath each subgoal frame structure we have the answer trie structure with the answers.

We can now formalize the Total Memory Usage (TMU) of YapTab's table space design. For this, we assume that all tabled predicates are completely evaluated, meaning that

the system does not allocate any further structures on the table space. Given NP tabled predicates, Equation 3.1 presents the TMU of the YapTab system (TMU_{YT}).

$$\begin{aligned} & TMU_{YT} = \sum_{i=1}^{NP} MU_{YT}(P_i) \\ & where \\ & MU_{YT}(P_i) = TE_{YT} + ST_{YT}(P_i) + \sum_{j=1}^{NC(P_i)} [SF_{YT} + AT_{YT}(P_{i.j})] \end{aligned} \tag{3.1}$$

The TMU of YapTab is given by the summatory of the memory used for each predicate, i.e, the $MU_{YT}(P_i)$ value. The $MU_{YT}(P_i)$ value is then given by the sum of each structure inside the table space for the corresponding predicate P_i. The TE_{YT}, $ST_{YT}(P_i)$, SF_{YT}, $AT_{YT}(P_i)$ represent the amount of the memory used by predicate P_i in its table entry, subgoal trie, subgoal frames and answer trie structures, respectively, and $NC(P_i)$ represents the number of subgoal calls created during the evaluation of the predicate. In the example shown in Figure 3.2, the value of $NC(P_1)$ would be x and the value of $NC(P_n)$ would be y.

In what follows, we will be using $MU_{YT}(P_i)$ to make a performance analysis comparison between the memory used by YapTab and the three new concurrent table space designs, so that we can understand better the benefits that each model has in terms of memory usage. For this performance analysis, we will not be considering any synchronization mechanisms, such as lock fields, since several synchronization techniques exist that do not require an actual lock field inside the table space structures[1], and we will be assuming that an arbitrary number of threads NT have completely evaluated the same tabled predicate P_i in all of its subgoal calls $NC(P_i)$.

3.2.2 No-Sharing Design

The No-Sharing (NS) design was the first design to be implemented and the starting point of our work. We consider the eagerest design in terms of memory consumption because the key idea is to give complete priority to the private information of the threads. This means that each thread allocates private tables for each new subgoal

[1]Two examples that do not require lock fields are (i) a global array of locks outside the concurrent structures and (ii) the usage of the low level Compare-And-Swap (CAS) operation that is widely available in the current hardware architectures.

called during its computation, regardless of the behavior of the remaining threads working in the system. In this design, only the TE structure is shared among threads.

Figure 3.3 shows the configuration of a table space using the NS design on a table space with n predicates. As before, the TE structures of all predicates are chained. For the sake of simplicity, in the figure we are only showing the configuration of the NS design for a particular predicate P_i and a particular subgoal call $P_{i.j}$.

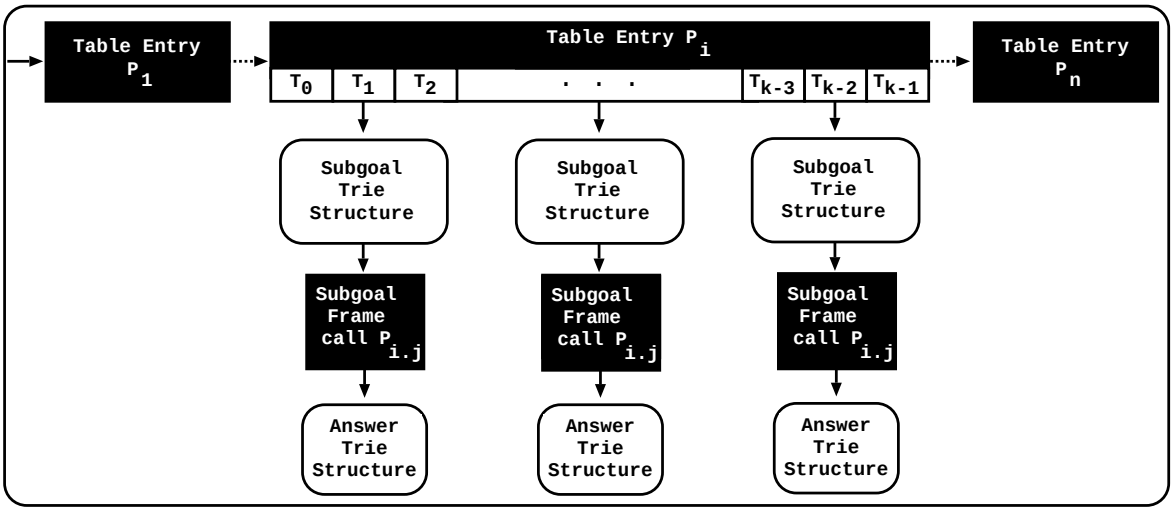

Figure 3.3: Table space organization for the NS design

In this design, the TE structure still stores the common information for the predicate (such as the predicate's arity or the predicate's evaluation strategy), but it is extended with a Bucket Array of Entries (BAE). Each thread t has its own cell T_t inside the BAE, which points to the private data structures of the thread. The ST, the SF and the AT structures are thus private to each thread and they can be removed when the thread finishes execution. As one can observe, only the TE is shared among threads. Since this structure is created by the main thread, when a program is being compiled, we can assume that, no synchronization or concurrent writing points exist.

Given NT threads, Equation 3.2 shows the memory usage analysis of the predicate P_i using the NS design ($MU_{NS}(P_i)$) and the conditions that determine the size of every structure in the design.

$$
\begin{aligned}
&MU_{NS}(P_i) = \\
&TE_{NS} + NT * [ST_{NS}(P_i) + \sum_{j=1}^{NC(P_i)} [SF_{NS} + AT_{NS}(P_{i.j})]] \\
&cond. \begin{cases} TE_{NS} = TE_{YT} + BAE \\ ST_{NS}(P_i) = ST_{YT}(P_i) \\ SF_{NS} = SF_{YT} \\ AT_{NS}(P_{i.j}) = AT_{YT}(P_{i.j}) \end{cases}
\end{aligned} \tag{3.2}
$$

The value is given by the sum of the sizes of the structures that are used by the NS design, i.e., the size of table entry structure (TE_{NS}) plus the sum of the sizes of the structures that are multiplied by the NT number of threads. The structures in the multiplication are the sum of the memory used in the subgoal trie structures($ST_{NS}(P_i)$) with the summatory of the memory used in the subgoal frame (SF_{NS}) and answer trie ($AT_{NS}(P_i)$) structures in the NC subgoal calls of the predicate P_i ($NC(P_i)$). Concerning the conditions that describe the size of the structures, the TE_{NS} size is given by the size of the table entry structure in YapTab ($TE_{YT}(P_i)$) added with the size of the bucket array of entries (BAE). The size of the remaining structures within the NS design, the subgoal trie ($ST_{NS}(P_i)$), answer trie ($AT_{NS}(P_{i.j})$) and subgoal frame (SF_{NS}) structures is equal to the size of the same structure type used in the original YapTab, the $ST_{YT}(P_i)$, $AT_{YT}(P_{i.j})$ and SF_{YT}, respectively.

When comparing the memory equations of the NS design and YapTab, the extra memory cost of the NS design to support concurrency is given by

$$\sum_{i=1}^{NP}[BAE + [NT - 1] * [ST_{NS}(P_i) + \sum_{j=1}^{NC(P_i)} [SF_{NS} + AT_{NS}(P_{i.j})]]].$$

This formula shows that the amount of memory spent by the design in ST, AT and SF is directly affected by the number of threads NT. Thus, for a particular $NT = 1$, the extra memory cost would be

$$NP * BAE.$$

The ST and AT structures can be considered to be the backbone of the table space, since often tabled evaluations can grow their size significantly and the Prolog system

spends most of its time in the table space on these two structures. This dependency over the NT value motivated us to create new designs that would decrease or remove this dependency, thus in the next two subsections we will be presenting two alternative designs that are aimed to be more efficient in the memory usage.

3.2.3 Subgoal-Sharing Design

In this second design, the threads share part of the table space. Figure 3.4 shows the configuration of a table space using the SS design on a table space with n predicates. As before, the TE structures of all predicates are chained. For the sake of simplicity, we are considering in the figure the configuration of the SS design for a particular predicate P_i and a particular subgoal call $P_{i.j}$. The ST structure is shared among the threads and the leaf data structures in each subgoal trie path, instead of referring to a SF, they point now to a BAE. Each thread t has its own cell T_t inside the bucket array which then points to private SF and AT structures.

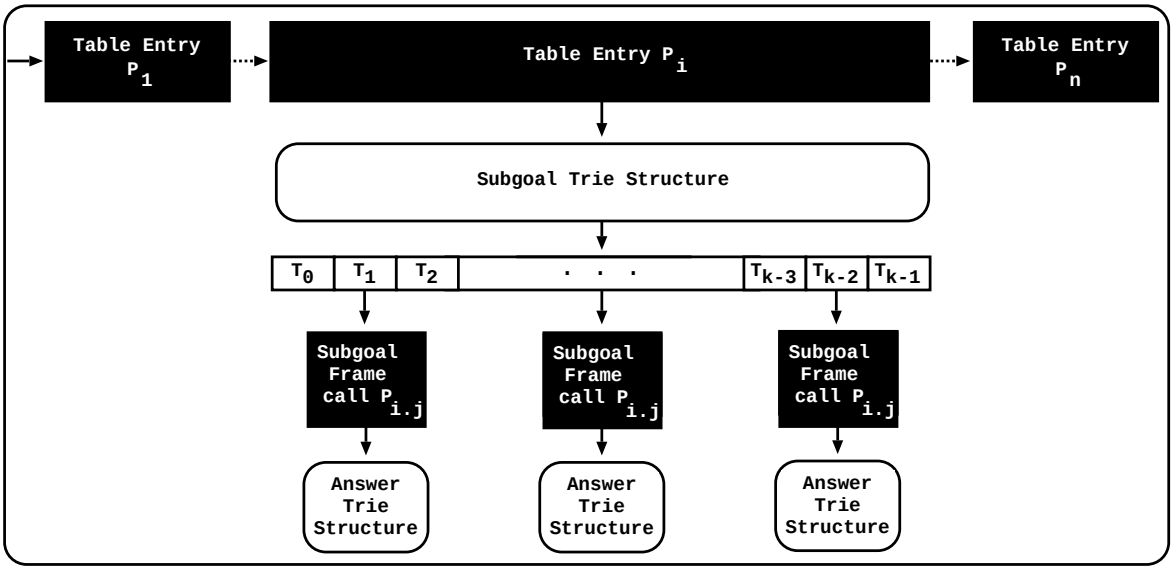

Figure 3.4: Table space organization for the SS design

In this design, concurrency among threads is restricted to the allocation of new entries on the ST structure. Whenever a thread finishes the execution, its private structures are removed, but the shared part remains present as it can be in use or be further used by other threads.

Given NT threads, Equation 3.3 shows the memory usage analysis of the predicate P_i using the SS design ($MU_{SS}(P_i)$) and the conditions that determine the size of every

structure in the design.

$$
\begin{aligned}
&MU_{SS}(P_i) = \\
&TE_{SS} + ST_{SS}(P_i) + \sum_{j=1}^{NC(P_i)} [BAE + NT * [SF_{SS} + AT_{SS}(P_{i.j})]] \\
&cond. \begin{cases} TE_{SS} = TE_{YT} \\ ST_{SS}(P_i) = ST_{YT}(P_i) \\ SF_{SS} = SF_{YT} \\ AT_{SS}(P_{i.j}) = AT_{YT}(P_{i.j}) \end{cases}
\end{aligned} \tag{3.3}
$$

The memory usage for the SS design is given by the sum of the size of table entry structure (TE_{SS}) with the size of the subgoal trie structure ($ST_{SS}(P_i)$) plus the summatory of the memory used in bucket array of entries (BAE) added with the multiplication of NT threads by the subgoal frame (SF_{NS}) and answer trie ($AT_{NS}(P_i)$) structures in the NC subgoal calls of the predicate P_i ($NC(P_i)$). Concerning the conditions that describe the size of the structures, Equation 3.3 shows that all structures in the SS design have the same size as the ones used in YapTab.

Lemma 3.2.1 shows the conditions where the SS design uses less memory than the NS design for an arbitrary number of threads $NT \geq 1$ and an arbitrary number of calls made to predicates, $NC(P_i) \geq 1$. To prove it, we begin by reducing both NS and SS designs to the canonical base, which is the YapTab with its structures, and then we proceed with the memory analysis to understand the structures that influence the behavior of the models. This analysis is useful to formalize the intuitive notions about the designs and whenever we want to know beforehand, which would be the best multithreaded design to be used on a particular tabled predicate.

Lemma 3.2.1. *If $NT \geq 1$ and $NC(P_i) \geq 1$ then the SS design uses less or equal memory than the NS design on a predicate P_i, i.e., $MU_{SS}(P_i) \leq MU_{NS}(P_i)$ if and only if the formula $[NC(P_i) - 1] * BAE \leq [NT - 1] * ST_{YT}(P_i)$ holds.*

Proof. The proof consists in two parts. On the first part we show the value of $MU_{SS}(P_i) - MU_{NS}(P_i)$, and then on the second part we use it to make the final statement of the proof.

Assume that all subgoal calls of the predicate P_i were completely evaluated using the SS and the NS designs.

For the SS design we have:

$MU_{SS}(P_i) =$

$$TE_{SS} + ST_{SS}(P_i) + \sum_{j=1}^{NC(P_i)} [BAE + NT * [SF_{SS} + AT_{SS}(P_{i.j})]] =_{SS\ cond.\ 1,2,3,4}$$

$$TE_{YT} + ST_{YT}(P_i) + NC(P_i) * BAE + NT * \sum_{j=1}^{NC(P_i)} [SF_{YT} + AT_{YT}(P_{i.j})]$$

For the NS design we have:

$MU_{NS}(P_i) =$

$$TE_{NS} + NT * ST_{NS}(P_i) + NT * \sum_{j=1}^{NC(P_i)} [SF_{NS} + AT_{NS}(P_{i.j})] =_{NS\ cond.\ 1,2,3,4}$$

$$TE_{YT} + BAE + NT * ST_{YT}(P_i) + NT * \sum_{j=1}^{NC(P_i)} [SF_{YT} + AT_{YT}(P_{i.j})]$$

The value of $MU_{SS}(P_i) - MU_{NS}(P_i)$ is given by:

$MU_{SS}(P_i) - MU_{NS}(P_i) =$

$$\underbrace{TE_{YT}}_{a} + ST_{YT} + NC(P_i) * BAE + \underbrace{NT * \sum_{j=1}^{NC(P_i)} [SF_{YT} + AT_{YT}(P_{i.j})]}_{b} - [\underbrace{TE_{YT}}_{-a} + BAE +$$

$$NT * ST_{YT}(P_i) + \underbrace{NT * \sum_{j=1}^{NC(P_i)} [SF_{YT} + AT_{YT}(P_{i.j})]]}_{-b} =$$

$$[NC(P_i) - 1] * BAE - [NT - 1] * ST_{YT}(P_i).$$

Now for the second and final part of the proof.

$MU_{SS}(P_i) \leq MU_{NS}(P_i) \Leftrightarrow$

$MU_{SS}(P_i) - MU_{NS}(P_i) < 0 \Leftrightarrow$

$[NC(P_i) - 1] * BAE - [NT - 1] * ST_{YT}(P_i) < 0 \Leftrightarrow$

$[NC(P_i) - 1] * BAE \leq [NT - 1] * ST_{YT}(P_i)$ □

Lemma 3.2.1 shows that the comparison between NS and the SS designs depends on the number of calls made to a predicate, the sizes of the BAE and ST structures, and on number of threads in evaluation. The NS design grows in the size of ST structures as we increase the number of threads in execution. The SS design grows in the size of BAE structures proportionally to the total number of the subgoal calls made to the predicate. The number of subgoal calls and the size of the ST structure is dependent

of the evaluation of the predicates, while the size of the BAE structures is fixed by the implementation provided by the YapTab-Mt and the number of threads is user-dependent. For the number of threads $NT = 1$ the following corollaries can be taken from the lemma:

Corollary 3.2.1. *If* $NT = 1$ *and* $NC(P_i) = 1$ *then* $MU_{SS}(P_i) = MU_{NS}(P_i)$.

Proof. From Lemma 3.2.1 we have:
$MU_{SS}(P_i) \leq MU_{NS}(P_i) \Leftrightarrow$

$[NC(P_i) - 1] * BAE \leq [NT - 1] * ST_{YT}(P_i) \Leftrightarrow_{NT=1,NC(P_i)=1}$

$[1 - 1] * BAE \leq [1 - 1] * ST_{YT}(P_i) \Leftrightarrow 0 = 0$

This means that $MU_{SS}(P_i) = MU_{NS}(P_i)$ must be true. □

Corollary 3.2.2. *If* $NT = 1$ *and* $NC(P_i) > 1$ *then* $MU_{SS}(P_i) > MU_{NS}(P_i)$.

Proof. From Lemma 3.2.1 we have:
$MU_{SS}(P_i) \leq MU_{NS}(P_i) \Leftrightarrow$

$[NC(P_i) - 1] * BAE \leq [NT - 1] * ST_{YT}(P_i) \Leftrightarrow_{NT=1}$

$[[NC(P_i) - 1] * BAE \leq 0$

Since $NC(P_i) > 1$ then $[[NC(P_i) - 1] * BAE > 0$, thus $MU_{SS}(P_i) > MU_{NS}(P_i)$. □

The conclusion for the comparison in the memory usage between the SS and the NS designs is that for one thread the SS is worst than or equal to the SS design. For a number of threads higher than one, the SS design performs better than the NS design when the formula presented in Lemma 3.2.1 is true. The best scenarios for the SS design are in predicates that have a small number of subgoal calls and ST structures that uses larger amounts of memory. In these scenarios the difference between both designs increases in proportion to the number of threads NT.

3.2.4 Full-Sharing Design

The FS design is the last design to be presented and is the most sophisticated among the three. Figure 3.5 shows the configuration of a table space using the FS design on a table space with n predicates. As before, the TE structures of all predicates are chained. For the sake of simplicity, we are considering in the figure the configuration

of the FS design for a particular predicate P_i and a particular subgoal call $P_{i.j}$. In this design, part of the subgoal frame information, the Subgoal Entry (SE) data structure, and the AT structure are now also shared among all threads. The previous SF data structure was split into two: the SE stores common information for the subgoal call (such as the pointer to the shared AT structure) and the BAE structure; the remaining information (the SF data structure) stores the private information about execution of the subgoal on each thread.

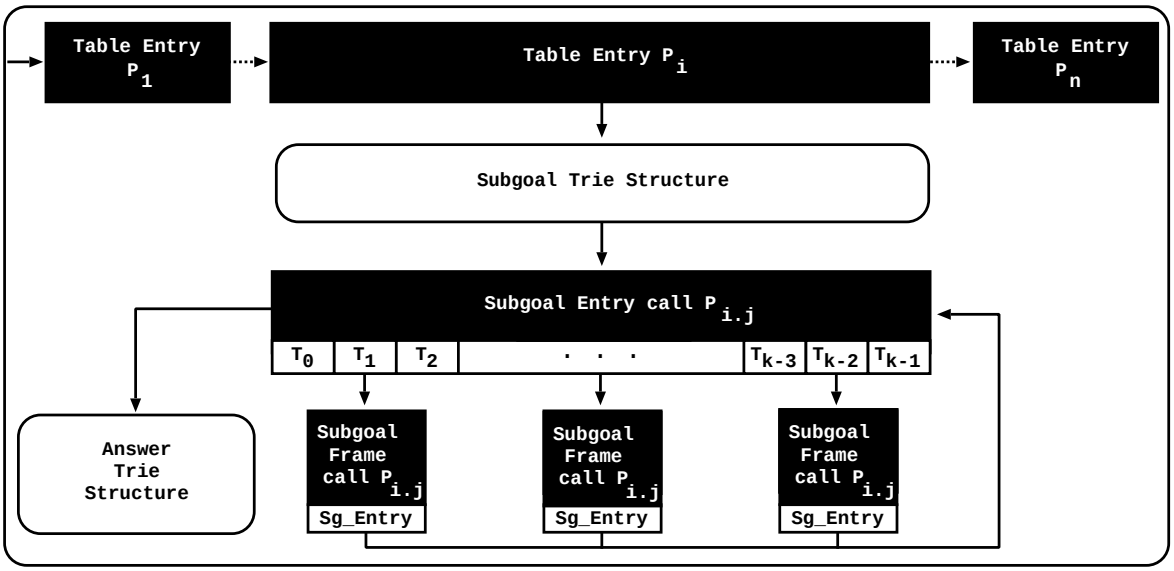

Figure 3.5: Table space organization for the FS design

The SE includes a BAE, in which each cell T_t points to the private subgoal frame of each thread t. The private subgoal frames include an extra field which is a back pointer to the common SE. This is important because, with that, we can keep unaltered all the tabling data structures that point to subgoal frames. To access the private information on the subgoal frames there is no extra cost (we still use a direct pointer), and only for the common information on the SE we pay the extra cost of following the indirect pointer. In this design, concurrency among threads includes then the access to the SE data structure and the allocation of new entries on the AT structures. However, this latest design has two major advantages. First, memory usage is reduced to a minimum. The only memory overhead, when compared with a single threaded evaluation, is the BAE associated with each subgoal entry, and apart from the split on the SF data structure, all the remaining structures remain unchanged. Second, since threads are sharing the same AT structures, answers inserted by a thread for a particular subgoal call are automatically made available to all other threads when they call the same subgoal.

Given NT threads, Equation 3.4 shows the memory usage analysis of the predicate P_i using the FS design ($MU_{FS}(P_i)$) and the conditions that determine the size of every structure in the design.

$$
\begin{aligned}
&MU_{FS}(P_i) = \\
&TE_{FS} + ST_{FS}(P_i) + \sum_{j=1}^{NC(P_i)} [SE_{FS} + BAE + NT * [SF_{FS} + BP_{FS}] + AT_{FS}(P_{i.j})] \\
&cond. \begin{cases} TE_{FS} = TE_{YT} \\ ST_{FS}(P_i) = ST_{YT}(P_i) \\ SE_{FS} + SF_{FS} = SF_{YT} \\ AT_{FS}(P_{i.j}) = AT_{YT}(P_{i.j}) \end{cases}
\end{aligned}
\tag{3.4}
$$

The memory usage for the FS design is given by the sum of the size of table entry structure (TE_{FS}) with the size of the subgoal trie structure ($ST_{FS}(P_i)$) plus the summatory of the memory used in the subgoal entry ($SE_{FS}(P_i)$) added with bucket array of entries (BAE), with the multiplication of the sum of the NT threads of the subgoal frame (SF_{FS}) and the back pointer size (BP_{FS}) and with the answer trie ($AT_{FS}(P_i)$) structures in the NC subgoal calls of the predicate P_i ($NC(P_i)$). Concerning the conditions that describe the size of the structures, Equation 3.4 shows that the size of the TE, ST and AT structures is the same as the YapTab original structures, and, the third condition shows that the sum of the size of the subgoal entry with the size of the subgoal frame is equal to the size of the subgoal frame structure used by the YapTab.

The FS is a refinement of the SS and next we use Lemma 3.2.2 to make a comparison in terms of memory usage between both designs. To prove it, we begin by reducing both FS and SS designs to the canonical base, which is the YapTab with its structures, and then we proceed with the memory analysis to understand the structures that most influences the behavior of the designs.

Lemma 3.2.2. *If $NT > 1$ and $NC(P_i) \geq 1$ then the FS design uses always less memory than the SS design on a predicate P_i, i.e., $MU_{FS}(P_i) < MU_{SS}(P_i)$ always holds.*

Proof. The proof consists in two parts. On the first part we show the value of $MU_{FS}(P_i) - MU_{SS}(P_i)$, and then on the second part we use it to make the final

statement of the proof.

Assume that all subgoal calls of the predicate P_i were completely evaluated using the FS and the SS designs.

For the FS design we have:
$MU_{FS}(P_i) =$
$TE_{FS}+ST_{FS}(P_i)+\sum_{j=1}^{NC(P_i)}[SE_{FS}+BAE+NT*[SF_{FS}+BP_{FS}]+AT_{FS}(P_{i.j})] =_{FS\ cond.\ 1,2,4}$

$TE_{YT}+ST_{YT}(P_i)+\sum_{j=1}^{NC(P_i)}[SE_{FS}+BAE+NT*[SF_{FS}+BP_{FS}]+AT_{YT}(P_{i.j})] =_{FS\ cond.\ 3}$

$TE_{YT}+ST_{YT}(P_i)+\sum_{j=1}^{NC(P_i)}[SF_{YT}-SF_{FS}+BAE+NT*[SF_{FS}+BP_{FS}]+AT_{YT}(P_{i.j})] =$

$TE_{YT}+ST_{YT}(P_i)+\sum_{j=1}^{NC(P_i)}[SF_{YT}+[NT-1]*SF_{FS}+BAE+NT*BP_{FS}+AT_{YT}(P_{i.j})] =$

$TE_{YT}+ST_{YT}(P_i)+NC(P_i)*BAE+NC(P_i)*NT*BP_{FS}+\sum_{j=1}^{NC(P_i)}[SF_{YT}+[NT-1]*SF_{FS}+AT_{YT}(P_{i.j})]$

For the SS design we have:
$MU_{SS}(P_i) =_{Lemma\ 3.2.2}$
$TE_{YT}+ST_{YT}(P_i)+NC(P_i)*BAE+NT*\sum_{j=1}^{NC(P_i)}[SF_{YT}+AT_{YT}(P_{i.j})]$

The value of $MU_{FS}(P_i)-MU_{SS}(P_i)$ is given by:
$MU_{FS}(P_i)-MU_{SS}(P_i) =$

$\underbrace{TE_{YT}+ST_{YT}(P_i)+NC(P_i)*BAE}_{a}+NC(P_i)*NT*BP_{FS}+\sum_{j=1}^{NC(P_i)}[SF_{YT}+[NT-1]*SF_{FS}+AT_{YT}(P_{i.j})]-[\underbrace{TE_{YT}+ST_{YT}(P_i)+NC(P_i)*BAE}_{-a}+NT*\sum_{j=1}^{NC(P_i)}[SF_{YT}+AT_{YT}(P_{i.j})]] =$

$NC(P_i)*NT*BP_{FS}+\sum_{j=1}^{NC(P_i)}[SF_{YT}+[NT-1]*SF_{FS}+AT_{YT}(P_{i.j})]-[NT*\sum_{j=1}^{NC(P_i)}[SF_{YT}+AT_{YT}(P_{i.j})]] =$

$NC(P_i)*NT*BP_{FS}+\sum_{j=1}^{NC(P_i)}[SF_{YT}+[NT-1]*SF_{FS}+AT_{YT}(P_{i.j})-NT*SF_{YT}-NT*AT_{YT}(P_{i.j})] =$

$$NC(P_i)*NT*BP_{FS}+\sum_{j=1}^{NC(P_i)}[[1-NT]*SF_{YT}+[NT-1]*SF_{FS}+[1-NT]*AT_{YT}(P_{i.j})] =$$

$$NC(P_i)*NT*BP_{FS}+NC(P_i)*[NT-1]*[SF_{FS}-SF_{YT}]-[NT-1]*\sum_{j=1}^{NC(P_i)}AT_{YT}(P_{i.j}) =$$

$$NC(P_i)*[[NT-1]*[SF_{FS}+BP_{FS}-SF_{YT}]+BP_{FS}]-[NT-1]*\sum_{j=1}^{NC(P_i)}AT_{YT}(P_{i.j})$$

Now for the second and final part of the proof.
$MU_{FS}(P_i) < MU_{SS}(P_i) \Leftrightarrow$

$MU_{FS}(P_i) - MU_{SS}(P_i) < 0 \Leftrightarrow$

$$NC(P_i)*[[NT-1]*[SF_{FS}+BP_{FS}-SF_{YT}]+BP_{FS}]-[NT-1]*\sum_{j=1}^{NC(P_i)}AT_{YT}(P_{i.j}) < 0 \Leftrightarrow$$

$$\underbrace{NC(P_i)*[[NT-1]*\underbrace{[SF_{FS}+BP_{FS}-SF_{YT}]}_{<0}+BP_{FS}]}_{<0} < \underbrace{[NT-1]*\sum_{j=1}^{NC(P_i)}AT_{YT}(P_{i.j})}_{>0}$$

□

Lemma 3.2.2 shows that the FS design uses always less memory than the SS design if the number of threads NT is higher than one. The proof has two parts, on the first part we prove the memory used by each design on each predicate and on the second part we compare both designs. Since the value of

$$[[NT-1]*[SF_{FS}+BP_{FS}-SF_{YT}]+BP_{FS}]$$

is always lower than zero and the value of $AT_{YT}(P_{i.j})$ is always positive (all tabled subgoal call have always an AT structure), the difference between both is always negative, which represents the fact that the memory used by the FS is always lower than the SS. The difference is multiplied by the number of subgoal calls and the number of threads. The difference between both designs occurs in two types of structures, on the subgoal frames and the answer tries. On the subgoal frames, the difference is that the size of the subgoal frames used by the FS design added with the back pointer is lower that the ones used by the SS design. For the answer trie structures, the FS design simply does not allocate as many of theses structures has the SS design. Remember from the previous subsection that the SS behavior was dependent on the amount of memory spent in BAE. The FS maintains this dependency, since this structure is co-

allocated inside the subgoal entry structure. For the number of threads $NT = 1$ the following corollary can be taken from the lemma:

Corollary 3.2.3. *If* $NT = 1$ *and* $NC(P_i) \geq 1$ *then* $MU_{FS}(P_i) > MU_{SS}(P_i)$.

Proof. From Lemma 3.2.2 we have:
$MU_{FS}(P_i) < MU_{SS}(P_i) \Leftrightarrow$

$$NC(P_i)*[[NT-1]*[SF_{FS}+BP_{FS}-SF_{YT}]+BP_{FS}] < [NT-1]* \sum_{j=1}^{NC(P_i)} AT_{YT}(P_{i.j}) \Leftrightarrow_{NT=1}$$

$$NC(P_i) * [[1-1] * [SF_{FS} + BP_{FS} - SF_{YT}] + BP_{FS}] < [1-1] * \sum_{j=1}^{NC(P_i)} AT_{YT}(P_{i.j}) \Leftrightarrow$$

$NC(P_i) * BP_{FS} < 0$, which is false and $NC(P_i) * BP_{FS} = 0$ is also false, because $NC(P_i) \geq 1$ and we know that $BP_{FS} > 0$. Thus, $MU_{FS}(P_i) < MU_{SS}(P_i)$ and $MU_{FS}(P_i) = MU_{SS}(P_i)$ can not be true, which means that $MU_{FS}(P_i) > MU_{SS}(P_i)$ must be true. □

The conclusion for the comparison in the memory usage between the FS and the SS designs is that, for one thread, the FS is always worst than the SS design and the difference increases in the proportion of the number of subgoal calls. For a number of threads higher than one, the FS design performs always better than the SS design (Lemma 3.2.2) and the difference increases as the number of threads NT and the number of subgoal calls increases.

3.3 Implementation Details

In this section, we discuss some low level details regarding the implementation of the three designs. We begin by describing the most important tabling operations and how they were extended for multithreaded tabling support. Next, we show how the BAE structure was implemented. We conclude with the discussion about the alternative locking schemes used to ensure mutual exclusion over the table space.

3.3.1 Tabling Operations

YapTab programs using tabling are compiled to include *tabling operations* that enables the tabling engine to properly schedule the evaluation process. In this subsection, we

revise the extensions involved in the two key tabling operations, in order to extend YapTab to YapTab-Mt.

We begin with the *tabled subgoal call* operation. As described in the previous chapter this operation inspects the table space looking for a subgoal similar to the current subgoal being called. If a similar subgoal is found, then the corresponding subgoal frame is returned. Otherwise, if no such subgoal exists, it inserts a new path into the subgoal trie structure, representing the current subgoal, and allocates a new subgoal frame as a leaf of the new inserted path. Algorithm 3.1 shows how we have extended the tabled subgoal call operation for multithreaded tabling support.

Algorithm 3.1 tabled_subgoal_call(table entry TE, subgoal call SC, thread id TI)

```
1:  root ← get_subgoal_trie_root_node(TE, TI)
2:  leaf ← subgoal_trie_check_insert(root, SC)
3:  if NS_design then
4:      sg_fr ← get_subgoal_frame(leaf)
5:      if sg_fr = Null then                          ▷ sg_fr does not exist
6:          sg_fr ← new_subgoal_frame(leaf)
7:      return sg_fr
8:  else if SS_design then
9:      bucket ← get_bucket_array(leaf)
10:     if bucket = Null then                         ▷ BAE does not exist
11:         bucket ← new_bucket_array(leaf)
12: else if FS_design then
13:     sg_entry ← get_subgoal_entry(leaf)
14:     if sg_entry = Null then                       ▷ SE does not exist
15:         sg_entry ← new_subgoal_entry(leaf)
16:     bucket ← get_bucket_array(sg_entry)
17: sg_fr ← get_subgoal_frame(bucket, TI)
18: if sg_fr = Null then                              ▷ sg_fr does not exist
19:     sg_fr ← new_subgoal_frame(bucket)
20: return sg_fr
```

Algorithm 3.1 receives three arguments: the table entry for the predicate at hand (TE), the current subgoal being called (SC), and the *id* of the working thread (TI). The *NS_design*, *SS_design* and *FS_design* macros define which table design is enabled.

The algorithm begins with the *get_subgoal_trie_root_node()* procedure, that receives as arguments a TE structure and a thread identifier TI (line 1). The aim of the procedure

is to return the root trie node for the subgoal trie structure that matches with the given thread identifier *TI*. Internally the procedure is quite simple. If the NS design is enabled, then it uses a *get_bucket_entry()* procedure to get the reference to the bucket of the thread with identifier TI within the BAE structure (we will give more details about this procedure in the next subsections). Then, if the bucket is empty, it creates and returns a root node for the bucket. Otherwise, the reference exists, and it simply returns the root node reference. Now, if the NS design is not enabled, then one of the other two designs is enabled (SS or FS). Since for both designs, only one root node is required, it creates a new root node or simply returns the node if it already exists.

At line 2, the *tabled_subgoal_call* operation calls the *subgoal_trie_check_insert()* procedure to check/insert a given *SC* into the subgoal trie structure whose path begins in the root node at hand. The procedure returns the leaf node for the path representing the *SC*. Internally, the *subgoal_trie_check_insert()* procedure calls a *trie_node_check_insert()* procedure to check/insert each token of the subgoal call *SC* within the subgoal trie structure.

In the continuation, if the NS design is enabled, the *tabled_subgoal_call* operation uses the leaf node to obtain the corresponding subgoal frame (line 4). If the subgoal call is new, no subgoal frame still exists and a new one is created (line 6). Then, the procedure ends by returning the subgoal frame (line 7). This code sequence corresponds to the standard tabled subgoal call operation.

Otherwise, for the SS design, the *tabled_subgoal_call* operation follows the leaf node to obtain the bucket array (line 9). If the subgoal call is new, no bucket exists and a new one is created (line 11). On the other hand, for the FS design, it follows the leaf node to obtain the subgoal entry (line 13) and, again, if the subgoal call is new, no subgoal entry exists and a new one is created (line 15). From the subgoal entry, it then obtains the bucket array (line 16).

Finally, for both SS and FS designs, the bucket array is then used to obtain the subgoal frame (line 17). To do so, it calls the *get_subgoal_frame()* procedure. The procedure receives a bucket array reference and a thread identifier TI, and returns a subgoal frame reference, if it exists. Internally, the procedure uses the *get_bucket_entry()* procedure to get the reference in the bucket for the thread with identifier TI. In the continuation, the *tabled_subgoal_call* operation checks if the subgoal frame exists (line 18) and, if the given subgoal call is new, a new subgoal frame needs to be created (line 19). The operation ends by returning the subgoal frame (line 20). Note that, for the sake of simplicity, we omitted some of the low level details in manipulating the

bucket arrays and internal manipulation of the trie. On the next subsections we will give more details about both features.

Another important tabling operation is the *tabled new answer*. This operation checks whether a newly found answer is already in the corresponding answer trie structure and, if not, it inserts it. Remember from Subsection 2.3.3 that, with local evaluation, the new answer operation always fails, regardless of the answer being new or repeated, and that, with batched evaluation, when new answers are inserted the evaluation should continue, failing otherwise. With the FS design, the answer trie structures are shared. Thus, since several threads can be inserting answers in the same trie structure, when an answer exists in the trie, it is not possible to determine if the answer is new or repeated for a certain thread. This is the reason why at this stage we will discuss the FS design only for local evaluation. On the next sections, we will show one technique to efficiently bypass this constraint and allows the YapTab-Mt system to support the FS design with batched scheduling. Algorithm 3.2 shows how we have extended the tabled new answer operation to support multithreading.

The operation receives two arguments: the new answer found during the evaluation (*ANS*) and the subgoal frame which corresponds to the call at hand (*SF*). The *NS_design*, *SS_design* and *FS_design* macros define again which table design is enabled.

The operation begins by checking/inserting the given *ANS* into the answer trie structure, which will return the leaf node for the path representing *ANS* (line 1). In line 2, it then tests whether the answer *ANS* already existed in the trie, i.e., if it was inserted or not by the current check/insert operation in line 1 and if the already answer existed, then it returns failure. Then, if one of the two NS or SS designs is enable (lines 5 to 10), it uses the leaf node to mark the answer as *found* and inserts the answer in the consumer chain, so that the answer can be consumed by the consumer nodes of the call, and returns accordingly to the current scheduling mode.

Otherwise, the FS design is enabled (lines 12 to 17), and the operation implements a critical region, for marking the answer as found and inserting it in to the consumer's chain. At the end (line 17), the operation simply fails (remember that at this stage we are only considering the local scheduling mode for the FS design).

3.3.2 Bucket Array of Entries

In the previous sections, we introduced the BAE structure. For the NS design, we included the BAE structure in the table entry (see Figure 3.3), for the SS design, the

Algorithm 3.2 tabled_new_answer(answer ANS, subgoal frame SF)

```
1: leaf ← check_insert_answer_trie(ANS, SF)
2: if is_answer_marked_as_found(leaf) = True then  ▷ the answer already exists
3:     return failure
   ▷ the answer is new
4: if NS_design or SS_design then
5:     mark_answer_as_found(leaf)
6:     consumer_chain_insert(leaf, SF)
7:     if local_scheduling_mode(SF) then
8:         return failure
9:     else                                        ▷ batched scheduling mode
10:        return proceed
11: else                                           ▷ FS design
12:    enter_critical_region(SF)                   ▷ critical region - begin
13:    if is_answer_marked_as_found(leaf) = False then
14:        mark_answer_as_found(leaf)
15:        consumer_chain_insert(leaf, SF)
16:    exit_critical_region(SF)                    ▷ critical region - end
17:    return failure                              ▷ local scheduling mode
```

BAE follows a subgoal trie path (see Figure 3.4), and for the FS design, the BAE is part of the new subgoal entry data structure (see Figure 3.5). The BAE structure is then a key structure that allows multiple threads to use concurrently the multithreaded table space. The BAE structure was presented as containing as much entry cells as the maximum number of threads (Yap's current version supports 1024 threads). However, in practice, this solution is highly inefficient and memory consuming, as we must always allocate this huge bucket array even when only few threads will be used. To solve this issue, we introduce a kind of *inode* pointer structure, where the bucket array is split into direct bucket cells and indirect bucket cells. The direct bucket cells are used as before and the indirect bucket cells are only allocated as needed. This new structure applies to all BAE structures in the three designs. Figure 3.6 shows an example on how this new structure is used in the FS design.

A BAE structure has now two operating modes. If it is being used by a thread with an identification number TI lower than a default starting size $DirectBuckets$ (32 in our implementation), then the buckets are used as before, meaning that the entry cell T_{TI} still points to the private information of the corresponding thread. But now, if a thread with an identification number equal or higher than $DirectBuckets$ appears, the thread

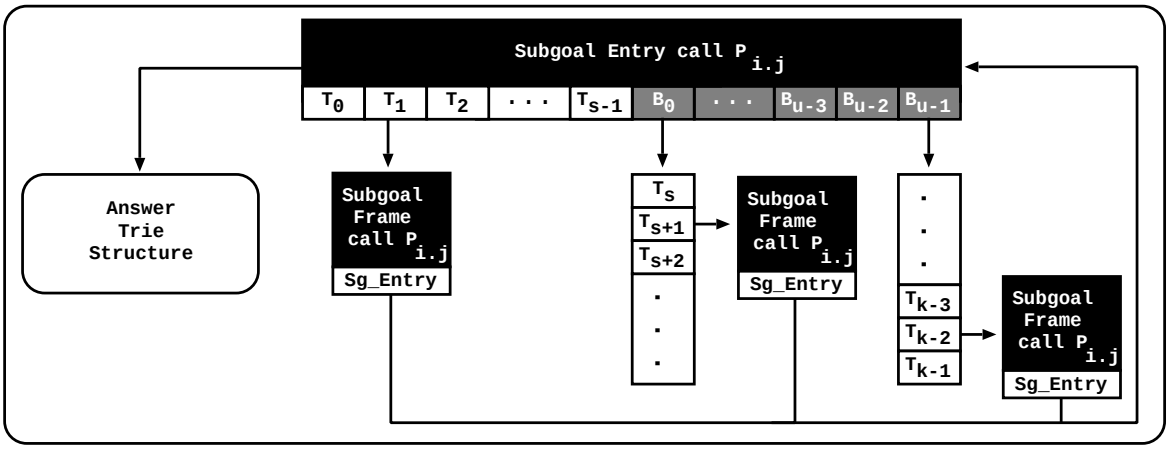

Figure 3.6: Using direct and indirect bucket cells in the FS design

is mapped into one of the u undirected buckets (entry cells B_0 until B_{u-1} marked with gray in Figure 3.6), which becomes a pointer to a second level bucket array that will now contain the entry cells referring to the private thread information. The second level buckets contains direct bucket entries. Given a thread TI ($TI \geq DirectBuckets$), its index in the first and in the second level bucket arrays is given by the division and the remainder of ($TI - DirectBuckets$) by direct buckets, respectively.

Algorithm 3.3 shows the pseudo-code for the check and insert procedure for the BAE structures using directed and undirected buckets. The procedure is used by the three multithreaded designs. For the moment, we are considering 32 direct buckets and 31 undirected buckets that expand to other BAE structures with 32 buckets, performing the total number of 1024 threads supported by Yap.

The procedure receives two arguments: the main bucket array of entries at hand (BAE) and the identifier of the working thread (TI). The procedures begins by checking whether the thread identifier TI is lower that the *DirectBuckets* threshold and if so, the procedures ends by returning the correspondent bucket (lines 1-2). Otherwise, the TI is in the undirect side, and the procedure follows by getting the undirect bucket position in the main BAE (line 3) and by getting the secondary BAE (line 4). If the secondary BAE does not exists (line 5), then the procedure creates a new secondary BAE (line 6) and enters in the critical region (line 7) of the main BAE structure, until line 13. While the procedure is in the critical region, it checks again if the undirect bucket is still empty and, if so, it updates the undirect bucket with the new secondary BAE (lines 8-9). Otherwise, the undirect bucket is no longer empty, meaning that other thread has inserted another secondary BAE in the meantime. In this case the procedure frees the new secondary BAE and gets the BAE that is in the undirect

Algorithm 3.3 get_bucket_entry(bucket array BAE, thread id TI)

```
1: if TI < DirectBuckets then
2:     return BAE + TI
3: UndirBkt ← BAE + DirectBuckets + (TI − DirectBuckets)/DirectBuckets
4: UndirBAE ← get_BAE(UndirBkt)
5: if UndirBAE = Null then                          ▷ bucket is empty
6:     UndirBAE ← NewBAE(DirectBuckets)
7:     enter_critical_region(BAE)                   ▷ critical region - begin
8:     if get_BAE(UndirBkt) = Null then             ▷ bucket still is empty
9:         set_BAE(UndirBkt) = UndirBAE
10:    else                                         ▷ failed to insert
11:        FreeBAE(UndirBAE)
12:        UndirBAE ← get_BAE(UndirBkt)
13:    exit_critical_region(BAE)                    ▷ critical region - end
14: return UndirBAE + (TI − DirectBuckets)%DirectBuckets
```

bucket (line 11-12). Finally, the procedure ends by returning the bucket that is in the secondary BAE.

In this subsection, we have discussed the implementation of BAE structures. Our purpose was to switch from a BAE structure without concurrent writes but using a huge amount of memory, to a BAE structure where part of the buckets are shared among threads. Maintaining a trade-off between both concurrency and memory usage is always a key to achieve an efficient concurrent system. On the next subsection, we discuss the table locking mechanisms that were implemented in the initial version of the YapTab-Mt.

3.3.3 Table Locking Schemes

Remember that the SS and FS designs introduce concurrency among threads when accessing shared resources of the table space. Here, we discuss how we use locking schemes to ensure mutual exclusion when manipulating such shared resources.

We can say that there are two critical issues that determine the efficiency of a locking scheme. One is the *lock duration*, that is, the amount of time a data structure is locked. The other is the *lock grain*, that is, the amount of data structures that are protected through a single lock request. It is the balance between lock duration and

lock grain that compromises the efficiency of different locking schemes.

The or-parallel tabling engine of Yap [109] already implemented four alternative locking schemes to deal with concurrent table accesses: the *Table Lock at Entry Level* (TLEL) scheme, the *Table Lock at Node Level* (TLNL) scheme, the *Table Lock at Write Level* (TLWL) scheme, and the Table Lock at Write Level - Allocate Before Check (TLWL-ABC) scheme. Currently, the first three are also available on our multithreaded engine. Figure 3.7(a) shows the TLEL scheme and Figure 3.7(b) shows the TLNL/TLWL schemes. The gray areas represent the areas that are locked by each lock. In the TLEL scheme, one can observe that the two lock fields (the *Lock L1* in the table entries and *Lock L2* in the subgoal frames) fully lock the complete access to the subgoal and answer trie structures, respectively. In the TLNL/TLWL schemes, the access to the subgoal and answer tries is locked per trie level and we use the parent trie nodes to lock the access to the list of sibling nodes. In Figure 3.7(b), *Lock L1* and *Lock L2* lock the sibling nodes for a common parent node.

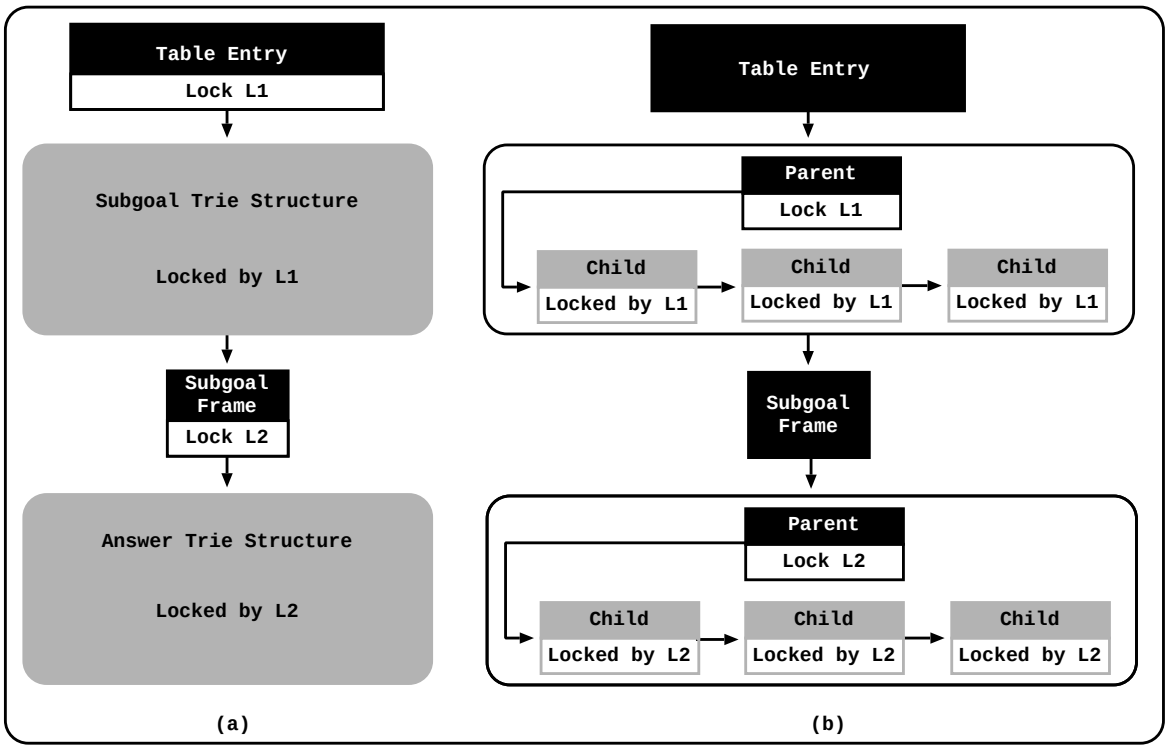

Figure 3.7: Table Locking Schemes : (a) TLEL vs (b) TLNL/TLWL

The TLNL/TLWL schemes allow a *single writer* per chain of sibling nodes that represent alternative paths from a common parent node. This means that each node in the subgoal/answer trie structures is expanded with a *locking field* that, once activated, synchronizes updates to the chain of sibling nodes, meaning that only one thread at a

time can be inserting a new child node starting from the same parent node.

In what follows, we will focus our attention on the TLWL locking scheme, since its performance showed to be clearly better than the other two [108]. With the TLWL scheme, the process of check/insert a token T in a chain of sibling nodes works as follows. Initially, the working thread starts by searching for T in the available child nodes (the non-critical region) and only if the token is not found, it will enter the critical region in order to insert it on the chain. At that point, it waits until the lock be available, which can cause a delay proportional to the number of threads that are accessing the same critical region at the same time.

In order to reduce the lock duration to a minimum, we have improved the original TLWL scheme to use *trylocks* instead of traditional locks. With trylocks, when a thread fails to get access to the lock, instead of waiting, it returns to the non-critical region, i.e., it traverses the newly inserted nodes, if any, checking if T was, in the meantime, inserted in the chain by another thread. If T is not found, the process repeats until the thread get access to the lock, in order to insert T, or until T be found. Algorithm 3.4 shows the pseudo-code for the implementation of this procedure using the TLWL scheme with trylocks.

Initially, the algorithm traverses the chain of sibling nodes, that represent alternative paths from the given parent node P, and checks for one representing the given token T. If such a node is found (line 6) then execution is stopped and the node returned (line 7). Otherwise, this process repeats (lines 3 to 10) until the working thread gets access to the lock field of the parent node P. In each round, the *last_child* auxiliary variable marks the last node to be checked. It is initially set to *Null* (line 1) and then updated, at the end of each round, to the new first child of the current round (line 9).

Whenever, the thread gets access to the lock, it enters the critical region (lines 11 to 19). Here, it first checks if T was, in the meantime, inserted in the chain by another thread (lines 12 to 16). If this is not the case, then a new trie node representing T is allocated (line 17) and inserted in the beginning of the chain (lines 18 and 19). The procedure then unlocks the parent node (line 20) and ends returning the newly allocated child node (line 21).

Another feature that we have implemented to improve the TLWL scheme was the usage of an external global array of locks that is shared among threads instead of using a lock field per trie node. The key idea is to save memory by reducing the size of the nodes inside the subgoal and answer tries, by removing the lock field from the nodes and pass it to an external fixed-size structure that is shared between all threads.

Algorithm 3.4 trie_node_check_insert(token T, parent trie node P)

```
1:  last_child ← Null                          ▷ used to mark the last child to be checked
2:  repeat                                                          ▷ non-critical region
3:      first_child ← TrNode_first_child(P)
4:      child ← first_child
5:      while child ≠ last_child do          ▷ traverse the chain of sibling nodes ...
6:          if TrNode_term(child) = T then                          ▷ ... searching for T
7:              return child
8:          child ← TrNode_sibling(child)
9:      last_child ← first_child
10: until (trylock(TrNode_lock(P)) = True)                  ▷ critical region, get lock
11: child ← TrNode_first_child(P)
12: while child ≠ last_child do              ▷ traverse the chain of sibling nodes ...
13:     if TrNode_entry(child) = T then                             ▷ ... searching for T
14:         unlock(TrNode_lock(P))                                  ▷ unlocking before return
15:         return child
16:     child ← TrNode_sibling(child)
17: child ← new_trie_node(T)                          ▷ create a new node to represent T
18: TrNode_sibling(child) ← TrNode_first_child(P)
19: TrNode_first_child(P) ← child
20: unlock(TrNode_lock(P))                                          ▷ unlocking before return
21: return child
```

Thus, whenever a thread wants to lock a particular level of the trie, it uses the value of the parent node of the trie level to feed a hash function that maps afterwards that value in to a bucket in the global array of locks. The bucket has then the lock to be used by the thread to lock the level, Figure 3.8 illustrates this idea. The correctness of usage of the global array of locks feature is ensured by the fact that the input value and the hash value is the same for all the threads that want to lock the same particular level of the trie. In our implementation, the size of the global array of locks is 512 buckets, which was the value that had the best performance results in terms of runtime. On the next subsections, we will give mode details about this performance analysis.

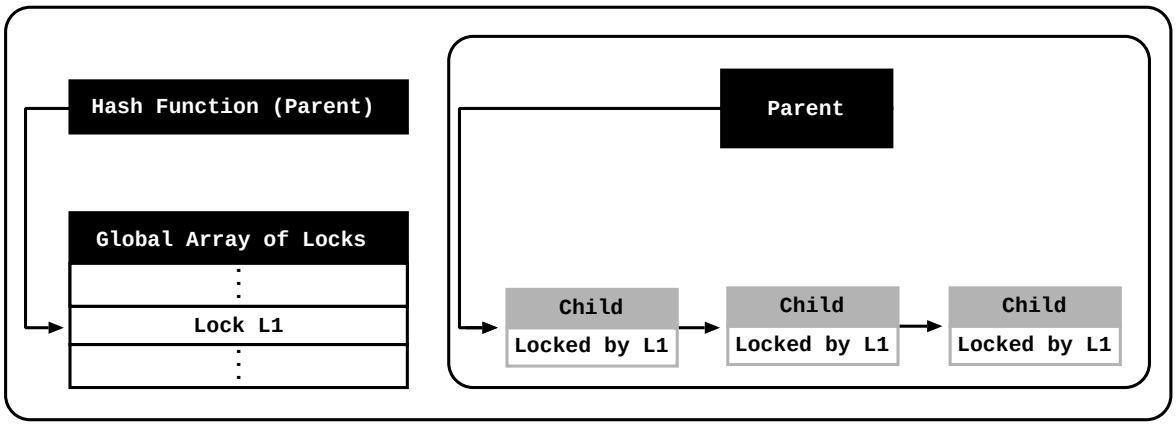

Figure 3.8: Combining a global array of locks with the TLWL scheme

3.4 Experimental Results

In this section, we present some experimental results obtained for the NS, SS and FS designs using the TLWL scheme with traditional locks, global locks and trylocks. The environment for our experiments was a machine with 32-Core AMD Opteron (TM) Processor 6274 (2 sockets with 16 cores each) with 32G of main memory, each processor with caches $L1$, $L2$ and $L3$ respectively with the sizes of 64K, 2048K and 6144K, running the Linux kernel is the 3.16.7-200.fc20.x86_64, with Yap 6.3 compiled with gcc 4.8.

3.4.1 Benchmark Programs

We used five sets of benchmarks. The *Large Joins* and *WordNet* sets were obtained from the OpenRuleBench project [73]; the *Model Checking* set includes three different specifications and transition relation graphs usually used in model checking applications; the *Path Left* and *Path Right* sets implement two recursive definitions of the well-known $path/2$ predicate, that computes the transitive closure in a graph, using several different configurations of $edge/2$ facts. Figure 3.9 shows an example for each configuration. We experimented the *BTree* configuration with depth 17, the *Pyramid* and *Cycle* configurations with depth 2000 and the *Grid* configuration with depth 35. All benchmarks find all the solutions for the problem.

In order to have a deeper insight on the behavior of each benchmark, and therefore clarify some of the results that are presented next, we first characterize the benchmarks. The columns in Table 3.1 have the following meaning:

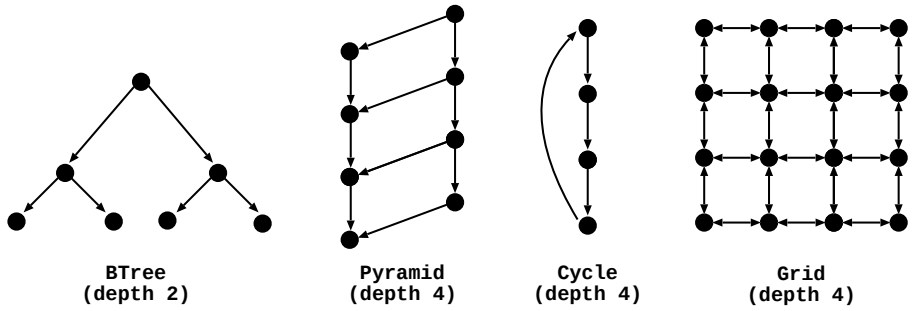

Figure 3.9: Edge configurations for the path benchmarks

- **calls:** is the number of different calls to tabled subgoals. It corresponds to the number of paths in the subgoal tries.
- **trie nodes:** is the total number of trie nodes allocated in the corresponding subgoal/answer trie structures.
- **trie depth:** is the minimum/average/maximum number of trie node levels required to represent a path in the corresponding subgoal/answer trie structures. Trie structures with smaller average values are more amenable to higher lock contention.
- **unique:** is the number of different tabled answers found. It corresponds to the number of paths in the answer tries.
- **repeated:** is the number of redundant tabled answers found. With the TLWL locking scheme, redundant answers do not lock the table space.
- **NS:** is the average execution time, in seconds, of ten runs for 1 thread with the NS design. In what follows, we will use these times as the base times when computing the overhead ratios for the other designs.

By observing Table 3.1, the *Mondial* benchmark, from the *Large Joins* set, and the three *Model Checking* benchmarks seem to be the benchmarks least amenable to lock contention since they are the ones that find less unique answers and that have the deepest trie structures. In this regard, the *Path Left* and *Path Right* sets correspond to the opposite case. They find a huge number of answers and have very shallow trie structures. On the other hand, the *WordNet* and *Path Right* sets have the benchmarks with the largest number of different subgoal calls, which can reduce the probability of lock contention because answers can be found for different subgoal calls and therefore be inserted with minimum overlap. On the opposite side are the *Join2* benchmark,

Table 3.1: Characteristics of the benchmark programs

Bench	Tabled Subgoals			Tabled Answers				Time (sec)
	calls	trie nodes	trie depth	unique	repeated	trie nodes	trie depth	NS
Large Joins								
Join2	1	6	5/5/5	2,476,099	0	2,613,660	5/5/5	2.85
Mondial	35	42	3/4/4	2,664	2,452,890	14,334	6/7/7	0.84
WordNet								
Clusters	117,659	235,319	2/2/2	166,877	161,853	284,536	1/1/1	0.83
Holo	117,657	235,315	2/2/2	74,838	54	192,495	1/1/1	0.75
Hyper	117,657	235,315	2/2/2	698,472	8,658	816,129	1/1/1	1.42
Hypo	117,657	117,659	2/2/2	698,472	20,341	816,129	1/1/1	1.53
Mero	117,657	117,659	2/2/2	74,838	13	192,495	1/1/1	0.74
Tropo	117,657	235,315	2/2/2	472	0	118,129	1/1/1	0.66
Model Checking								
IProto	1	6	5/5/5	134,361	385,423	1,554,896	4/51/67	2.70
Leader	1	5	4/4/4	1,728	574,786	41,788	15/80/97	3.51
Sieve	1	7	6/6/6	380	1,386,181	8,624	21/53/58	18.50
Path Left								
BTree	1	3	2/2/2	1,966,082	0	2,031,618	2/2/2	1.53
Cycle	1	3	2/2/2	4,000,000	2,000	4,002,001	2/2/2	3.52
Grid	1	3	2/2/2	1,500,625	4,335,135	1,501,851	2/2/2	1.93
Pyramid	1	3	2/2/2	3,374,250	1,124,250	3,377,250	2/2/2	3.08
Path Right								
BTree	131,071	262,143	2/2/2	3,801,094	0	3,997,700	1/2/2	2.33
Cycle	2,001	4,003	2/2/2	8,000,000	4,000	8,004,001	1/2/2	3.55
Grid	1,226	2,453	2/2/2	3,001,250	8,670,270	3,003,701	1/2/2	2.32
Pyramid	3,000	6,001	2/2/2	6,745,501	2,247,001	6,751,500	1/2/2	3.17

from the *Large Joins* set, and the *Path Left* benchmarks, which have only a single tabled subgoal call. On the next subsection we will present the performance analysis that we have done using these five sets of benchmarks.

3.4.2 Performance Analysis on Worst Case Scenarios

For the performance analysis, we used the three multithreaded tabling designs that were presented in the previous sections, the NS, the SS and the FS. To deal with the concurrency in the SS and FS designs, we have used the standard TLWL scheme and the modified version of the TLWL using *trylocks*, using *global locks* and using the combination of both. For the global locks strategies, we used a global array of 512 lock entries with a hash function that maps trie nodes to lock entries in the global array.

Note that for the moment our goal is to evaluate the robustness of our implementation when exposed to *worst case* scenarios. We will leave for later chapters the discussion that the system is scalable and able to speedup the execution of multithreaded tabled

programs. By focusing on *worst case* scenarios, we show the lowest bounds in terms of performance that each design might achieve when applied/used with other real world applications/programs. Moreover, by testing the framework with worst case scenarios, we avoid the peculiarities of the program at hand and we try to focus on measuring the real value of our designs.

Thus, we will follow a common approach to create worst case scenarios and we will run all threads starting with the same query goal. By doing this, it is expected that all threads will access the table space, to check/insert for subgoals and answers, at similar times, thus causing a huge stress on the same critical regions. To put the results in perspective, we experimented with 1, 8, 16, 24 and 32 threads (the maximum number of cores available in our machine) with local scheduling, for the combination of the multithreaded tabled designs with the *trylocks* and *global locks* on the five sets of benchmarks presented in the early subsection.

Table 3.2 shows the overhead ratios for the five sets of benchmarks, where each benchmark was executed ten times. The columns in the table have the following meaning: NS (NS design), *SS* (SS design without *global locks* and *trylocks*), SS_G (SS design with *global locks*), SS_T (SS design with *trylocks*), SS_{GT} (SS design with *global locks* and *trylocks*), *FS* (FS design without *global locks* and *trylocks*), FS_G (FS design with *global locks*), FS_T (FS design with *trylocks*) and FS_{GT} (FS design with *global locks* and *trylocks*). The rows in the table show the minimum (*Min*), the average (*Avg*), the maximum (*Max*), and the standard deviation (*StD*) overhead values when comparing with the NS design with one thread as presented in Table 3.1. The values marked with bold represent the best overhead (the lowest value) by row and by design. For example, for one thread the best maximum overhead with the SS design was 1.26, using the global locks scheme, while with the FS design was 1.49, using the combination of global locks with try locks.

In order to give a fair weight to each benchmark, the overhead ratio is calculated as follows. We begin by running ten times each benchmark B for each design D with T threads. Then, we calculate the average of those ten runs and use that value (D_{BT}) to put it in perspective against the base time, which is the average of the ten runs of the NS design with one thread (NS_{B1}). For that, we use the following formula for the overhead $O_{DBT} = D_{BT}/NS_{B1}$. After calculating all the overheads O_{DBT} for a certain design D and number of threads T corresponding to the several benchmarks B, we calculate the respective minimum, average, maximum and standard deviation overhead ratios.

Table 3.2: Overhead ratios, when compared with the NS design with 1 thread, for the NS, SS, SS_G, SS_T, SS_{GT}, FS, FS_G, FS_T and FS_{GT} designs, when running 1, 8, 16, 24 and 32 threads with local scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**NS**	**SS**				**FS**			
			SS	**SS_G**	**SS_T**	**SS_{GT}**	**FS**	**FS_G**	**FS_T**	**FS_{GT}**
1	**Min**	**1.00**	**0.99**	**0.99**	**0.99**	**0.99**	1.05	**1.01**	**1.01**	1.03
	Avg	**1.00**	1.11	**1.09**	**1.09**	**1.09**	1.39	**1.22**	1.38	1.24
	Max	**1.00**	1.40	**1.26**	1.42	1.35	1.73	1.56	1.75	**1.49**
	StD	0.00	0.14	0.10	0.14	0.12	0.17	0.16	0.18	0.14
8	**Min**	**1.07**	1.00	**0.99**	1.01	**0.99**	1.07	**1.02**	1.06	1.04
	Avg	**2.35**	2.50	**2.44**	2.46	2.53	3.58	**3.35**	3.68	3.43
	Max	**5.06**	5.37	**5.00**	5.23	5.11	7.12	6.50	7.49	**6.48**
	StD	1.23	1.29	1.23	1.28	1.28	1.81	1.68	1.93	1.67
16	**Min**	**1.02**	1.09	1.05	1.10	**1.01**	1.06	**1.02**	1.07	1.11
	Avg	**5.13**	**5.01**	5.03	5.06	5.14	4.48	4.29	4.46	**4.18**
	Max	**11.17**	**11.19**	11.31	11.50	11.43	9.30	8.23	9.32	**7.56**
	StD	3.12	3.11	3.17	3.14	3.20	2.43	2.23	2.40	2.02
24	**Min**	**1.24**	1.22	**1.08**	1.16	1.13	1.27	**1.22**	**1.22**	1.24
	Avg	**8.42**	8.02	**7.91**	8.19	8.08	5.13	4.96	5.18	**4.86**
	Max	**18.33**	18.50	**17.89**	19.01	18.38	10.56	9.30	10.33	**8.83**
	StD	5.24	5.31	5.27	5.37	5.33	2.69	2.43	2.62	2.19
32	**Min**	**1.33**	1.32	**1.18**	1.25	1.21	1.36	**1.34**	**1.34**	1.36
	Avg	**12.94**	11.43	**11.16**	12.07	12.05	5.88	**5.72**	6.46	5.92
	Max	**26.67**	25.96	**25.91**	25.97	26.24	12.32	10.87	11.92	**10.02**
	StD	7.52	7.98	7.72	7.50	8.06	3.15	2.83	3.04	2.59

By observing Table 3.2, we can see that, for one thread, on average, the SS and the FS designs perform worst than the NS design. For the SS design, we have an average between 1.09 and 1.11 and, for the FS the average is between 1.22 and 1.39. These overheads are a consequence of the extra complexity required to support concurrency, in particular, the cost incurred with the extra code necessary to implement the TLWL locking scheme that, even with a single working thread, has to be executed. For one thread, the usage of global locks shows lower overheads. The reason is that, with global locks, the number of lock fields is fixed to the size of the array, and thus not dependent on the number of nodes inside the tries. As the memory size of the trie

nodes is kept unaltered, the total amount of memory required do not increases as we allocate more trie nodes, which is not the case without global locks.

As we scale the number of threads, one can observe that, on average, the NS and SS designs show very poor results when compared with the FS. In particular, these bad results are more clear in the benchmarks that allocate a higher number of trie nodes. The explanation for this is the fact that we are using Linux's default memory allocator *malloc*, which can be a problem, when making a lot of memory requests, since these requests require synchronization at the low level implementation.

For the FS design, the results are significantly better and, in particular for FS_{GT}, the results show that the trylocks and global lock implementation is quite effective in reducing contention for 16 and 24 threads. For 32 threads, the global locks design alone (FS_G) is the best design. This can be explained by the fact that trylocks are known to have poor performances when the number of threads is equal or higher than the number of cores available in the hardware architecture. Thus, concerning the usage of trylocks and global locks, we can say that both of them have a positive impact in the designs and in some situations combining both of them shows to be the best option. But in the general picture, the global locks used solely seems to be the best option.

In summary, we can say that there are two main reasons for the good results of the FS design. The first, and most important, is that the FS design can effectively reduce the memory usage of the table space, almost linearly in the number of threads, which confirm the memory usage formulas introduced on Section 3.2. having the collateral effect of also reducing the impact of Yap's memory allocator. The second reason is that, since threads are sharing the same answer trie structures, answers inserted by a thread are automatically made available to all other threads when they call the same subgoal. We observed that this collateral effect can also lead to reductions on the execution time.

3.5 Chapter Summary

This chapter presented three new designs to multithreaded tabled evaluation of logic programs and their implementation on the YapTab-Mt framework. The chapter presented also several locking techniques that were aimed to improve the performance of the designs. The chapter concluded with a performance analysis of the designs in worst case scenarios.

Chapter 4

Concurrent Memory Allocation

This chapter describes TabMalloc, which is an efficient and scalable user-level memory allocator specially aimed for environments with the characteristics of multithreaded tabled evaluation of logic programs. TabMalloc is the current default YapTab-Mt's memory allocator.

4.1 Introduction

After the initial implementation of the YapTab-Mt system, we used the profiling tools Intel VTune [60], Valgrind [86] and OProfile [72], in order to better understand the performance results initially obtained. We observed that there is still considerable space for improvements in the concurrent memory allocation of our YapTab-Mt. In this chapter, we will present a new memory allocator, whose key idea is to implement strategies that pre-allocate bunches of memory in order to minimize the performance degradation that the YapTab-Mt framework showed, when exposed to simultaneous memory requests made by multiple threads.

Using the profiling tools, we detected some problems related to Yap's memory allocator, mainly, when running programs that allocated a higher number of data structures in the table space. Yap's memory allocator is based on the operating system's default memory allocator, which can be a problem when making a lot of memory requests, since such requests may require synchronization at the low-level implementation.

TabMalloc memory allocator was designed aiming to be a more efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs [6]. It is

based on local and global pages, to split memory among specific data structures and different threads, together with a strategy where data structures of the same type are pre-allocated within a page, so the goal is to minimize the performance degradation that YapTab-Mt suffers when it is exposed to simultaneous memory requests made by multiple threads.

In order to avoid memory contention, TabMalloc follows the general approach of the current state-of-the-art user-level memory allocators, such as PtMalloc [46], Hoard [20], TcMalloc [44] and JeMalloc [38], but instead of using thread caches, local and global heaps with different block sizes, we use proper *local* and *global* pages, to split memory among specific data structures and different threads, together with a kind of *slab allocation* [22] mechanism where tabled data structures of the same type are pre-allocated within a page. When a page P is made local to a thread T, this means that T has exclusive permission to allocate and deallocate data structures from P. On the other hand, global pages have no owners and, thus, they are free from allocate/deallocate operations. In both cases, all threads can access (for read or write operations) the data structures on local or global pages. This is very important since it allows to significantly reduce memory contention without introducing any overhead for multithreaded tabled evaluation.

Experimental results showed that TabMalloc can effectively reduce the execution time and scale better, when increasing the number of threads, than the original allocator [6]. Due to the good performance shown, TabMalloc became the current default memory allocator of the YapTab-Mt system for multithreaded tabled programs using the NS, SS or FS designs. We describe next its most important key ideas and implementation details.

4.2 Related Work

The performance of User-level Memory Allocator (UMA) can be crucial and can limit the application. Many UMA subsystems were written in a time when multiprocessor systems were rare. They use memory efficiently but are highly serial and constitute an obstacle to throughput for parallel applications. Evidence of the importance of UMA comes from the wide array of aftermarket UMA replacement packages that are currently available. Thus, the efficient usage of memory has an important impact in the development of complex frameworks such has the YapTab-Mt, because it requires the multiple *allocation* and *deallocation* of different sized chunks of memory. In a

conceptual level, there are two categories of memory managers: the *kernel level* and the *user level* memory managers. The kernel level memory managers are responsible for managing memory inside the protected sub-systems/resources of the OS while the user level memory managers are responsible for managing memory inside the applications. TabMalloc fits in the second category, i.e., TabMalloc is a UMA.

4.2.1 UMA Memory Management

The main goal of a UMA is to manage the *heap* area, which is an area that is inside the addressing space of each process where the dynamic allocation of memory is directly done. In a multithreaded environment, all threads share the same heap, thus the allocation and deallocation of objects on this area of memory must be executed in a concurrent fashion. UMA implementations exist in a wide spectrum. At one extreme we find a single *global heap* protected by one mutex. The default UMA in the Solaris Operating Environment is of this design. This type of allocator can organize the heap with little wastage, but operations on the heap are serialized, so this type of design might not scale well. At the other extreme, a UMA can provide a *private heap* for each thread. UMA operations that can be satisfied from a thread's private heap do not require synchronization and have low latency. When the number of threads grows large, however, the amount of memory reserved in per-thread heaps can become unreasonable. Various solutions have been suggested, such as adaptive heap sizing or trying to minimize the number of heaps in circulation by handing off heaps between threads as they block and unblock [37].

UMA can be seen as an interface between a process and an OS. Different UMAs have different interactions with an OS, but in a nutshell the activity of a UMA begins upon the creation of a process within a OS. At this stage, the UMA connected to the process sends a request to the OS, asking for an area of memory. After the memory request be satisfied by the OS, the UMA creates and initializes a header for the heap's area [1]. Typically, a header is a structure that has meta-information about the area of memory, such as for example the size of the blocks that are within the area of memory, or the number of blocks that are available in the heap. Now, when a memory allocation request is done within the process, the UMA satisfies the request using the memory in the heap. If the memory in the heap is not sufficient, then the UMA does a memory request for the OS and satisfies the memory allocation request within the process,

[1] As described in the previous paragraph, a UMA might have more than one heap. In this cases it creates the headers for all the heaps that is using.

increasing this way the amount of dynamic memory that can be allocated. Upon a request for the deallocation of chunk memory within a process, depending again of the UMA's approach, it might choose between the integration of the chunk of memory in its heap (to satisfy other memory allocation requests) or free immediately the chunk of memory to the OS.

The process of managing the heap is important to address problems in the area of memory management. An important optimization goal of a good UMA is to minimize *fragmentation*, i.e., minimize the amount of free memory that cannot be used (allocated) by the process. Fragmentation is classified as either *internal* or *external*. Internal fragmentation is free memory wasted when the allocator gives to the process a larger memory block than the process requested. External fragmentation is free memory that have been split into non-contiguous blocks too small to be used to satisfy the requests from the process. Moreover, multithreaded programs add more complications to the UMA. Obviously some kind of synchronization has to be added to protect the heap during concurrent requests. There are also other problems which have significant impact on application performance when the application is run on a multiprocessor, such as *heap blowup*, *false sharing* or *memory contention* [20, 19, 79, 45].

The heap blowup problem consists in an overconsumption of memory by a process. This occurs in situations where the memory in use is not being recycled or exists chucks of memory that were requested to the OS but are simply not being used. When the process has multiple threads, if the memory allocator fails to make memory deallocated by threads running on one processor available to threads running on other processors the consumption of memory can blowup. A typical source of heap blowup is a process that has producer and consumer threads, where the producers allocate memory and pass it to the consumers which in turn free the memory. If the memory blocks freed by the consumers are not made available to the producers the heap blowup problem can occur.

The false sharing problem occurs when different parts of the same cache-line end up being used by threads running on different processors. This will put a potentially large and completely unnecessary load on the cache-coherence mechanism. False-sharing can never be avoided completely since application threads may pass allocated memory between themselves but a memory allocator should avoid to actively induce false-sharing by satisfying memory requests from different processors with memory from the same cache-line [45]. A practical example of the false sharing problem would be two threads running in two different processors. A thread reads a memory position and another thread writes to a memory position that shares the same line of cache.

In this case, the first thread would be forbidden to access the corresponding chunk of memory until the cache line be completely updated. This is a practical example using just two threads, but one can easily understand that this might become a huge problem in environments with an arbitrary number of threads.

Finally, the memory contention problem occurs whenever multiple threads using the same heap to allocate or deallocate memory, thus requiring some sort of synchronization mechanism. If one thread accesses the heap then the remaining threads would be forbidden to access the heap due to mutual execution, because no more than one request can be done simultaneously in the same heap. Thus, a UMA has to ensure *efficiency* and *scalability*. For a memory allocator to be scalable, its performance has to scale well with the number of processors, threads and the load in the system. In terms of speed, the concurrent memory allocator should be about as fast as a good sequential one in order to ensure good performance even when a multithreaded program is executed on a single processor [45].

4.2.2 Concurrent Memory Allocators

In this section, we describe some of the state-of-the-art concurrent user-level memory allocators that were the base for our proposal.

The PtMalloc [46] memory allocator, developed by Wolfram Gloger and based on Doug Lea's *dlmalloc* sequential allocator [68], is used in most modern distributions of Linux that use *glibc*. Lea's memory allocator had several goals, including improving portability, space and time utilization, and adding tunable parameters to control allocation behavior. Gloger's update to Lea's original allocator retains these desirable behaviors, and adds the multithreading ability. PtMalloc memory allocator uses *arenas* with different bins for small and large objects requested by the threads. All arenas are shared by all threads. The allocation and deallocation of objects is always done inside the arenas. Whenever a thread needs to allocate memory and all the arenas are completely full, then a new arena is created and it becomes immediately available to all threads. The current version 3 of Ptmalloc improves the previous version, mainly because it adopts a different method to meet memory requests for larger blocks, by keeping small bins in a linked list and the large bins in a binary tree, thus that the search for a large bin can run in a logarithmic time using the binary tree.

The Hoard [20] memory allocator, developed by Emery Berger, uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks, and

each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread identifiers to decide which processor heap to use for a memory request. When a thread frees a memory block, it returns the block to its original superblock and updates the fullness statistics for the superblock as well as the heap that owns it. Typically, allocating and deallocating memory requires one and two lock acquisitions, respectively.

The TcMalloc [44] memory allocator, developed by Google, uses a thread cache for small objects and a global heap for larger objects. The requests for small objects within the thread cache are done without synchronization, while requests for larger objects are done using fine grained spinlocks. A key feature of the system is to allow the threads to execute their own garbage collection over their thread cache structures. The garbage collection operation is activated by a thread whenever its thread cache reaches an *adjustable threshold*, i.e., to face with different memory demands by each thread, the threshold of the garbage collection of each thread is adjustable by itself.

Finally, the JeMalloc [38] memory allocator, developed by Jason is used in many well known applications (for example, FreeBSD, Firefox and Facebook), has a thread cache for small objects and arenas with different bins for small and large objects. Also it uses bin locking for small objects and arena locking for larger objects. The allocation and deallocation of objects is also done inside the same arena. A key feature of the system, is that it uses *red-black trees* to improve the execution time on the allocation and deallocation of objects.

In general we can resume, some of the common characteristics that memory allocators use to address the heap blowup, false sharing and memory contention problems, as follows:

- Separate handling of thread-local allocations. It is advantageous to distinguish between thread-local allocations and allocations of memory that is to be shared between threads. In particular, the thread-local memory allocator might not need any synchronization.

- Avoid contention and false sharing through the usage of private and shared heaps or other structures such as arenas.

- Avoid to actively inducing false-sharing by satisfying memory requests from different processors with memory from the same cache-line.
- Use chunks of memory which are multiples of the cache line size.
- Avoid heap blowup through the migration of chunks of memory between heaps.

4.3 Our Proposal

In this section, we show the details of the TabMalloc memory allocator and how we have integrated it in the YapTab-Mt system. Our TabMalloc [6] proposal has local and global page heaps per object type. In addition it uses global and local void heaps for the allocation of objects when the local heaps run empty. Each global heap has its own locking mechanism and the deallocation of shared objects is done on global page heaps. TabMalloc takes advantage of running inside the YapTab-Mt engine, i.e., the allocation and deallocation of objects is always done via local page heaps, except for the main thread that performs garbage collection on the global page heaps.

4.3.1 Key Ideas

Modern computer architectures use *pages* to handle memory. Pages are fixed size blocks of contiguous memory cells. Based on this characteristic, we adopted an allocation scheme based also on pages, where each memory page only contains data structures of the same type. In order to split memory among different threads, in our approach, a page can be considered a *local page*, if owned by a particular thread, or a *global page*, otherwise. Figure 4.1 gives an overview of the new memory allocator based on pages.

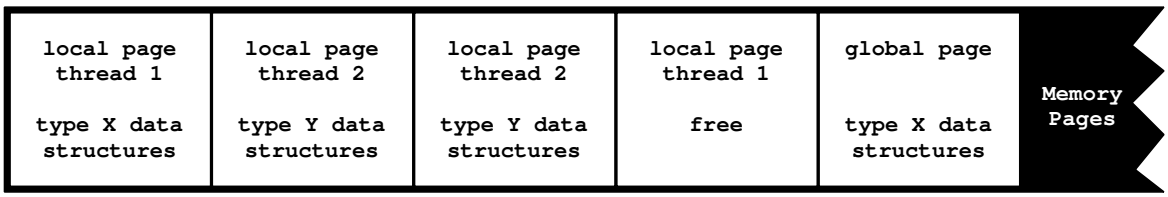

Figure 4.1: Using pages as the basis for the new memory allocator

A thread can own any number of pages of the same type, of different types and/or free pages. Any type of page (including free pages) can be local to a thread or global, and

each particular page only contains data structures of the same type. When a page P is made local to a thread T, this means that T has exclusive permission to allocate and deallocate data structures from P. On the other hand, global pages have no owners and, thus, they are free from allocate/deallocate operations. To allocate/deallocate data structures on global pages, first the corresponding pages should be moved to a particular thread. All running threads can access (for read or write operations) the data structures allocated on a page, independently of being a local or global page.

Access to the chain of available pages for a given data type is synchronized by a *page entry* data structure. For each different data type, there is a *global page entry* and a *local page entry* per thread. For example, for the subgoal frames, there is a *GB_PG_sg_fr* global page entry and a *LC_PG_sg_fr* local page entry per thread. Access to free pages (i.e., pages with all data structures unused) is also synchronized by proper global/local page entries, named *GB_PG_void* and *LC_PG_void*, respectively. Full pages (i.e., pages with all data structures in use) are not accessed from any local or global page entry. A page entry data structure includes a *PgEnt_first* and a *PgEnt_last* field that point, respectively, to the first and last page in the chain of pages. For the global pages, an extra *PgEnt_lock* field implements a lock mechanism that synchronizes access to the respective chain of pages.

The management of pages and data structures within pages is achieved by allocating a special *page header* structure at the beginning of each page and by uniformly dividing the remaining space in equal-size data structures of the data type being handled. Figure 4.2 shows an example that better illustrates how page entries and page headers work together. A page header consists of four fields. The *PgHd_next* and *PgHd_prev* fields point, respectively, to the next and previous pages in the chain of pages. The *PgHd_strs_in_use* field stores the number of data structures in use within a page. When it reaches zero the page is freed and moved to the *LC_PG_void* page entry of the thread at hand. The *PgHd_first* field points to the first unused data structure within a page and the remaining unused data structures are linked through their *next* fields. When all data structures are in use (i.e., when a page is full and *PgHd_first* is *Null*), the page is simply released from the respective chain.

Allocating and freeing data structures are constant-time operations, all we have to do is to move a structure to or from a list of free structures. Whenever a thread T requests to allocate memory for a data structure of type S, it can instantly satisfy the request by returning the first unused slot on the first available local page with type S. If there are no available local pages with type S, then a new page must be requested. If there are free local pages in *LC_PG_void*, then the first one is made to be of type S.

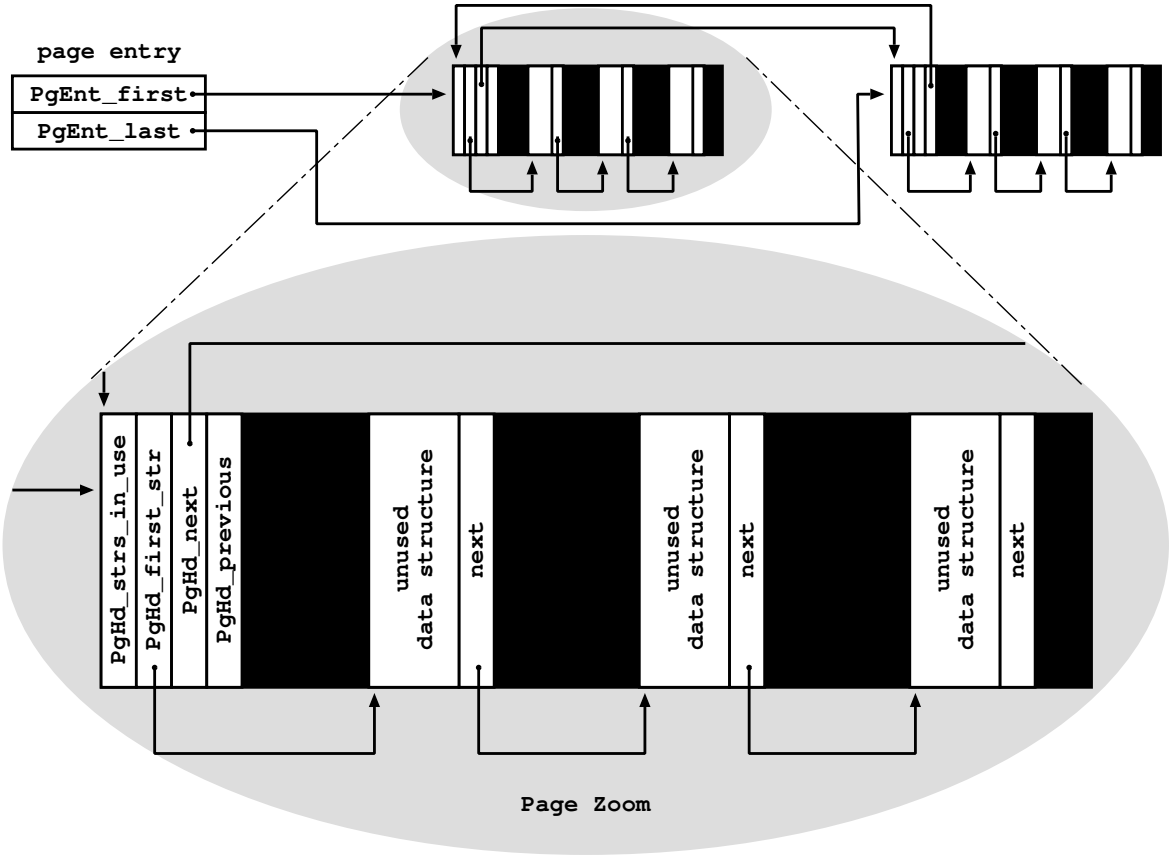

Figure 4.2: Page entries and page headers in the new memory allocator

Otherwise, thread T must synchronize with the other threads in order to access the shared resources. Then, it first tries the *GB_PG_void* chain of free global pages and, if no free page exists there, it asks for new memory pages from the operating system's memory allocator (such pages are then chained in the *GB_PG_void* page entry).

Deallocation of a data structure of type S does not free up the memory, but only opens an unused slot on the chain of available local pages for type S. Further requests to allocate memory of type S will later return the now unused memory slot. When all data structures in a page are unused, the page is moved to the chain of free local pages. A free local page can be reassigned later to a different data type. This process eliminates the need to search for suitable memory space and greatly alleviates memory fragmentation. The only wasted space is the unused portion at the end of a page when it cannot fit exactly with the size of the corresponding data structures.

When a thread finishes execution, it deallocates all its private data structures and then moves its local pages to the corresponding global page entries. Shared structures are only deallocated when the last running thread (usually thread 0) abolish the tables.

Thus, if a thread T allocates a data structure D, then it will be also responsible for deallocating D, if D is private to T, or D will remain live in the tables, if D is shared, even when T finish execution. In the latter case, D can be only deallocated by the last running thread L. In such case, D is made to be local to L and the deallocation process follows as usual.

In general, TabMalloc follows the common characteristics of the other memory allocators to address the heap blowup, false sharing and memory contention problems. It separates local and shared memory allocation, and uses local and global heaps with pages that are formatted in blocks with the sizes of the structures that are used by the YapTab-Mt. The page formatting ensures also that TabMalloc avoids to actively inducing false-sharing, because different threads in different processors do not share the same cache line and the heap blowup problem is avoided through the migration of pages between local and global heaps.

4.3.2 Implementation Details

In this section, we present in more detail the algorithms that implement the key aspects of the new memory allocator.

Algorithm 4.1 shows the pseudo-code for allocating a new data structure given the corresponding local page entry PE. Initially, it checks for available pages and, if no page exists, a new one is requested through a call to the *alloc_page()* procedure (lines 1–3). Next, it increases the number of structures in use in the page (line 4) and gets the first unused structure from the page obtained and updates the page header to point to the next unused structure (lines 5–6). If no more unused structures exist then the page is full and the page entry at hand is updated to point to the next available page (lines 8–12).

Algorithm 4.2 shows the pseudo-code for the *alloc_page()* procedure. Initially, the procedure checks for free local pages (lines 1–2). If there is at least one such page, it updates the chain of free local pages (lines 3–5) and returns it. Otherwise, it locks the free global pages and tries to get a page from there (lines 7–15) and, if no free page exists, it asks the operating system for new memory pages (procedure *alloc_init_new_pages_from_OS()*).

Algorithm 4.3 shows the pseudo-code for the *free_struct()* procedure given a data structure DS and the corresponding local page entry PE. Initially, it determines the corresponding page pg for DS (line 1) and checks if pg contains other structures in

Algorithm 4.1 alloc_struct(local page entry PE)

```
1: pg ← PgEnt_first(PE)
2: if pg = Null then                                   ▷ no available pages
3:     pg ← alloc_page()
4: PgHd_strs_in_use(pg) + +
5: str ← PgHd_first(pg)
6: PgHd_first(pg) ← struct_next(str)
7: if PgHd_first(pg) = Null then                       ▷ page is full
8:     if PgHd_next(pg) = Null then
9:         PgEnt_last(PE) ← Null
10:    else
11:        PgHd_prev(PgHd_next(pg)) ← Null
12:    PgEnt_first(PE) ← PgHd_next(pg)
13: return str
```

use (lines 2–3). If so, DS is chained in the list of unused structures within the page (lines 10–11), and if DS is the first structure being made available, then *pg* is also inserted in the chain of available pages of that type (lines 4–9). Otherwise, if *pg* does not contain other structures in use, the page stops being of the current type and is moved to the chain of free local pages (lines 13–25). The *free_page()* procedure inserts a page into the chain of available free pages.

4.4 Performance Analysis on Worst Case Scenarios

We now present experimental results about the usage of the TabMalloc memory allocator on the NS, SS and FS designs. For the sake of simplicity, for the designs SS and FS, we will be presenting only the results for the designs with global locks (SS_G and FS_G), since they were the ones that presented the lowest overheads in the previous chapter. Yet, the reader can keep the idea that TabMalloc affects all designs presented in the previous chapter similarly. Concerning the benchmarks, we will be using the same five sets of benchmarks presented also in Subsection 3.4.1 with the same number of runs per benchmark and the same formula to calculate the metric of overhead ratios. Thus, we will be using again the worst case scenarios, where all threads start with the same query goal. By doing this, it is expected that all threads will access the table space, to check/insert for subgoals and answers, at similar times, thus causing a huge stress on the same critical regions. To put the results in perspective, we experimented

Algorithm 4.2 alloc_page()

```
1: pg ← PgEnt_first(LC_PG_void)
2: if pg ≠ Null then
3:     PgEnt_first(LC_PG_void) ← PgHd_next(pg)
4:     if PgEnt_last(LC_PG_void) = pg then
5:         PgEnt_last(LC_PG_void) ← Null
6: else                                          ▷ no free local pages
7:     lock(PgEnt_lock(GB_PG_void))
8:     pg = PgEnt_first(GB_PG_void)
9:     if pg = Null then                         ▷ no free global pages
10:        alloc_init_new_pages_from_OS()
11:        pg = PgEnt_first(GB_PG_void)
12:    PgEnt_first(GB_PG_void) ← PgHd_next(pg)
13:    if PgEnt_last(GB_PG_void) = pg then
14:        PgEnt_last(GB_PG_void) ← Null
15:    unlock(PgEnt_lock(GB_PG_void))
16: return pg
```

with 1, 8, 16, 24 and 32 threads (the maximum number of cores available in our machine) with local scheduling.

Table 4.1 shows the overhead ratios for the NS design alone and combined with TabMalloc for the five sets of benchmarks using underneath, at the OS level, the PtMalloc 3 (column PtMa), Hoard 3.10 (column Hoard), TcMalloc 4.2 (column TcMa) and JeMalloc 3.6 (column JeMa) memory allocators. PtMalloc 3 is the default memory allocator and is already installed in the *glib* library of the OS, thus no special procedure was used for enable it. In this regard, please observe that the first column of the table (PtMa with the NS design) is equal to the first column of Table 3.2 presented in the previous chapter, which means that the results presented on the previous chapter were already obtained using PtMalloc.

For the Hoard, TcMalloc and JeMalloc, first we have downloaded them from the respective sites [18, 39, 44], then we compiled them, using the gnu compiler gcc version 4.8 in the machine where we executed the benchmarks and finally, when running each benchmark, we used the *LD_PRELOAD* instruction to use each one instead of the default memory allocator. The rows in the table show the minimum (*Min*), the average (*Avg*), the maximum (*Max*), and the standard deviation (*StD*) overhead values when comparing with the NS design with one thread as presented in Table 3.1. Again, the

Algorithm 4.3 free_struct(data structure DS, local page entry PE)

```
1: pg ← get_page(DS)
2: PgHd_strs_in_use(pg) − −
3: if PgHd_strs_in_use(pg) ≠ 0 then
4:     if PgHd_first(pg) = Null then                    ▷ first unused struct
5:         PgHd_next(pg) ← Null
6:         PgHd_prev(pg) = PgEnt_last(PE)
7:         if PgHd_prev(pg) ≠ Null then
8:             PgHd_next(PgHd_prev(pg)) ← pg
9:         PgEnt_last(PE) ← pg
10:    struct_next(DS) ← PgHd_first(pg)
11:    PgHd_first(pg) ← DS
12: else                                                ▷ no other structures in use
13:    if PgHd_prev(pg) ≠ Null then
14:        if PgHd_next(pg) = Null then
15:            PgEnt_last(PE) ← PgHd_prev(pg)
16:        else
17:            PgHd_prev(PgHd_next(pg)) ← PgHd_prev(pg)
18:        PgHd_next(PgHd_prev(pg)) ← PgHd_next(pg)
19:    else
20:        if PgHd_next(pg) = Null then
21:            PgEnt_last(PE) ← Null
22:        else
23:            PgHd_prev(PgHd_next(pg)) ← Null
24:        PgEnt_first(PE) ← PgHd_next(pg)
25:    free_page(pg, LC_PG_void)
```

best overheads by row and by design are marked with bold. For example, for one thread, the best value for the minimum overhead, found for the NS design was 0.74 obtained with the TcMalloc memory allocator, while for the NS design combined with the TabMalloc was 0.53, again obtained with the TcMalloc.

The results on Table 4.1 clearly show that TabMalloc has a big impact in reducing the overheads of the NS design. When comparing the values of the PtMa columns with and without TabMalloc, one can observe that for one thread, TabMalloc reduces on average the overhead in 0.14. As we scale the number of threads, the difference between both overheads increases significantly, ending for 32 threads with the result

Table 4.1: Overhead ratios, when compared with the NS design with 1 thread, for the NS design alone and combined with TabMalloc, using the PtMalloc 3, Hoard 3.10, TcMalloc 4.2 and JeMalloc 3.6 memory allocators, when running 1, 8, 16, 24 and 32 threads with local scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**NS**				**NS + TabMalloc**			
		PtMa	**Hoard**	**TcMa**	**JeMa**	**PtMa**	**Hoard**	**TcMa**	**JeMa**
1	**Min**	1.00	0.96	**0.74**	0.94	0.61	0.54	**0.53**	0.54
	Avg	1.00	1.02	**0.91**	0.99	0.86	0.80	**0.78**	**0.78**
	Max	**1.00**	1.19	1.19	1.16	1.08	1.11	**1.06**	**1.06**
	StD	0.00	0.06	0.12	0.06	0.14	0.16	0.15	0.15
8	**Min**	1.07	1.15	**0.90**	1.02	0.85	0.68	**0.66**	0.67
	Avg	2.35	4.24	1.19	**1.18**	0.99	0.92	**0.85**	0.87
	Max	5.06	9.41	1.89	**1.46**	1.32	**1.11**	1.12	1.21
	StD	1.23	2.52	0.26	0.14	0.14	0.12	0.13	0.16
16	**Min**	**1.02**	1.15	1.04	1.09	0.88	0.91	0.85	**0.78**
	Avg	5.13	13.29	2.09	**1.39**	1.61	1.18	0.98	**0.95**
	Max	11.17	33.49	5.16	**1.84**	3.19	1.76	**1.16**	1.20
	StD	3.12	9.87	1.10	0.23	0.78	0.24	0.09	0.12
24	**Min**	1.24	1.28	1.24	**1.16**	0.95	1.07	0.91	**0.90**
	Avg	8.42	28.79	3.16	**1.79**	2.37	1.54	**1.15**	1.18
	Max	18.33	80.99	8.64	**2.65**	5.43	2.82	**1.72**	1.96
	StD	5.24	23.31	1.98	0.48	1.50	0.54	0.20	0.26
32	**Min**	1.33	1.43	1.35	**1.23**	1.24	1.15	**1.05**	**1.05**
	Avg	12.94	47.06	4.40	**1.92**	3.45	2.11	**1.51**	1.56
	Max	26.67	121.39	13.11	**3.00**	8.24	4.32	**2.52**	2.73
	StD	7.52	36.08	3.08	0.50	2.36	0.93	0.45	0.52

of 12.94, on average for the NS design used solely and 3.45 for the NS design with TabMalloc.

Comparing now the best combination with alternative memory allocators, both TcMalloc and JeMalloc showed good performances, but the best combination seems to be the usage of TabMalloc with the TcMalloc memory allocator since, as we scale the threads up to 32, the overheads remain quite low, ending with an average of 1.51 for 32 threads, which is a quite significant achievement.

Table 4.2 shows the impact of the TabMalloc in the YapTab-Mt for the $\texttt{SS}_G$ design. We use the same metrics and the same memory allocators.

Table 4.2: Overhead ratios, when compared with the NS design with 1 thread, for the SS_G design alone and combined with TabMalloc, using the PtMalloc 3, Hoard 3.10, TcMalloc 4.2 and JeMalloc 3.6 memory allocators, when running 1, 8, 16, 24 and 32 threads with local scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**SS_G**				**SS_G + TabMalloc**			
		PtMa	**Hoard**	**TcMa**	**JeMa**	**PtMa**	**Hoard**	**TcMa**	**JeMa**
1	**Min**	0.99	0.95	**0.71**	0.86	0.54	**0.53**	0.54	0.54
	Avg	1.09	1.14	**0.93**	1.07	0.86	0.88	**0.84**	0.86
	Max	1.26	1.44	**1.06**	1.24	1.11	1.18	**1.03**	1.11
	StD	0.10	0.12	0.11	0.11	0.19	0.21	0.17	0.20
8	**Min**	0.99	0.97	0.99	**0.88**	0.84	0.73	**0.66**	0.72
	Avg	2.44	3.05	**1.30**	**1.30**	1.17	1.15	**0.99**	1.04
	Max	5.00	11.59	1.99	**1.74**	1.62	1.82	**1.36**	1.54
	StD	1.23	2.87	0.23	0.24	0.26	0.37	0.22	0.28
16	**Min**	**1.05**	1.07	1.16	0.97	0.98	0.93	0.81	**0.80**
	Avg	5.03	7.78	2.13	**1.52**	1.84	1.48	**1.13**	1.15
	Max	11.31	32.77	5.30	**1.87**	3.21	2.74	**1.50**	1.73
	StD	3.17	9.70	1.07	0.26	0.72	0.55	0.21	0.29
24	**Min**	1.08	1.14	1.19	**1.02**	1.15	1.05	1.02	**0.99**
	Avg	7.91	15.85	3.10	**1.91**	2.62	1.78	**1.34**	1.37
	Max	17.89	80.65	8.77	**2.68**	5.56	2.76	**1.77**	1.92
	StD	5.27	23.01	2.04	0.45	1.39	0.57	0.23	0.27
32	**Min**	1.18	1.33	1.24	**1.06**	1.24	1.14	**1.07**	**1.07**
	Avg	11.16	25.13	4.18	**2.05**	3.64	2.43	**1.71**	1.72
	Max	25.91	120.79	13.14	**3.03**	8.28	3.98	**2.61**	2.74
	StD	7.72	35.48	3.18	0.48	2.22	0.95	0.45	0.46

The results on Table 4.2 show again that for the $\texttt{SS}_G$ design using the TabMalloc is always better than not using it. Remember that, on the one hand, the SS design requests less trie nodes for the subgoal tries (thus reducing synchronization when requesting memory for the memory allocator) but, on the other hand, we are introducing a new cost when synchronizing the insertion of nodes into the shared subgoal trie structures. This cost is more clear for the benchmarks that allocate an higher number of subgoal

trie nodes. However in general, the TabMalloc always decreases the overhead ratios, and its impact is more significant as we scale the number of threads. As for the NS design, the combination that has the best results is the one that uses TabMalloc with the TcMalloc memory allocator.

Finally, Table 4.3 shows the same study for the $\texttt{FS}_G$ design. Remember that the FS design also has the answer tries shared among threads. On one hand, it requests less trie nodes for the answer tries (thus reducing synchronization when requesting memory for the memory allocator) but, on the other hand, it introduces an extra cost when synchronizing the insertion of nodes into the shared answer trie structures. Again, the results show that, in general, TabMalloc always decreases the overhead ratios, and its impact is more significant as we scale the number of threads. As for the previous designs, the combination that has the best results is the one that uses again TabMalloc with TcMalloc memory allocator.

In conclusion, our experimental results clearly show that the TabMalloc memory allocator performs always better than the equivalent implementation not taking advantage of it and that, it can achieve significant reductions on the execution time on all the YapTab-Mt designs. The experiments also show that the new memory allocator scales better when we increase the number of threads and if combined with the TcMalloc memory allocator, it can improve even further the execution times.

4.5 Chapter Summary

In this chapter, we have presented TabMalloc which is a novel, efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs. TabMalloc is based on local and global pages, that splits memory among specific data structures and different threads, together with a page based mechanism, where data structures of the same type are pre-allocated within a page. Our experimental results showed that we were successful in our goal of minimizing the performance degradation that YapTab-Mt suffered, when exposed to simultaneous memory requests made by multiple threads.

Table 4.3: Overhead ratios, when compared with the NS design with 1 thread, for the FS_G design alone and combined with TabMalloc, using PtMalloc 3, Hoard 3.10, TcMalloc 4.2 and JeMalloc 3.6 memory allocators, when running 1, 8, 16, 24 and 32 threads with local scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**FS_G**				**FS_G + TabMalloc**			
		PtMa	**Hoard**	**TcMa**	**JeMa**	**PtMa**	**Hoard**	**TcMa**	**JeMa**
1	**Min**	1.01	1.02	0.98	**0.93**	0.84	**0.81**	0.85	0.83
	Avg	1.22	1.30	**1.10**	1.25	1.01	1.01	**0.99**	1.01
	Max	1.56	1.63	**1.29**	1.58	1.14	1.17	**1.10**	1.12
	StD	0.16	0.17	0.08	0.16	0.09	0.11	0.08	0.08
8	**Min**	1.02	1.05	**0.99**	1.08	**0.99**	1.10	1.09	1.11
	Avg	3.35	3.15	**2.93**	3.14	2.47	2.49	**2.46**	2.47
	Max	6.50	6.11	**5.83**	6.03	4.50	**4.35**	4.48	4.51
	StD	1.68	1.48	1.51	1.54	1.07	1.04	1.08	1.05
16	**Min**	**1.02**	1.14	1.17	1.04	**1.01**	1.10	1.14	1.14
	Avg	4.29	3.80	**3.60**	3.89	2.90	2.92	**2.89**	**2.89**
	Max	8.23	**7.18**	7.21	7.97	5.63	**5.53**	5.67	5.62
	StD	2.23	1.76	1.95	2.15	1.39	1.37	1.40	1.40
24	**Min**	**1.22**	1.26	1.24	**1.22**	**1.23**	1.27	**1.23**	1.24
	Avg	4.96	4.37	**4.03**	4.40	3.18	3.16	**3.15**	3.16
	Max	9.30	8.23	**8.17**	8.95	6.32	**6.26**	6.34	6.37
	StD	2.43	2.02	2.27	2.46	1.58	1.54	1.58	1.59
32	**Min**	**1.34**	1.36	1.36	1.37	**1.36**	1.37	1.37	1.38
	Avg	5.72	5.12	**4.78**	5.02	3.63	3.58	**3.51**	3.61
	Max	10.87	10.41	**9.92**	10.50	7.48	**7.42**	7.47	7.51
	StD	2.83	2.56	2.82	2.90	1.89	1.85	1.85	1.89

Chapter 5

Lock-Free Data Structures

In this chapter, we present two proposals for lock-free data structures that address concurrency within the ST data structure of the SS design and within the ST and AT data structures of the FS design. For each proposal, we will discuss the implementation of the concurrent search and insert operations, the correctness of the proposal and its efficiency in the context of the YapTab-Mt framework.

5.1 YapTab-Mt Table Space Data Structures

A critical component in the design of an efficient concurrent tabling system is the implementation of the data structures and algorithms that manipulate tabled data. As observed in the previous chapters, YapTab-Mt implements a two-level trie data structure, where one trie level stores the tabled subgoal calls and the other stores the computed answers. Recall that a trie is a tree structure where each different path corresponds to a term described by the tokens labeling the nodes traversed. Figure 5.1 shows an example for the tabled predicate $p/3$ presented in the previous sections.

On the ST data structure, each different path corresponds to a subgoal call described by the tokens labeling the nodes traversed. For example, the tokenized form of the subgoal call $p(1, X, Y)$ is the sequence of 3 tokens 1, VAR_0 and VAR_1. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second subgoal call $p(1, 2, 3)$. Since the first argument, the token 1, is common to both terms, only two additional nodes will be required to fully represent this second call in the trie. On the AT structure, the behavior is similar to the ST structure when the trie structure is accessed in a top-down fashion. The

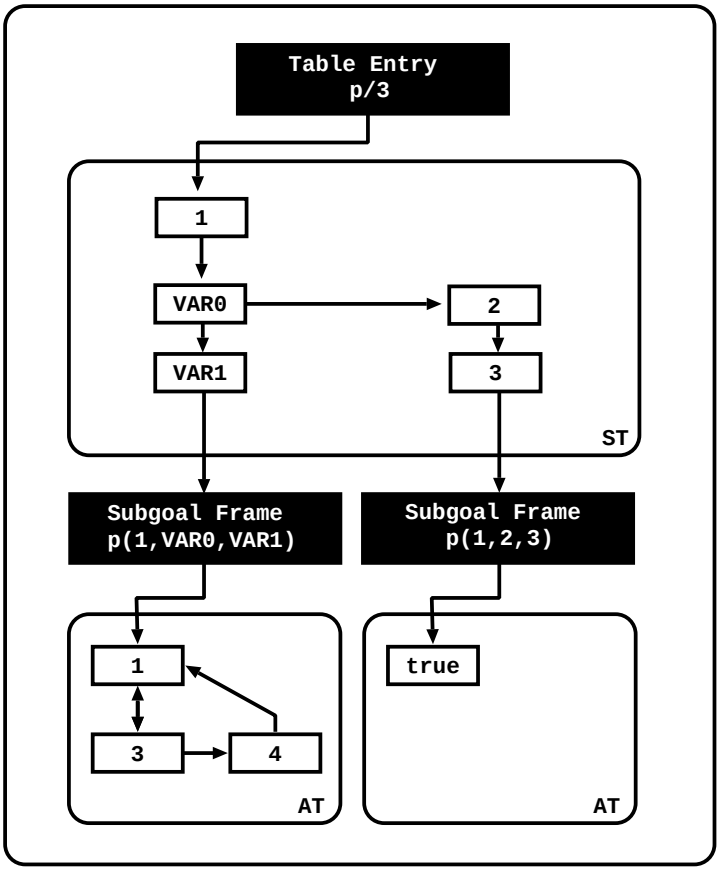

Figure 5.1: Trie hierarchical levels overview

difference is that each path corresponds to a different answer to the tabled subgoal call. For the subgoal call $p(1, VAR0, VAR1)$, the answers shown are $p(1, 1, 3)$, i.e., $VAR0 = 1$ and $VAR1 = 3$, and $p(1, 1, 4)$, i.e., $VAR0 = 1$ and $VAR1 = 4$, while for the subgoal call $p(1, 2, 3)$ the answer is *true*. Internally, each particular ST and AT data structure has as many trie levels as the number of parent/child relationships. For example in Figure 5.1, ST data structure has 3 levels, and the AT data structure for the subgoal calls $p(1, VAR0, VAR1)$ and $p(1, 2, 3)$ have 2 and 1 levels, respectively.

On both ST and AT data structures, the trie nodes are 4-field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Whenever a level of the trie becomes saturated, i.e., the chain of sibling nodes with a common parent node becomes larger than a predefined threshold value, a *hash mechanism* is used to provide direct node access and therefore optimize the search for the node's token. Figure 5.2 shows a hashing mechanism for a trie level within the ST and AT data structures.

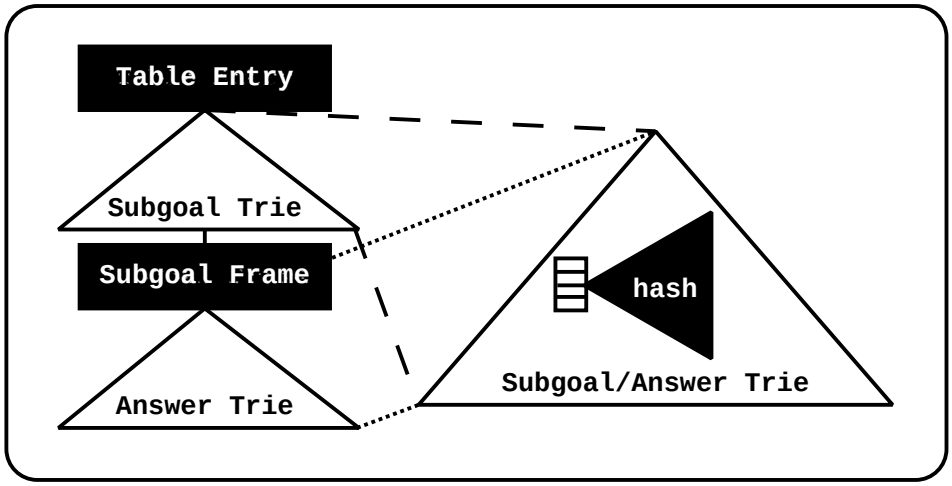

Figure 5.2: The hashing mechanism within a trie level

Several approaches for hashing mechanisms exist. The most important aspect of a hashing mechanism is its behavior in terms of hash collisions, i.e., when two keys collide and occupy the same hash table location. Multiple solutions exist that address the collision problem. Among these are the *open addressing* and *closed addressing* approaches [128, 63].

In open addressing, the hash table stores the objects directly within the hash table internal array. The term *open* indicates that the index (also known as address) at which the object is stored in the array is not completely determined by its key. Instead, the index varies depending on what's already in the hash table. A hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target object is found, or an unused entry is found, which indicates that there is no such key in the hash table.

In closed addressing, every object is stored directly at an index in the hash table's internal array. The term *closed* guarantees that the index at which the object is stored in the array is completely determined by its key. This means that collisions are resolved by storing potentially several objects at the same index. In closed addressing, collisions are solved by using other arrays or linked lists. One well known mechanism to solve hash collisions is the *separate chaining* [63]. In the separate chaining mechanism, the hash table is implemented as an array of linked lists. The basic idea of separate chaining techniques is to apply linked lists for collision management, thus in case of a conflict a new key is appended to the linked list. Each hash table entry has its own list for collision resolution. The advantage of chaining techniques rely in the ability to easily resolve collisions since new keys can be always inserted without resizing the hash

table, thus they are not dependent on choosing beforehand a proper hash table size. Multiple optimizations exist that improve the behavior of the hashing mechanism, such as *move to the front* and *exact fit* [80], but for the moment we will not consider them in this work. For more details about these mechanisms and others please consult [63, 145].

YapTab-Mt is a general-purpose framework, which means that a user can use it for any kind of tabled logic program. The best option for our framework is thus the hashing mechanism with separate chaining, since we can not know the proper size of a hash table in advance, and no fixed size suits all system configurations and workloads. Additionally, since the framework's needs may change at runtime and the performance of the hash mechanism depends heavily on the number of hash buckets[1], we use a dynamic resizing support over the hash table to improve the behavior of the hashing mechanism whenever it becomes saturated. We will give more details about this in what follows.

Using Figure 5.3, we go one step deeper in YapTab-Mt's trie internals and pinpoint the operations that occur on each trie level. We will be showing the operations for one level only, since all levels have the same behavior.

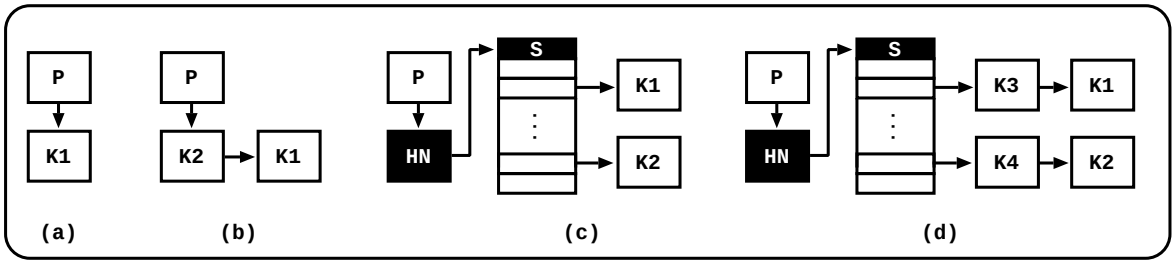

Figure 5.3: Hash tables inside the trie levels

Figure 5.3 shows how starting from a common parent node P, the trie is adapted to the insertion of child nodes with distinguish keys K_1, K_2, K_3 and K_4. Figure 5.3(a) shows the trie level representation after the insertion of K_1 and Figure 5.3(b) shows the trie representation after the insertion of K_2. Note that new nodes are inserted on the head for the level. Whenever the number of nodes in a level reaches a predefined threshold value MAX_NODES, the trie level is extended to include a hash mechanism with separate chaining. For simplicity of illustration, in this example, we are using a MAX_NODES value of 2. Figure 5.3(c) shows the trie level representation using the

[1]Making a hash table too small might lead to excessively long hash chains and poor performance. Making a hash table too large might consume too much memory, increasing hardware requirements and reducing performance-improving caches.

hashing mechanism. Each hashing mechanism includes two data structures, a specific node HN that contains generic information about the trie level, such as the total number of nodes in the level, and a structure that contains a value S and a bucket array with a size of S entries. Then, using a hash function $Hash(S, key) = key\ mod\ S$ over the keys K_1 and K_2 the respective nodes with keys K_1 and K_2 are mapped into the bucket array of entries. In this example, the keys K_1 and K_2 do not collide, thus they are mapped into two different entries. From this point on, the access to the child nodes of the parent node P is done through the hashing mechanism. Finally, Figure 5.3(d) shows the trie level representation after the insertion of keys K_3 and K_4. In our hashing mechanism, new nodes are inserted in the head of the bucket entries of the array. In this example, the hash function resulted in the collision of keys K_3 and K_4 with keys K_1 and K_2, respectively.

When the number of nodes in a bucket entry exceeds the MAX_NODES value and the total number of nodes exceeds S, we consider that the hash bucket array is saturated and in such cases we expand the hash by doubling the number S of entries of the bucket array. Figure 5.4 shows more details about the expansion procedure. Figure 5.4(a) shows the trie level representation after the bucket array expansion to a new one with the double number of entries ($2 * S$ in this case) followed by the adjustment process of nodes with keys K_1, K_2, K_3 and K_4 to the new bucket array of entries. The hash function is now called as $Hash(2 * S, key)$ and the nodes are now mapped in the range of 0 to $2 * S - 1$. In this example, we are assuming that the keys K_1 and K_3 do not collide, thus they are mapped into two different entries, and that the keys K_2 and K_4 collide in the same entry.

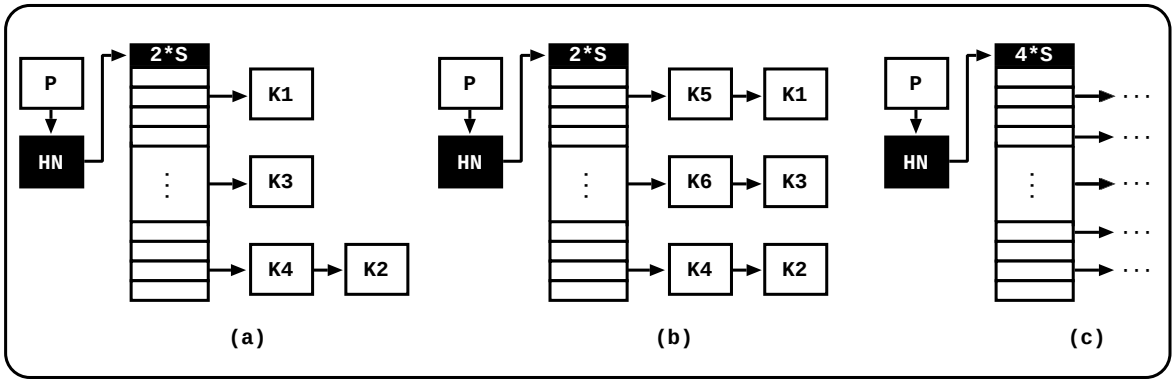

Figure 5.4: Expanding the hash tables inside the trie levels

Figure 5.4(b) shows the trie level representation after the insertion of new nodes with keys K_5 and K_6. In the example, we are assuming that they collide with the keys K_1 and K_3, respectively. Finally, Figure 5.4(c) shows a situation where the hash bucket

array becomes saturated again and, as consequence, the bucket array is expanded to a new one, this time with $4 * S$ entries, and the hash expansion process continues to be executed as long as it is necessary.

5.2 Concurrent Data Structures

To address concurrency inside the ST and AT data structures, our initial approach, was to use *lock-based* data structures. Our lock-based approach allowed a *single writer* per chain of sibling nodes that represent alternative paths from a common parent node, meaning that only one thread at a time can be inserting a new child node starting from the same parent node. To implement locking, we used either a *locking field* per trie node or a *global array* of lock entries [6]. To reduce the lock duration, we also tried with *trylocks* instead of traditional locks. However, in the context of multithreading, the lock-based data structures have their performance restrained by multiple problems, such as:

- **Priority inversion**. A lower-priority thread is preempted while holding a lock needed by higher-priority threads.
- **Convoying**. A thread holding a lock is descheduled, perhaps by exhausting its scheduling quantum, by a page fault, or by some other kind of interrupt. When such an interruption occurs, other threads capable of running may not be able to do so, due to the thread holding the lock.
- **Deadlock**. Threads attempt to lock the same set of objects in different orders. Deadlock avoidance can be awkward if threads must lock multiple data objects, particularly if the set of objects is not known in advance.
- **Mutual exclusion**. Can needlessly restrict parallelism by serializing non-conflicting updates. This can be greatly mitigated by using fine-grained locks, but lock convoying and cache performance may then become an issue, along with the extra cost of acquiring and releasing those locks.
- **Contention**. Even when an operation does not modify shared data, the required lock manipulations can cause memory coherency conflicts, and contribute to contention on the memory interconnect. This can have enormous impact on system performance: In the work [88], Larson and Krishnan observed that *reducing the frequency of access to shared, fast-changing data items* is critical to

prevent *cache ping-pong* effects from limiting system throughput. Cache ping-pong effects occur when a cache line with exclusive ownership moves rapidly among a set of processors.

5.2.1 Compare-And-Swap Operations

YapTab-Mt's framework supports the evaluation of tabled programs according to the semantics of SLG resolution [27]. The practical significance of this is that, in general, we know that a concurrent tabled program will only execute search and insert operations over the table space shared data structures and no delete operations are performed, thus the size of the shared trie data structures always grow monotonically during an evaluation. YapTab-Mt's shared data structures are only removed when the last running thread abolishes the tables.

To deal with concurrency within the shared trie data structures, we are interested in taking advantage of the fact that, no concurrent delete operations are performed in YapTab-Mt's framework, and combine it with the low-level CAS operation, that nowadays can be widely found on many common architectures. CAS is a fine grained and fully synchronized operation that dates back to the *IBM System 370* and it is still available on many modern processors including Intel *IA-64 (x86)* and *Sun SPARC* architectures. Processors, like the *IBM PowerPC* that do not support the CAS operation, often support directly Load-Linked and Store-Conditional (LL/SC) operation instead, which is sufficient to implement the CAS operation [83].

The CAS operation is an *atomic instruction* that compares the contents of a memory location to a given value and, if they are the same, updates the contents of that memory location to a given new value. The atomicity guarantees that the new value is calculated based on up-to-date information, i.e., if the value had been updated by another thread in the meantime, the write would fail. The CAS result indicates whether it has successfully performed the substitution or not. Internally, the CAS operation uses a **UPDATE** procedure that receives two arguments: a *reference* (also known as address) to a memory position and a *value*. Then, the **UPDATE** procedure writes the value in the memory position. For the sake of clarity, we distinguish the **UPDATE** procedure from a standard attribution procedure. The **UPDATE** procedure updates a memory position which is shared among threads, while a standard attribution procedure updates a memory position which is private to a thread[2].

[2]We will use this notion later, when we prove the correctness of our proposals using linearization.

Algorithm 5.1 shows the pseudo-code of a Boolean CAS operation.

Algorithm 5.1 CAS(memory reference M, expected value E, new value N)

```
1: if Val(M) = E then
2:     UPDATE(M, N)          ▷ updates the value of memory position M to N
3:     return True
4: else
5:     return False
```

The CAS operation receives three arguments: the reference of the memory location M, the expected value E in the memory location and the new value N to replace the expected value E. Then, the CAS operation *atomically* checks if the value in M has the expected value E (line 1) and if so, it replaces E with the new value N (line 2). Otherwise, M remains unchanged. At the end, the operation returns the Boolean result of *True* or *False*, whether the operation succeed or not (lines 3 and 5).

5.2.2 Lock-Freedom and Linearization

Besides reducing the granularity of the synchronization, the CAS operation is at the heart of many *lock-free* (also known as non-blocking) data structures [56]. Non-blocking data structures offer several advantages over their blocking counterparts, such as being immune to deadlocks, tolerant to priority inversion, kill-tolerant availability (threads are immune to the dead of other threads during the execution) and preemption-tolerant (which ensures the performance regardless of the arbitrary thread scheduling), and convoying. Additionally, they have been shown to work well in practice in many different settings [130, 122]. They have been included in Intel's Threading Building Blocks Framework [103], the NOBLE library [122] and the Java concurrency package [69].

A data structure is lock-free if it can be accessed by multiple threads concurrently without using any type of standard *locking mechanism*, such as spinlocks, mutexs or semaphores. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. A shared object is lock-free if it guarantees that whenever a thread executes some finite number of steps, at least one operation on the object by some thread must have made *progress* during the execution of these steps. In the work [55], Herlihy and Shavit presented a novel *grand unified explanation* for the progress properties, using *linearizability* which is an important correctness condition for the implementation of concurrent data structures [57]. Linearizability ensures

the correctness of concurrent data structures by proving that semantically consistent (non-interfering) operations may execute in parallel. An operation is linearizable if it appears to take effect instantaneously at some moment of time t_{LP} between its invocation and response. The literature often refers to t_{LP} as a Linearization Point (LP) and, for lock-free implementations, a linearization point is typically a single instant where its effects become visible to all the remaining operations. Linearizability guarantees that if all operations individually preserve an invariant, the system as a whole also will. Thus, linearizability is a local property, and is therefore independent of any underlying scheduling policy or interaction between objects. Locality improves the portability and modularity of large concurrent systems, and can simplify reasoning about concurrent data structures.

The progress is seen as the number of steps that threads take to complete methods within a concurrent object, i.e., the number of steps that threads take to execute methods between their *invocation* and their *response*. The execution of a concurrent object is then modeled by a *history* H, a finite sequence of method invocation and response events, a *subhistory* of H is a sub-sequence of the events of H and an *interval* is a subhistory consisting of contiguous events. Progress conditions are placed in a two-dimensional *periodical table*, where one of the axis defines the assumptions of the OS scheduler, which might be *scheduler independent* or *scheduler dependent*, and the other axis defines the *maximal progress* and *minimal progress* provided by a method in a history H. Since we will be using this notion of progress later when we present the proof of correctness of our proposals, we analyze next using Figure 5.5, the *periodic table* of progress conditions defined by Herlihy and Shavit [55].

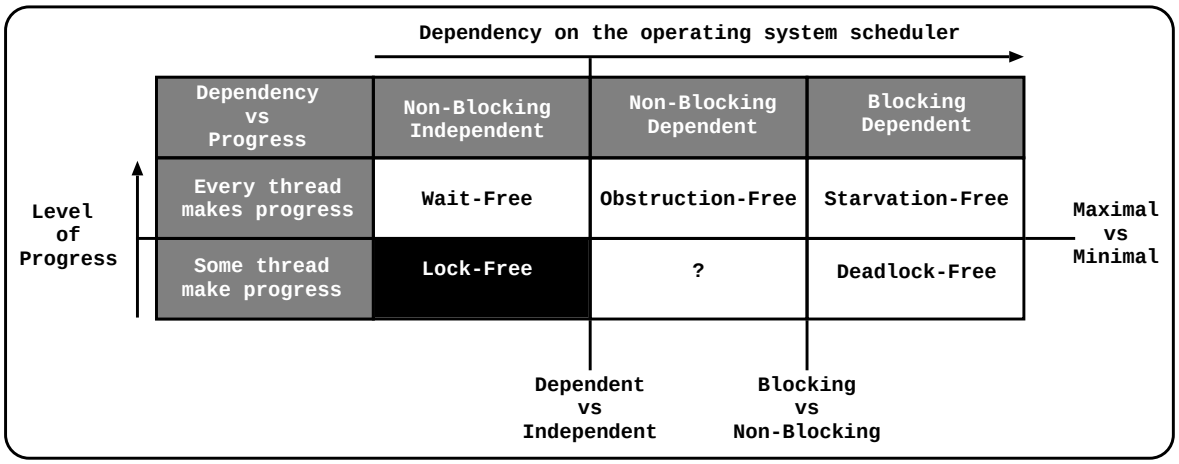

Figure 5.5: The *Periodic Table* of progress conditions

For the assumptions about the OS scheduler, a scheduler independent assumption,

guarantees progress as long as threads are scheduled and no matter how they are scheduled. A scheduler dependent assumption, means that the progress of threads rely on the OS scheduler to satisfy certain properties. For example, the deadlock-free (threads cannot delay each other perpetually) and starvation-free (a critical region cannot be denied to a thread perpetually) properties guarantee progress, however, they depend on the assumption that the OS scheduler will let each thread within a critical region to be able to run a sufficient amount of time, so that it can leave the critical section. The obstruction-free property [54] (a thread runs within a critical region in a bounded number of steps) requires the OS scheduler to allow each thread to run in isolation for a sufficient amount of time.

In a nutshell, the progress conditions are divided in to blocking, if a thread blocks all remaining threads during a access to critical region and non-blocking, otherwise. Herlihy and Shavit define the progress conditions as the level of progress provided by methods within objects. A method provides the minimal progress in H, if in every suffix of H, *some pending active invocation* has a matching response. In other words, there is no point in the history where *all threads that called the method* take an infinite number of concrete steps without returning. An abstract method provides maximal progress in a history H if in every suffix of H, *every pending active invocation* has a matching response. In other words, there is no point in the history where *a thread that calls the abstract method* takes an infinite number of concrete steps without returning. The lock-free data structures are mapped in the periodical table as scheduler independent and providing minimal progress.

5.2.3 Related Work

We next briefly describe some of the state-of-the-art approaches for concurrent trie data structures and for lock-free hash tables using linked lists. Historically, a number of so-called *universal methods* [56, 92, 16, 58] for constructing non-blocking data structures of any type have been discussed in the literature. In the work [58], Herlihy presented the first widely accepted universal method. He maintains a write-ahead log of operations for each shared data structure. The order of entries in the log determines the serialization order. Private copies of structures, built by applying a sequential reconstruction algorithm to the operations in the log, are updated and then finally added to the log itself. Check-pointing the log is used to decrease the cost of state reconstruction.

The first non-blocking linked list proposal was introduced by Valois [132]. In Valois

proposal, the consistency is maintained by using *auxiliary nodes* which are defined as nodes that did not store values. Every list node has an auxiliary node that is used during concurrent insertions and deletions to help maintain the consistency of the list. Each auxiliary node consists of a single pointer to the next regular node in the list and every normal node in the list is required to have an auxiliary node as its predecessor and its successor. Thus, in this proposal, the consistency of the lock-free linked list is maintained at the cost of a two times storage overhead. One of the major drawbacks of Valois proposal was that it suffered from the ABA problem[3], since the proposal used the CAS operation, but the concurrent delete of a node operation, required two simultaneous atomic operations. Greenwald [47] suggested a stronger Double-Compare-And-Swap (DCAS) operation that atomically updates two memory locations after confirming that both of them contained the expected values. But, until now the DCAS operation is not commonly available in multiprocessor architectures. Later, Harris [52] presented the first correct CAS-based lock-free list-based set proposal. Harris proposal does not use the auxiliary nodes proposed by Valois, instead, it uses a two stage approach to deal with deletion of nodes. Whenever a list node is to be deleted, in the first stage the node is marked as *logically deleted* and only on the second stage the node is *physically deleted*. Thus, the delete operation, first marks a node as deleted using CAS to prevent new nodes from being linked to it, and then removes it from the list by swinging the next pointer of the previous node to the next node in the list, also using CAS.

Michael [81] presented an improved proposal for lock-free list-based sets and hash tables. In the work [81], Michael improves Harris work by presenting the first CAS-based lock-free list-based set proposal that was compatible with all lock-free memory management methods and Michael uses this proposal has the building block for lock-free hash tables. Thus, the proposal used fixed sized arrays for the hashing operations and list-based sets to deal with the collisions in the hash. The proposal allowed the search, insert and delete operation of nodes in the lists, so in that sense the proposal was dynamic, but the size of the arrays was fixed. Nevertheless, experimental results showed that the lock-free implementation outperformed, by significant margins, the best lock-based implementations, both under low and high contention.

Michael's work was latter extended by Shalev and Shavit [119], when they presented their lock-free algorithm for expandable hash tables. The algorithm is based in split-ordered lists and allows the number of hash buckets to vary dynamically according to the number of nodes inserted or deleted, preserving the read-parallelism. Later, both

[3]We will give more details about this problem in the next section.

Gao el al. [43] and Purcell and Harris [95] presented lock-free open address hash tables. More recently, Triplett et al. presented a set of algorithms that allow concurrent wait-free[4], linear scalable searches while shrinking and expanding hash tables [129]. The experimental results showed a good performance even when the hash table is under resizing.

Regarding concurrent trie data structures, recently Prokopec et al. presented recently the Concurrent Hash Tries (CTries) [94]. CTries can be used to implement, efficient concurrent, lock-free maps and sets. They have the lock-freedom property and support lookup, insert and delete operations based on CAS instructions and the support of a constant time, linearizable, lock-free snapshot operation. The snapshot operation provides a consistent view of a data structure at a single point in time. Snapshots can be used to implement operations requiring global information about the data structure - in this case, the performance of the data structure is limited by the performance of the snapshot. The CTries snapshot operation is used to parallelize CTries operations without the need for quiescence. Its aim is to improve the performance of the CTrie through the usage of Generation-Compare-And-Swap (GCAS) operations. At the structural level, CTries are trees composed by multiple types of data structures . Some of the most relevant nodes are the following: The *indirection node* contains a reference to a single node called a main node and there are several types of main nodes. The *tomb node* which is a special node used to ensure proper ordering during removals. The *list node* which is a leaf node used to handle hash code collisions by keeping such keys in a list. The *CTries node* which is an internal main node containing a bitmap and the array with references to branch nodes. A branch node is either another internal node or a singleton node, which contains a single key and a value. Singleton nodes are leaves in the CTries. For more information about the CTries please consult [93]. All these types of data structures are mostly used for the support of the delete concurrent operation. Since our proposals do not require the support for this operation, we were able to reduce the number of required data structures to two, bucket arrays of entries for hashing and chain nodes for values.

[4]In the periodical table, the wait-free data structures are mapped as independent of the operating system scheduler and as providing maximal progress.

5.3 Motivation

Although state-of-the-art approaches exist and are well documented, to the best of our knowledge, none of them is specifically aimed for an environment with the characteristics of the SS and FS designs. One reason for that is the complexity of the tabling engine, which integrates trie structures with generator and consumer evaluation nodes. Recall from Figure 2.13 that these evaluation nodes access the table space (and the trie data structures) in different fashions, some of them top-bottom and others bottom-up. For our multithreaded tabling framework, this means that nodes inside the trie data structure cannot be replaced by new nodes with the same key (and respective information), since replacing a trie node would imply to visit all the generator and consumer evaluation nodes for all threads under execution and update their information with the new node, for the ones referring the older node. Such procedure, would cause an huge overhead in a multithreaded tabling framework. Thus, some of the state-of-the-art approaches, such as CTries could not be used because, to update the trie state, they use techniques that require making a private copy of nodes inside the trie and replace them with new nodes.

Another reason for not using the state-of-the-art approaches is that, in general, the evaluation of a tabled program is deterministic, finite and only executes lookup and insert operations over the table space data structures. In YapTab-Mt, the table space is recovered when the last running thread abolishes a table. Since no delete operations are performed, the size of the tables always grows monotonically during an evaluation. Therefore, since the nodes inside the ST and AT are persistent during a tabled evaluation, the SS and FS designs do not require any support for concurrent delete operations. Concurrent delete operations often require extra computational steps. For example, in Harris's work for lock-free linked lists [52], the delete operation requires two CAS operations, the first to mark the node to be deleted as *logically deleted* and the second to *physically delete* the node. In trees, several practical examples exist where the concurrent delete operation interferes with lookup and insert operations. Examples are the approaches of lock-free binary trees [26] and Bw-Trees [71].

With both reasons in mind, we have created two fresh proposals for lock-free linearizable data structures aimed to be as effective as possible in the search and insert operations, by exploiting the full potentiality of lock-freedom on those operations, to minimize the bottlenecks and performance problems mentioned in Section 5.2 and without introducing significant overheads in the sequential execution. Figure 5.6 resumes the architecture of our lock-free proposals. The proposals have two types

of data structures, the chain nodes and the bucket arrays and one algorithm that implements the search/insert key operation. The external operation (or method) that is callable (the call and the return back arrows in figure) by the threads is the search/insert key operation which is lock-free and linearizable. In order to keep the efficiency in accessing the chain nodes, during the search/insert key operation a thread might be *elected to optimize (or expand)* a bucket array data structure if it becomes saturated (shown in gray in the figure). The elected thread is chosen through a single atomic CAS operation, that chooses the elected thread and blocks the election procedure simultaneously. Thus, after CAS operation only one thread is elected for doing the optimization. The elected thread proceeds as follows: first it expands the bucket arrays, then it adjusts the chain nodes to the expanded bucket arrays, and finally it unblocks the election procedure. As we will be proving in the next sections, the lock-freedom property will hold in all instants of the execution of the search/insert key operation, once either the elected and the non-elected threads will be proven to be doing their work in a lock-free fashion.

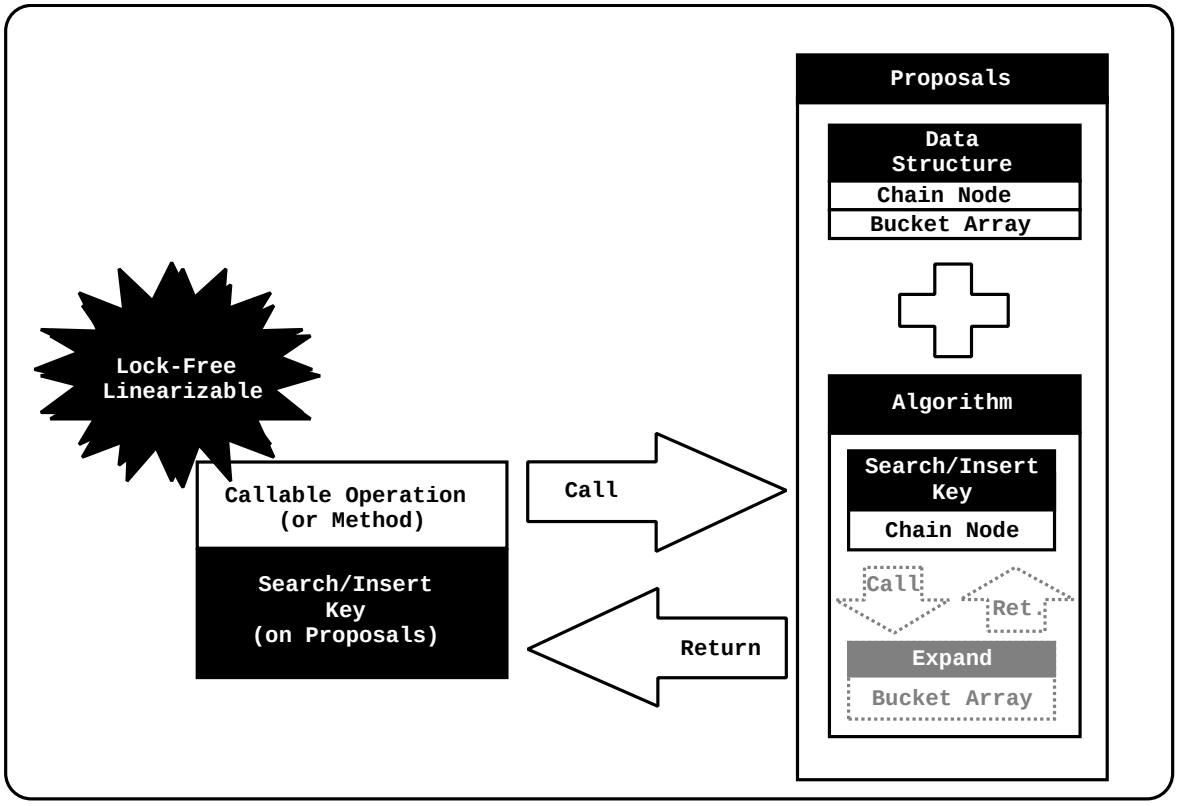

Figure 5.6: Architecture of the lock-free proposals

To resume the search/insert key operation in the proposals, a thread calls this operation to search for a key in the chain nodes. If the key is present then the operation

returns the chain node with the key, otherwise it inserts the key in a new chain node and returns the new chain node. During the operation, a thread might be elected to do some extra work, such as optimizing the bucket arrays and adjusting chain nodes. For the elected thread the time between the call and the return of operation increases in the proportion of the extra amount of work that the elected thread has to do. In all instants of the execution of the operation, the search, insert and optimization procedures, will guarantee that at least one threads does progress in their execution.

To support concurrency within the proposals we use the CAS operation. The usage of the CAS operation must be properly measured, since it can lead to multiple problems. Arguably, one of the best-known is the ABA problem[5]. The ABA problems occurs when the fact that a memory location has not changed between two readings is used to assume that nothing has changed during the period of time from the first to the second reading. Although, this is a common assumption when using the CAS operation, in some cases, it can lead to the ABA problem. An example of that would be: a thread T reads a value V_1 from a memory location L, uses V_1 to do some work, updates L to a new value V_2 and, at the end of the work, changes the value of L again to V_1. In such case, if another thread has read the memory location L before and after the work done by T, then it will be deceived by the fact that the memory location has not changed. In our trie data structures, a practical consequence of this would be to insert more than once the same value on the same level of the trie. To address the ABA problem, several techniques already exist, such as *version tagging* [35], *hazard pointers* [82] or *value semantics* [53]. In general, these kind of techniques rely on the fact that a writing over a memory position always cause a transition from the current state of the system to a uniquely new different state. Both our proposals will be proved to be correct using linearization, which ensures that they are ABA-free. One of the property in both proposals is the fact that every concurrent memory location L that is used to insert new structures (chain nodes and bucket arrays) refers only once to the same value V_1, i.e., if L is updated from V_1 to V_2 than L will never refer to V_1 again. Next, we describe our first lock-free trie data structure proposal.

5.4 Lock-Free Trie - LF_1

The LF_1 proposal is aimed to deal with concurrency inside the trie data structures in a lock-free fashion. In what follows, we describe the key ideas of this proposal, we

[5]Note that we have already mentioned this problem in Valois's approach.

discuss important implementation details and we present a proof of correctness. At the end of the section, we draw some conclusions about the proposal and motivate for the creation of a second proposal.

In Figure 5.3, we observed that to maintain an efficient performance on accessing nodes inside a trie level, each trie level is expanded with a hashing mechanism whenever it reaches a predefined threshold of nodes. The hashing mechanism is composed by a bucket array of S entries and a hash function that maps the nodes into the entries of the bucket array. Whenever the hash bucket array becomes saturated, i.e., when the number of nodes in a bucket entry exceeds the threshold value and the total number of nodes exceeds S, then the bucket array is expanded to a new one with $2 * S$ entries. Our LF_1 proposal maintains this constraints. The difference is that it implements all insert operations inside the trie structure using a CAS. In particular, the search and insert operations of nodes is done in a lock-free fashion. The expansion of the bucket arrays of entries of the hashing mechanism is also done using the CAS operation, that whenever is successfully executed by one thread, the expansion is blocked to all of the remaining threads. This means that no more than one expansion can be done at a time, and if the thread that is doing the expansion suspends by some reason (for example, the OS scheduler), then all the remaining threads can still be searching and inserting keys in the trie level that is being expanded in a lock-free fashion, but no other thread will be able to expand the same trie level.

Figure 5.7 shows the progress stages that a thread passes by in the search and insert key operation, according with the decisions (oval boxes) that it has to do while executing the operation in the LF_1 proposal. We define three types of stages that specify the type of progress that a thread can make:

- The private stages that do not change the configuration of data structures (white rectangular boxes). A thread progresses in a private fashion, searching for keys in chain nodes or returning from the operation;
- The public stage where a thread *might or might not change* the configuration of data structures (gray rectangular box). In this stage a thread progresses if it successfully inserts a key in the data structure;
- The public stages that *must* change the configuration of data structures (black rectangular boxes). A thread progresses in this stage whenever it changes successfully the configuration of data structures.

The search/insert key operations of the LF_1 proposal begin with a search key stage.

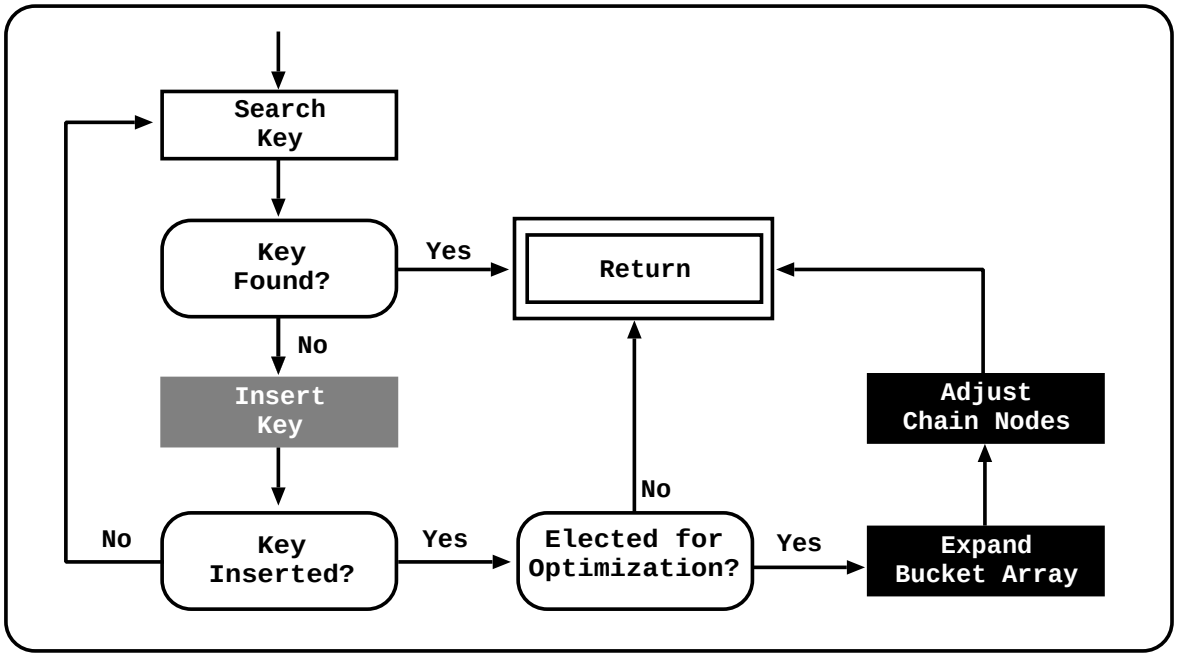

Figure 5.7: Progress stages of a thread in the search/insert key operation of the LF_1 proposal

In this stage, the thread searches for a key in the chain of nodes and if the key is found then the procedure ends the algorithm and moves to the return stage. Otherwise, the key is not found and the thread passes to the insert stage. In the insert stage, if the key is not inserted, then the thread moves again to the search stage. Otherwise, the key was inserted, thus the thread leaves the insert stage and checks whether it is elected for optimizing the data structure or not. If elected, then the thread passes to the optimization stage where it expands the current bucket array to a new one and adjusts all nodes to the new bucket array. Otherwise, if the thread is not elected, then it simply moves to the return stage and exits from the operation.

5.4.1 Our Proposal By Example

This section presents our LF_1 proposal to support the concurrent search, insertion, hash creation and expansion inside the ST and AT data structures. We begin with Figure 5.8 showing a small example that illustrates how the concurrent insertion of nodes in the new lock-free trie structure is done. Again, for the sake of simplicity, we are only considering one level of the trie data structure.

Figure 5.8(a) shows the trie configuration after the insertion of the child nodes K_1

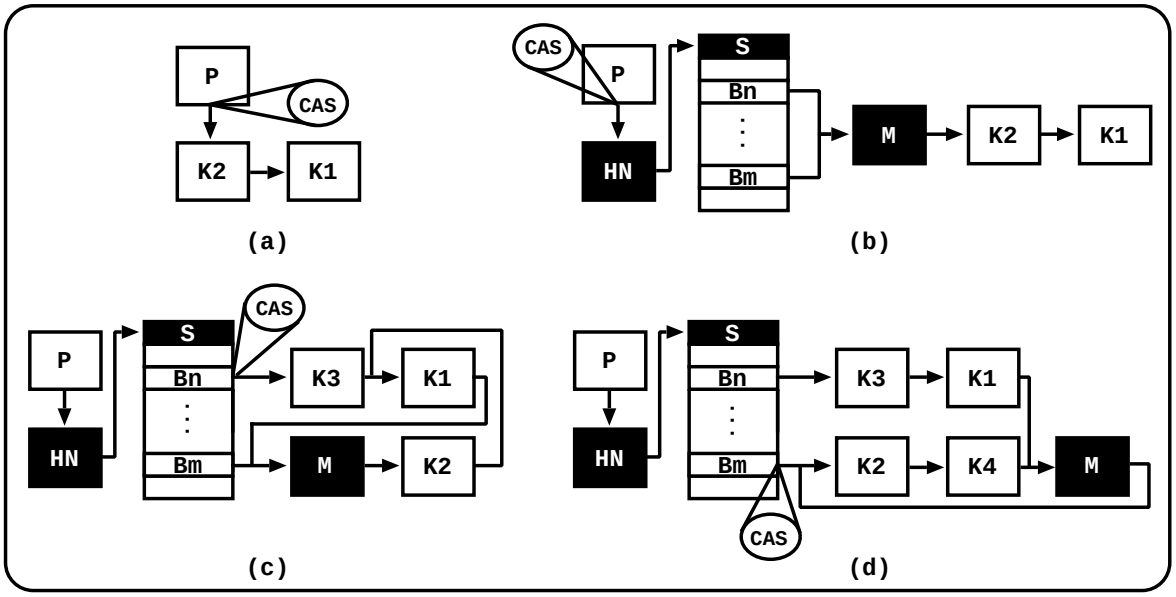

Figure 5.8: Concurrent insertion of nodes in a trie level using LF_1

and K_2 in the parent node P. At this stage, the search/insert operation for a node with a key is straightforward. Initially, a thread follows the pointer of P to access the next level of the trie. Then, the chain of sibling nodes is searched for the key at hand. If no such node exists, the pointer of P is used in a CAS operation to guarantee the synchronization of the insertion of the key in the chain. During the search, a local counter is used to count the number of nodes on the level which, in the case of a node insertion, is then used to verify if the trie level has reached the predefined threshold value required for hash creation. For this count, no synchronization is required, since only one thread will be able to have its local counter equal to the threshold value.

Figure 5.8(b) shows then the trie configuration in the case where a thread has started the hash creation process for a trie level. The thread first creates the special node HN, the initial bucket array with size S and initializes all entries in the bucket array referring to a special marking node M. The node M is then used to implement a synchronization point with the first child node of P (node K_2 in the figure) that, whenever both are synchronized, will correspond to a successful CAS operation on P that updates it to HN. This means that, from this point on, the access to the trie level will be done through the new hash node HN. If a thread has accessed the trie level before the hash creation, which means that it has not seen HN, in such case, when trying to insert a new node, the CAS operation on P will fail because P is now refering to HN.

In the continuation, Figure 5.8(c) and Figure 5.8(d) show the adjustment process of

placing the child nodes in the correct bucket entries. To ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read the correct values (starting from any bucket entry) and insert new values without any delay from the adjustment process. To guarantee both properties, we use M as a way to mark the beginning of the nodes not yet adjusted and we execute the adjustment process in reverse order. Figure 5.8(c) shows the case where node K_1 is first adjusted to be in the bucket entry Bn and Figure 5.8(d) shows the case where node K_2 is then adjusted to be in the bucket entry Bm. Concurrently with the adjustment process, other threads can be inserting nodes in the same bucket entries. In Figure 5.8(c), a new node K_3 is inserted after K_1 in entry Bn and, in Figure 5.8(d), a new node K_4 is inserted before K_2 in entry Bm. To ensure that the nodes not yet adjusted (after M) can always be accessed from any bucket entry, the adjustment process may lead to cycles between the nodes. For example, in Figure 5.8(c), node K_1 is made to point to node M and since M is referring to K_2 and K_2 is still referring to K_1, we have a temporary cycle between these nodes.

At the end of the adjustment process, all bucket entries still access M. To complete the hash creation process, the last operation is thus to remove M from all entries. For each bucket entry B, if M is on the head of B, then a CAS operation updating M to $Null$ is necessary. Otherwise, if M is not on the head of B, then we can simply mark as $Null$ the pointer of the node that is referring to M (nodes K_1 and K_4 in Figure 5.8(d)). This can be safely done without any CAS operation since no other thread can write on those nodes.

We complete the presentation of the LF_1 proposal with the description about how a hash table with a bucket array of size S is expanded to a new one with size $2 * S$. The decision of performing hash expansion is similar to the hash creation process. During the search, a local counter is used to count the number of nodes on a bucket entry which, in the case of a node insertion, is then used to verify the conditions for hash expansion (please refer to Section 5.1). In order to ensure that only one thread gains access to the hash expansion operation, we use a CAS operation to tag a specific field on HN. Figure 5.9 illustrates the hash expansion of Figure 5.8(d) after the insertion of a new node K_5 on the bucket entry Bn.

The thread that gains access to the hash expansion operation starts by creating a new hash H' of size $2 * S$ entries. Next, for each old bucket entry Bn, it recomputes the hash function for the nodes on Bn and redistributes them on H' accordingly to the new hash values. In particular, for our hash function, this means that a node on the nth entry of the old bucket array B (Bn on Figure 5.9) will be assigned to

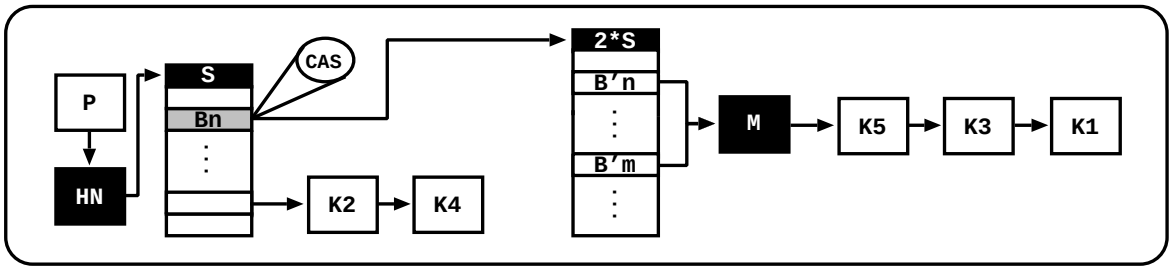

Figure 5.9: Expanding the hash tables in a trie level using LF_1

the nth or $(n + K)$th entry of H' (entries $B'n$ and $B'm$ on Figure 5.9). As before, we use again a marking node M to implement a synchronization point between the old bucket entry Bn and the new bucket entries $B'n$ and $B'm$ that, whenever both are synchronized, will correspond to a successful CAS operation that updates Bn to H' (situation illustrated on Figure 5.9). In the continuation, we follow the same adjustment process as before and, at the end, we remove M from $B'n$ and $B'm$. When the process of bucket expansion is completed for all S bucket entries, we update HN to point to the new hash H' (and remove simultaneously - same memory position - the tagging mark for hash expansion).

5.4.2 Implementation Details

We now present in more detail the algorithms that implement the key aspects of the LF_1 proposal. We begin with Algorithm 5.2 that shows the pseudo-code for the search/insert operation of a given key K in a hash node HN.

In a nutshell, the algorithm executes in a loop until one of the following situations occurs: (a) the search operation is successful, meaning that there is already a node in the trie level with the same key K (lines 19–21); or (b) a $newNode$ with key K is successfully inserted in the trie (lines 30–32).

In more detail, the algorithm starts by allocating a new node with the key K (lines 1–2), then it gets the hash H from the hash node HN, the size of the hash S and the bucket entry B in the hash where the key should be stored (lines 3-5). Then, the algorithm reads the reference R on B (line 9) and checks whether it references a chain node or a second hash. If the bucket entry B is referencing another hash (this happens when another thread is doing hash expansion). In such case, it moves to the new hash (variable H at line 11) and updates B (by recomputing the hash function using the key K), $oldFirst$, R and $markingNodeVisited$ accordingly (lines 13–16). The auxiliary

Algorithm 5.2 search_insert_key_on_hash(key K, hash node HN)

```
1:  newNode ← AllocNode()
2:  Key(newNode) ← K
3:  H ← GetHash(HN)
4:  S ← Size(H)
5:  B ← GetHashBucket(H, Hash(S, K))                    ▷ get the bucket entry
6:  markingNodeVisited ← False
7:  oldFirst ← Null
8:  repeat                          ▷ critical region only when CAS is executed
9:      R ← EntryRef(B)
10:     while IsHash(R) do                          ▷ R references a second hash
11:         H ← GetHash(R)
12:         S ← Size(H)
13:         B ← GetHashBucket(H, Hash(S, K))        ▷ get the next bucket entry
14:         markingNodeVisited ← False
15:         oldFirst ← Null
16:         R ← EntryRef(B)
17:     chain ← R                                   ▷ get the first node in the chain
18:     while chain ≠ Null and chain ≠ oldFirst do  ▷ traverse chain nodes
19:         if Key(chain) = K then                  ▷ key already exists
20:             FreeNode(newNode)
21:             return chain
22:         else if IsMarkingNode(chain) then
23:             if markingNodeVisited then          ▷ second time in the marking node
24:                 break
25:             else                                ▷ first time in the marking node
26:                 markingNodeVisited ← True
27:         chain ← NextRef(chain)
28:     if not IsMarkingNode(R) then                ▷ mark the last node visited
29:         oldFirst ← R
30:     NextRef(newNode) ← R
31: until CAS(EntryRef(B), R, newNode)
32: return newNode
```

variable $oldFirst$ marks the beginning of the chain of nodes on B that were already searched in a previous round and the auxiliary variable $markingNodeVisited$ denotes if the marking node was already visited.

On the second part of the algorithm, it then searches if there is a node with the same key K already in the chain (lines 17–27). Note that this search is done while the nodes in the chain were not yet searched in a previous round (while condition at line 18) and while the marking node was not visited twice (lines 22–26). This second condition allows to break any potential cycle between the nodes, as a result of a hash creation/expansion operation being done by another thread. Finally, if K is not found, the algorithm tries to insert $newNode$ on the bucket entry B by using a CAS operation that updates R to $newNode$ (line 31). In case of failure, this means that the head of B has changed in the meantime, thus leading to a new round.

Next, Algorithm 5.3 shows the pseudo-code for the hash expansion operation given a hash node HN (since it is quite similar, we will leave aside the algorithm for hash creation). Please remember that to ensure that only the elected thread executes the hash expansion operation for HN, we use a CAS operation to tag a specific field on HN (not shown here for the sake of simplicity).

The algorithm begins by initializing a set of local variables and by allocating a new bucket array (lines 1–5). Next, for each old bucket entry $oldB$, it redistributes the chain of nodes on $oldB$ to the corresponding bucket entries on the hash $newH$ (lines 7–20). At line 9, it executes a CAS operation on $oldB$ trying to update a value of $Null$ to $newH$. A successful CAS operation means that $oldB$ was empty and thus no redistribution is necessary (it just becomes a reference to the new hash). An unsuccessful CAS operation means that $oldB$ has nodes to be expanded. In such case, the algorithm then computes the entries on $newH$ in which the nodes from $oldB$ will fall (entries $newB1$ and $newB2$) and initializes them to refer to the marking node M (lines 10–13). The marking node M is then used to implement a synchronization point between the old bucket entry $oldB$ and the new bucket entries $newB1$ and $newB2$ that, whenever both are synchronized, will correspond to a successful CAS operation that updates $oldB$ to $newH$ (lines 14–16). In the continuation (lines 17–19), the algorithm proceeds by adjusting the nodes on the old chain (Algorithm 5.4 below) and by removing M from the $newB1$ and $newB2$ chains (Algorithm 5.5 below). At the end, when the process of bucket expansion is completed for all entries in $oldH$, HN is updated to point to the new bucket array $newH$ (line 21).

Algorithm 5.4 shows the pseudo-code for the process of adjusting a chain of nodes, starting from a given node N, into a given hash H. One can observe that the algorithm traverses the chain of nodes recursively and that the base case for recursion is the last node on the chain (lines 1–3). For each *chain* node, it then calculates the bucket entry B in which it will fall (lines 4–5). The bucket entry B is then used in repeated CAS

Algorithm 5.3 hash_expansion(hash node HN)

```
1:  M ← GetMarkingNode(HN)
2:  oldH ← GetHash(HN)
3:  oldS ← Size(oldH)                          ▷ get the size of the old bucket array
4:  newS ← 2 * oldS                            ▷ double the size for the new bucket array
5:  newH ← AllocInitHash(newS)
6:  i ← 0
7:  while i < oldS do                          ▷ traverse all entries in the old bucket array
8:      oldB ← GetHashBucket(oldH, i)
9:      if not CAS(EntryRef(oldB), Null, newH) then   ▷ the entry is not empty
10:         newB1 ← GetHashBucket(newH, i)
11:         newB2 ← GetHashBucket(newH, i + oldS)
12:         EntryRef(newB1) ← M
13:         EntryRef(newB2) ← M
14:         repeat          ▷ chain the head of the old entry with the marking node
15:             NextRef(M) ← EntryRef(oldB)
16:         until CAS(EntryRef(oldB), NextRef(M), newH)
17:         adjust_chain_nodes(M, newH)
18:         remove_marking_node(M, newB1)
19:         remove_marking_node(M, newB2)
20:     i ← i + 1
21: UPDATE(HashRef(HN), newH)                  ▷ set the hash node with the newH
22: return
```

Algorithm 5.4 adjust_chain_nodes(node N, hash H)

```
1:  chain ← NextRef(N)
2:  if NextRef(chain) ≠ Null then              ▷ N is not the last node in the chain
3:      adjust_chain_nodes(chain, H)
4:  S ← Size(H)
5:  B ← GetHashBucket(H, Hash(S, Key(chain)))
6:  repeat                                     ▷ insert the chain node in the head of the entry
7:      UPDATE(NextRef(chain), EntryRef(B))
8:  until CAS(EntryRef(B), NextRef(chain), chain)
9:  return
```

operations until successfully insert the *chain* node on the head of B (lines 6–8).

Finally, Algorithm 5.5 shows the pseudo-code for the operation of removing a given

marking node M from a given bucket entry B.

Algorithm 5.5 remove_marking_node(marking node M, bucket entry B)

```
1: if not CAS(EntryRef(B), M, Null) then   ▷ M is not in the head of the chain
2:     chain ← EntryRef(B)
3:     next ← NextRef(chain)
4:     while (next ≠ M) do                 ▷ traverse the chain until M is found
5:         chain ← next
6:         next ← NextRef(chain)
7:     UPDATE(NextRef(chain), Null)        ▷ remove M from the chain
8: return
```

Initially, Algorithm 5.5 executes a CAS operation on B trying to update an expected value M to $Null$. A successful CAS operation means that no nodes were adjusted to be on B (and B just becomes a reference to $Null$). An unsuccessful CAS operation means that at least one node was adjusted to be on B. In such case, the algorithm then follows the chain of nodes on B until reaching M and updates the node previous to M to point to $Null$ (thus removing M from the chain). This can be safely done without any CAS operation, because at this stage no other thread can be writing at this node.

5.4.3 Proof of Correctness

In this section, we discuss the correctness of the LF_1 proposal. For the sake of simplicity, we will keep the discussion in a particular trie level with a hash node HN already in place (the starting point will be Figure 5.8(b)), since, as described earlier, all trie levels and all chains of nodes have the same behavior. The proof consists in two parts: first we prove that the LF_1 proposal is *linearizable* and then we prove that the proposal is *lock-free* for the search and insert operations.

Linearizability is an important correctness condition, since it guarantees that if every operation within every algorithm that manipulates a concurrent data structure individually preserve an *invariant* (or a set of invariants), then the system as a whole also will. A operation is linearizable if it appears to take effect instantaneously at some moment of time t_{LP} between its invocation and response. The literature often refers to t_{LP} as a linearization point and, for lock-free implementations, a LP is typically a single instant where its effects become visible to all the remaining operations. The linearization proof has then three parts. On the first part, we enumerate the linearization points of the

proposal. On the second part, we describe the set of invariants that define a correct state of the concurrent trie data structure within the LF_1 proposal. On the third part, we prove that every linearization point *preserves* the set of invariants. We thus start by enumerating, the linearization points in the algorithms of the LF_1 proposal which are:

***LP*$_1$** Algorithm 5.2 (search_insert_key_on_hash) is linearizable at successful CAS in line 31.

***LP*$_2$** Algorithm 5.3 (hash_expansion) is linearizable at successful CAS in line 9.

***LP*$_3$** Algorithm 5.3 (hash_expansion) is linearizable at successful CAS in line 16:

***LP*$_4$** Algorithm 5.3 (hash_expansion) is linearizable at the UPDATE in line 21.

***LP*$_5$** Algorithm 5.4 (adjust_chain_nodes) is linearizable at the UPDATE in line 7.

***LP*$_6$** Algorithm 5.4 (adjust_chain_nodes) is linearizable at successful CAS in line 8.

***LP*$_7$** Algorithm 5.5 (remove_marking_node) is linearizable at successful CAS in line 1.

***LP*$_8$** Algorithm 5.5 (remove_marking_node) is linearizable at the UPDATE in line 7.

Next, we describe the set of invariants that must be *preserved* on every state of the LF_1 data structures:

***Inv*$_1$** The hash node HN refers always to a bucket array of entries H.

***Inv*$_2$** A bucket entry B must comply with the following semantics: (i) its initial reference refers to the marking node M; (ii) after an update, it must be referring to one of the following: $Null$, a node N or a hash H.

***Inv*$_3$** A node N_1 in a chain of nodes starting from a bucket entry B belonging to a bucket array of entries H must be referring always to one of the following: $Null$, another node N_2 or a marking node M.

***Inv*$_4$** A chain of nodes must always end with one of the following references: $Null$ or the marking node M.

***Inv*$_5$** The value of every concurrent memory location L that is **used for insertion of new structures** (nodes and bucket array of entries) refers only once to the same value V_1. Once it changes to another value V_2 it will never refer again to V_1.

Finally, on the third part of the linearization proof, we prove that every linearization point *preserves* the set of invariants.

Lemma 5.4.1. *In the initial state of the data structure the set of invariants hold.*

Proof. In the initial state the hash already have some chain nodes (please check Figure 5.8(b)). The hash node HN refers to a hash H, so Inv_1 holds, and all entries of H are referring to the marking node M, so Inv_2 also holds. The Inv_3 holds because the chain node K_2 is referring to the chain node K_1 and K_1 is referring to $Null$. The Inv_4 holds because the last node in the chain is K_1 which is referring to $Null$. The Inv_5 is not affected. □

Lemma 5.4.2. *The linearization point LP_1 preserves the set of invariants.*

Proof. After a successful CAS operation in the linearization point LP_1, a bucket entry B is updated to a new node $newNode$. Algorithm 5.2 shows that the node $newNode$ was allocated in the line 1, thus Inv_2 and Inv_5 hold. The Inv_4 also holds because if $newNode$ is the only node in the chain then it must be referring to $Null$ or to the marking node M (given by line 30), otherwise if $newNode$ is not the only node in the chain and since it was inserted in the entry of the chain, $newNode$ can not be at the end of the chain. The remaining invariants are not affected. □

Lemma 5.4.3. *The linearization point LP_2 preserves the set of invariants.*

Proof. After a successful CAS operation in the linearization point LP_2, a bucket entry $oldB$ is updated from $Null$ to a hash $newH$. Algorithm 5.3 shows that the $newH$ is allocated in the line 5, thus Inv_2 and Inv_5 hold. The remaining invariants are not affected. □

Lemma 5.4.4. *The linearization point LP_3 preserves the set of invariants.*

Proof. After a successful CAS operation in the linearization point LP_3, a bucket entry $oldB$ is updated from the node referred by the marking node to a hash $newH$. Algorithm 5.3 shows that the $newH$ is allocated in the line 5, thus Inv_2 and Inv_5 hold. The remaining invariants are not affected. □

Lemma 5.4.5. *The linearization point LP_4 preserves the set of invariants.*

Proof. After the UPDATE operation in the linearization point LP_4, the hash node HN is updated from an old hash to a new one $newH$. Algorithm 5.3 shows that the

newH is allocated in the line 5, thus Inv_1 and Inv_5 hold. The remaining invariants are not affected. □

Lemma 5.4.6. *The linearization point LP_5 preserves the set of invariants.*

Proof. After the UPDATE operation in the linearization point LP_5 the reference in a node *chain* is updated to refer to $EntryRef(B)$, which is necessarily another node or a marking node M, thus Inv_3 and Inv_4 hold. The Inv_5 holds because *chain*, where the reference is being updated, is not a memory location used for concurrent insertions of new data structures (remember that concurrent insertions are done only in the bucket entries). The remaining invariants are also not affected. □

Lemma 5.4.7. *The linearization point LP_6 preserves the set of invariants.*

Proof. After a successful CAS operation in the linearization point LP_6, a bucket entry B is updated from the reference given by $NextRef(chain)$ to the reference of node *chain*, thus Inv_2 holds. The Inv_5 also holds, because B was never referring to node *chain* before the update. This is ensured because only the elected thread (remember the election process in the beginning of this section using Figure 5.7) can be doing the adjustment process. In the adjustment process, each node is adjusted only once, and since B belongs to a new hash created by the elected thread, then B could not have referred to *chain* before the CAS operation. The remaining invariants are not affected. □

Lemma 5.4.8. *The linearization point LP_7 preserves the set of invariants.*

Proof. After a successful CAS operation in the linearization point LP_7, a bucket entry B is updated from the marking node M to $Null$, thus Inv_2 holds. The Inv_5 also holds, because Algorithm 5.5 is called from Algorithm 5.3 at lines 18 and 19, which means that the bucket entry B was initialized with the reference to the marking node M, so B never referred to $Null$ before the CAS operation. The remaining invariants are not affected. □

Lemma 5.4.9. *The linearization point LP_8 preserves the set of invariants.*

Proof. After the UPDATE operation in the linearization point LP_8 the reference of a node *chain* is updated to refer to $Null$, thus Inv_3 and Inv_4 hold. The Inv_5 holds, because the *chain* node, where the reference if being updated, is not a memory location used for concurrent insertions of new data structures, thus to *chain* is allowed to refer to $Null$ more than once. The remaining invariants are not affected. □

Corollary 5.4.1. *The invariants hold on every configuration of the LF_1 proposal due to Lemmas 5.4.1 to 5.4.9.*

Theorem 5.4.1. *The LF_1 proposal is linearizable.*

As described previously the lock-freedom property is very important because, although it allows individual threads to starve, it guarantees system-wide throughput. Using the notion of progress presented in the Subsection 5.2.2, we next prove that the LF_1 proposal is lock-free.

At the beginning of the section, we defined progress as the steps that a thread takes to complete the search/insert key operation (with the stages defined in the beginning of the section) in the data structures, i.e., the steps that a thread takes between the instant of the *call* and the instant of the *return* of the operation that it is calling. Formally, progress is given by: either (i) a thread searching for keys within the data structure and finding the key that it is searching between the two instants; or by (ii) a thread changing the configuration of the data structures through insertion of new nodes or optimizing the data structures though the expand and adjust procedures. As observed before, changes in the configuration of the data structure occur in the linearization points. Some of the linearization points use the CAS operation, which might fail in successfully update a memory position, and in those cases we will prove that progress still exists, because at least an other thread has made progress to the configuration of the data structure. The proof of lock-freedom of the LF_1 proposal has two parts, on the first part we discuss progression in the search_insert_key_on_hash, hash_expansion, adjust_chain_nodes, and remove_marking_node algorithms and in the second part we prove the lock-freedom property.

For the progress in the linearization points we have.

Lemma 5.4.10. *The execution of the operations defined by the linearization points LP_4, LP_5, and LP_8 lead to change in the configuration of data structure because such operations are composed by unconditional updates.*

Lemma 5.4.11. *When a thread executes the operations defined by the linearization points LP_1, LP_2, LP_3, LP_6 and LP_7 then configuration of the data structures has changed.*

Proof. All linearization points correspond to CAS operations on a given memory location L trying to update an initial reference I to a new reference R corresponding to an adjusted node, a new node or a new bucket array. Assuming that t_i is the

instant of time where a thread T first reads I from L and that t_f is the instant of time where T executes the CAS operation trying to update I to R, then a successful CAS execution leads to a change in the configuration of a data structure because L was updated to R. Otherwise, if the CAS operation fails, that means that between instants t_i and t_f, the reference on L was changed, which means that at least another thread has changed L between the instants of time t_i and t_f, thus also leading to a change in the configuration of the data structures. □

Corollary 5.4.2. *When a thread executes one of the linearization points LP_1 to LP_8 then, due to Lemma 5.4.10 and to Lemma 5.4.11, the state of the configuration of the data structure has made progress.*

For progress in the search and insert procedures, we prove that every key is only inserted once. To do so, we must prove that for a given key, if it exists in the data structure, then the algorithms are able to find it. Otherwise, if the key does not exist, then the algorithms are able to insert it. For the sake of clarity, we next introduce three lemmas: two for the search procedure, where we distinguish the cases where the expand procedure interferes or not with the search procedure, and one for the insert operation.

Lemma 5.4.12. *Consider Algorithm 5.2 with a given key K. Assuming that the insert and expand procedure do not interfere with the search operation, if K exists in the data structure, then the node with K is found, otherwise the algorithm moves from the search procedure to the insert procedure.*

Proof. Given the assumption above, the condition in line 10 of the Algorithm 5.2 fails, and thus the search for K begins in line 18 with *chain* and $oldFirst$ referring to the first node in the chain and to $Null$ (end of the chain), respectively. Next, at line 19, if the key of *chain* is equal to K then the node with K is found. Otherwise, the keys are different, the condition fails and the next condition at line 22 also fails, because no marking node can be found during the search (the marking node belongs to the expand procedure). Finally, the next node in the chain is visited (line 30) and the search procedure continues until either K is found or the *chain* is equal to $Null$ and the algorithm passes from the search procedure to the insert procedure. The invariant Inv_4 ensures that *chain* reaches $Null$ whenever a marking node is not present. □

Lemma 5.4.13. *Consider Algorithm 5.2 with a given key K. Assuming that the insert and expand procedure interfere with the search procedure, if K exists in the data structure, then the node with K is found, otherwise the algorithm moves from search procedure to the insert procedure.*

Proof. Given the assumption above the search procedure might be influenced in two situations:

- References to second hashes H or a marking node M might be found. If a reference to H if found, then the condition at line 10 in the Algorithm 5.2 is true, thus the search procedure traverses all second hashes until a reference to a chain of nodes is found (using lines 10 to 16). If a reference to M is found, then the key in M is always different from K, thus M is always skipped from the search.
- Cycled chains of nodes might be found, such as the one presented in Figure 5.8(c). Cycled chains of nodes do not end with a marking node M or $Null$. However, the invariant Inv_4 ensures that in any given instant one of both have to be present in a chain of nodes. Thus, to prove that the search procedures always ends and finds K if it is in the chain, we consider two situations:
  - If M is found then Algorithm 5.2 uses a Boolean $markingNodeVisited$ variable to detect if M was visited more than once (lines 23 and 24), and uses the $oldFirst$ to detect the reference that ends the search, i.e. the reference to the first node in the chain that was already visited. Thus, the algorithm passes from the search procedure to the insert procedure. During the search procedure, if K is the chain then it is found (proven by Lemma 5.4.12).
  - If $Null$ is found, then the condition at line 18 in Algorithm 5.2 fails, and the search procedure ends. Since K was not found, the algorithm advances to the insert procedure.

□

Lemma 5.4.14. *Consider Algorithm 5.2 with a given key K. If K does not exist, then K is inserted in the data structure and the insert procedure ends, otherwise the algorithm moves to the search procedure.*

Proof. Given the assumption above, consider that the bucket entry B marks the insertion point of the K and that R is the reference in B that was read in the initial instant T_i of the execution of the search procedure. In Lemma 5.4.12 and Lemma 5.4.13, we proved that the search procedure either finds a node with key K and ends at line 21 in Algorithm 5.2 or, otherwise, moves to the insert procedure without finding K. The insert procedure is executed in the instant T_f at line 31 in

Algorithm 5.2, using B, R and the new node to be inserted $newNode$. During the instants T_i to $T_f - 1$ the algorithm executed the search procedure. At instant T_f one of the two situations might have occurred:

- The reference in B is equal to R. The invariant Inv_5 ensures that nothing has changed in the chain, thus K is not in the chain of nodes and K is inserted in the data structure with the CAS operation.

- The reference in B is different from R. The invariant Inv_5 ensures that R must be $Null$, a node N or a hash H. In all situations the CAS operation will fail and the algorithm moves again to the search procedure.

□

Corollary 5.4.3. *On every instant of the execution of the search/insert key operation, at least one of the threads T executing the operation does progress.*

- *Corollary 5.4.2 shows that progress always exists in every linearization point, since if T has not made progress, then another thread has made progress in changing the configuration of the data structures.*

- *Lemma 5.4.12, Lemma 5.4.13 and Lemma 5.4.14, shows that progress always exists in the search and insert procedures because:*

  - *without the expand procedure, if T executes the search procedure with a key K, then T finds K if it is in the data structure, otherwise T successfully inserts K using the insert procedure or calls back the search procedure.*

  - *with the expand procedure, T might be or might not be the elected thread for the expansion. If T is the elected thread, then T progresses according with the changes that it does in data structures using the linearization points LP_2 to LP_8. If the CAS operation fails in some of the linearization points, then at least other thread as made progress in that particular linearization point. Otherwise, T is not the elected thread, thus it simply returns from the search/insert key operation.*

Theorem 5.4.2. *The LF_1 proposal is lock-free.*

5.4.4 Discussion

We have presented the LF_1 lock-free proposal specially aimed for environments that do not require the support for the delete operation. Our main motivation was to reduce the granularity of the previous lock-based proposal in order to be as efficient as possible in the concurrent search and insert operations and to maintain an efficient average node access as the size of the trie data structures increases, independently of the number of running threads. We discussed the relevant implementation details and proved the correctness of the implementation.

The LF_1 proposal implements a dynamic resizing of the hash tables by doubling the size of the bucket entries in the hash, whenever a trie level becomes saturated. Since the size of the hashes doubles, it is highly inappropriate to integrate this proposal with the TabMalloc memory allocator which requires the usage of fix-sized data structures and pages.

To implement lock-freedom we took advantage of the CAS operation which reduces the granularity of the synchronization when threads access concurrent areas. However, we observed that, by also implementing the LF_1 proposal in an external framework written in C, the LF_1 proposal suffers from problems such as false sharing or cache memory ping pong effects. We detected this behavior in four different x86 architectures: a 24-Core AMD Opteron(TM) Processor 8425 HE, a 32-Core AMD Opteron (TM) Processor 6274, a 64-Core AMD Opteron(TM) Processor 6376 and a 24-Core Intel Xeon(TM) Processor E5645. These problems are mostly due to the fact that nodes are inserted in bucket entries, and since bucket entries are close to each other they share the same cache lines in memory which causes false sharing and cache ping-pong overheads during execution.

5.5 Lock-Free Hash Trie - LF_2

This section presents the LF_2 proposal. This proposal is based on *hash tries* and is aimed to be a simpler and more efficient lock-free proposal that disperses the concurrent areas as much as possible in order to minimize problems such as false sharing or cache memory ping pong effects. In what follows, we describe the key ideas of this proposal, discuss important implementation details and we present a proof of correctness. At the end of the section, we describe how we have used the LF_2 proposal to deal with the consumer chain of nodes in the AT data structures.

Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash tables [15]. An essential property of the trie data structure is that common prefixes are stored only once [40], which in the context of hash tables allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions. To implement hashing inside the ST and AT data structures, thus use hash trie structures. In a nutshell, a hash trie is composed by *internal hash arrays* and *leaf nodes*. The leaf nodes store *key* values and the internal hash arrays implement a hierarchy of hash levels of fixed size 2^w. To map a *key* into this hierarchy, we first compute the hash value h for *key* and then use chunks of w bits from h to index the entry in the appropriate hash level. Hash collisions are solved by simply walking down the tree as we consume successive chunks of w bits from the hash value h.

The aim of the LF_2 proposal is then to be as effective as possible in heavily concurrent environments and improve the efficiency of the previous LF_1 proposal in three major aspects: (i) integration with the TabMalloc memory allocator, (ii) reduce false sharing and ping-pong effects, and (iii) improve liveness. With LF_2, the integration with the TabMalloc memory allocator, is now possible because, as we will observe, the size of the bucket arrays is fixed. To integrate LF_2 with TabMalloc, we have created a new queue of pages, for the pages holding the bucket array of entries.

For reducing false sharing and ping-pong effects, each bucket array of entries has the size of a line of cache of the hardware architecture that is supporting the execution. Additionally, we disperse the concurrent memory locations for insertion and the insert procedure is done now in the tail of the chain of nodes instead of the head. As a result, it is expected that, threads working in chains of nodes with bucket array entries that share the same cache line will interfere less with each other when compared with the LF_1 proposal.

Finally, for improving liveness, in the LF_1 proposal, only one thread T could be expanding a hash and the expanding procedure would be done by T in all bucket entries. The OS scheduler can affect the performance of the LF_1 since the preemption of T stops the expansion procedure which can potentially saturate all chains of nodes in all bucket entries. This might occur in heavily concurrent environments where other threads, different from T, insert an arbitrary large number of nodes during the preemption of T. Consequently, a chain of nodes could potentially be much more higher that the maximum number of nodes MAX_NODES that we have predefined has threshold. In the LF_2 proposal, we reduce the granularity of the expanding operation to a single bucket entry. Thus, if a thread T activates the expansion

operation, T is only responsible for expanding only one chain of nodes and that chain of nodes will have exactly the number of nodes given by the MAX_NODES threshold.

Figure 5.10 shows the progress stages that a thread passes by in the search and insert key operation, according with the decisions (oval boxes) that it has to do while executing the operation in the LF_2 proposal. We define three types of stages that specify the type of progress that a thread can make:

- The private stages that do not change the configuration of data structures (white rectangular boxes). A thread progresses in a private fashion, searching for keys in chain nodes or returning from the operation;
- The public stage where a thread *might or might not change* the configuration of data structures (gray rectangular box). In this stage a thread progresses if it successfully inserts a key in the data structure;
- The public stages that *must* change the configuration of data structures (black rectangular boxes). A thread progresses in this stage whenever it changes successfully the configuration of data structures.

The search/insert key operation of the LF_2 proposal begins with a search stage. In this stage, the thread searches for a key in the chain of nodes and if the key is found then the thread moves to the return stage. Otherwise, the key is not found and the thread passes to the election for optimization process. If elected, then the thread passes to the optimization stage, where it expands the current bucket array to a new one and adjusts all nodes to the new bucket array, and, at the end of the optimization process, it goes for a new election. If elected, it begins a new optimization process, otherwise, it returns to the search procedure. For the search procedure, if the thread does not find the key and it is not elected, it passes to the insert stage. In the insert stage, if the key is not inserted, then the thread moves again to the search stage. Otherwise, the key is inserted and the thread moves to the return stage.

5.5.1 Our Proposal By Example

This section presents our LF_2 proposal to support the concurrent search, insertion and expansion procedures inside the ST and AT data structures. Again, for the sake of simplicity, we are only considering one trie level of the ST and AT data structures. We begin with Figure 5.11 showing a small example that illustrates how the concurrent

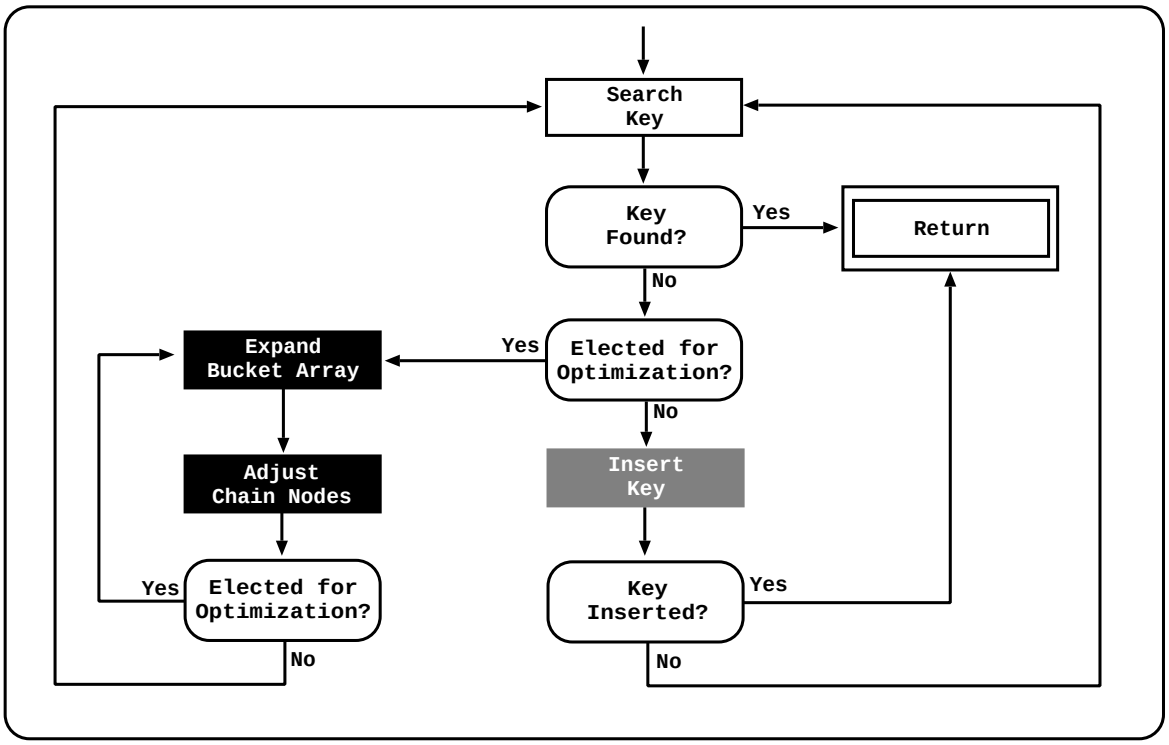

Figure 5.10: Progress stages of a thread in the search/insert key operation of the LF_2 proposal

insertion of nodes is done in a level of the ST and AT data structures starting with the initial configuration containing one hash level. We will use three examples to illustrate the different configurations that the hash trie assumes for one, two and three hash levels (for more hash levels, the same idea applies).

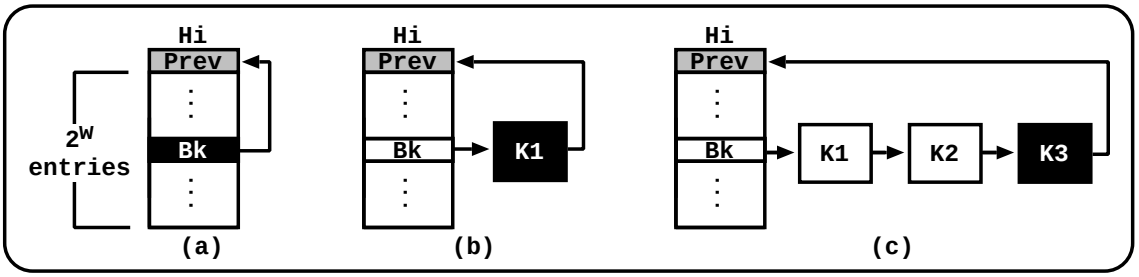

Figure 5.11: Insert procedure in a hash level

Figure 5.11(a) shows the initial configuration for a hash level. Each hash level H_i is formed by a bucket array of 2^w entries and by a backward reference to the previous hash level (represented as $Prev$ in the figures). For the root level, the backward reference is $Null$. In Figure 5.11(a), B_k represents a particular bucket entry of the hash level. B_k and the remaining entries are all initialized with a reference to the current level

H_i. During execution, each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of internal nodes, that deals with the hash collisions for that entry. Each internal node holds a *key* value and a reference to the next-on-chain internal node. Figure 5.11(b) shows the hash configuration after the insertion of node K_1 on the bucket entry B_k and Figure 5.11(c) shows the hash configuration after the insertion of nodes K_2 and K_3 also in B_k. Note that the insertion of new nodes is done at the end of the chain and that any new node being inserted closes the chain by referencing back the current level.

During execution, the different memory locations that form a hash trie are considered to be in one of the following states: *black*, *white* or *gray*. A black state represents a memory location that can be updated by any thread (concurrently). A white state represents a memory location that can be updated only by one (specific) thread (not concurrently). A gray state represents a memory location used only for reading purposes. As the hash trie evolves during time, a memory location can change between black and white states until reaching the gray state, where it is no further updated.

The initial state for B_k is black, because it represents the next synchronization point for the insertion of new nodes. After the insertion of node K_1, B_k moves to the white state and K_1 becomes the next synchronization point for the insertion of new nodes. To guarantee the property of lock-freedom, all updates to black states are done using CAS operations. Since we are using single word CAS operations, when inserting a new node in the chain, first we set the node with the reference to the current level and only then the CAS operation is executed to insert the new node in the chain.

When the number of nodes in a chain exceeds a MAX_NODES threshold value, then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. Thus, instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. To map our key values into this hierarchy, we use chunks of w bits from the hash values computed by our hash function. For example, consider a *key* value and the corresponding hash value h. For each hash level H_i, we use the $w * i$ least significant bits of h to index the entry in the appropriate bucket array, i.e., we consume h one chunk at a time as we walk down the hash levels. Starting from the configuration in Figure 5.11(c), Figure 5.12 illustrates the expansion mechanism with a second level hash H_{i+1} for the bucket entry B_k.

The expansion procedure is activated whenever a thread T meets the following two conditions: (i) the key at hand was not found in the chain and (ii) the number of nodes

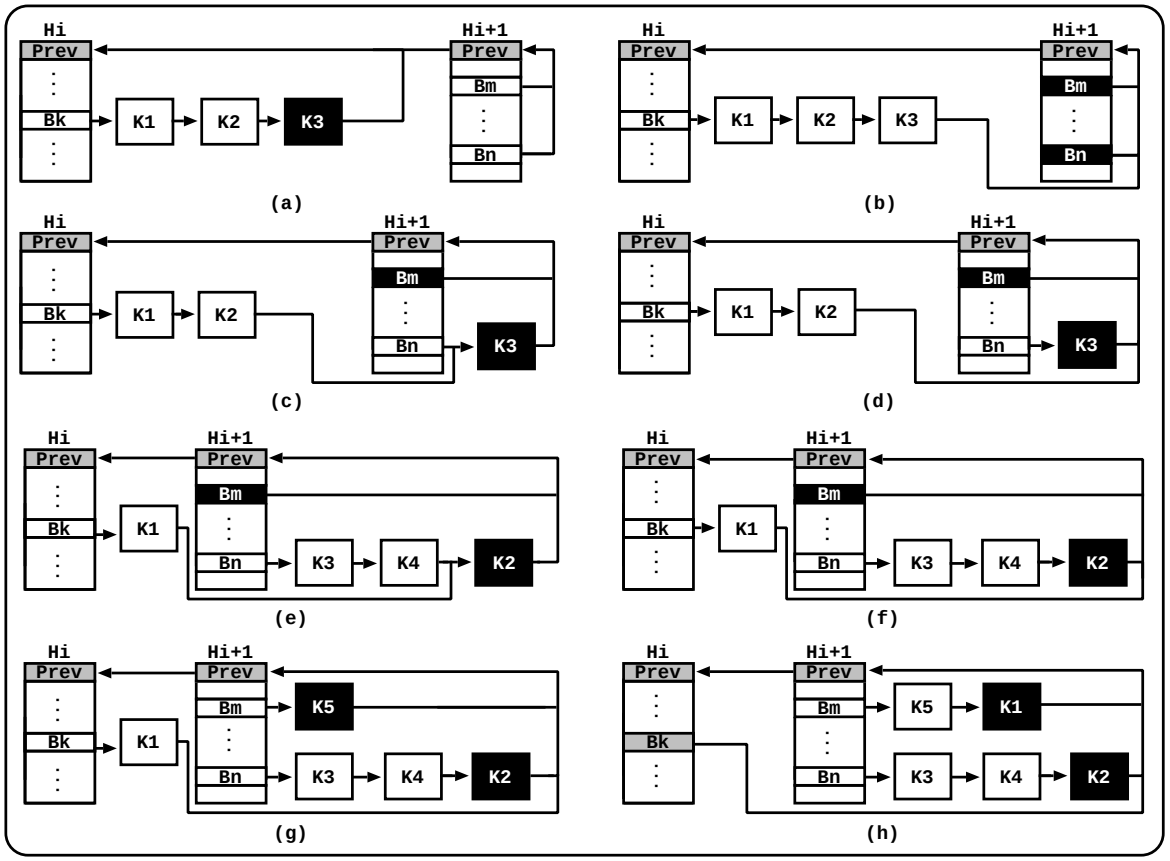

Figure 5.12: Expanding a bucket entry with a second level hash

in the chain is equal to the threshold value (in what follows, we consider a threshold value of three nodes). In such case, T starts by pre-allocating a second level hash H_{i+1}, with all entries referring the respective level (Figure 5.12(a)). At this stage, the bucket entries in H_{i+1} can be considered white memory locations, because the hash level is still not visible for the other threads. The new hash level is then used to implement a synchronization point with the last node on the chain (node K_3 in the figure) that will correspond to a successful CAS operation trying to update H_i to H_{i+1} (Figure 5.12(b)). From this point on, the insertion of new nodes on B_k will be done starting from the new hash level H_{i+1}.

If the CAS operation fails, that means that another thread has gained access to the expansion procedure and, in such case, T aborts its expansion procedure. Otherwise, T starts the remapping process of placing the internal nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 5.12(c) to 5.12(h) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries B_m and B_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to entries

B_m, B_n and B_n, respectively. In order to ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read all the available nodes and insert new nodes without any delay from the remapping process. To guarantee both properties, the remapping process is thus done in reverse order, starting from the last node on the chain, initially K_3.

Figure 5.12(c) then shows the hash trie configuration after the successful CAS operation that adjusted node K_3 to entry B_n. After this step, B_n moves to the white state and K_3 becomes the next synchronization point for the insertion of new nodes on B_n. Note that the initial chain for B_k has not been affected yet, since K_2 still refers to K_3. Next, on Figure 5.12(d), the chain is broken and K_2 is updated to refer to the second level hash H_{i+1}. The process then repeats for K_2 (the new last node on the chain for B_k). First, K_2 is remapped to entry B_n (Figure 5.12(e)) and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_{i+1} (Figure 5.12(f)). Finally, the same idea applies to the last node K_1. Here, K_1 is also remapped to a bucket entry on H_{i+1} (B_m in the figure) and then removed from the original chain, meaning in this case that the bucket entry B_k itself becomes a reference to the second level hash H_{i+1} (Figure 5.12(h)). From now on, B_K is also a gray memory location since it will be no further updated.

Concurrently with the remapping process, other threads can be inserting nodes in the same bucket entries for the new level. This is shown in Figure 5.12(e), where a new node K_4 is inserted before K_2 in B_n and, in Figure 5.12(g), where a node K_5 is inserted before K_1 in B_m. As mentioned before, lock-freedom is ensured by the use of CAS operations when updating black state memory locations.

To ensure the correctness of the remapping process, we also need to guarantee that the nodes being remapped are not missed by any other thread traversing the hash trie. Please remember that any chaining of nodes is closed by the last node referencing back the hash level for the node. Thus, if when traversing a chain of nodes, a thread U ends in a second level hash H_{i+1} different from the initial one H_i, this means that U has started from a bucket entry B_k being remapped, which includes the possibility that some nodes initially on B_k were not seem by U. To guarantee that no node is missed, U simply needs to restart its traversal from H_{i+1}.

We conclude the description of our proposal with a last example that shows a expansion procedure involving three hash levels. Starting from the configuration on Figure 5.12(b), Figure 5.13 assumes a scenario where a set of nodes (K_4, K_5, K_6 and K_7 in the figure) are inserted in the bucket entries B_m and B_n before the beginning

of the remapping process of nodes K_1, K_2 and K_3. Again, we will consider only the entries B_m and B_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to entries B_m, B_n and B_n, respectively.

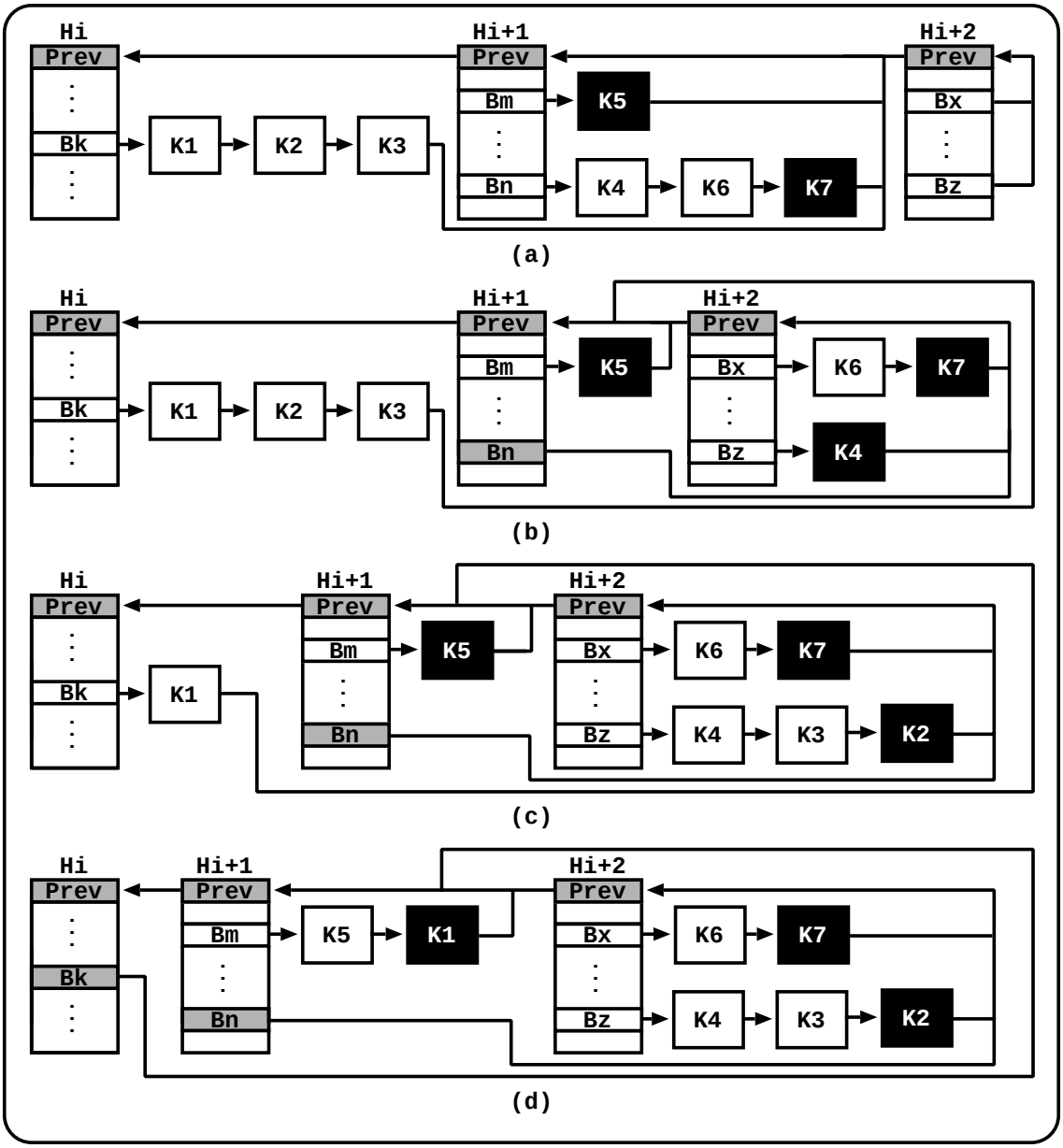

Figure 5.13: Adjusting nodes on a third level hash

Figure 5.13(a) shows the situation where K_3 is scheduled to be remapped to entry B_n on level H_{i+1} but, since the number of nodes on B_n is equal to the threshold value, a preliminary expansion procedure for B_n should be done, which leads to the pre-allocation of a third level hash H_{i+2}. Figure 5.13(b) then shows the hash trie configuration after the remapping of the nodes on B_n to the level H_{i+2}. Please note

that B_n became a gray state memory location since it is now referring the third level hash H_{i+2}, which means that any operation scheduled to B_n should be rescheduled to H_{i+2}. This is the case shown in Figure 5.13(c), where K_3 and K_2 were both rescheduled to entry B_z on H_{i+2}. Despite this third level remapping, the chaining reference of the last node on the chain (for example, K_1 in Figure 5.13(c)) is still made to refer to the second level hash H_{i+1}. To conclude the example, Figure 5.13(d) shows the configuration at the end of the remapping process. Here, K_1 is remapped to the bucket entry B_m on H_{i+1} and removed from the initial chain, meaning that B_k itself becomes a reference to H_{i+1} and moves to a gray state.

For each configuration shown, the reader is encourage to verify that, at any moment, all threads are able to access all available nodes. Consider, for example, the configuration shown in Figure 5.13(c) and a thread entering on level H_i searching for a node with the key K_7. The thread would begin by hashing the key K_7 on level H_i and obtain the bucket entry B_k. Then, it would follow the chain of nodes (K_1 in this case) and reach level H_{i+1}. At level H_{i+1}, it would hash again the key K_7, obtain the bucket entry B_n and follow the reference to level H_{i+2}. Finally, it would hash one more time the key K_7, now for level H_{i+2}, obtain the entry B_x and follow the chain until reaching node K_7.

We argue that a key design decision in the LF_2 proposal is thus the combination of hash tries with the use of a separate chaining (with a threshold value) to resolve hash collisions. Also, to ensure that nodes being remapped are not missed by any other thread traversing the hash trie, any chaining of nodes is closed by the last node referencing back the hash level for the node, which allows to detect the situations where a node changes level. This is very important because it allows to implement a clean design to resolve hash collisions by simply moving nodes between the levels. In this proposal, updates and expansions of the hash levels are never done by using replacement of data structures (i.e., create a new one to replace the old one), which also avoids the complex mechanisms necessary to support the recovering of the unused data structures. Another important design decision which minimizes the low-level synchronization problems leading to false sharing or cache memory side-effects, is the insertion of nodes done at the end of the separate chain. Inserting nodes at the end of the chain allows for dispersing as much as possible the memory locations being updated concurrently (the last node is always different) and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels (each bucket entry is at most only updated twice).

5.5.2 Implementation Details

This section presents the algorithms that implement our new lock-free hash trie proposal. We begin with Algorithms 5.6 and 5.7 that show the pseudo-code for the search/insert operation of a given key K in a hash level H. In a nutshell, the algorithms execute recursively, moving through the hierarchy of hash levels until K is found or inserted in a hash level H (for the entry call, H is the root level). Algorithm 5.6 deals with the hash level data structures and Algorithm 5.7 deals with the internal nodes in a separate chaining.

Algorithm 5.6 search_insert_key_on_hash(key K, hash H)

```
1:  B ← GetHashBucket(H, Hash(Level(H), K))
2:  if EntryRef(B) = H then                          ▷ B is an empty bucket
3:      newNode ← AllocNode()
4:      Key(newNode) ← K
5:      NextRef(newNode) ← H
6:      if CAS(EntryRef(B), H, newNode) then
7:          return newNode
8:      else
9:          FreeNode(newNode)
10: R ← EntryRef(B)
11: if IsNode(R) then                          ▷ start traversing the chain
12:     return search_insert_key_on_chain(K, H, R, 1)
13: else                                       ▷ R references a second level hash
14:     return search_insert_key_on_hash(K, R)
```

In more detail, Algorithm 5.6 starts by applying the hash function that allows obtaining the appropriate bucket entry B of H that fits K (line 1). Next, if B is empty (i.e., if B is referencing back the hash level H), then a new node $newNode$ representing K is allocated and properly initialized (lines 3–5). Then, the algorithm tries to insert K on the head of B by using a CAS operation that updates H to $newNode$ (line 6). If the operation is successful, then the node was successfully inserted and the algorithm ends by returning it (line 7). Otherwise, in case of failure, the head of B has changed in the meantime, so B is not empty (lines 10–14). Here, the algorithm then reads the reference R on B (line 10) and checks whether it references an internal node or a second hash level. If R is a node, then it calls Algorithm 5.7 to traverse the chain of nodes (line 12). Otherwise, it calls itself, but now for the second level hash represented by R (line 14).

Algorithm 5.7 shows the search/insert operation of a given key K in a hash level H starting from a reference R (which is a chain node) at position C in a separate chaining (for the entry call, C is 1 and R is the head node in the chain). Initially, the algorithm simply checks if R holds the key K, in which case, it ends by returning R (lines 1–2). Otherwise, it checks if R is the last node in the chain (line 3). If so, then two situations might occur: (i) the chain is full, in which case, the expansion procedure should be activated (lines 5–13); or (ii) the chain is not full, in which case, a new node representing K should be inserted in the chain (lines 15–21).

For the former situation, a second level hash $newHash$ is first allocated and initialized (lines 5–6) and then used to implement a synchronization point that will correspond to a CAS operation trying to update the next reference of R from H to $newHash$ (line 7). If the CAS operation fails, that means that another thread has gained access to the expansion procedure. Otherwise, if successful, the algorithm starts the remapping process of adjusting the internal nodes on the separate chaining, corresponding to the bucket entry B at hand, to the new hash level (line 9) and, for that, it calls the *adjust_chain_nodes*() procedure (see Algorithm 5.8 next). After that, it updates the bucket entry B to refer to the new level (line 10) and then Algorithm 5.6 is called again, this time to search/insert for K in the new hash level (line 11).

For the latter situation (lines 15–21), a new node representing K is allocated and properly initialized (lines 15–17), and a CAS operation tries to insert it at the end of the chain. If successful, the reference to the new node is returned. Otherwise, this means that another thread has inserted another node in the chain in the meantime, which lead us to the situation in the last block of code (lines 22–28), where R is not last in the chain.

In the last block of code, the algorithm then updates R to the next reference in the chain (line 22) and, as in Algorithm 5.6, it checks whether R references an internal node or a second hash level. If R is still a node, then the algorithm calls itself to continue traversing the chain of nodes (line 24). Otherwise, it returns to Algorithm 5.6, but now for the hash level after the given hash H (lines 26–28). Note that, if other threads are simultaneously expanding the hash tries, it might happen that we end in a hash level several levels deeper and thus incorrectly miss the node we are searching for. This is why we need to move backwards to the hash level after the given hash H (lines 26–27).

Algorithms 5.8, 5.9 and 5.10 show the pseudo-code for the remapping process of adjusting a chain of nodes to a new hash level H. Algorithm 5.8 is the entry procedure

Algorithm 5.7 search_insert_key_on_chain(key K, hash H, reference R, counter C)

```
1: if Key(R) = K then                                   ▷ we have found K in the chain
2:     return R
3: if NextRef(R) = H then                               ▷ R is last in the chain
4:     if C = MAX_NODES then                            ▷ chain is full
5:         newHash ← AllocInitHash(Level(H) + 1)
6:         PrevHash(newHash) ← H
7:         if CAS(NextRef(R), H, newHash) then
8:             B ← GetHashBucket(H, Hash(Level(H), K))
9:             adjust_chain_nodes(EntryRef(B), newHash)
10:            UPDATE(EntryRef(B), newHash)
11:            return search_insert_key_on_hash(K, newHash)
12:        else
13:            FreeHash(newHash)
14:    else
15:        newNode ← AllocNode()
16:        Key(newNode) ← K
17:        NextRef(newNode) ← H
18:        if CAS(NextRef(R), H, newNode) then
19:            return newNode
20:        else
21:            FreeNode(newNode)
22: R ← NextRef(R)
23: if IsNode(R) then                                   ▷ keep traversing the chain
24:    return search_insert_key_on_chain(K, H, R, C + 1)
25: else                                                ▷ R references a second level hash
26:    while PrevHash(R) ≠ H do                         ▷ move backwards
27:        R ← PrevHash(R)
28:    return search_insert_key_on_hash(K, R)
```

that ensures that the remapping process is done in reverse order, Algorithm 5.9 deals with the adjustment on hash level data structures and Algorithm 5.10 deals with the adjustment on a separate chaining.

In more detail, Algorithm 5.8 starts by traversing the nodes in the chain until reaching the last one. Then, for each node R in the chain (from last to first), it calls *adjust_node_on_hash()* in order to remap R to the given new hash level H.

Algorithm 5.8 adjust_chain_nodes(reference R, hash H)

```
1: if NextRef(R) ≠ H then
2:     adjust_chain_nodes(NextRef(R), H)
3: adjust_node_on_hash(R, H)
4: return
```

Algorithm 5.9 shows the pseudo-code for the process of remapping a given node N into a given hash H. It is quite similar to Algorithm 5.6, except for the fact that there is no need to allocate and initialize a new node with the key at hand (here, we already have the node). It starts by updating the next reference of N to H (line 1), next it applies the hash function that allows obtaining the appropriate bucket entry B of H that fits the key on N (line 2), and then, if B is empty, it tries to successfully insert N on the head of B by using a CAS operation (lines 3–5). Otherwise, B is not empty, and the same procedure as in Algorithm 5.6 applies (lines 6–10). The difference is that here it calls the *adjust_node_on_chain()* and *adjust_node_on_hash()* algorithms, instead of the *search_insert_key_on_chain()* and *search_insert_key_on_hash()* algorithms.

Algorithm 5.9 adjust_node_on_hash(node N, hash H)

```
1: UPDATE(NextRef(N), H)
2: B ← GetHashBucket(H, Hash(Level(H), Key(N)))
3: if EntryRef(B) = H then                          ▷ B is an empty bucket
4:     if CAS(EntryRef(B), H, N) then
5:         return
6: R ← EntryRef(B)
7: if IsNode(R) then                                ▷ start traversing the chain
8:     return adjust_node_on_chain(N, H, R, 1)
9: else                                             ▷ R references a second level hash
10:    return adjust_node_on_hash(N, R)
```

Algorithm 5.10 then concludes the presentation. It shows the pseudo-code for the process of remapping a given node N into a hash level H starting from a node R at position C in a separate chaining. As before, Algorithm 5.10 also shares similarities, but now with Algorithm 5.7, except for the fact that there is no need to check if N already exists in the chain (lines 1–2 in Algorithm 5.7) and, as before, that there is no need to allocate and initialize a new node with the key at hand (lines 15–17 in Algorithm 5.7). The last block of code (lines 14–20) is also identical to Algorithm 5.7, except for the fact that it calls the *adjust_node_on_chain()* and *adjust_node_on_hash()* algorithms, instead of the *search_insert_key_on_chain()* and *search_insert_key_on_hash()* algorithms.

Algorithm 5.10 adjust_node_on_chain(node N, hash H, reference R, counter C)

```
1: if NextRef(R) = H then                                   ▷ R is last in the chain
2:   if C = MAX_NODES then                                  ▷ chain is full
3:     newHash ← AllocInitHash(Level(H) + 1)
4:     PrevHash(newHash) ← H
5:     if CAS(NextRef(R), H, newHash) then
6:       B ← GetHashBucket(H, Hash(Level(H), K))
7:       adjust_chain_nodes(EntryRef(B), newHash)
8:       UPDATE(EntryRef(B), newHash)
9:       return adjust_node_on_hash(N, newHash)
10:    else
11:      FreeHash(newHash)
12:  else if CAS(NextRef(R), H, N) then
13:    return
14: R ← NextRef(R)
15: if IsNode(R) then                                       ▷ keep traversing the chain
16:   return adjust_node_on_chain(N, H, R, C + 1)
17: else                                                    ▷ R references a second level hash
18:   while PrevHash(R) ≠ H do                              ▷ move backwards
19:     R ← PrevHash(R)
20:   return adjust_node_on_hash(N, R)
```

5.5.3 Proof of Correctness

In this section, we discuss the correctness of the LF_2 proposal. For the sake of simplicity, we will keep the discussion similar to the proof of the LF_1 proposal, thus consisting in two parts: first we prove that the LF_2 proposal is *linearizable* and then we prove that the proposal is *lock-free* for the search and insert operations.

Again, the linearization proof has then three parts. On the first part, we describe the linearization points of the LF_2 proposal. On the second part, we describe the invariants that define a correct state of the concurrent trie data structure within the LF_2 proposal. On the third part, we prove that every linearization point *preserves* a set of invariants.

The linearization points in the algorithms of the LF_2 proposal are:

***LP*$_1$** Algorithm 5.6 (search_insert_key_on_hash) is linearizable at successful CAS in

line 6.

***LP*$_2$** Algorithm 5.7 (search_insert_key_on_chain) is linearizable at successful CAS in line 7.

***LP*$_3$** Algorithm 5.7 (search_insert_key_on_chain) is linearizable at the UPDATE in line 10.

***LP*$_4$** Algorithm 5.7 (search_insert_key_on_chain) is linearizable at successful CAS in line 18.

***LP*$_5$** Algorithm 5.9 (adjust_node_on_hash) is linearizable at the UPDATE in line 1.

***LP*$_6$** Algorithm 5.9 (adjust_node_on_hash) is linearizable at successful CAS in line 4.

***LP*$_7$** Algorithm 5.10 (adjust_node_on_chain) is linearizable at successful CAS in line 5.

***LP*$_8$** Algorithm 5.10 (adjust_node_on_chain) is linearizable at the UPDATE in line 8.

***LP*$_9$** Algorithm 5.10 (adjust_node_on_chain) is linearizable at successful CAS in line 12.

The set of invariants that must be *preserved* on every state of the data structure are:

***Inv*$_1$** For every hash level H, $PrevHash(H)$ always refers to the previous hash level.

***Inv*$_2$** A bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial reference is H; (ii) after the first update, it must refer to a node N; (iii) after the second (and final) update, it must refer to a hash level H_d such that $PrevHash(H_d) = H$.

***Inv*$_3$** A node N in a chain of nodes starting from a bucket entry B belonging to a hash level H must comply with the following semantics: (i) its initial reference is H; (ii) after an update, it must refer to another node in the chain or to a hash level H_d (at least one level) deeper than H.

***Inv*$_4$** For a chain of nodes in a bucket entry B belonging to a hash level H, the number C of nodes in the chain is always lower or equal than a predefined threshold value MAX_NODES ($MAX\_NODES \geq 1$).

***Inv*$_5$** The value of every concurrent memory location L that is used for insertion of new structures (nodes and bucket array of entries) refers only once to the same value V_1. Once it changes to another value V_2 it will never refer again to V_1.

Next, we prove that every linearization point *preserves* the set of invariants.

Lemma 5.5.1. *In the initial state of the data structure the set of invariants hold.*

Proof. Consider that H represents the root level for a hash trie (its initial configuration is the same as the one represented in Figure 5.11(a)). Since H is the root level, the reference $PrevHash(H)$ is $Null$ (Inv_1), each bucket entry B is referring H (Inv_2) and the number C of nodes in any chain is 0 (Inv_3 and Inv_4). The invariant Inv_5 is not affected. □

Lemma 5.5.2. *The linearization point LP_1 preserves the set of invariants.*

Proof. After the successful execution of the CAS operation at line 6, the bucket entry B refers to $newNode$ (Inv_2), $newNode$ refers to H (Inv_3), as initialized at line 5, and $C = 1$ (Inv_4). Inv_5 holds because $newNode$ is allocated at line 3 of Algorithm 5.6, thus it represents a new memory location. Inv_1 is not affected. □

Lemma 5.5.3. *The linearization point LP_2 preserves the set of invariants.*

Proof. After the successful execution of the CAS operation at line 7, the node R refers to a deeper hash level $newHash$ (Inv_3) and $PrevHash(newHash)$ refers to the current hash level H, as initialized at line 6 (Inv_1). Inv_5 holds because $newHash$ is allocated at line 5 of Algorithm 5.7, thus it represents a new memory location. Inv_2 and Inv_4 are not affected. □

Lemma 5.5.4. *The linearization point LP_3 preserves the set of invariants.*

Proof. After the UPDATE operation in the linearization point LP_3, a bucket entry reference B is updated to refer to a $newHash$, which is a new memory location, thus Inv_5 holds because for the linearization point LP_3 be executed, then the condition $C = MAX\_NODES$ must be true, which means that B was referring a chain node (Inv_2). The remaining invariants are not affected. □

Lemma 5.5.5. *The linearization point LP_4 preserves the set of invariants.*

Proof. After the successful execution of the CAS operation at line 18, the node R refers to $newNode$ and $newNode$ refers to H, as initialized at line 17 (Inv_3). Inv_4 also holds because since the condition at line 4 failed, meaning that initially $C_i < MAX\_NODES$, the insertion of a new node in the chain after R leads to $C_f = C_i + 1 \leq MAX\_NODES$. Inv_5 holds because $newNode$ is allocated at line 15 of

Algorithm 5.7, thus it represents a new memory location. Inv_1 and Inv_2 are not affected. □

Lemma 5.5.6. *The linearization points LP_5, LP_6, LP_7, LP_8 and LP_9 preserve the set of invariants.*

Proof. The linearization points LP_5, LP_6, LP_7, LP_8 and LP_9 belong to the optimization procedure of adjusting a chain of nodes to a new hash level. This procedure is initially called at line 9 of Algorithm 5.7, for a chain of nodes in the bucket entry B and for the deeper hash level $newHash$ with the previous hash reference updated to the previous hash level (line 6) and B is updated to refer to the $newHash$ (line 10), thus Inv_1 and Inv_2 hold. The adjust_chain_nodes() algorithm then calls Algorithms 5.9 and 5.10. In Algorithm 5.9, at line 1, the node N being adjusted is made to refer to a deeper hash level, thus Inv_3 holds. During the adjustment procedure, the number of nodes is always lower or equal than a predefined threshold value MAX_NODES (line 2 at Algorithm 5.10), thus Inv_4 holds. Finally, every linearization point update memory locations L to newly allocated bucket array structures or to chain nodes that were never referred by L, thus Inv_5 holds. For the remaining parts of Algorithms 5.9 and 5.10, the proofs are similar to the proofs for Algorithms 5.6 and 5.7, as shown on the previous lemmas. Thus, the invariants still hold for Algorithms 5.9 and 5.10. □

Corollary 5.5.1. *The invariants hold on every configuration of the LF_2 proposal due to Lemmas 5.5.1 to 5.5.6.*

Theorem 5.5.1. *The LF_2 proposal is linearizable.*

The proof of lock-freedom of the LF_2 proposal has two parts, on the first part we discuss progression in the search_insert_key_on_hash(), search_insert_key_on_chain(), adjust_chain_nodes(), adjust_node_on_hash() and adjust_node_on_chain() algorithms and in the second part we prove the lock-freedom property.

For the progress in the linearization points we have.

Lemma 5.5.7. *The execution of the operations defined by the linearization points LP_3, LP_5 and LP_8 lead to progress in the configuration of the data structures because such operations are composed by unconditional updates.*

Lemma 5.5.8. *When a thread executes the operations defined by the linearization points LP_1, LP_2, LP_4, LP_6, LP_7 and LP_9 then the configuration of the data structure has made progress.*

Proof. All linearization points correspond to CAS operations on a given memory location M trying to update an initial reference to a hash level H with a reference R corresponding to a new node or hash level. Assuming that t_i is the instant of time where a thread T first reads H from M and that t_f is the instant of time where T executes the CAS operation trying to update H to R, then a successful CAS execution leads to progress in the state of the hash trie configuration because M was updated to R. Otherwise, if the CAS operation fails, that means that between instants t_i and t_f, the reference on M was changed, which means that at least another thread has changed M between the instants of time t_i and t_f, thus leading to progress in the state of the hash trie configuration. □

Corollary 5.5.2. *When a thread executes one of the linearization points LP_1 to LP_9 then, due to Lemmas 5.5.7 to 5.5.8, the state of the configuration of the data structure has made progress.*

For progress in the search and insert operation, we prove that every key is only inserted once. To do so, we must prove that for a given key K, if K exists in the hash trie, then the algorithms are able to find it. Otherwise, if K does not exist, then the algorithms are able to insert it.

Lemma 5.5.9. *Consider Algorithm 5.6 with a given key K and a hash level H. If K exists in a chain of nodes in a hash level deeper than H, then Algorithm 5.6 computes the next hash level H_d where K can be found, and calls itself for H_d. When K exists in a chain of nodes in H, then Algorithm 5.6 maps K to the correct bucket B of H that holds K and calls Algorithm 5.7 to search for K in the separate chaining of B.*

Proof. Since we are assuming that K already exists in a chain of nodes, the code between lines 2–9 can be ignored because the condition at line 2 is always false. If R is then a reference to a hash trie, the algorithm calls itself for the next hash level (as defined by Inv_2) and the process continues recursively until the condition at line 11 be true. At that stage, Algorithm 5.7 is called to search for K starting from the first node R in the corresponding separate chaining B of H. □

Lemma 5.5.10. *Consider Algorithm 5.7 with a given key K, a hash level H and a reference to the first node R in a chain of nodes. If K exists in the chain, then Algorithm 5.7 finds the node with K.*

Proof. Since we are assuming that K already exists in a chain of nodes, the code between lines 3–21 can be ignored because the condition at line 3 is always false. If

the condition at line 1 succeeds then K was found in the chain. Otherwise, if the chain is not being remapped to a second hash level, the algorithm uses the lines 22–24 to call itself recursively until it finds K at line 1. If the chain is being remapped, Inv_3 ensures that we will reach a reference to a hash level H_d which is deeper than H. Thus, at some point in the execution, the algorithm reads H_d at line 22, calling Algorithm 5.6 in the continuation with a hash level H_a one level deeper than H (not that H_d can be in a deeper level than H_a). The search process then continues using Lemma 5.5.9. Since K exists and was not found yet, Algorithm 5.7 will be called again, this time for H_a or for a deeper level and the process will be repeated until K be found in a node. □

Lemma 5.5.11. *If a given key K does not exist in the hash trie, then it will be inserted in the linearization points LP_1 or LP_4.*

Proof. Since we are assuming that K does not exist in the hash trie, then the search procedure will necessarily end when it finds an empty bucket entry (line 2 in Algorithm 5.6) or when it reaches the last node in a chain of nodes not being remapped (line 3 in Algorithm 5.7). If the CAS operation at line 6 for Algorithm 5.6 (LP_1) or at line 18 for Algorithm 5.7 (LP_4) then succeeds, a new node with the key K was inserted in the hash trie. Otherwise, in case of CAS failure, the separate chaining at hand was changed by another thread T in the meantime. In particular, it could happen that T had inserted a node for K. The search process is then resumed and if K was inserted by another thread then, using Lemmas 5.5.9 and 5.5.10, Algorithm 5.7 will find it. Otherwise, the search process will end again in the lines mentioned above until K be successfully inserted in the hash trie. □

Corollary 5.5.3. *Progress exists in every search and insert key operation because:*

- *Corollary 5.5.2 shows that progress always exists in every linearization point even if a particular thread T does not progress another thread has made progress.*
- *Lemma 5.5.9, Lemma 5.5.10 and Lemma 5.5.11, shows that progress always exists in every search and insert operation because:*
  - *if T executes a search and insert operation with a given key K then, the thread finds K if it exists in the data structure (T progresses).*
  - *otherwise, if K does not exist, then T inserts it in the data structure (T progresses because the configuration of the data structure changes).*

– *in both cases, T might also expand the hash and in that cases, T changes the configuration of the data structure and consequently T progresses.*

Theorem 5.5.2. *The LF_2 proposal is lock-free.*

5.5.4 Private Consumer Chaining

In this subsection, we describe how we have used the data structures of LF_2 proposal to improve the performance of the FS design when dealing with the chain of nodes representing tabled answers. We named this procedure Private Consumer Chaining (PCC). The key idea is similar to the idea proposed by Costa and Rocha [31] for the global trie data structure, where the answers are represented only once on a global trie and then each subgoal call has private pointers to its set of answers. With the PCC procedure we apply the same key idea of representing only once each answer (as given by the FS design), but now since we are in a multithreaded environment, we use a private chain of answers per thread to represent the answers for each subgoal call.

As we have shown in the previous chapters, the FS design minimizes memory usage. On the other hand, it showed overheads in the execution time mainly due to the synchronization that it needs whenever one thread is updating an AT data structure and other threads are also updating/accessing the same answer trie data structure. Throughout a deeper study we have identified that most of the FS overheads are caused by the consumer chaining in the answer trie data structure. As explained in Figure 2.13, the consumer chaining is used to chain the answers to be consumed by the consumer nodes. This chaining procedure takes place whenever a new answer is found for a call, and since it is done inside the AT, it requires synchronization at the write level in multithreaded environments. The key idea of the PCC is then to improve the behavior in the execution time of the FS design, by moving the chaining procedure from public to private, i.e., we remove the chaining procedure from the answer trie and we moved it to a private procedure that only affects the thread that is doing it. At the end, when the evaluation is complete, i.e, when a subgoal call is marked as complete, we put one of the private chains as public, so that from that point on all threads can use that chain in complete (only reading) mode.

Figure 5.14 shows the key data structures for supporting the implementation of the PCC procedure during the evaluation of a tabled subgoal call $P_{i.j}$ using the FS design. The FS design uses a subgoal entry data structure to store common information for a subgoal call and a subgoal frame (SF) data structure to store private information

about the execution of each thread. The PCC procedure works at the subgoal frame level, since its aim is to avoid the sharing problems described above.

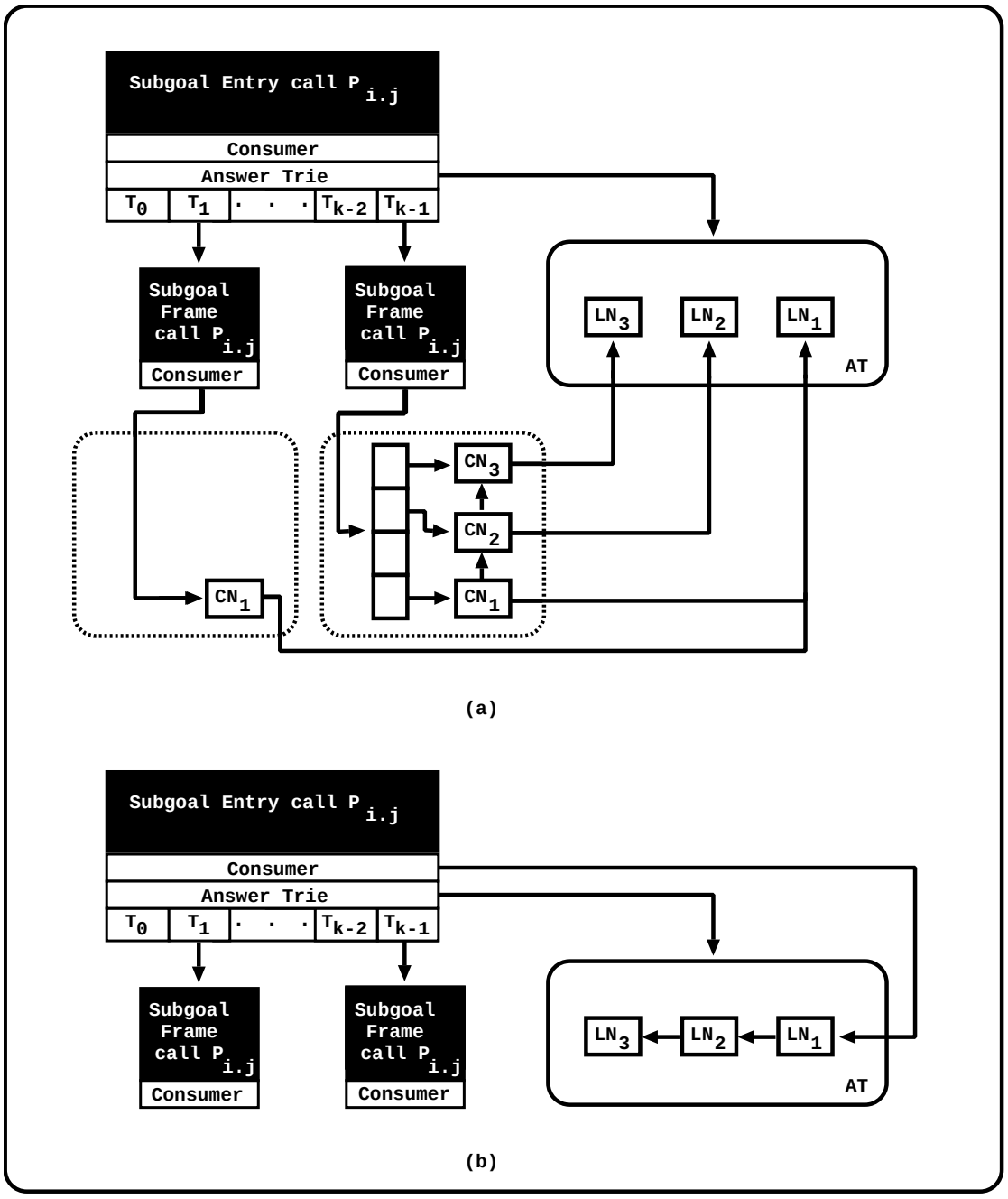

Figure 5.14: The FS design with the PCC optimization - (a) private chaining and (b) public chaining

Figure 5.14(a) shows then a situation where two threads, T_1 and T_{k-1}, are sharing the same subgoal entry call $P_{i.j}$ when the subgoal is still under evaluation, i.e., the subgoal is not yet complete. The current state of the evaluation shows an answer trie

(AT) with 3 answers found for the subgoal call $P_{i.j}$. For the sake of simplicity, we are omitting the internal AT nodes and we are only showing the leaf nodes on the AT data structure, nodes LN_1, LN_2 and LN_3, in the figure. With the PCC optimization, the leaf nodes are no longer chained in the AT data structure. Now, the chaining process is done privately, and for that, we use the SF structure of each thread. On the SF structure we added a new field, called *consumer*, to store the answers found within the execution of the thread. In order to minimize the impact of the PCC optimization, each node within the new consumer answer structure has two fields: (i) an entry pointer, which points to the corresponding leaf node in the AT data structure; (ii) a next pointer to chain the answers within the consumer structure. To maintain a good performance, when the number of nodes exceeds a certain threshold, we use a hash trie mechanism design similar to the LF_2 proposal. However, since this mechanism is private to each thread, it did not requires any of the tools that were necessary to support lock-freedom. In particular, on each hash trie level, we have removed the previous pointer from the hashes and from the nodes within the separate chaining mechanism and, for writing, we not use the CAS operation. We have chosen the LF_2 proposal instead of the LF_1, because the LF_2 proposal showed better balance between lookup and insert operations [10, 11, 14], but the major reason was mostly because of the integration in the TabMalloc memory allocator.

Going back to Figure 5.14(a), the consumer answer structures represent then two different situations where threads can be evaluating a subgoal call. Thread T_1 has only found one answer and it is using a direct consumer answer chaining to access the node LN_1. Thread T_{k-1} was already found three answers for the subgoal call and it is already using the hash trie mechanism within its consumer answer structure. The consumer nodes are chained between themselves, thus that consumer nodes belonging to thread T_{k-1} can consume the answers as in the original mechanism.

Figure 5.14(b) shows the state of the subgoal call after completion (recall that after completion of a subgoal call, the threads use loader nodes to consume the answers). When a thread T completes a subgoal call, it frees its private consumer structures, but before doing that, it checks whether another thread as already marked the subgoal as completed. If no other thread has done that, then thread T not only follows its private chaining mechanism as it would for freeing its private nodes, but also, follows the pointers to the answer trie leaf nodes in order to reproduce the chain inside the answer trie. Since this procedure is done inside a critical region, no more than one thread can be doing this chaining process. Thus, in Figure 5.14(b), we are showing a situation where the subgoal call is completed and both threads T_1 and T_{k-1} have

already removed their consumer answer structures and chained the leaf nodes inside the answer trie.

5.6 Performance Analysis on Worst Case Scenarios

In this section, we analyze the performance of the LF_1 and LF_2 proposals, when applied to the SS design and the FS design used solely and combined with the PCC optimization, and compare them against the best lock-based strategy presented in the previous chapters, which has the one using global locks. For benchmarking, we used the same set of tabling benchmarks presented in the previous chapters. Again, since these benchmarks have characteristics that cover a wide number of scenarios in terms of trie usage, they have different demands in terms of trie traversing and create different trie configurations with lower and higher number of nodes and depths. Since all threads are executing the same query goal, it is expected that the aforementioned problems of false sharing and cache memory effects to show up and thus penalize the less robust designs.

Table 5.1 shows the overhead ratios (minimum, maximum and average) of the five sets of benchmarks, using the combination of TabMalloc with the TcMalloc memory allocators, which showed to be the best combination in the previous chapter, when comparing against the NS design with one thread. The columns of the table are divided by designs, the first set of three columns represent the SS design, the second set of three columns represent the FS design and the third set represent the FS design using the PCC optimization. For each set, the first column represents the design using global locks (SS_G and FS_G), the second and third columns represent the usage of the first (SS_{LF_1} and FS_{LF_1}) and the second (SS_{LF_2} and FS_{LF_2}), lock-free proposals, respectively. For both proposals, we have experimented with several configurations of sizes for the initial hash tables and the threshold values. Table 5.1 shows the overhead ratios for the configurations that showed the best results, which were the following: For the LF_1 proposal, both the initial number of bucket entries of the hash table and the MAX_NODES constant have the same value which is 8. For the LF_2 proposal, each hash trie level has 8 bucket entries and the MAX_NODES constant is 4. As described in the previous sections, the hash tries of the LF_2 proposal are integrated in the TabMalloc memory allocator. The values in bold, represent the best overhead ratios by row and by design. For example, the minimum overhead ratio obtained for one thread was: 0.54 for the SS design, 0.85 for the FS design and 1.01 for the FS design combined with PCC.

Table 5.1: Overhead ratios, when compared with the NS design with 1 thread, for the SS, FS and FS + PCC designs using global locks and the LF_1 and the LF_2 proposals, when running 1, 8, 16, 24 and 32 threads with local scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**SS**			**FS**			**FS + PCC**		
		SS_G	**SS_{LF_1}**	**SS_{LF_2}**	**FS_G**	**FS_{LF_1}**	**FS_{LF_2}**	**FS_G**	**FS_{LF_1}**	**FS_{LF_2}**
1	**Min**	**0.54**	**0.54**	**0.54**	**0.85**	0.96	**0.85**	1.03	1.02	**1.01**
	Avg	**0.84**	**0.84**	**0.84**	**0.99**	1.07	1.06	**1.28**	1.32	1.30
	Max	**1.03**	1.05	1.04	**1.10**	1.17	1.16	**1.71**	**1.71**	1.76
	StD	0.17	0.17	0.17	0.08	0.05	0.08	0.22	0.25	0.22
8	**Min**	**0.66**	0.68	**0.66**	1.09	1.04	**0.99**	**0.99**	1.15	1.16
	Avg	0.99	**0.91**	0.92	2.46	2.36	**2.30**	2.55	2.29	**1.88**
	Max	1.36	**1.05**	1.20	**4.48**	4.55	4.54	4.96	4.54	**2.82**
	StD	0.22	0.14	0.15	1.08	1.17	1.13	1.24	1.11	0.60
16	**Min**	**0.81**	0.82	0.82	1.14	1.12	**1.11**	**1.01**	1.14	1.17
	Avg	1.13	**1.02**	1.04	2.89	2.59	**2.50**	2.88	2.69	**1.97**
	Max	1.50	**1.14**	1.31	5.67	5.19	**4.88**	5.84	6.07	**3.14**
	StD	0.21	0.08	0.12	1.40	1.33	1.22	1.48	1.62	0.65
24	**Min**	**1.02**	**1.02**	**1.02**	1.23	**1.07**	**1.07**	1.28	**1.15**	1.16
	Avg	1.34	**1.20**	1.22	3.15	2.79	**2.71**	3.11	2.84	**2.06**
	Max	1.77	**1.72**	1.81	6.34	5.90	**5.65**	6.45	6.89	**3.49**
	StD	0.23	0.16	0.18	1.58	1.50	1.41	1.62	1.74	0.70
32	**Min**	**1.07**	1.08	**1.07**	1.37	1.31	**1.28**	1.40	**1.33**	**1.33**
	Avg	1.71	1.55	**1.54**	3.51	3.12	**3.03**	3.46	3.07	**2.24**
	Max	2.61	2.53	**2.52**	7.47	6.81	**6.54**	7.23	7.47	**3.71**
	StD	0.45	0.43	0.42	1.85	1.72	1.63	1.80	1.85	0.74

Analyzing the results for the SS design, one can observe that both LF_1 and LF_2 proposals have similar results for one thread when comparing with the global locks approach, but as we scale the number of threads up to 32 threads, the best average ratios are found for both lock-free proposals. In particular, for 32 threads the best overheads (minimum, maximum and average) are found for the LF_2 proposal. Having a closer look into the benchmark characteristics (please refer to Table 3.1), we observed during experimentation that on the *WordNet* set of benchmarks, the LF_2 proposal clearly outperforms the remaining strategies on the SS design (on the remaining

benchmark sets it is still better, however the difference is not that clear). This is explained by the fact that on the *WordNet* set of benchmarks, the ratio of time that the threads spend in ST data structures is actually higher than on the remaining sets, thus the impact of the LF_2 proposal becomes more visible when compared with the global locks and the LF_1 proposals.

For the FS design used solo, the best proposal is again LF_2, which as we scale the number of threads, improves its difference against the other two proposals. For 32 threads, the average overhead of the LF_2 proposal is 3.03, which is a good result when compared with the 3.51 overhead of the global locks and the 3.12 overhead of the LF_1 proposal. By using the LF_2 proposal, we are removing part of the synchronization overhead from inside the AT data structure, but the weight in terms of overhead that the chaining mechanism introduces, still restrains the performance of the LF_2 proposal when combined with the FS design.

Finally, we discuss the results of the FS design combined with the PCC optimization. In this strategy, the results of the LF_2 proposal become more visible as we scale the number of threads. For 32 threads, the LF_2 has an average overhead of 2.24 which is by far better than 3.46 and 3.07 of the global locks and LF_1, respectively. However, we would like to draw the reader's attention to the worst results obtained, which are the ones represented by the maximum rows. For 32 threads, the LF_2 has 3.71, which is an outstanding result, when comparing with the 7.23 and 7.47 of the global locks and LF_1, respectively.

In conclusion, the design that showed the best behavior from all combinations was the SS design with the LF_2 proposal. For the FS design, we have highly improved the overhead ratios, using the PCC strategy with the LF_2 hash trie design. Foremost, we have shown that for 32 threads, on the worst case, we had an overhead of 3.71, which we consider to be an considerable result, if we consider the fact that on the first approach of the FS design, shown in the Table 3.2, we had an initial overhead of 12.32.

5.7 Performance Analysis in a External Framework

For the LF_2 proposal, we have also compared it against some of the best-known currently available implementations of lock-free hash tables [10, 14], and for that we used a publicly available framework[6] developed to evaluate lock-free hash tables.

[6]Available at `https://github.com/axel22/Ctries`

Using the Oracle's JDK version 1.7.0_25, we tested the following implementations: two CTries [94] proposals (CT_1 is the original approach and CT_2 is a second proposal with improved snapshots); and the CHM and CSL (both implemented by Doug Lea) from the Java's concurrency package [70]. In CHM, the keys are mapped by any order and the hashes re-size (expand and contract) according with the number of keys in the data structure. The execution time to perform search and insert operations is expected to be constant whatever the size of the map (may vary with the hash re-sizing operation). In the CSL, the keys are mapped by a order (for example, if keys are integers, they may be mapped according with their natural order), thus the execution time to perform search and insert operations is expected to be not constant, once they depend on the number of keys in the data structures. Both CHM and CSL allow parallel search access by multiple threads without any blocking and for the insert operation, they block only a portion of the data structure during the insertion of new keys.

For the experiments, we used two benchmarks already available in the framework, *Insert(N)* and *Lookup(N)* for a numeric data-set with $N = 10^7$ different elements. The *Insert(N)* benchmark starts with an empty set and inserts the N elements. The *Lookup(N)* benchmark does N searches on a previously created data structure containing the same N elements. For both benchmarks, the work of inserting/searching the N elements is equally divided between the working threads. In addition, we created a new benchmark, named *Worst(N)*, for testing a worst case scenario where all threads fully insert the same N elements (we used a numeric data-set with $N = 2 * 10^6$ different elements). By doing this, it is expected that all threads will access the same data structures, to search/insert for elements, at similar times, thus stressing the synchronization on common memory locations, which can increase the aforementioned problems of false sharing and cache memory effects. We experimented with intervals of 8 threads up to 32 threads (the number of cores in the machine) and all results are the average of 10 runs for each benchmark.

Table 5.2 shows the execution time, in milliseconds, and the speedup, compared against the respective execution time with one thread, for the five proposals when running the *Insert(N)* and *Lookup(N)* benchmarks.

For the *Insert(N)* benchmark, LF_2 has the best results for the execution time, showing a significant difference to all other proposals. On average, LF_2 is around three times faster than the second best proposal, which is CT_2. Regarding the speedup, CT_2 competes with LF_2 for the best results, but for most cases, LF_2 still gets the best speedup. The top speedups for LF_2 are 10.76 and 11.00 for 16 and 32 threads. For CT_2, the top speedup is 9.96 for 32 working threads.

Table 5.2: Execution time, in milliseconds, and speedup, against the execution time with one thread, for the *Insert(N)* and *Lookup(N)* benchmarks using Java's standard library JDK version 1.7.0_25, when running 1, 8, 16, 24 and 32 threads with LF_2, CT_1, CT_2, CSL and CHM proposals (best ratios by row and by execution time and speedup are in bold)

Thrs	**Time (T_P(w))**					**Speedup ($T_P(1)/T_P$(w))**				
(w)	**LF_2**	**CT_1**	**CT_2**	**CSL**	**CHM**	**LF_2**	**CT_1**	**CT_2**	**CSL**	**CHM**
Insert(N) Benchmark										
1	**3,057**	7,231	9,613	4,701	4,983	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
4	**976**	2,836	3,017	3,198	3,314	3.13	2.55	**3.19**	1.47	1.50
8	**582**	1,693	1,726	2,552	2,654	5.25	4.27	**5.57**	1.84	1.88
16	**284**	1,453	1,441	2,339	2,815	**10.76**	4.98	6.67	2.01	1.77
24	**466**	1,521	1,072	2,088	3,031	6.56	4.75	**8.97**	2.25	1.64
32	**278**	1,285	965	1,910	3,340	**11.00**	5.63	9.96	2.46	1.49
Lookup(N) Benchmark										
1	6,175	6,856	6,905	6,046	**1,898**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
4	1,581	1,834	1,806	1,572	**540**	**3.91**	3.74	3.82	3.85	3.51
8	845	924	942	735	**283**	7.31	7.42	7.33	**8.23**	6.71
16	445	515	524	346	**151**	13.88	13.31	13.18	**17.47**	12.57
24	327	479	431	512	**119**	**18.88**	14.31	16.02	11.81	15.95
32	335	492	469	302	**123**	18.43	13.93	14.72	**20.02**	15.43

For the *Lookup(N)* benchmark, CHM achieves the best results for the execution time followed by CSL and LF_2 as third placed. When the work is split among multiple threads, LF_2 is up to 1.5 times faster than CT_1 and CT_2. For the speedup, CSL and LF_2 show the best results. The top speedup for both proposals is achieved for 32 threads with a 20.02 value for CSL and 18.43 for LF_2.

Table 5.3 shows the execution time, in milliseconds, and the speedup, compared against the respective execution time with one thread, for the five proposals when running the *Worst(N)* benchmark.

For the *Worst(N)*, we are interested in evaluating the robustness of the implementations when exposed to worst case scenarios. As it is expected that the execution time with multiple threads will result in worst results when compared with the base execution time with one thread, we thus show the overhead (not the speedup) for comparing the execution with increasing number of threads (values close to 1.00 are thus better). For the execution time, LF_2 shows again the best results with CT_2 being

Table 5.3: Execution time, in milliseconds, and speedup, against the execution time with one thread, for the *Worst(N)* benchmark using Java's standard library JDK version 1.7.0_25, when running 1, 8, 16, 24 and 32 threads with LF_2, CT_1, CT_2, CSL and CHM proposals (best ratios by row and by execution time and overhead are in bold)

Thrs	**Time (T_P(w))**					**Overhead (T_P(w)/T_P(1))**				
(w)	**LF_2**	**CT_1**	**CT_2**	**CSL**	**CHM**	**LF_2**	**CT_1**	**CT_2**	**CSL**	**CHM**
1	**495**	987	1,442	818	827	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
4	**1,579**	2,840	1,720	3,786	2,388	3.19	2.88	**1.19**	4.63	2.89
8	**2,019**	2,971	2,667	7,698	3,395	4.08	3.01	**1.85**	9.41	4.11
16	**2,346**	3,276	2,518	8,018	7,936	4.74	3.32	**1.75**	9.80	9.60
24	**2,502**	3,802	3,223	8,304	12,864	5.05	3.85	**2.24**	10.15	15.56
32	**2,730**	4,111	3,181	8,620	16,420	5.52	4.17	**2.21**	10.54	19.85

very close. For the overhead, CT_2 and CT_1 are better than LF_2 mostly because the base execution times with one thread are significantly higher than LF_2 (495, 987 and $1,442$ milliseconds, respectively, for the LF_2, CT_1 and CT_2 proposals). The CSL and CHM proposals show a poor performance for this benchmark. In particular, CHM has the worst results with an overhead almost linear to the number of working threads.

In summary, these experiments show that the LF_2 proposal clearly outperforms all the other proposals for the execution times and that, in general, it also achieves the best results for the speedup/overhead in most experiments.

5.8 Chapter Summary

This chapter introduced several key concepts about the LF_1 and the LF_2 proposals which are lock-free data structures. We presented the algorithms, the formalization of the proposals and the PCC optimization for the FS proposal. At the end of the chapter, we discussed the performance analysis of the proposals.

Both lock-free data structure proposals were implemented in the YapTab-Mt framework. Our main motivation for the creation of the proposals was to refine the previous lock-based proposals in order to be as effective as possible in the concurrent search and insert operations over the trie structures of the table space. We discussed the relevant details of each proposal, described the main algorithms and proved the correctness of both. We based our discussion on the YapTab-Mt framework, but both proposals can

be applied to general purpose applications, such as word counting, compilers, language run-times and some components of game development, that only require search and insert operations on their hash mapping mechanisms.

For the LF_2 proposal, a key decision was the combination of hash tries with the use of a separate chaining closed by the last node referencing back the hash level for the node. This allowed us to implement a clean proposal to solve hash collisions by simply moving nodes between the levels. In this proposal, updates and expansions of the hash levels are never done by using data structure replacements (i.e., create a new one to replace the old one), which also avoids the need for memory recovery mechanisms. Another key proposal decision that minimizes the bottlenecks leading to false sharing or cache memory effects, is the insertion of nodes done at the end of the separate chain. This allows for dispersing the memory locations being updated concurrently as much as possible and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels. A final motivation was the complete integration of hashes and nodes within the TabMalloc memory allocator, thus that we could minimize the cost of having multiple and simultaneous memory allocation requests.

Experimental results obtained in a external framework (i.e., not within YapTab-Mt) showed that this proposal can effectively reduce the execution time and scales better than some of the best-known currently available lock-free hashing implementations. In the context of YapTab-Mt framework, our proposal clearly achieved the best results for the overhead ratios. In particular, for worst case scenarios, our proposal clearly outperformed the previous proposals with superb overheads in some cases.

Chapter 6

Batched Scheduling

In this chapter we discuss the problem of supporting multithreaded batched scheduling and we present a performance analysis comparing local scheduling with batched scheduling.

Local and batched evaluations differ in that batched evaluation eagerly returns answers while local evaluation may not return any answers out of an SCC until that SCC is completely evaluated. Thus, batched scheduling schedules the evaluation of a program in a depth-first manner as does the WAM, favoring the forward execution first instead of backtracking, leaving the consumption of answers and completion for last. Thus, the key idea is to return an answer for a subgoal call to the GN that called the subgoal call as soon as the answer is derived.

6.1 Implementation Details

At the implementation level, the major difference between local and batched scheduling is in the tabling operation *tabled new answer*, where we decide what to do when an answer is found during the evaluation. This operation checks whether a newly found answer is already in the corresponding answer trie structure and, if not, inserts it. Remember that in Chapter 3, we showed how to extend the tabled new answer operation to support multithreading (Algorithm 3.2). For the NS and SS designs the support for batched scheduling was immediate, since the AT data structure is not shared among threads, but for the FS design we have omitted how to handle batched scheduling. The usage of local scheduling with the FS design was straightforward, because for repeated and new answers, local scheduling always fails. The usage of

batched scheduling with the FS design requires further support since with batched scheduling, answers are immediately propagated and we have to ensure that the propagation of an answer occurs on all subgoal calls one and only once. To do so, we take advantage of the PCC procedure, presented in Section 5.5.4, as a way to keep, for every subgoal call of every thread, track of all the answers that were already propagated. This requires minor changes to the *tabled new answer* tabling operation. Algorithm 6.1 shows how we have extended the tabled new answer operation to support the FS design with batched scheduling.

Algorithm 6.1 tabled_new_answer(answer ANS, subgoal frame SF)

```
1:  leaf ← check_insert_answer_trie(ANS, SF)
2:  if NS_design or SS_design then
3:      ...                                          ▷ same as Algorithm 3.2
4:  else                                                          ▷ FS design
5:      chain ← check_insert_consumer_chain(leaf, SF)
6:      if is_answer_marked_as_found(chain) = True then
7:          return failure
8:      else                                                ▷ the answer is new
9:          mark_answer_as_found(chain)
10:         if local_scheduling_mode(SF) then
11:             return failure
12:         else                                       ▷ batched scheduling mode
13:             return proceed
```

The algorithm receives two arguments: the new answer found during the evaluation (*ANS*) and the subgoal frame which corresponds to the call at hand (*SF*). The *NS_design*, *SS_design* and *FS_design* macros define which table design is enabled.

The algorithm begins by checking/inserting the given *ANS* into the answer trie structure, which will return the leaf node for the path representing *ANS* (line 1). In line 2, it then tests whether one of the NS or SS designs are active, and in such a case, the algorithm is the same as Algorithm 3.2.

Otherwise, for the FS design (lines 4 to 13), it checks/inserts the given *leaf* node into the private consumer chain for the current thread, which will return the corresponding chain node. In line 6, it then tests whether the chain node already existed in the consumer chain, i.e., if it was inserted or not by the current check/insert operation in order to return failure (line 7), or it proceed with marking the answer *ANS* has found (line 9). At the end (lines 10 to 13), it returns failure if local scheduling is active (line

11), otherwise, the batched scheduling is active, thus it propagates the answer *ANS* (line 13).

6.2 Performance Analysis on Worst Case Scenarios

We now present experimental results about the usage of the batched scheduling on the NS, SS and FS designs. For the sake of simplicity, for the SS and FS designs, we will be presenting only the results for the lock free LF_2 proposal (SS_{LF_2} and FS_{LF_2}), since they were the ones that presented the lowest overheads in the previous chapters. For the FS_{LF_2} design, we will use it with the PCC procedure enabled.

Concerning the benchmarks, we will be using the same five sets of benchmarks presented before with the same number of runs per benchmark, the same formula to calculate the overhead ratios, and the same worst case scenario approach, where all threads begin with the same query goal. To put the results in perspective, we experimented with 1, 8, 16, 24 and 32 threads (the maximum number of cores available in our machine) with batched and local scheduling.

Table 6.1 shows the overhead ratios, when compared with the NS design with 1 thread (running with local scheduling, PtMalloc and without TabMalloc), for the NS, SS_{LF_2} and FS_{LF_2+PCC} designs (all running with TabMalloc and TcMalloc), when running 1, 8, 16, 24 and 32 threads with local and batched scheduling on the five sets of benchmarks. For each design, the table has then two columns, a column **Local** that shows results already presented in previous chapters for the local scheduling and a column **Batched** that shows the new results with batched scheduling. The overhead results presented in both **Local** and **Batched** columns use as base time the execution times presented in the NS column of the Table 3.1.

By observing Table 6.1, we can see that, for one thread, on average, local scheduling is sightly better than batched on the three designs. For the NS design, we have 0.78 and 0.82, for the SS_{LF_2} design we have 0.84 and 0.90 and for the FS_{LF_2+PCC} design we have 1.30 and 1.46 average overhead ratios, for the local and batched scheduling strategies, respectively.

As we scale the number of threads, one can observe that, for the NS and SS_{LF_2} designs both scheduling strategies have similar minimum, average and maximum overhead ratios. For the FS_{LF_2+PCC} design, the best minimum overhead ratio is always for batched scheduling. During experimentation we observed that the minimum overhead

Table 6.1: Overhead ratios, when compared with the NS design with 1 thread (running with local scheduling, PtMalloc and without TabMalloc) for the NS, SS_{LF_2}, FS_{LF_2+PCC} designs (with TabMalloc and TcMalloc), when running 1, 8, 16, 24 and 32 threads with local and batched scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads		**NS**		**SS_{LF_2}**		**FS_{LF_2+PCC}**	
		Local	**Batched**	**Local**	**Batched**	**Local**	**Batched**
1	**Min**	**0.53**	0.55	**0.54**	0.55	1.01	**0.95**
	Avg	**0.78**	0.82	**0.84**	0.90	**1.30**	1.46
	Max	1.06	**1.05**	**1.04**	**1.04**	**1.76**	2.33
	StD	0.15	0.14	0.17	0.16	0.22	0.44
8	**Min**	0.66	**0.63**	0.66	**0.63**	1.16	**0.99**
	Avg	**0.85**	0.88	**0.92**	0.93	**1.88**	1.95
	Max	**1.12**	1.14	1.20	**1.15**	**2.82**	3.49
	StD	0.13	0.14	0.15	0.14	0.60	0.79
16	**Min**	0.85	**0.75**	0.82	**0.77**	1.17	**1.06**
	Avg	**0.98**	1.00	**1.04**	1.05	**1.97**	2.08
	Max	**1.16**	1.31	1.31	**1.28**	**3.14**	3.69
	StD	0.09	0.17	0.12	0.13	0.65	0.83
24	**Min**	**0.91**	0.93	1.02	**0.98**	1.16	**1.09**
	Avg	**1.15**	1.16	1.22	**1.19**	**2.06**	2.19
	Max	1.72	**1.60**	1.81	**1.61**	**3.49**	4.08
	StD	0.20	0.21	0.18	0.16	0.70	0.91
32	**Min**	1.05	**1.04**	**1.07**	1.12	1.33	**1.26**
	Avg	1.51	**1.49**	1.54	**1.51**	**2.24**	2.41
	Max	**2.52**	2.63	**2.52**	2.62	**3.71**	4.51
	StD	0.45	0.45	0.42	0.43	0.74	1.02

values for 8, 16, 24 and 32 threads were given by the benchmark belonging to the model checking set (see Table 3.1 for the characteristics of the benchmarks). For the average and maximum overhead ratio, local scheduling is always better than batched scheduling. During experimentation we observed also that the maximum overhead values for 8, 16, 24 and 32 threads were given by the pyramid benchmark in the path right set.

In summary, we can say that both the local and batched scheduling strategies have similar overhead results on worst case scenarios for the NS, SS_{LF_2} and FS_{LF_2+PCC}

designs.

6.3 Chapter Summary

So far we have observed that both scheduling strategies have similar results for worst case scenarios, even though that local scheduling showed to be on average slightly better that batched scheduling for the NS, SS and FS designs. In the next chapter, we will be discussing answer subsumption and mode-directed tabling features. Previous works [42, 125] showed that local is the best scheduling strategy for answer subsumption, because it restricts all operations to a maximal SCC. This property implies that no non-maximal answer will be used outside of the SCC in which it was derived. For this reason, the use of local evaluation can be critical for efficient answer subsumption. A good theoretical example was given by Freire in the work [42], where answer subsumption is used to find the shortest paths in a graph G. When local evaluation is used, the complexity of evaluation is proportional to the number of edges in G. When batched evaluation is used, the complexity of the evaluation is proportional to the number of paths in G, which is exponential in the number of edges of G. This example was used later by Swift and Warren in their work about answer subsumption [125].

In a different work, Santos and Rocha compared local and batched evaluations using mode-directed tabled predicates on several benchmarks and showed that, on average, batched evaluation is around 31% worse than local evaluation for the execution time [116]. Additionally, they observed that batched evaluation allocated/deleted more trie nodes and inserted/deleted more tabled answers than local evaluation. In particular, batched evaluation got worse as more answers were inserted into the table space.

Accordingly, based on our results and on these previous works, in the next chapter we will continue to use local scheduling as our default scheduling strategy for multi-threaded tabled evaluations.

Chapter 7

Subgoal-Sharing with Shared Answers

In this chapter, we focus on two well-known dynamic programming problems, the 0-1 Knapsack and Longest Common Subsequence (LCS) problems, and we discuss how we were able to scale their execution by taking advantage of the SS design. For each problem, we present a multithreaded tabled top-down and bottom-up approach. For the top-down approach, we use YapTab's mode-directed tabling support [116] that allows to aggregate answers by specifying pre-defined modes such as *min* or *max*. For the bottom-up approach, we use YapTab's standard tabling support.

7.1 Dynamic Programming

Dynamic programming [17] is a general recursive strategy that consists in dividing a problem in simpler sub-problems that, often, are the same. The idea behind dynamic programming is to reduce the number of computations: once an answer to a given sub-problem has been computed, it is memorized and the next time the same answer is needed, it is simply looked up. Dynamic programming is especially useful when the number of overlapping sub-problems grows exponentially as a function of the size of the input, such as problems where functional equations can provide a mechanism to obtain optimal solutions to sub-problems. Dynamic programming problems, may be classified in terms of the functional equation. A functional equation that contains a single recursive term yields a *monadic* dynamic programming formulation. Formulations whose cost functions contains multiple recursive terms are called *polyadic* formulations.

Formulations can also be categorized as *serial*, if the sub-problems at all levels depend only on the results at the immediate preceding levels, or *non-serial* if the sub-problems at all levels depend on the results of other than the immediately preceding levels [67].

Dynamic programming can be implemented using either a *bottom-up* or a *top-down* approach. In bottom-up, it starts from the base sub-problems and recursively computes the next level sub-problems until reaching the answer to the given problem. On the other hand, the top-down approach starts from the given problem and uses recursion to subdivide a problem into sub-problems until reaching the base sub-problems. Answers to previously computed sub-problems are reused rather than being recomputed. An advantage of the top-down approach is that it might not need to compute all possible sub-problems.

Most of the proposals that can be found in the literature to parallelize dynamic programming problems follow the parallelization of a sequential bottom-up algorithm. All these proposals are usually based on a careful analysis of the sequential algorithm in order to find the best way to minimize the data dependencies in the supporting data structure for memorization, often a matrix or an array, resulting in a parallelization that requires a synchronization mechanism before recursively computing the next level sub-problems. Alternatively, a generic proposal to parallelize top-down dynamic programming algorithms is Stivala et al.'s work [121], where a set of threads solve the entire dynamic program independently but with a randomized choice of sub-problems, i.e., each thread runs exactly the same function, but the randomization choice of sub-problems results in the threads diverging to compute different sub-problems while reusing the sub-problem's results computed in the meantime by the other threads.

7.2 Subgoal-Sharing with Shared Answers

In this chapter, we introduce our improved approach of the SS design, where threads view their tables as private but are able to use the answers of a sub-problem, if another thread has already computed them. The idea is as follows. Whenever a thread calls a new tabled subgoal, first it searches the table space to lookup if any other thread has already computed the full set of answers for that call. If so, then the thread reuses the available answers. Otherwise, it computes the subgoal call itself in a private fashion. Several threads can work on the same subgoal simultaneously, i.e., we do not protect a subgoal from further evaluation while other threads have picked it up already. The first thread completing a computation, shares the results by making them available

(public) to the other threads.

Furthermore, we aim to improve the memory usage of the SS design by removing the BAE data structure. The decision of removing the bucket array is a direct consequence of the memory analysis made in Equation 3.3 where we have shown that the performance of the SS design is directly affected by the size of the memory used in the bucket array of entries. Thus, in problems with a considerable amount of subgoal calls, the time and memory used in the allocation bucket arrays can have a significant impact in the performance of the SS design. Thus, assuming that NT threads have evaluated a predicate P_i, then the memory usage analysis of the SS design with answer sharing support is given by the following Equation 7.1:

$$
\begin{aligned}
& MU_{SS}(P_i) = \\
& TE_{SS} + ST_{SS}(P_i) + \sum_{j=1}^{NC(P_i)} [NT_1 * [SF_{SS} + AT_{SS}(P_{i.j})]] \\
& \textit{cond.} \begin{cases} TE_{SS} = TE_{YT} \\ ST_{SS}(P_i) = ST_{YT}(P_i) \\ NT_1 \leq NT \\ SF_{SS} = SF_{YT} \\ AT_{SS}(P_{i.j}) = AT_{YT}(P_{i.j}) \end{cases}
\end{aligned}
\tag{7.1}
$$

The memory usage is given by the sum of the size of table entry structure (TE_{SS}) with the size of the subgoal trie structure ($ST_{SS}(P_i)$) plus the summatory of the memory used in the multiplication of NT_1 threads by subgoal frame (SF_{NS}) and answer trie ($AT_{NS}(P_i)$) data structures in the NC subgoal calls of the predicate P_i ($NC(P_i)$). Concerning the conditions that describe the size of the structures, Equation 7.1 shows that all structures in the SS design have the same size as the ones used in YapTab, and that NT_1 is always lower or equal to NT, since the NT_1 value is the number of threads that have completely evaluated the subgoal calls of P_i in a private fashion. The NT_1 value will be clarified later when we present this new approach.

In more detail, when comparing both equations of the SS design, one can observe that Equation 7.1 is always lower (or equal if $NT = NT_1$) than Equation 3.3 due to the condition $NT \leq NT_1$. Additionally, we have optimized even further this design, and allow threads to delete their private AT data structures at the end of the evaluation of a call. So, in practice at the end of the execution of the NT threads, the Equation 7.1 will be even lower than Equation 3.3, since the memory used in the AT data structures

will be not multiplied by NT.

Figure 7.1 illustrates the table data structures for the SS design with shared answers for a predicate P_i. As before, threads access the subgoal trie structures, for both read and write operations concurrently, but for the answer trie structures, only after completion they are concurrently accessed for reading (black data structures in Figure 7.1).

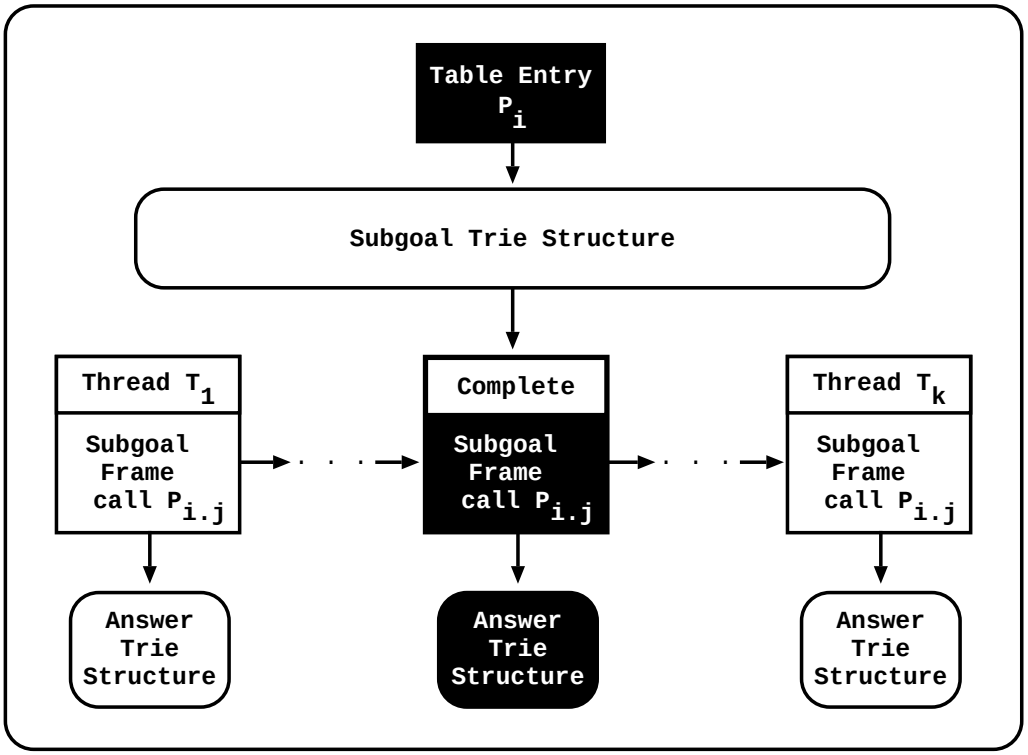

Figure 7.1: The key idea for the SS design with shared answers

All subgoal frames and answer tries are initially private to a thread. Thus, if k threads are evaluating a subgoal call $P_{i.j}$, each thread has its own subgoal frame and answer trie structures. Later, when the first subgoal frame is completed, i.e., when we have found the full set of answers for it, it is marked as completed and put in the beginning of the list of private subgoal frames (configuration shown in Figure 7.1). Following calls made by other threads to this subgoal call simply consume the answers from the completed subgoal frame, thus avoiding recomputing the subgoal call at hand. By sharing only completed answer tries, we avoid the problem of dealing with concurrent updates to the answer tries, the problem of managing the different set of answers that each thread has found and, more importantly, the problem of dealing with concurrent deletes, as in the case of using mode-directed tabling.

Remember that mode directed tabled predicates require the deletion of nodes inside the

answer tries, when answers are updated as *better ones* appear during the evaluation of the call. Since the SS design keeps the answer tries private to each thread, the deletion of nodes can be done without any complex machinery to deal with concurrent delete operations. We use all these advantages at our favor and improve the efficiency of the SS design by extending it to share the answer tries between the threads as soon as the tables are completed, i.e., as soon as no more changes can occur in the answer trie of the subgoal call.

7.2.1 Mode Directed Tabling

Mode-directed tabling is an extension to the tabling technique that supports the definition of modes for specifying how answers are inserted into the table space. The idea is to use the modes to define the arguments to be considered for variant checking and to define how variant answers should be tabled regarding the remaining arguments. Figure 7.2 shows the example for the tabled predicate $p/3$ represented in the previous sections, using mode-directed tabling with modes (*index*, *index*, *min*). At the entry point we have the TE data structure extended with a *mode array*. The mode array stores information about the modes defined for the predicate's arguments, which in the example are index for the first and second arguments and maximum for the third argument. Underneath the TE data structure, we have the ST data structure, that stores the tokenized form of the calls $p(1, X, Y)$ and $p(1, 2, 3)$, and call has it own SF data structure.

With mode-directed tabling the SF data structure is extended with a *substitution array*. The substitution array stores the mode information together with the number of free variables associated with each argument within the subgoal call. In the example, the call $p(1, X, Y)$ has 0 variables in the first argument, 1 variable in the second argument and 1 variable in the third argument, while the $p(1, 2, 3)$ does not have any variables on its arguments. Finally, each SF has its own AT data structure, with the answers for the subgoal call. For $p(1, X, Y)$, the answers shown are $p(1, 1, 3)$ and $p(1, 1, 4)$, but now since the mode operator in the third argument is the maximum, the answer $p(1, 1, 3)$ is marked as *invalid* (black box), and thus the only valid answer is $p(1, 1, 4)$. For $p(1, 2, 3)$, the AT data structure remains with the answer *true*.

Next, we describe how the call $p(1, X, Y)$ is represented in the SS design using mode-directed support. To do so, we use Figure 7.3 with two threads T_1 and T_k evaluating the call $p(1, X, Y)$. The figure is divided in two types of data structures, the data structures that belong to the table space and the local stack of thread T_1 (for the sake

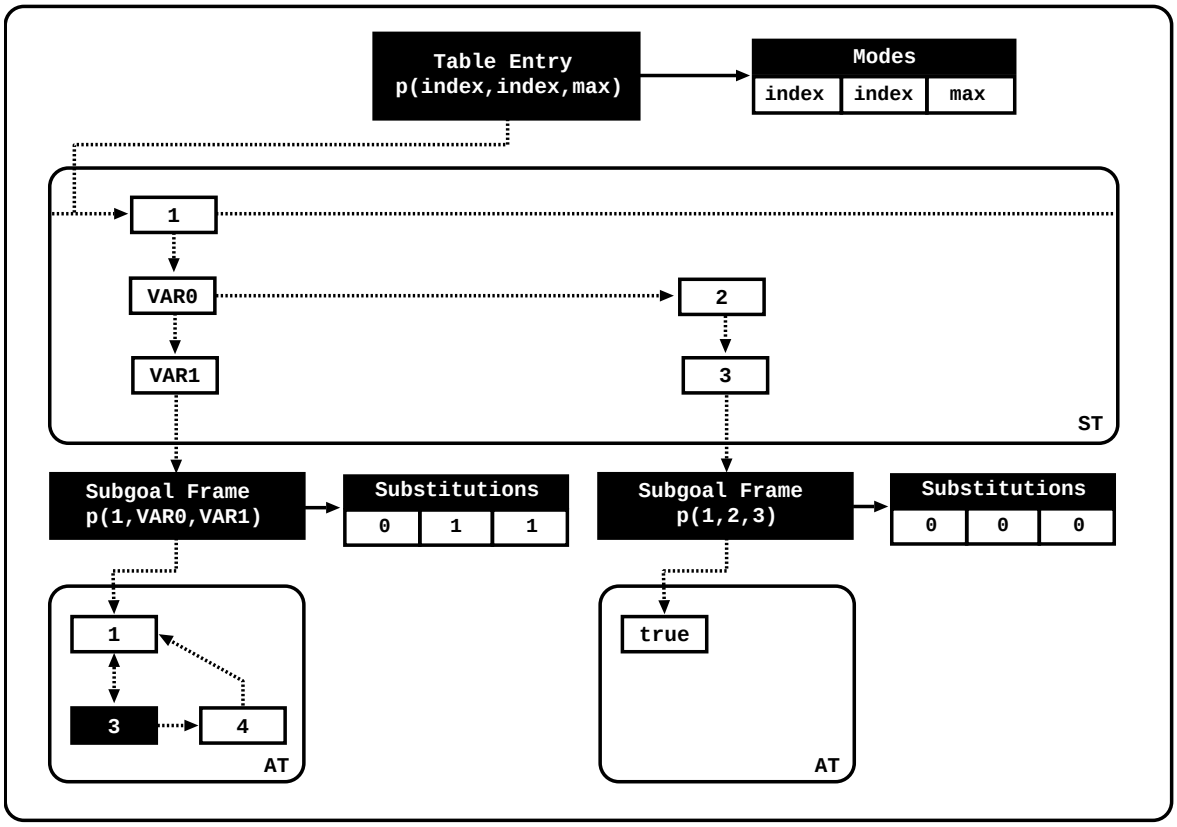

Figure 7.2: Table space organization for mode-directed tabling

of simplicity, we are only showing the local stack of thread T_1 since the local stack of thread T_k is similar). As expected, the table space data structures have the ST data structure shared among threads and the leaf node of the call $p(1, X, Y)$ refers to a bucket array of entries. Each thread has its own cell inside the bucket array, which refers to the private structures of each thread, which are the SF data structure with the substitution array and the AT data structure.

For the local stack of thread T_1, Figure 7.3 shows the generator node referring the SF data structure, and the consumer node referring the AT data structure (remember that generator nodes are allocated in the first call to a term, while consumer nodes are allocated in follower calls). The allocation of both generator/consumer nodes is done independently by both threads T_1 and T_k, i.e., each thread allocates its own nodes referring the private SF and AT data structures. This allocation is done regardless of the fact that one of the threads might already have completely evaluated the subgoal call. One can observe, that in the example, thread T_k has already marked as complete the SF data structure and the AT data structure is in its final state (the invalid answer $p(1, 1, 3)$ has been already deleted from the data structure), but this information is

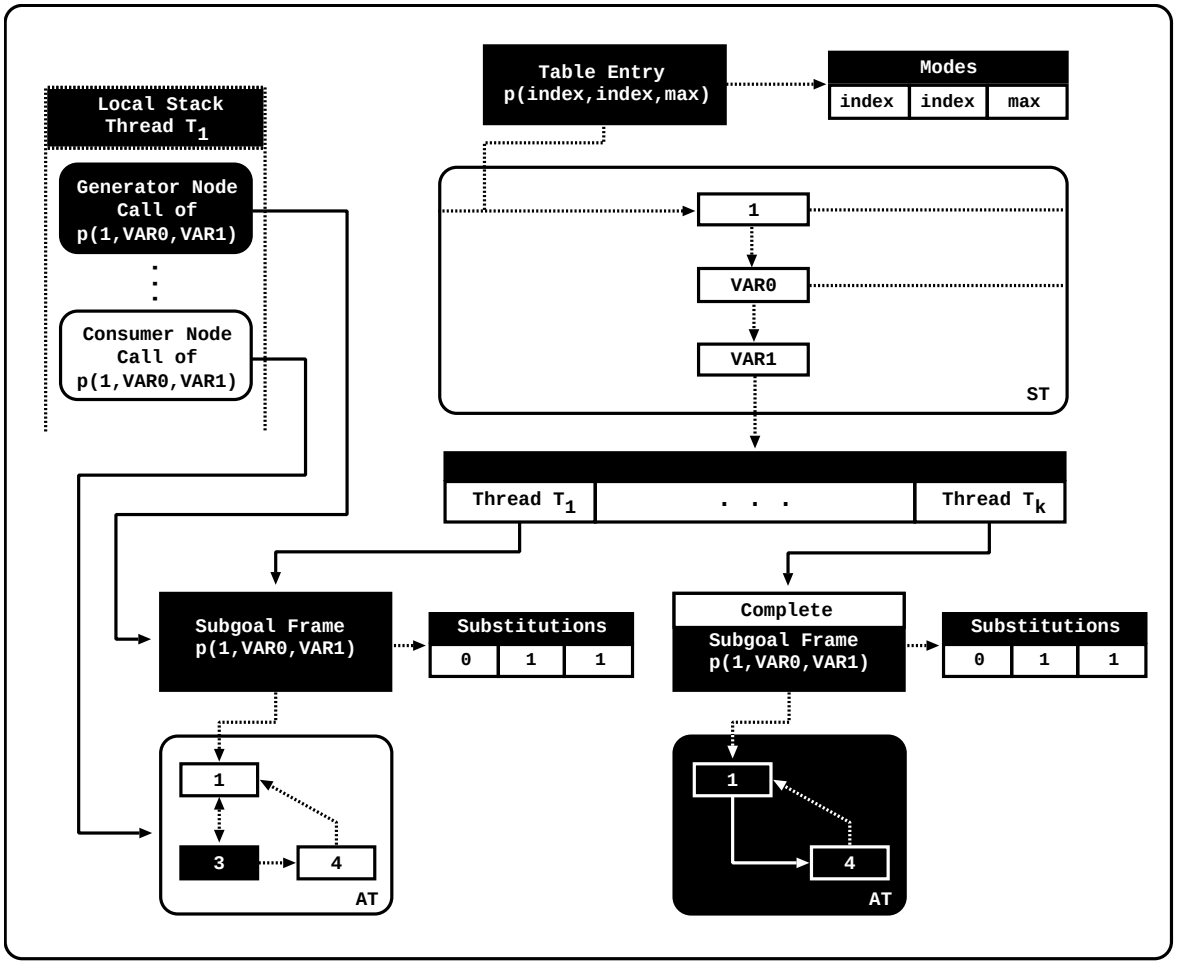

Figure 7.3: The SS design with mode-directed tabling

not shared with thread T_1.

7.2.2 Support for Shared Answers

In this section, we describe how we have extended the SS design, thus that it can support answer sharing for a subgoal call after it is completely evaluated. The description has three steps. On the first step we show how we compacted the table space to be more efficient in terms of memory usage. On the second step, we show how answers from a thread must be public before they can be shared with other threads. On the third step, we show how a thread publishes its answers as public and how efficient is the access to the shared answers.

Figure 7.4 shows then the first step of changes that we have implemented in the SS design. In this step, we allocate only once the substitution array and share it among

threads, and next we remove the bucket array of entries between the ST and the SF data structures.

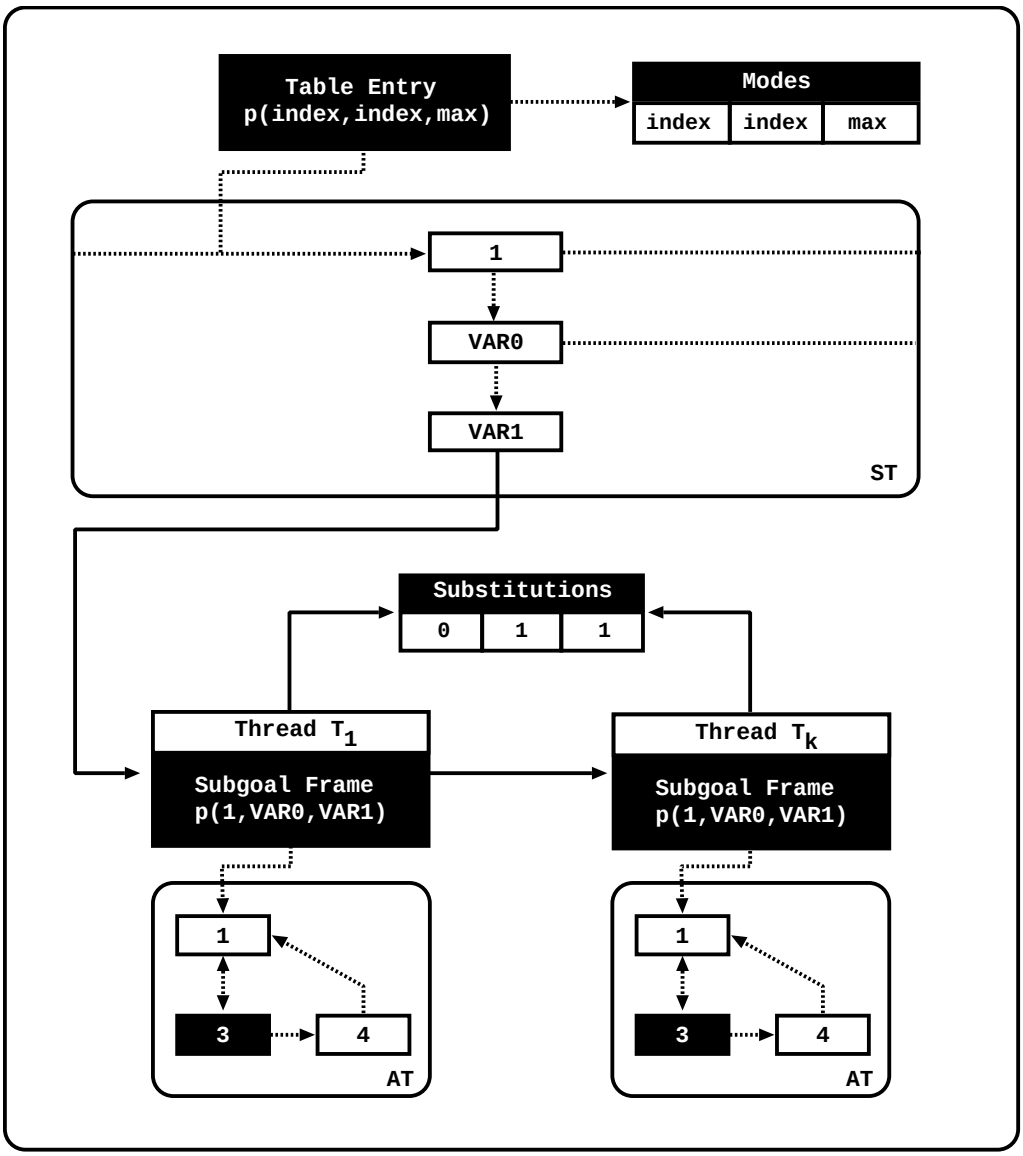

Figure 7.4: The first step for sharing answers in the SS design

For the substitution array, instead of allocating one of this structure by subgoal frame, we allocate it only once and share it among all threads and their respective SF data structures. The substitution array is then allocated when the first SF data structure is allocated for a subgoal call, and the follower SF data structures for the same subgoal call simply refer to it whenever they are allocated. Furthermore, since the initial values in the substitution array data structure do not change during the evaluation of a subgoal call, the data structure is used only for reading, thus no synchronized mechanism is required when multiple threads access it.

For the removal of the bucket array of entries, we included the information about the thread identifier in the SF data structure and chained these structures. The downside of this removal is the fact that we have now a linear search over the thread identifiers whenever a thread is searching for its subgoal frame, if exists more than one thread using the same subgoal call. In the future, we hope to implement a lock-free hashing mechanism similar to the LF_2 proposal to solve this issue, but for the moment the reader should keep in mind that the user-defined scheduler must be efficient enough to take advantage of the SS design with shared answers, otherwise if multiple threads evaluate the same subgoal calls, the overall performance might end up in overheads instead of speedups.

In the example shown in Figure 7.4 for the call $p(1, X, Y)$, the threads T_1 and T_k are sharing the same substitution array and both of their SF data structures are chained. Thus, if the thread T_k wants to reach to its SF data structure, it has to traverse the SF data structure of thread T_1. Furthermore, the AT data structures of both threads are similar and none of them is complete.

Next, on the second step, we show how answers from a thread can be shared with other threads using Figure 7.5. The key idea of sharing answers among threads is to allow the threads to use a AT data structure as soon as it is in its final state, i.e., as soon as its SF data structure is marked as complete. For this idea to the true, every generator and consumer node in the local stack of every thread must be able to access the AT data structure which is being shared, regardless of the state of its own SF data structure. An important remark is the fact that a SF data structure marked as complete is not enough for allowing the remaining threads to begin consuming from its AT data structure, since both data structures *must be public* before being used. This is imperative, since by default, in the SS design both data structures are private to each thread. We require that data structures must be made public before being used, therefore ensuring the correctness of the implementation and avoid situations where a thread is consuming answers from AT data structure which is being deleted by an other thread.

Figure 7.5 shows then an example where the thread T_k as already completed the evaluation of the call $p(1, X, Y)$. But, thread T_1 is still using its own data structures, thus both the generator and consumers nodes in the local stack are using its own SF and AT data structures.

On the third and last step, we show how a thread publishes its answers as public. To do so, we use Figure 7.6 which is a continuation of the previous example. The publishing

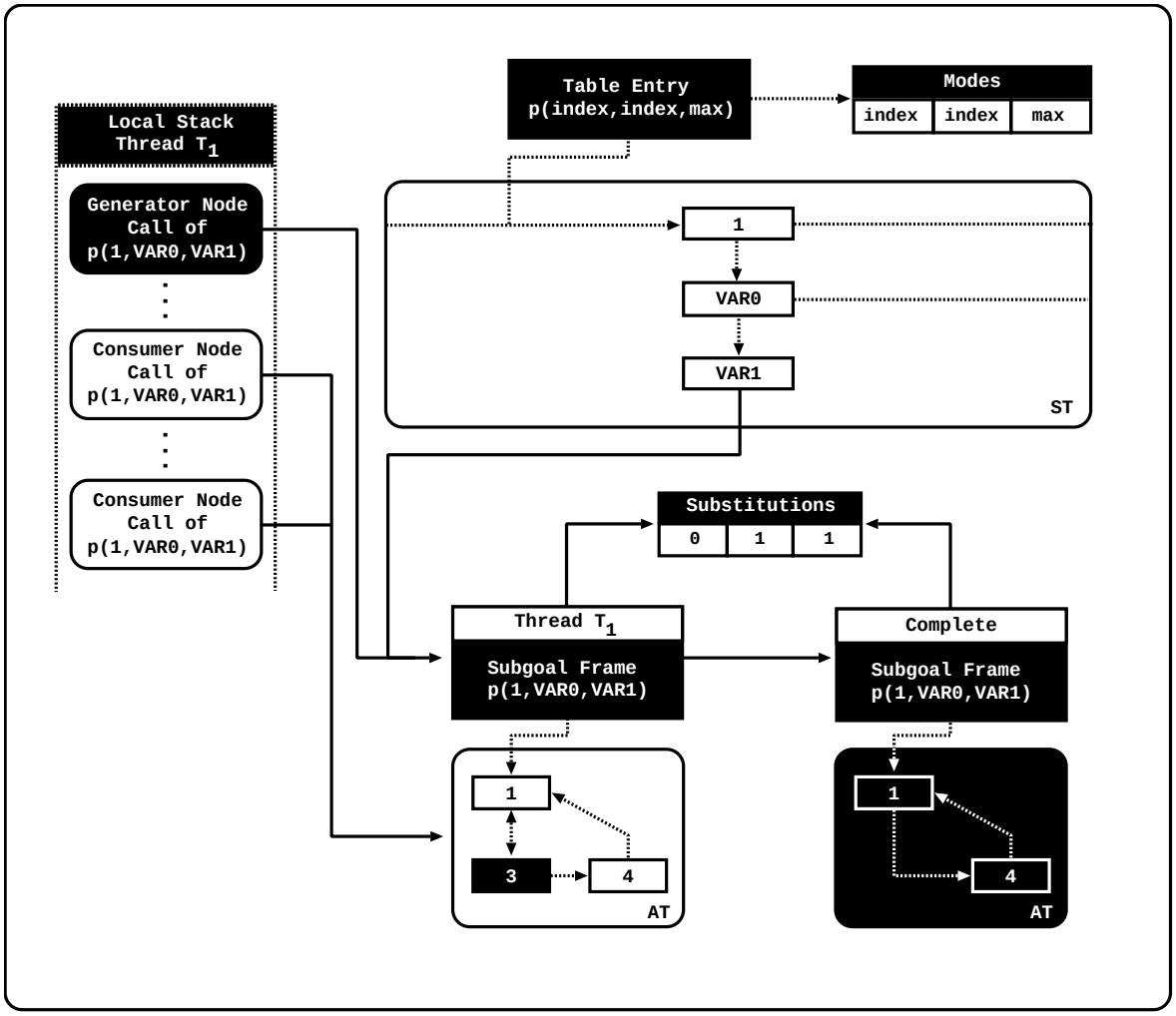

Figure 7.5: The second step for sharing answers in the SS design

process is done with a simple CAS operation over the reference of leaf node in the ST data structure (the *VAR*1 node in the example) that marks the head of the chain of SF data structures. A successful CAS operation means that the SF becomes public, otherwise it means that the SF remains private to the thread T_k. When the CAS operation fails, it means that another thread has inserted a newly SF data structure in the chain. So, thread T_k checks the state of the newly inserted SF. If the state is complete, then SF is public and thread T_k finishes its publishing process. Otherwise, the SF is private and thread T_k executes again the CAS operation over the leaf node. The publishing process finishes when thread T_k finds a public SF or succeeds inserting its own SF. Consequently, the search for a public SF data structure is only done on the head of the chain.

In the example shown in Figure 7.6, we have the complete SF data structure already

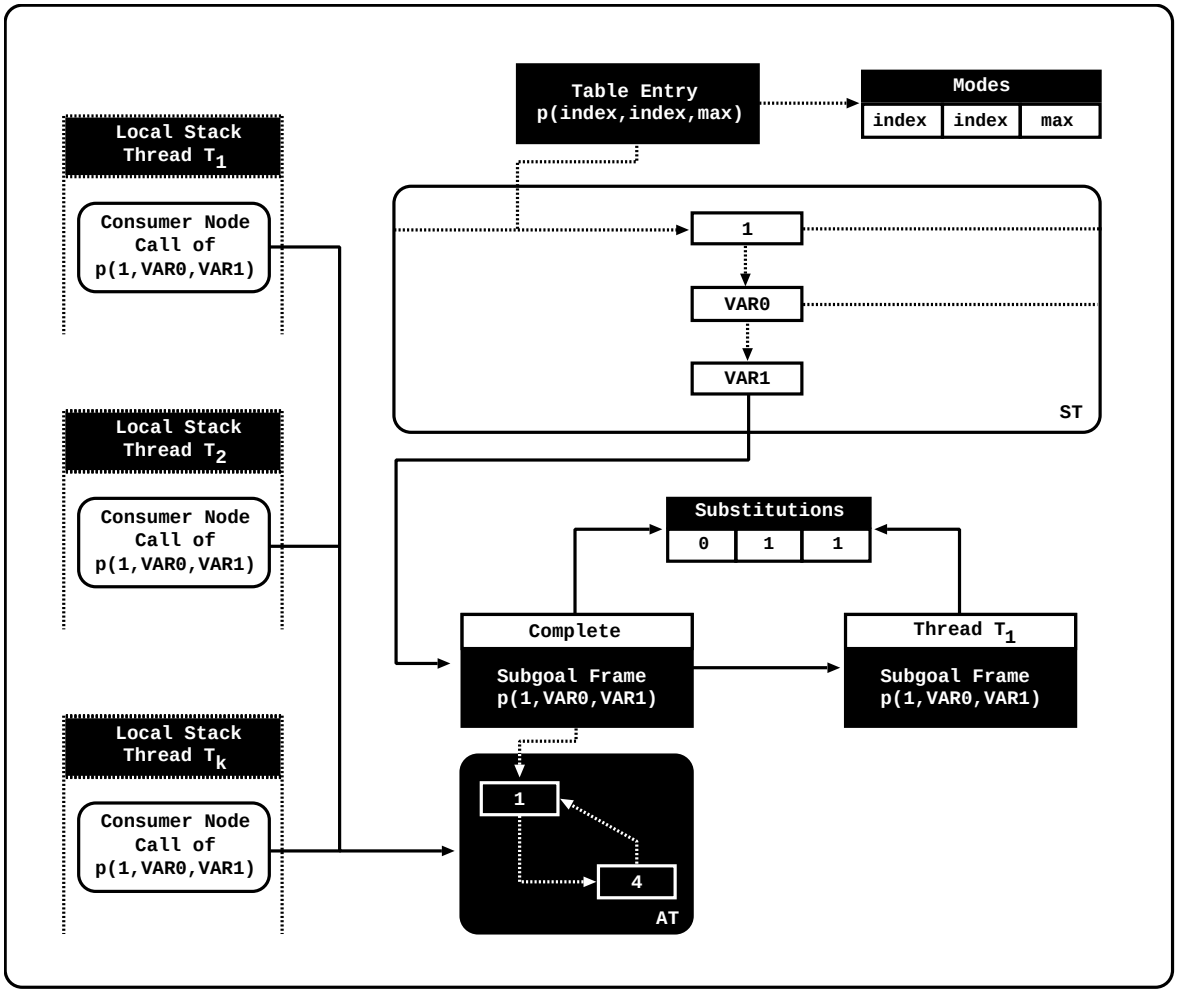

Figure 7.6: The third step for sharing answers in the SS design

public and the threads T_1, T_2 and T_k already using it in their consumer nodes [1]. The allocation of these consumer nodes have two major advantages: The first advantage is the fact that all answers become available sooner for all threads. The second advantage is that these consumer nodes are not in the process of subgoal call completion check, since when they were allocated the SF was already complete. For the sake of simplicity, we will not be discussing the completion check process, but the reader can keep the idea that the process is simpler with a low number of consumer nodes than with a higher number of consumer nodes. The intuitive notion is that the completion process requires that all consumer nodes to have consumed all answers in the AT data structure and when this is not the case, the evaluation is resumed in those consumer nodes.

[1]The values of NT and NT_1 in Equation 7.1 would be 3 and 2, respectively, since three threads used the subgoal call, but only two have evaluated the subgoal call in a private fashion. Additionally, to optimize the memory usage, we allowed thread T_1 to delete its private AT data structure.

Note that in Figure 7.6, thread T_1 is still alive and the SF data structure is still in the chain of the subgoal call, but the AT data structure was already deallocated. The deallocation of structures, in the original SS design, occurred only when a thread existed. With the shared answers support, the deallocation of structures is sightly different, since the SF and AT data structures might be allocated as private and later passed to public. Furthermore, since all threads can traverse the chain of subgoal frames, with the shared answers support, this type of structure is actually always public. Thus, upon the completion procedure of its own SF, thread T_1 proceeds as follows. It begins by the publishing procedure, but it finds other public SF and AT data structures. Hence, thread T_1 deallocates its AT data structure, but keeps SF in the chain to be deallocated later by the last living thread in the environment, which is always the main thread. At the end of the execution, the main thread proceeds as in the original SS design and deallocates all public data structures.

7.3 0-1 Knapsack Problem

The Knapsack problem [78] is a well-known problem in combinatorial optimization that can be found in many domains such as logistics, manufacturing, finance or telecommunications. Given a set of items, each with a weight and a profit, the goal is to determine the number of items of each kind to include in a collection so that the total weight is equal or less than a given capacity and the total profit is as much as possible. The most common variant of the problem is the *0-1 Knapsack problem*, which restricts the number of copies of each kind of item to be zero or one. In what follows, we will focus on this variant. The 0-1 Knapsack problem can be described by the following formulation:

$$KS = \begin{cases} max \sum_{i=1}^{N} p_i.x_i, \\ \sum_{i=1}^{N} w_i.x_i \leq C, x_i \in \{0,1\}. \end{cases} \qquad KS_R = \begin{cases} \forall_{i \in \{1,...,N\}}, w_i \leq C, \\ \sum_{i=1}^{N} w_i > C. \end{cases}$$

Given a set of items $i \in \{1, ..., N\}$, each with a weight $w_i \in \mathbb{N}^*$ and a profit $p_i \in \mathbb{N}^*$, and a Knapsack with capacity $C \in \mathbb{N}^*$, the formula KS defines the Knapsack problem, which is the maximum value obtained for the summatory of profits of the items in the Knapsack not exceeding the capacity C. The formula KS_R defines the restriction that avoids any trivial solution, by insuring that each item fits into the Knapsack and that the total weight of all items exceeds the Knapsack capacity.

7.3.1 Top-Down Approach

In a standard top-down approach that solves the *0-1 Knapsack problem*, an item i is excluded from or included in the knapsack whether it does not belong or belongs to the best solution of the problem. Figure 7.7 shows the evaluation tree of a Knapsack with N items and a capacity C. As expected, the tree is the binary combination of excluding and/or including the N items.

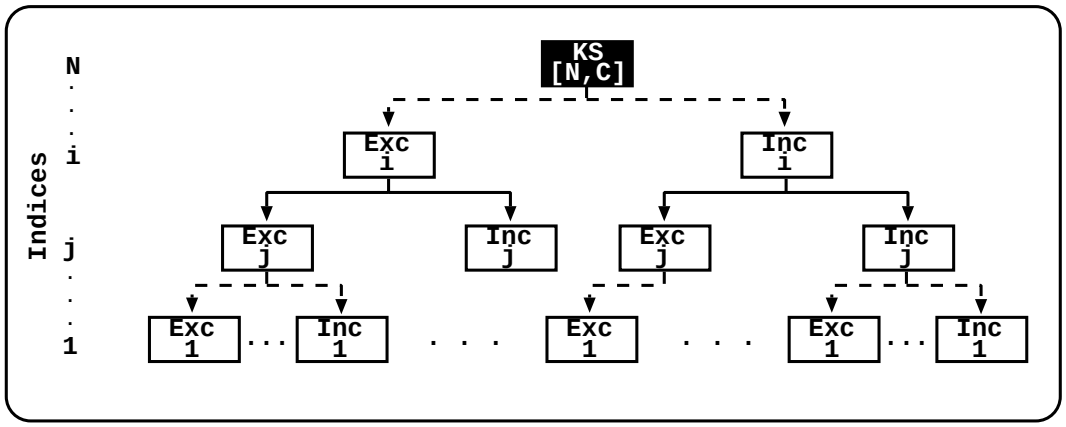

Figure 7.7: Knapsack top-down evaluation tree

In the black box of Figure 7.7 we have the top query call $KS[N, C]$, and from that point the evaluation traverses all items from the Nth item to the first item. The evaluation stores two temporary values, the capacity c and the profit p, which store the capacity available in the knapsack and the corresponding accumulated profit. In each level of the tree, an item i can be excluded or included in the Knapsack. When the condition $c - w_i \geq 0$ is false then the item can only be excluded. When the item i is included, the capacity c is updated with the value $c - w_i$, the profit p is updated with the value $p + p_i$ and the evaluation passes to the next level, i.e., passes to the item $j = i - 1$ in the figure. Finally, whenever the evaluation reaches the last level of the tree, the maximum value in p stores the best solution of the Knapsack.

Now the reader can observe that some of the sub-trees under the Exc_i and Inc_i nodes are equal, which means that the capacity and profits can be shared by both sub-trees, and to do so we will be using tabling. Next we introduce a standard top-down approach that solves the Knapsack problem using mode-directed tabling. Figure 7.8 shows our Yap's implementation adapted from [49] to include the profitability dimension.

The table directive declares that predicate ks with arity 3 (or $ks/3$ for short) is to be tabled using modes $(index, index, max)$, meaning that the third argument (the profit)

```
% tabling declaration
:- table ks(index, index, max).
% base case
ks(0, C, 0).
% exclude case
ks(I, C, P) :-
  I > 0, ks_exc(I, C, P, 1).
% include case
ks(I, C, P) :-
  I > 0, ks_inc(I, C, P, 1).
% exclude N items starting from I
ks_exc(I, C, P, N) :-
  J is I - N, ks(J, C, P).
% include I and
% exclude the next N-1 items
ks_inc(I, C, P, N) :-
  item(I, Ci, Pi), Cj is C - Ci,
  Cj >= 0, J is I - N,
  ks(J, Cj, Pj), P is Pi + Pj.
```

Figure 7.8: A top-down approach for the Knapsack problem with mode-directed tabling

should store only the maximal answers for the first two arguments (the index of the number of items being considered and Knapsack's capacity). The remaining part of the program implements a recursive top-down definition of the Knapsack problem. The first clause is the base case and defines that the empty set is a solution with profit 0. The second clause excludes the current item from the solution set and the third includes the current item in the solution if its inclusion does not overcome the current capacity of the Knapsack. For simplicity of integration with the parallel approach presented next, we are already using two auxiliary predicates, *ks_exc*/4 and *ks_inc*/4, as a way to implement the exclude and include cases. These auxiliary predicates take an extra argument N (fourth argument) that represents the number of items to jump (or exclude) in the recursion procedure. Here, for the sequential version of the problem, N is always 1, i.e, we always move to the next item.

To parallelize top-down dynamic programming algorithms, we followed Stivala et al.'s

work [121] where a set of threads solve the entire program independently but with a randomized choice of the sub-problems. Figure 7.9 shows a small example with two threads T_1 and T_2. Threads begin in the top query call $KS[N, C]$, but now on each level of the evaluation tree, they use a random function to decided which branch will be evaluated first (the exclude item branch or the include item branch). This random decision is aimed to disperse the threads through the evaluation tree.

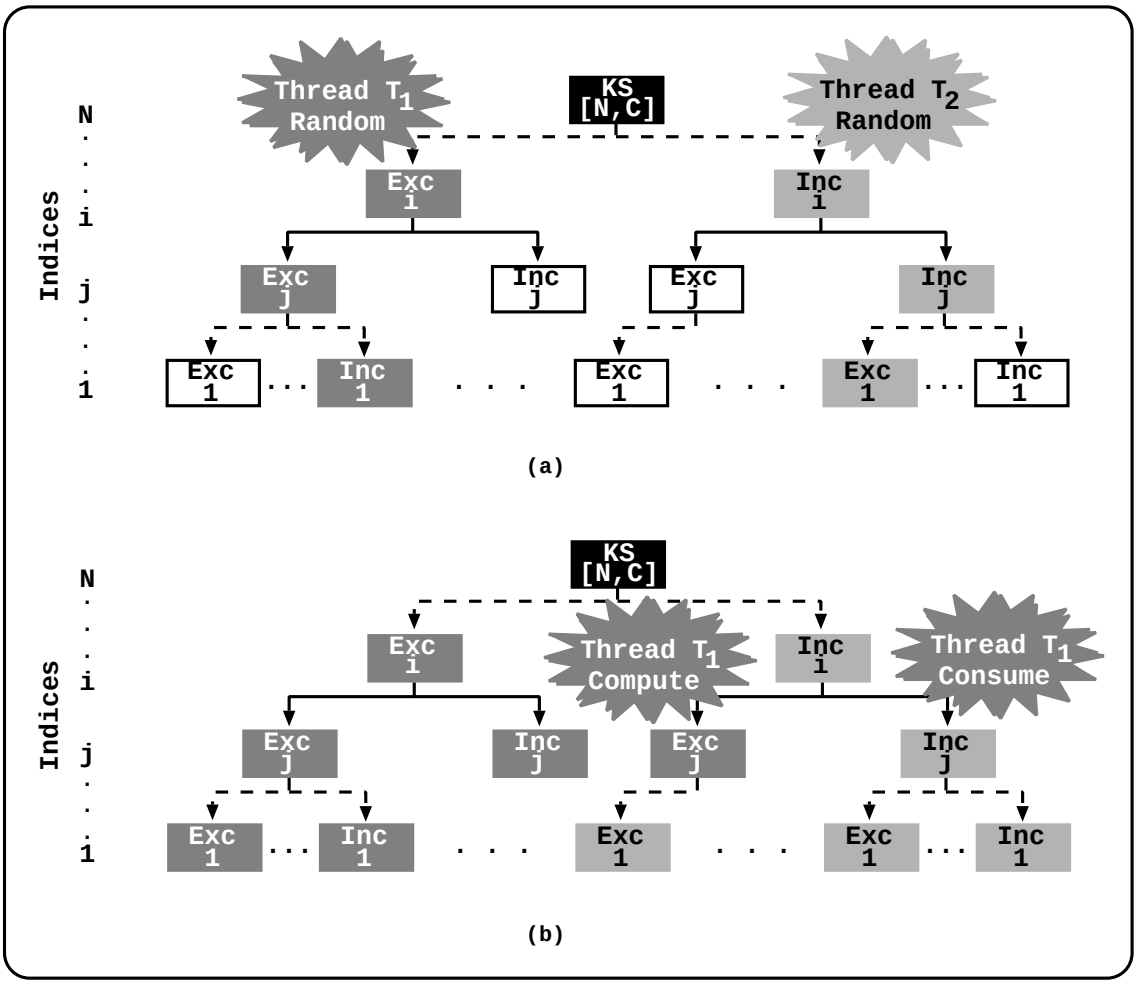

Figure 7.9: Knapsack top-down parallel evaluation tree

Figure 7.9(a) shows a situation where the thread T_1 is evaluating a left branch of the tree, while the thread T_2 is evaluating a right branch. Remember that although the threads are evaluating the branches of the tree in a random order, they have to evaluate all branches thus that they can find the optimal solution for the Knapsack. So, the random decision is only about the evaluation order of the branches and not about skipping branches. Figure 7.9(b) shows then a situation where thread T_1 has completely evaluated the Exc_i branches of the tree and is now evaluating the Inc_i

branches. Since the threads are using dynamic programming, thread T_1 computes the branch Exc_j that was not yet evaluated and consumes the results that were already evaluated for the branch Inc_j, which in this example was evaluated by thread T_2.

We can thus consider two alternative execution choices at each step: (i) exclude first and include next (as in the sequential version presented in Figure 7.8), or (ii) include first and exclude next. The randomized choice of sub-problems results in the threads diverging to compute different sub-problems simultaneously while reusing the sub-problem's results computed in the meantime by the other threads. Since the number of overlapping sub-problem is usually high in these kind of problems, it is expected that the available set of sub-problems to be computed will be evenly divided by the number of available threads resulting in less computation time required to reach the final result.

For the parallel version of the Knapsack problem, we have implemented two alternative versions. The first version simply follows Stivala et al.'s original random approach. The second version extends the first one with an extra step where the computation is first moved forward using a random displacement of the number of items to be excluded and only then the computation is performed for the next item, as usual. By doing this, it is expected that the sub-problems closer to the base cases are computed earlier, meaning that their subgoal frames are also marked as completed earlier, which avoids recomputation when other threads call the same sub-problems. Figure 7.10 shows the implementation. The difference between the two versions is that the first version does not consider the first extra clause in the $aux\_exc/4$ and $aux\_inc/4$ auxiliary predicates.

7.3.2 Bottom-Up Approach

A straightforward method to solve the Knapsack problem bottom-up, for a fixed capacity c, is to consider all 2^N possible subsets of the N items and choose the one that maximizes the profit. The recursive application of this algorithm to increasing capacities $c \in \{1, ..., C\}$, yields a Knapsack of maximum profit for the given capacity C [66]. The bottom-up characteristic comes from the fact that, given a Knapsack with capacity c and using i items, $i < N$, the decision to include the next item j, $j = i + 1$, leads to two situations: (i) if j is not included, the Knapsack profit is unchanged; (ii) if j is included, the profit is the result of the maximum profit of the Knapsack with the same i items but with capacity $c - w_j$ (the capacity needed to include the weight w_j of item j) increased by p_j (the profit of the item j being included). The algorithm then decides whether or not to include an item based on which choice

```
% tabling declaration
:- table ks(index, index, max).
% base case
ks(0, C, 0).
% random choice
ks(I, C, P) :-
  I > 0, random(2, maxRandom, N),
  R is N mod 2,
  ( R == 0 ->
    aux_exc(I, C, P, N)
  ;
    aux_inc(I, C, P, N)).
% try exclude first and include next
% not in the first version
aux_exc(I, C, P, N) :- ks_exc(I, C, P, N).
aux_exc(I, C, P, _) :- ks_exc(I, C, P, 1).
aux_exc(I, C, P, _) :- ks_inc(I, C, P, 1).
% try include first and exclude next
% not in the first version
aux_inc(I, C, P, N) :- ks_inc(I, C, P, N).
aux_inc(I, C, P, _) :- ks_inc(I, C, P, 1).
aux_inc(I, C, P, _) :- ks_exc(I, C, P, 1).
```

Figure 7.10: A top-down parallel version of the Knapsack problem with mode-directed tabling

leads to maximum profit. Thus, the equation in the formulation of the Knapsack problem is *serial monadic* [67], once all levels require solutions to sub-problems at the immediate preceding level (serial) and the equation has a single recursive term (monadic). Figure 7.11 shows the $KS[N, C]$ matrix that represents the dependencies in this approach. The rows define the items and the columns define the Knapsack capacities. The first column and row are filled with zeros, which are the initial profit for the Knapsack with no items or no capacity.

The sequential version of the algorithm can be constructed row by row or column by column. The computation of each sub-problem $KS[j, c]$ considers the maximum profitability obtained between $KS[j-1, c]$ and $KS[j-1, c-w_{j-1}]+p_j$. Thus, the black

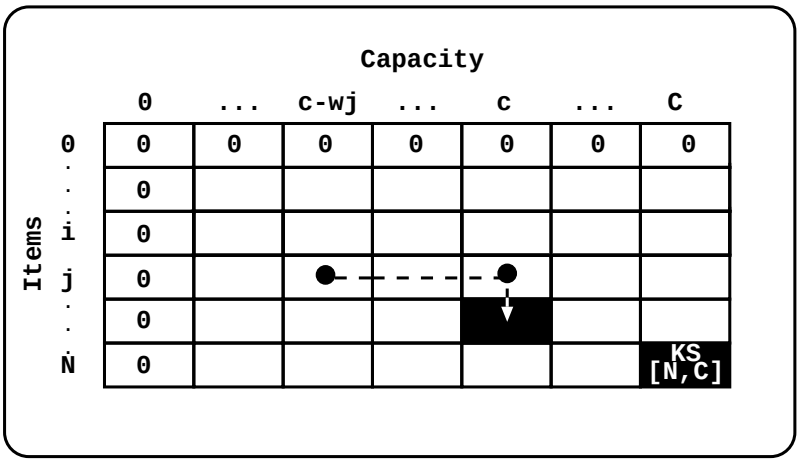

Figure 7.11: Knapsack bottom-up matrix

dots and the black cell within the evaluation matrix represents this dependency for the maximum profitability of the black cell that depends on the cells that have the dots. When all sub-problems are computed, $KS[N, C]$ holds the best profitability for the full problem. Figure 7.12 shows Yap's implementation. For simplicity of presentation, we are omitting the predicate that implements the main loop used to recursively traverse the matrix and launch the computation for each sub-problem.

```
% tabling declaration
:- table ks/3.
% base cases
ks(0, C, 0).
ks(I, 0, 0).
% item I exceeds capacity C
ks(I, C, P) :-
  I > 0, item(I, Ci, Pi), Ci > C,
  J is I - 1, ks(J, C, P).
% item I fits in capacity C
ks(I, C, P) :-
  I > 0, item(I, Ci, Pi), Ci =< C,
  Cj is C - Ci, Cj >= 0, J is I - 1,
  ks(J, Cj, Pj), P1 is Pj + Pi,
  ks(J, C, P2), max(P1, P2, P).
```

Figure 7.12: A bottom-up approach for the Knapsack problem with standard tabling

The table directive declares that predicate $ks/3$ is to be tabled using standard tabling. Since here a sub-problem can be computed from the results of its sub-problems, standard tabling is enough and there is no need for mode-directed tabling. The first two clauses of $ks/3$ are the base cases and define that the Knapsacks with no items or no capacity have profit 0. The third clause deals with the cases where an item's weight exceeds the Knapsack capacity and the fourth clause is the one that implements the main case discussed above.

Filling cells in subsequent rows requires accessing two cells from the previous row: one from the same column and one from the column offset by the weight of the current item. Thus, computing a row i depends only on the sub-problems at row $i-1$. A possible parallelization is, for each row, to divide the computation of the C columns between the available threads and then wait for all threads to complete in order to synchronize before computing the next row. Figure 7.13 shows an example with two threads T_1 and T_2, where the computation of the C columns within the evaluation matrix was divided in smaller chunks and those chunks were evaluated in the threads.

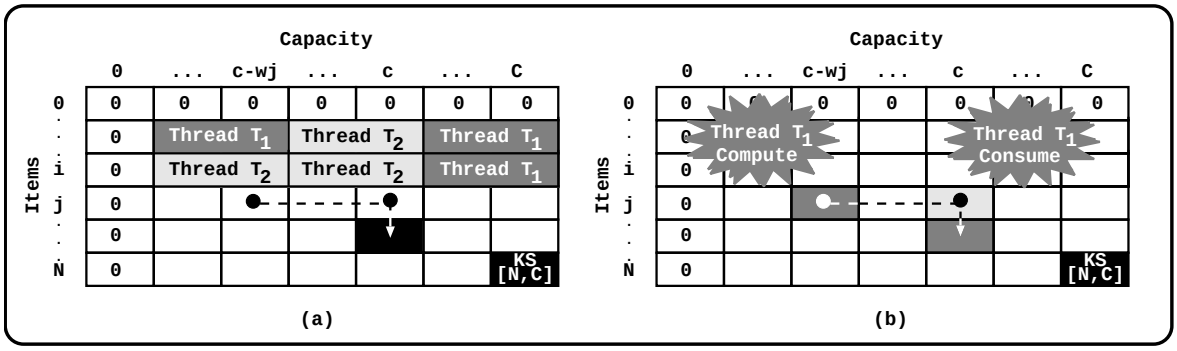

Figure 7.13: Knapsack bottom-up parallel matrix

Figure 7.13(b) shows a situation where the black cell in Figure 7.13(a) is evaluated by the thread T_1. To do so, it must have the values of the cell to which this cell depends. Since the threads are using dynamic programming, thread T_1 computes the cell that was not yet evaluated and consumes the results of the cell that was already evaluated, which in this example was evaluated by thread T_2.

Here, since we want to take advantage of the built-in tabling mechanism, which is implicit and cannot be controlled by the user, we want to avoid this kind of synchronization between iterations. Hence, when a sub-problem in the previous row was not computed yet (i.e., marked as completed in one of the subgoal frames for the given call), instead of waiting for the corresponding result to be computed by another thread, the current thread starts also its computation and for that it can recursively

call many other sub-problems not computed yet. Despite this can lead to redundant sub-computations, it avoids synchronization. In fact, as we will see, this strategy showed to be very effective.

We next introduce our generic multithreaded scheduler used to load balancing the access to a set of concurrent tasks. We assume that the number of tasks is known before execution starts and that tasks are numbered incrementally starting at 1. For the Knapsack problem, we will consider that the number of tasks is the number of capacities $c \in \{1, ..., C\}$ (alternatively, we could have considered the number of items $i \in \{1, ..., N\}$). In a nutshell, the scheduler uses a user-level *mutex* to protect a concurrent *queue* that stores the indices of the available tasks. In fact, since tasks are numbered incrementally, the queue simply needs to store the index of the next available task. When a thread gets access to the queue of tasks, it picks a chunk of consecutive tasks and updates the queue's stored index accordingly. Figure 7.14 shows the Prolog code that implements the main execution loop of each thread.

```
% initialize mutex
:- mutex_create(queueLock).
% initialize queue
:- set_value(queueIndex, 0).
% main execution loop
do_work(NumberOfTasks, ChunkSize) :-
  mutex_lock(queueLock),
  get_value(queueIndex, Current),
  ( Current = NumberOfTasks ->
    % terminate execution
    mutex_unlock(queueLock)
  ;
    First is Current + 1,
    Last is Current + ChunkSize,
    set_value(queueIndex, Last),
    mutex_unlock(queueLock),
    compute_tasks(First, Last),
    % get more work
    do_work(NumberOfTasks, ChunkSize)
  ).
```

Figure 7.14: The generic execution loop of each thread for the bottom-up approach

The top declarations initialize the *queueLock* mutex and the *queueIndex* queue. The predicate *do_work*/2 implements the main execution loop of each thread and is recursively executed until no more tasks exist in the queue. It receives two arguments: the total number of tasks in the problem ($NumberOfTasks$); and the chunk size to be considered when retrieving tasks from the queue ($ChunkSize$). In each loop, a thread starts by gaining access to the mutex and then it checks the queue. If the queue is empty, case in which the test $Current = NumberOfTasks$ succeeds[2], the mutex is released and the thread terminates execution. Otherwise, the thread picks a new chunk of consecutive tasks and updates the queue's stored index accordingly. Variables $First$ and $Last$ define the lower and upper bounds of the chunk of tasks obtained. The tasks are then evaluated using the *compute_tasks*/2 predicate, which calls the *ks*/3 predicate for the set of Knapsack sub-problems associated with the task. After the *compute_tasks*/2 finishes, the *do_work*/2 predicate is called again to get more tasks from the queue. The process repeats until no more tasks exist.

7.4 Longest Common Subsequence Problem

The problem of computing the length of the Longest Common Subsequence (LCS) is representative of a class of dynamic programming algorithms for string comparison that are based on getting a similarity degree. A good example is the sequence alignment, which is a fundamental technique for biologists to investigate the similarity between species. The problem can be described as follows: given a finite set of symbols S and two sequences $U = \langle u_0, u_1, ..., u_N \rangle$ and $V = \langle v_0, v_1, ..., v_M \rangle$ such that $\forall_{i \in 0,...,N}, u_i \in S$ and $\forall_{i \in 0,...,M}, v_i \in S$, we say that U has a common subsequence with V of length k if there are indices $i_0, i_1, ..., i_k, j_0, j_1, ..., j_k : 0 \leq i_0 < i_1 < ... < i_k \leq N$ and $0 \leq j_0 < j_1 < ... < j_k \leq M$ such that $\forall_{l \in 0,...,k}, u_{i_l} = v_{j_l}$. The length k is considered to be the longest common subsequence if it is maximal.

7.4.1 Top-Down Approach

In a standard top-down approach that solves the LCS problem, a symbol with an index i is included or excluded from the longest common subsequence whether it belongs or not belongs to the best solution of the problem. Figure 7.15 shows a level of the

[2]In order to avoid low-level details which are not relevant to this work, the reader can assume that $NumberOfTasks$ is a multiple of $ChunkSize$.

evaluation tree of the problem with two sequences u and v, such that the size of the sequence u is N and the size of the sequence v is M. The evaluation tree is then a 3-ary tree, which is the combination of changing the indices in the sequences u and v, whether the symbols in the indices are equal or different in both sequences.

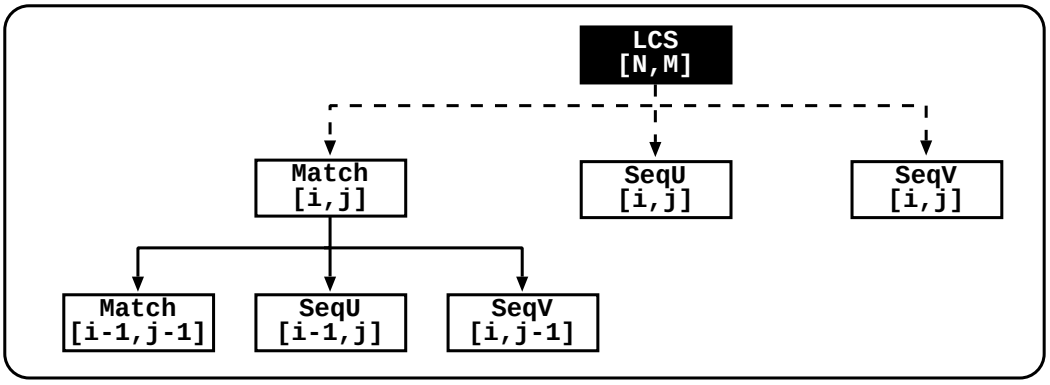

Figure 7.15: LCS top-down evaluation tree

In the black box of Figure 7.15 we have the top query call $LCS[N, M]$, and from that point the evaluation traverses all symbols of both sequences u and v from the Nth and Mth index to the last index, which is 0. The evaluation keeps one temporary value, that stores the maximum accumulated length l. In each level of the tree, a symbol with index i is included or excluded, whether the condition $u_i = v_j$ matches or not. When the condition is true (left-most sub-tree in the figure), both the index in both sequences decreases in one passing to the next index $i - 1$ and $j - 1$, and the l value is updated to $l + 1$. When the condition is false (middle and right branches in the sub-tree), the evaluation decreases the index of only of the sequences, thus in the middle branch the sequence u passes to the index $i - 1$, while the sequence v remains unchanged. In the right branch, the sequence u remains unchanged, while the sequence v passes to the index $j - 1$. Whenever the evaluation reaches the last level of the tree, the maximum value in l stores the best solution of the problem.

We next introduce a standard top-down approach that solves the LCS problem using mode-directed tabling. Figure 7.16 shows Yap's implementation adapted from [49].

The first two clauses of $lcs/3$ are the base cases defining that for empty sequences the LCS (third argument) is 0. The third clause deals with the cases where the current symbols in both sequences match (arguments I_u and I_v represent, respectively, the current indices in sequences U and V to be considered). The fourth and fifth clauses represent the opposite case, where the symbols do not match, and each clause moves one of the sequences to the next symbol (note that recursion is done in descending order

```
% tabling declaration
:- table lcs(index, index, max).
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0, symbol_u(Iu, S),
  symbol_v(Iv, S), Ju is Iu - 1,
  Jv is Iv - 1, lcs(Ju, Jv, Lj),
  L is Lj + 1.
% sequence U case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  lcs_u(Iu, Iv, L, 1).
% sequence V case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  lcs_v(Iu, Iv, L, 1).
% jump N symbols in sequence U
lcs_u(Iu, Iv, L, N) :-
  symbol_u(Iu, Su), symbol_v(Iv, Sv),
  Su =\= Sv,  Ju is Iu - N,
  lcs(Ju, Iv, L).
% jump N symbols in sequence V
lcs_v(Iu, Iv, L, N) :-
  symbol_u(Iu, Su), symbol_v(Iv, Sv),
  Su =\= Sv, Jv is Iv - N,
  lcs(Iu, Jv, L).
```

Figure 7.16: A top-down approach for the LCS problem with mode-directed tabling

until reaching index 0). Again, for simplicity of integration with the parallel approach presented next, we are already using two auxiliary predicates, *lcs_u*/4 and *lcs_v*/4, as a way to implement the unmatched cases. As for the Knapsack problem, these two auxiliary predicates take an extra argument N (fourth argument) that represents the number of symbols to jump in the recursion procedure. For the sequential version of

the problem, N is always 1, meaning that we always move to the next symbol.

Similarly to Knapsack's problem, to parallelize the LCS sequential top-down approach, we have implemented two alternative versions. The first version follows Stivala et al.'s original random approach. The second version extends the first one with an extra step where the computation is first moved forward using a random displacement of the number of symbols to jump and only then the computation is performed for the next symbol, as usual. The parallel evaluation tree of the LCS problem is similar to the Knapsack problem, the difference is that it has three branches to jump on each level instead of two. Figure 7.17 shows the implementation. The difference between the two versions is that the first version does not consider the first extra clause in the $aux\_u/4$ and $aux\_v/4$ auxiliary predicates.

7.4.2 Bottom-Up Approach

We now introduce our bottom-up approach to the LCS problem, which is based on [66]. In a nutshell, the bottom-up characteristic comes from the fact that, the maximum length of a common subsequence between two sequences u and v is: (i) if the initial symbols of both sequences match, then they are part of the longest common subsequence and the length of the longest common subsequence is the length of u and v both without the initial symbols plus one; (ii) if the initial symbols do not match then two situations arise: the longest common subsequence may be obtained from: sequence u and sequence v without its initial symbol; or sequence v and sequence u without its initial symbol. Since we want the longest subsequence, the maximum of these two must be selected. The following equation formulates the LCS problem as described above:

$$LCS[j,l] = \begin{cases} LCS[j-1,l-1]+1, & \text{if } u_j = v_l. \\ max\ \{LCS[j,l-1], LCS[j-1,l]\}, & \text{otherwise.} \end{cases}$$

The formulation is *non-serial monadic* [67], once each problem depends on sub-problems at the same or preceding level (non-serial) and the equation has a single recursive term (monadic). Figure 7.18 shows the LCS matrix that represents the dependencies in this approach. The rows define the indices to be considered in sequence u and the columns define the indices in sequence v. The first column and the first row are filled with zeros, meaning that for empty sequences the LCS is 0. The sequential version of the algorithm can be constructed row by row or column by column, since the computation

```
% tabling declaration
:- table lcs(index, index, max).
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  symbol_u(Iu, S), symbol_v(Iv, S),
  Ju is Iu - 1, Jv is Iv - 1,
  lcs(Ju, Jv, Lj), L is Lj + 1.
% random choice
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0,
  random(2, maxRandom, N),
  R is N mod 2,
  ( R == 0 ->
    aux_u(Iu, Iv, L, N)
  ;
    aux_v(Iu, Iv, L, N)).
% try sequence U first and V next
% not in the first version
aux_u(Iu, Iv, L, N) :-  lcs_u(Iu, Iv, L, N).
aux_u(Iu, Iv, L, _) :-  lcs_u(Iu, Iv, L, 1).
aux_u(Iu, Iv, L, _) :-  lcs_v(Iu, Iv, L, 1).
% try sequence V first and U next
% not in the first version
aux_v(Iu, Iv, L, N) :-  lcs_v(Iu, Iv, L, N).
aux_v(Iu, Iv, L, _) :-  lcs_v(Iu, Iv, L, 1).
aux_v(Iu, Iv, L, _) :-  lcs_u(Iu, Iv, L, 1).
```

Figure 7.17: A top-down parallel version of the LCS problem with mode-directed tabling

of each sub-problem $LCS[j, l]$ only depends on the sub-computations done for the preceding row and column. At the end, $LCS[N, M]$ holds the LCS for the problem.

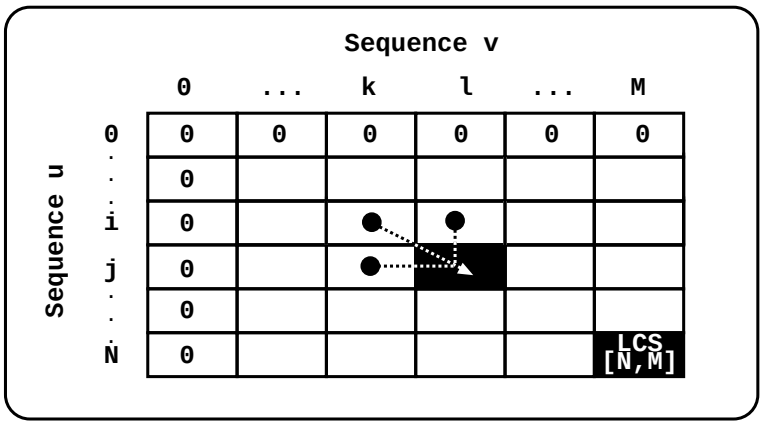

Figure 7.18: LCS bottom-up matrix

Figure 7.19 shows Yap's implementation. Again, for simplicity of presentation, we are omitting the predicate that implements the main loop used to recursively traverse the matrix and launch the computation for each sub-problem.

```
% tabling declaration
:- table lcs/3.
% base cases
lcs(I, 0, 0).
lcs(0, I, 0).
% matched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0, symbol_u(Iu, S),
  symbol_v(Iv, S), Ju is Iu - 1,
  Jv is Iv - 1, lcs(Ju, Jv, Lj),
  L is Lj + 1.
% unmatched case
lcs(Iu, Iv, L) :-
  Iu > 0, Iv > 0, symbol_u(Iu, Su),
  symbol_v(Iv, Sv), Su =\= Sv,
  Ju is Iu - 1, Jv is Iv - 1,
  lcs(Ju, Iv, L1), lcs(Iu, Jv, L2),
  max(L1, L2, L).
```

Figure 7.19: A bottom-up approach for the LCS problem with standard tabling

The table directive declares that predicate *lcs*/3 is to be tabled using standard tabling.

The first two clauses of *lcs*/3 are the base cases and the third and fourth clauses deal with the cases where the initial symbols of both sequences match and do not match, respectively.

Concerning the parallelization of the matrix, a possible approach is, for each row, divide the computation of the M columns between the available threads or, for each column, divide the computation of the N rows between the available threads. Here, we will follow the same approach as for the Knapsack problem and we will use the generic multithreaded scheduler that implements the thread execution loop presented in Figure 7.14. The number of concurrent tasks to be considered is the size of sequence u (alternatively, we could have considered the size of sequence v) and the evaluation of the *compute_tasks*/2 predicate calls the *lcs*/3 predicate for the set of LCS sub-problems associated with a given task.

7.5 Performance Evaluation

In this section, we evaluate the performance of the YapTab-Mt framework on both problems using the multithreaded top-down and bottom-up approaches. To do so, we used the SS design with support for shared answers[3] together with the LF_2 proposal to support concurrency within the ST data structure, and the TabMalloc combined with the TcMalloc as the memory allocator. To put our results in perspective, we also experimented with XSB Prolog version 3.4.0, using the shared tables model.

For the Knapsack problem, we fixed the number of items and capacity, respectively, 1600 and 3200. For the LCS problem, we used both sequences with a fixed size of 3200 symbols each. Then, for each problem, we created three different datasets, D_{10}, D_{30} and D_{50}, meaning that the values for the weights/profits for the Knapsack problem and the symbols for LCS problem where randomly generated in an interval between 1 and 10%, 1 and 30% and, 1 and 50% of the total number of items/symbols, respectively. For the top-down approaches, we only experimented with YAP since XSB does not support mode-directed tabling. We tested YAP in both problems, without randomization (YAP_{TD_0}), and with randomization using Stivala et al.'s original random version (YAP_{TD_1}) and the extended version using the extra random displacement clause (YAP_{TD_2}). For both Knapsack and LCS problems, in the randomized versions we used a *maxRandom* value corresponding to 10% of the total number of items/symbols

[3]Note that without this support, the YapTab-Mt framework would have overhead results instead of speedups, once every thread would be required to compute every answer for every subgoal call.

in the problem. For the bottom-up approaches, we experimented with Yap (YAP_{BU}) and XSB (XSB_{BU}) and we used a *ChunkSize* value of 5.

Table 7.1 and Table 7.2 show the average of 10 runs results obtained, respectively, for the Knapsack and LCS problems for both top-down and bottom-up approaches using the YAP and XSB Prolog systems. In particular, the columns of both tables show the following information: (i) the first two columns show the system and the dataset used; (ii) the third column shows the sequential execution time (T_{seq}). In T_{seq}, the Prolog systems where compiled without multithreaded support and all multithreaded instructions were removed from the Prolog code; (iii) the next five columns show the execution time for one thread (column **Time ($\mathbf{T}_1$)**) and the corresponding speedup, for the execution with 8, 16, 24 and 32 threads (columns **Speedup ($\mathbf{T}_1/\mathbf{T}_p$)**); (iv) The last column resumes the best execution time (T_{best}) obtained from the previous columns, where the results in bold highlight the best execution time (or speedup) obtained for each system/dataset configuration.

Analyzing the general picture of both tables, one can observe that for both problems, the speedup columns of the top-down YAP_{TD_0} approach show not considered (*n.c.*) results, because without randomization the approach is unable to scale because threads would evaluate every subgoal call in the same order, thus causing a worst case scenario similar to the ones presented in the previous chapters. When comparing the T_{seq} and T_1 results of YAP_{TD_0} with YAP_{TD_1} and YAP_{TD_2} approaches, we can observe that the randomized evaluation has an important cost. This can be explained by two situations, one is the usage of random function itself and other if the fact that the Prolog code is sightly more complex than without randomization. However, the top-down YAP_{TD_2} and bottom-up YAP_{BU} approaches have the best results with excellent speedups for 8, 16, 24 and 32 threads. In particular, for 32 threads, they obtain speedups around 21 and 20, respectively, for the Knapsack and LCS problems. The results for the top-down YAP_{TD_1} approach are not so interesting, regardless of the fact that it can slightly scale for the Knapsack problem up to 16 threads.

Note that, despite the similar average speedups for the YAP_{TD_2} and YAP_{BU}, their execution times are quite different. For example, consider the D_{50} dataset of the Knapsack problem with 32 threads, while the speedup 20.62 of YAP_{TD_2} corresponds to an execution time of 1.233 seconds, the speedup 21.76 of YAP_{BU} only corresponds to 0.804 seconds. Similarly for the LCS problem, if considering the D_{50} dataset with 32 threads, while the speedup 19.58 of YAP_{TD_2} corresponds to $2,255$ seconds, the speedup 20.52 of the YAP_{BU} only corresponds to $1,406$ seconds.

Table 7.1: Execution time, in milliseconds, for one thread (sequential and multi-threaded version) and corresponding speedup against one thread the multithreaded version, for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the Knapsack problem using the YAP and XSB Prolog systems

System/Dataset		Seq. Time (T_{seq})	# Threads (p)					Best Time (T_{best})
			Time (T_1)	Speedup (T_1/T_p)				
			1	8	16	24	32	
Top-Down Approaches								
YAP_{TD_0}	D_{10}	**9,508**	12,415	n.c.	n.c.	n.c.	n.c.	9,508
	D_{30}	**9,246**	12,177	n.c.	n.c.	n.c.	n.c.	9,246
	D_{50}	**9,480**	12,589	n.c.	n.c.	n.c.	n.c.	9,480
YAP_{TD_1}	D_{10}	14,330	19,316	1.96	**2.12**	2.04	1.95	9,115
	D_{30}	14,725	19,332	3.57	**4.17**	4.06	3.93	4,639
	D_{50}	14,729	18,857	4.74	6.28	**6.44**	6.41	2,930
YAP_{TD_2}	D_{10}	19,667	24,444	6.78	12.35	15.44	**18.19**	1,344
	D_{30}	19,847	25,609	7.15	13.83	17.37	**20.47**	1,251
	D_{50}	19,985	25,429	7.27	13.70	17.35	**20.62**	1,233
Bottom-Up Approaches								
YAP_{BU}	D_{10}	12,614	17,940	7.17	13.97	18.31	**22.15**	0,810
	D_{30}	12,364	17,856	7.23	13.78	18.26	**21.94**	0,814
	D_{50}	12,653	17,499	7.25	14.01	18.34	**21.76**	0,804
XSB_{BU}	D_{10}	**32,297**	38,965	0.87	0.66	0.62	0.55	32,297
	D_{30}	**32,063**	38,007	0.86	0.61	0.56	0.53	32,063
	D_{50}	**31,893**	38,534	0.84	0.58	0.57	0.57	31,893

Regarding the base execution times with one thread, YAP_{TD_2} clearly pays the cost of the extra clause with an average execution time around 1.3 to 1.5 times slower than YAP_{TD_1} and YAP_{BU}. In this regard, comparing the execution time for one thread (T_1) with the execution time of the sequential Prolog engine (T_{seq}), i.e., without thread support and without mutex in the Prolog code, we observed an average overhead (T_1/T_{seq}) around 1.3 to 1.4 times. For example, if we consider the D_{50} dataset of the Knapsack problem, this means that the speedups to T_{seq} for the execution with 8, 16, 24 and 32 threads are, respectively, 5.71, 10.77, 13.63 and 16.21 for YAP_{TD_2} and 5.25, 10.13, 13.26 and 15.74 for YAP_{BU}. We thus argue that, even if we consider the sequential Prolog engine as the base for comparison, our results still show excellent speedups for the execution with 8, 16, 24 and 32 threads. The executions times in the

Table 7.2: Execution time, in milliseconds, for one thread (sequential and multi-threaded version) and corresponding speedup against one thread the multithreaded version, for the execution with 8, 16, 24 and 32 threads, for the top-down and bottom-up approaches of the LCS problem using the YAP and XSB Prolog systems

System/Dataset		Seq. Time (T_{seq})	# Threads (p)					Best Time (T_{best})
			Time (T_1)	Speedup (T_1/T_p)				
			1	8	16	24	32	
Top-Down Approaches								
	D_{10}	**21,191**	26,225	n.c.	n.c.	n.c.	n.c.	21,191
YAP_{TD_0}	D_{30}	**20,809**	26,146	n.c.	n.c.	n.c.	n.c.	20,809
	D_{50}	**20,775**	26,028	n.c.	n.c.	n.c.	n.c.	20,775
	D_{10}	26,030	33,969	**1.58**	1.53	1.50	1.42	21,509
YAP_{TD_1}	D_{30}	26,523	34,213	**1.60**	1.54	1.50	1.42	21,424
	D_{50}	26,545	34,234	**1.60**	1.54	1.51	1.40	21,408
	D_{10}	34,565	44,371	7.23	13.23	16.45	**19.74**	2,248
YAP_{TD_2}	D_{30}	34,284	44,191	7.12	13.09	16.52	**19.77**	2,235
	D_{50}	33,989	44,158	7.06	13.30	16.49	**19.58**	2,255
Bottom-Up Approaches								
	D_{10}	20,799	28,909	6.47	12.21	16.48	**20.32**	1,423
YAP_{BU}	D_{30}	21,174	28,904	6.94	12.61	16.63	**20.40**	1,417
	D_{50}	21,166	28,857	6.44	12.31	16.44	**20.52**	1,406
	D_{10}	**60,983**	74,108	n.a.	n.a.	n.a.	n.a.	60,983
XSB_{BU}	D_{30}	**59,496**	74,410	n.a.	n.a.	n.a.	n.a.	59,496
	D_{50}	**59,700**	74,628	n.a.	n.a.	n.a.	n.a.	59,700

T_{best} column confirm this idea, showing that YAP_{BU} has the best execution times of all systems in all datasets.

Regarding the comparison with XSB's shared tables model, YapTap-Mt's results clearly outperform those of XSB. For the execution time with one thread, XSB shows higher times than all Yap's approaches (around two times the execution times for YAP_{TD_1} and YAP_{BU}). For the parallel execution of the Knapsack problem, XSB shows no speedups and for the parallel execution of the LCS problem we have no results available (*n.a.*) since we got *segmentation fault* execution errors. From our point of view, XSB's results are a consequence of the *usurpation operation* [77] that restricts the potential of concurrency to non-mutually dependent sub-computations. As the parallel versions of the Knapsack and LCS problems create mutual dependent sub-computations, which

can be executed in different threads, the XSB is actually unable to execute them in a parallel fashion. By other works, even if we launch an arbitrary large number of threads on those programs, the system would tend to use only one thread at the end to evaluate most of the computations.

7.6 Non-Prolog Related Work

Our framework provides the ground technology for multithreaded dynamic programming. From the user's point of view, it can be enabled through the use of single instructions of the form '*:- table p/n*', meaning that common sub-computations for *p/n* will be synchronized and shared between threads at the engine level, i.e., at the level of the tables where the results for such sub-computations are stored. Nevertheless, the user still needs to explicitly implement the thread management and scheduler policy for task distribution, which is orthogonal to the focus of our work. In any case, high-level predicates or libraries, like the generic multithreaded scheduler presented in Figure 7.14, can be easily develop on top of our framework to accomplish such tasks. To put our framework in perspective, we next briefly discuss and compare it with others available outside Prolog's world.

For functional programming languages, the Eden [75] and HDC [59] Haskell based frameworks allow the users to express their programs using polymorphic higher-order functions. Eden is a general-purpose parallel functional language suitable for developing sophisticated skeletons as well as for exploiting more irregular parallelism that cannot be easily captured by a predefined skeleton. HDC stands for *higher-order divide-and-conquer* and was originally developed for the parallelization of divide-and-conquer recursions, but is also appropriate for programming skeletons of any kind. Both frameworks showed the efficiency of these type of languages by presenting relevant speedups in benchmarks such as the Karatsuba, the N-Queens and the parallel computation of the Gröbner basis.

For object-oriented programming languages, the MALLBA [2] and DPSKEL [89] frameworks also showed relevant speedups in the parallel evaluation of combinatorial optimization benchmarks. MALLBA tackles the resolution of combinatorial optimization problems using algorithmic skeletons implemented in C++. Several skeletons are available, such as, divide-and-conquer, branch-and-bound, dynamic programming, hill climbing, among many others. DPSKEL is a skeleton tool for dynamic programming problems. In particular, DPSKEL used dynamic programming to solve the Knapsack

and LCS problems in a shared memory architecture, where it obtained maximum speedups of 6.63 and 8.15 for 8 threads, on the Knapsack benchmark with 1600 items and a capacity of 3200 and the LCS benchmark with sequences 3000 items, respectively. These speedups are in line with the speedups obtained with our approach.

Comparing our top-down results with Stivala et al.'s work [121], we can observe comparable results for the Knapsack problem and slight worst results for the LCS problem with YAP_{TD_1}, but significant better results with YAP_{TD_2}. For the Knapsack problem, Stivala et al.'s presented results for speedups over the sequential time (time without the multithreaded support, i.e., same T_{seq} presented in the Table 7.1) in 100 instances each of uncorrelated, weakly correlated, strongly correlated, inverse strongly correlated and almost strongly correlated Knapsack problems, each with 500 items and weights in the interval $[1, 500]$. The maximum speedups obtained were 8.31 with 31 threads on a *UltraSPARC T1* architecture, 3.11 with 8 threads on a *IBM PowerPC - 8 cores* architecture and 3.21 with 8 threads on a *AMD Quad Core Opteron - 8 cores* architecture.

Comparing our bottom-up results, they are also quite relevant when compared with similar approaches in the literature. For example, for the Knapsack problem, our bottom-up YAP_{BU} approach has similar speedups for 8 threads and better for 16 threads if compared with a multithreaded implementation using the classic and Morales parallelization of the Knapsack problem [101]. The work used *Intel Core 2 Duo - 2 cores* and *Intel Core 2 Quad - 4 cores* architectures and the classic parallelization used OpenMP, while the Morales used Pthreads. The Knapsack problem with capacity 10000 and 10000 items generated using the procedure described in [91]. The maximum speedup obtained over the sequential execution was about 7.80 for the classic parallelization and about 5.10 for the Morales parallelization, both of them obtained for 8 threads. For the LCS problem, our bottom-up YAP_{BU} approach shows similar base execution times (with one thread) for sequences of identical sizes, but far better speedups than parallel CUDA, OpenCL and OpenMP versions of the problem [36]. The work used *Intel Core(TM) 2 Quad - 4 cores* with *Nvidia GT 430 - 96 cores* architecture, with parallelization based on [66] (same as our bottom-up approach). For two sequences with fixed sizes of 4000 symbols, the best results were obtained using CUDA with a speedup of about 13.80, while OpenCL and OpenMP had speedups of about 10.20 and 3.20, respectively.

7.7 Chapter Summary

Starting from two well-known dynamic programming problems, the Knapsack and the Longest Common Subsequence problems, we have discussed how we were able to scale their execution by taking advantage of the multithreaded tabling engine of the Yap Prolog system. A key contribution of this work is our new asynchronous version of the table space data structures, where threads view their tables as private but are able to use the answers of a sub-problem, if another thread has already computed them.

We have presented multithreaded tabled top-down and bottom-up approaches using, respectively, Yap's mode-directed tabling support and Yap's standard tabling support. Our experiments, showed that using either top-down or bottom-up techniques, we were able to scale the execution of both problems by taking advantage of the state-of-the-art multithreaded tabling engine of the YapTab-Mt framework.

Chapter 8

Concluding Remarks

In this final chapter, we summarize the main contributions of the thesis and we outline some directions for the further work.

8.1 Main Contributions

The ultimate goal, of this thesis was to answer the question of either a Prolog system with a tabling engine could or could not scale effectively the execution of logic programming applications using tabling and multithreading. So far, the only system that combined multithreading with tabling was XSB Prolog [125] but, from our point of view, the results obtained by XSB Prolog were far from the potentialities of the combination. We believe that this thesis contributes to support the idea the answer that *yes, we can* be able to scale effectively the execution time of logic programming applications using tabling and multithreading combined.

The starting point of our work was XSB's approach for multithreaded tabling. We have studied both XSB's designs, *private tables* and *shared tables*, and understood their limitations. In the private tables design, the threads simply do not share any information, thus no scalability can be achieved with this design. In the shared tables design, the XSB Prolog system uses an *usurpation operation* [77] that restricts the potential of concurrency to non-mutually dependent sub-computations, since when a set of subgoals computed by different threads is mutually dependent, then a usurpation operation synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads. Thus, in applications that create mutual dependent sub-computations, XSB is actually unable to execute in

a concurrent parallel fashion due to this operation. By other works, even if we launch an arbitrary large number of threads on those programs, the system would tend to use only one thread at the end to evaluate most of the sub-computations. Additionally, we believe that the usurpation algorithm is also too costly, to be used with either local and batched scheduling, because, during the execution, it has to keep track of the dependent subgoals calls being called by different threads. For batched scheduling, this can be potentially a bigger problem, since batched scheduling is known to create higher subgoal call dependency than local scheduling.

Having both XSB designs in mind, we have defined two main goals for this thesis. The first was to present alternative designs for concurrent tabling and to effectively understand their advantages and limitations. The second goal was to implement an efficient multithreaded tabling framework that could use both local and batched scheduling and be used in multiple domains. Thus, the system had be as robust as possible, meaning that it had to be capable of correctly evaluate an huge class of problems written in Prolog.

To support our experiments we took advantage of a test suite engine that we had previously created and we started adjusting it to support multithreaded tabling evaluations. Actually, the engine has about 5 GBytes of data between several different tests, benchmarks and their solutions/tables produced. The engine is capable of comparing running time results and test the correctness of the program's solutions and tables obtained for either the Yap, XSB or B-Prolog systems. The test suite includes sets of different path problem definitions and transition graphs, model checking tests and basic tests to evaluate particular situations and programs obtained from the OpenRuleBench project [73].

While adjusting the test suite engine, we started the journey of combining multithreading with tabling. This journey took us to different domains in the parallel programming paradigm, such as concurrent memory allocators, concurrent data structures and top-down and bottom parallelization techniques applicable to dynamic programming problems. We then summarize the main contributions of our work.

Novel concurrent table space designs. We have presented three novel designs for concurrent table spaces and implemented them in the YapTab-Mt framework. For each design we have shown also a detailed memory usage analysis. These designs can be seen as alternative trade-offs between concurrency and memory usage. The first design (No Sharing), avoids concurrency by allowing threads to use all table space in a private fashion. In the second design (Subgoal Sharing),

threads share part of the table space in a concurrent fashion, while in the third design (Full Sharing), threads fully share the table space.

Lock-based concurrent table space designs. For the initial implementation of the concurrent table space designs in YapTab-Mt, we used four types of locking schemes to support the SS and FS designs: standard locks, try-locks, global locks and global try-locks, and we have compared them against the NS design with 1 thread using worst case scenarios. To do so, we scaled in intervals of 8 threads starting with 1 thread and ending with 32 threads. The best (minimum, average and maximum) overhead ratios obtained on worst case scenarios with 32 threads were, respectively:

- 1.33, 12.94 and 26.67 for the NS design;
- 1.18, 11.16 and 25.91 for the SS design;
- 1.34 (global locks and try-locks), 5.72 (try-locks) and 10.02 (global try-locks) for the FS design.

The TabMalloc memory allocator. We have presented a novel, efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs, and combined it with four different state-of-the-art memory allocators, namely, PtMalloc, Hoard, TcMalloc and JeMalloc. TabMalloc is based on local and global pages, that splits memory among specific data structures and different threads, together with a page based mechanism, where data structures of the same type are pre-allocated within a page. Our experimental results showed that we were successful in our goal of minimizing the performance degradation that YapTab-Mt suffered, when exposed to simultaneous memory requests made by multiple threads. The best (minimum, average and maximum) overhead ratios obtained on worst case scenarios with 32 threads were, respectively:

- 1.05, 1.51 and 2.52 (all best ratios obtained with TabMalloc combined with TcMalloc) for the NS design;
- 1.07, 1.71 and 2.61 (all best ratios obtained with TabMalloc combined with TcMalloc) for the SS design;
- 1.34 (PtMalloc used solely), 3.51 (TabMalloc combined with TcMalloc) and 7.42 (TabMalloc combined with Hoard) for the FS design.

Novel proposals for lock-free tries. We have presented two proposals, named LF_1 and LF_2, for lock-free tries, specially aimed for environments that do not require support for concurrent delete operations. The LF_1 proposal implements dynamic

resizing of the hash tables by doubling the size of the bucket entries in the hash, whenever a threshold limit for hash collisions is reached. Since the size of the hashes doubles, we could not efficiently integrate this proposal with the TabMalloc memory allocator, which requires the usage of fix-sized data structures and pages. The LF_2 proposal is based on lock-free hash tries and is aimed to be a simpler and more efficient lock-free proposal that disperses the concurrent areas as much as possible in order to minimize problems such as false sharing or cache memory ping-pong effects. Experimental results obtained with a external framework (i.e., not within YapTab-Mt) showed that the LF_2 proposal could effectively reduce the execution time and scale better than some of the best-known currently available lock-free hashing implementations. Within YapTab-Mt, the best (minimum, average and maximum) overhead ratios obtained on the worst case scenarios with 32 threads were, respectively:

- 1.07, 1.54 and 2.52 (all best ratios obtained with LF_2 proposal) for the SS design;
- 1.28, 3.03 and 6.54 (all best ratios obtained with LF_2 proposal) for the FS design.

Private consumer chaining. During the implementation of the FS design, we observed a bottleneck in the procedure of chaining answers to be used by the consumer nodes. To avoid this bottleneck, we moved the chaining procedure from public to private, i.e., we removed the chaining procedure from the answer tries and we moved it to a private chaining procedure that only affects the thread that is doing it. Later, when a evaluation is complete, i.e, when a subgoal call is marked as complete, we put one of the private chains as public, so that, from that point on, all threads can use that chain in complete mode (only for reading). Experimental results showed that by using a private consumer chaining process we were able to improve significantly the behavior of the FS design. The best (minimum, average and maximum) overhead ratios obtained on the worst case scenarios with 32 threads were, respectively, 1.33, 2.24 and 3.71.

Batched scheduling. We have presented a performance analysis comparison between local and batched scheduling for the NS, SS and FS designs. Experimental results showed that both local and batched scheduling perform similarly in worst case scenarios, even though that local scheduling showed to be on average slightly better that batched scheduling. For batched scheduling, the best (minimum, average and maximum) overhead ratios obtained on the worst case scenarios with 32 threads were, respectively:

- 1.04, 1.49 and 2.63 for the NS design;
- 1.12, 1.51 and 2.62 for the SS design;
- 1.26, 2.41 and 4.51 for the FS design.

Sharing completed tables. Small extension to the SS design to support answer sharing after a subgoal call is complete. A key contribution of this extension is that threads view their tables as private but are able to use the answers of a subgoal call, if another thread has already computed them. We showed how to take advantage of this extension in two real world dynamic programming problems using multithreaded tabled top-down and bottom-up approaches. Our experiments, showed that using either top-down or bottom-up techniques, we were able to scale the execution of both problems. We hope that in the future this simple design could be adopted by other Prolog systems, such as the XSB Prolog. The best speedups obtained for the YapTab-Mt, with 32 threads, for the Knapsack and LCS problems were, respectively, 22.15 and 20.52.

The results obtained in YapTab-Mt throughout this thesis reinforced our belief that multithreaded combined with tabling is a very good combination that can contribute to expand the range of applications in Logic Programming.

8.2 Further Work

We hope that the work resulting from this thesis will be a basis to conduct further improvements and further research in this area. YapTab-Mt has achieved our initial goal. Even so, the system still has some limitations that may reduce its use elsewhere and its contribution to general Prolog applications. Current limitations are mostly related with issues not within the scope of the present work, but that are very important for wider use throughout the logic programming community. We next suggest some topics for future work:

Lock-free bucket array of entries. An alternative approach for the implementation of the bucket array of entries data structure presented in the Subsection 3.3.2 would be to apply the LF_2 proposal. Applying the LF_2 proposal would be possible because no deletion operation is required in this data structure.

Extending FS design to support mode-directed tabling. This feature would allow the FS design to exploit the advantages of mode-directed tabling, such as the

usage of modes to specify how the answers are inserted into the table space [116]. In the previous chapter we observed the advantages of combining the SS design with mode-directed tabling. However, in the SS design, the answers of tables are only shared among threads after they are completed. With FS design threads would be able to share the answers sooner, once this design does not require the completion of tables to share the answers. Thus, it would interesting to analyze the performance results of the YapTab-Mt framework, when combining the FS design with mode-directed tabling, using modes that prune the evaluation space, such as *min* and *max* modes. Using these modes, we would expect that the pruning effect of inserting an answer in a table by one thread, would be propagated to all remaining threads, and this would be expected to improve the overall performance of the YapTab-Mt framework.

Extending LF_2 to support concurrent delete operations. This feature would allow the LF_2 proposal to be used in other domains and applications outside the YapTab-Mt, such as dictionaries or set comparison. Inside the tabling world, this extension could be applied in concurrent incremental tabling [113], where specific subgoal calls and answers are deleted during the evaluation of tabled logic programs.

Implicit parallelism. In the work [105], Rocha presented the OPTYap framework exploits implicit or-parallelism from tabled logic programs by considering all subgoals as being parallelizable (subgoals from tabled or non-tabled predicates). Due to the good performance results obtained with OPTYap, one possible direction for a further work, would be to combine YapTab-Mt with OPTYap and allow the Yap system to support the simultaneous usage of implicit and explicit parallelism. This would imply an extensive research about both systems and how they could be integrated by taking advantage of the good features of both systems and without penalizing the performance results obtained with each system.

XSB's shared tables design. The key idea of sharing tables, as proposed by Marques et. al [76] for the XSB system, seems to be a good approach for parallelization of tabled logic programs. As we argued before, the problem with sharing tables is the usurpation algorithm, which seems to be too complex and restrains the potentialities of parallelization. During a period of this thesis we have worked on a different approach for XSB's sharing tables view. The key idea was to evaluate a subgoal call in a thread and let other threads consume answers of that subgoal call. We studied the possibility of using a call graph to keep record of all subgoal calls in evaluation and the threads where they were being

evaluated, and then use a termination algorithm that could allow the completion of dependent subgoal calls, being evaluated in different threads. However, due to lack, of time we were not able to implement this approach.

Concurrent linear tabling. Since the evaluation of programs with a linear tabling engine is less complex than the evaluation with a suspension-based engine, it should be interesting to study how different linear tabled strategies [5, 7, 9], could run concurrently within such a model and take advantage of the different linear tabling optimizations. Also, it should be interesting to compare those results with the results already obtained in this work.

Concurrent negation. A wide range on problems that use tabling require the possibility to manipulate negative subgoals. Extending our implementation with this feature can be one major step forward to make it usable for a large community.

More experimentation. Explore the impact of applying our strategies to more complex problems, seeking real-world experimental results allowing us to improve and consolidate even further the current implementation.

8.3 Final Remark

The research in this thesis involved great motivation, dedication and pertinence. However, there is still too much work that can be done and this thesis is only a small step towards that direction. We end this thesis, by leaving the reader with the answer to the Prolog query *multithreaded_tabling_is(Quality, Reason)* that resumes our view about this topic:

```
?- multithreaded_tabling_is(Quality, Reason).

Quality = powerful      Reason = 'Combines Prolog
                                  with tabling
                                  and concurrency.' ?
;
Quality = elegant       Reason = 'At user level uses
                                  the Prolog language.' ?
;
Quality = complex       Reason = 'At structural level uses
                                  a complex combination of
                                  data structures.' ?
;
Quality = challenging   Reason = 'Many other features can
                                  still be implemented.'
```

References

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

[2] E. Alba, F. Almeida, M. J. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. M. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In *International Euro-Par Conference*, number 2400 in LNCS, pages 927–932. Springer, 2002.

[3] K. Apt and M. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[4] M. Areias. On Applying Linear Tabling to Logic Programs. MSc Thesis, University of Porto, Portugal, September 2010. Available: http://repositorio-aberto.up.pt/handle/10216/74598.

[5] M. Areias and R. Rocha. On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 11(4–5):681–696, 2011.

[6] M. Areias and R. Rocha. An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs. In *International Conference on Parallel and Distributed Systems*, pages 636–643. IEEE Computer Society, 2012.

[7] M. Areias and R. Rocha. On Extending a Linear Tabling Framework to Support Batched Scheduling. In A. Simões, R. Queirós, and D. Cruz, editors, *Proceedings of the Symposium on Languages, Applications and Technologies (SLATE 2012)*, pages 9–24, Braga, Portugal, June 2012.

[8] M. Areias and R. Rocha. Towards Multi-Threaded Local Tabling Using a Common Table Space. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 12(4 & 5):427–443, 2012.

[9] M. Areias and R. Rocha. Batched Evaluation of Linear Tabled Logic Programs. *Journal of Computer Science and Information Systems, Special Issue on Advances in Model Driven Engineering, Languages and Agents*, 10(4):1775–1797, October 2013.

[10] M. Areias and R. Rocha. A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs. In C. Grelck, editor, *7th International Symposium on High-level Parallel Programming and Applications (HLPP 2014)*, pages 259–278, Amsterdam, Netherlands, July 2014.

[11] M. Areias and R. Rocha. A Simple and Efficient Lock-Free Hash Trie Design for Concurrent Tabling. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the 30th International Conference on Logic Programming (ICLP 2014)*, Vienna, Austria, July 2014.

[12] M. Areias and R. Rocha. On Scaling Dynamic Programming Problems with a Multithreaded Tabling System. In A. Jannesari, W. F. Tichy, and F. Wolf, editors, *1st Workshop on Software Engineering for Parallel Systems (SEPS 2014)*, pages 103–114, Portland, Oregon, USA, October 2014.

[13] M. Areias and R. Rocha. On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs. In M. Flatt and Hai-Feng Guo, editors, *International Symposium on Practical Aspects of Declarative Languages*, number 8324 in LNCS, pages 168–183, San Diego, California, USA, January 2014. Springer.

[14] M. Areias and R. Rocha. A lock-free hash trie design for concurrent tabled logic programs. *International Journal of Parallel Programming*, pages 1–21, 2015.

[15] P. Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.

[16] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93. ACM, 1993.

[17] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[18] E. D. Berger. Hoard The Memory Allocator. Available: http://www.hoard.org/.

[19] E. D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, University of Texas, 2002. Available: http://people.cs.umass.edu/ emery/pubs/berger-phd-thesis.pdf.

[20] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[21] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[22] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Usenix Summer 1994 Technical Conference*, pages 87–98. Usenix Association, 1994.

[23] F. Bueno, D. Cabeza, M. Carro, M. V. Hermenegildo, P. López, and G. Puebla. *Ciao Prolog System Manual*. Available: http://clip.dia.fi.upm.es/Software/Ciao.

[24] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[25] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology, 1990.

[26] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331. ACM, 2014.

[27] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[28] P. Chico, M. Carro, M. V. Hermenegildo, C. Silva, and R. Rocha. An Improved Continuation Call-Based Implementation of Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 4902 in LNCS, pages 197–213. Springer, 2008.

[29] K. Clark. Predicate Logic as a Computational Formalism. Research monograph, Imperial College, 1979.

[30] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme–machine en francais. Technical report cri 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1973.

[31] J. Costa, J. Raimundo, and R. Rocha. A Term-Based Global Trie for Tabled Logic Programs. In *International Conference on Logic Programming*, number 5649 in LNCS, pages 205–219. Springer, 2009.

[32] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *ACM Conference on Programming Language Design and Implementation*, pages 117–126. ACM, 1996.

[33] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in LNCS, pages 21–35. Springer, 1998.

[34] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, 2000.

[35] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-Free Reference Counting. In *ACM Symposium on Principles of Distributed Computing*, pages 190–199. ACM, 2001.

[36] A. Dhraief, R. Issaoui, and A. Belghith. Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability. In *International Conference on Advanced Communications and Computation*, pages 143—-148, 2011.

[37] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. *SIGPLAN Not.*, 38:163–174, 2002.

[38] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *The Technical BSD Conference*, 2006.

[39] Jason Evans. JeMalloc. Available: http://www.canonware.com/jemalloc/.

[40] E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.

[41] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *International Symposium on Programming Languages: Implementations, Logics and Programs*, number 982 in LNCS, pages 115–132. Springer, 1995.

[42] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in LNCS, pages 243–258. Springer, 1996.

[43] H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

[44] S. Ghemawat and P. Menage. TCMalloc: Thread-Caching Malloc. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[45] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NB`malloc`: Allocating memory in a lock-free manner. *Algorithmica*, 58(2):304–338, 2010.

[46] W. Gloger. Ptmalloc. Available: http://www.malloc.de/en/.

[47] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[48] Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer, 2001.

[49] Hai-Feng Guo and G. Gupta. Simplifying Dynamic Programming via Mode-directed Tabling. *Software Practice and Experience*, 38(1):75–94, 2008.

[50] Hai-Feng Guo, B. Jayaraman, G. Gupta, and M. Liu. Optimization with Mode-Directed Preferences. In *ACM International Conference on Principles and Practice of Declarative Programming*, pages 242–251. ACM, 2005.

[51] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[52] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314. Springer-Verlag, 2001.

[53] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-free Stack Algorithm. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215. ACM, 2004.

[54] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, Washington, DC, USA, 2003. IEEE Computer Society.

[55] M. Herlihy and N. Shavit. On the nature of progress. In *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin Heidelberg, 2011.

[56] M. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, pages 13–26. ACM, 1987.

[57] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[58] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5), November 1993.

[59] C. A. Herrmann and C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2/3):239–250, 2000.

[60] Intel. Intel VTune Performance Analyzer, 2008. Available: http://software.intel.com/en-us/intel-vtune/.

[61] M. Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

[62] R. Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, The Royal Institute of Technology, 1992.

[63] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison-Wesley Longman, 1998.

[64] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[65] R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2(3/4):227–260, 1971.

[66] V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2002.

[67] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994.

[68] Doug Lea. DLMalloc. Available: http://gee.cs.oswego.edu/dl/html/malloc.html.

[69] Doug Lea. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package, 2005. Available: http://gee.cs.oswego.edu/dl/concurrency-interest/.

[70] Doug Lea. The java concurrency package (JSR-166), 2005. Available: http://gee.cs.oswego.edu/dl/concurrency-interest/.

[71] Justin Levandoski, David Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. International Conference on Data Engineering, April 2013.

[72] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004. Available: http://oprofile.sourceforge.net/doc/index.html.

[73] S. Liang, P.Fodor, H. Wan, and M.Kifer. OpenRuleBench: An Analysis of the Performance of Rule Engines. In *Internacional World Wide Web Conference*, pages 601–610. ACM, 2009.

[74] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[75] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[76] R. Marques and T. Swift. Concurrent and Local Evaluation of Normal Programs. In *International Conference on Logic Programming*, number 5366 in LNCS, pages 206–222. Springer, 2008.

[77] R. Marques, T. Swift, and J. C. Cunha. A Simple and Efficient Implementation of Concurrent Local Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 5937 in LNCS, pages 264–278. Springer, 2010.

[78] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[79] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. A comparison of memory allocators for real-time applications. In *Proceedings of the 4th International*

Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '06, pages 68–76. ACM, 2006.

[80] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Comput. Surv.*, 7(1):5–19, March 1975.

[81] M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82. ACM, 2002.

[82] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[83] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.

[84] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[85] P. Moura. ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates, 2008.

[86] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[87] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[88] P. A. Larson and K. Murali. Memory Allocation for Long-running Server Applications. *SIGPLAN Not.*, 34(3):176–185, 1998.

[89] I. Peláez, F. Almeida, and F. Suárez. DPSKEL: A Skeleton Based Tool for Parallel Dynamic Programming. In *International Conference on Parallel Processing and Applied Mathematics*, number 4967 in LNCS, pages 1104–1113. Springer, 2007.

[90] G. Pemmasani, Hai-Feng Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online Justification for Tabled Logic Programs. In *International Symposium on Functional and Logic Programming*, number 2998 in LNCS, pages 24–38. Springer, 2004.

[91] David Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47:570–575, 1994.

[92] S. Prakash, Y. Lee, and T. Johnson. Non-Blocking Algorithms for Concurrent Data Structures. Technical Report TR91-002, Department of Computer and Information Sciences, University of Florida, 1991.

[93] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2014.

[94] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 151–160. ACM, 2012.

[95] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2005.

[96] C. Pusch. Verification of compiler correctness for the WAM. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics(TPHOL-96)*, number 1125 in LNCS, Turku, Finland, 1996. Springer-Verlag.

[97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, number 1254 in LNCS, pages 143–154. Springer, 1997.

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

[99] 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):31–54, 1999.

[100] P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, number 1265 in LNCS, pages 431–441. Springer, 1997.

[101] H. Rashid, C. Novoa, and A. Qasem. An Evaluation of Parallel Knapsack Algorithms on Multicore Architectures. In *International Conference on Scientific Computing*, pages 230–235. CSREA Press, 2010.

[102] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[103] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[104] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

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

[106] R. Rocha, Nuno A. Fonseca, and V. Santos Costa. On Applying Tabling to Inductive Logic Programming. In *European Conference on Machine Learning*, number 3720 in LNAI, pages 707–714. Springer, 2005.

[107] R. Rocha, C. Silva, and R. Lopes. Implementation of Suspension-Based Tabling in Prolog using External Primitives. In *Local Proceedings of the 13th Portuguese Conference on Artificial Intelligence*, pages 11–22, 2007.

[108] R. Rocha, F. Silva, and V. Santos Costa. Concurrent Table Accesses in Parallel Tabled Logic Programs. In *International Euro-Par Conference*, number 3149 in LNCS, pages 662–670. Springer, 2004.

[109] R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1 & 2):161–205, 2005.

[110] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[111] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM International Conference on the Management of Data*, pages 442–453. ACM, 1994.

[112] K. Sagonas, D. S. Warren, T. Swift, P. Rao, S. Dawson, J. Freire, E. Johnson, B. Cui, M. Kifer, B. Demoen, and L. F. Castro. *XSB Programmers' Manual*. Available: http://xsb.sourceforge.net.

[113] Diptikalyan Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, Department of Computer Science, State University of New York, 2006.

[114] J. Santos. Tabulação com Operadores de Modo em Programas Lógicos. MSc Thesis, University of Porto, Portugal, December 2010. In Portuguese.

[115] J. Santos and R. Rocha. Mode-Directed Tabling and Applications in the YapTab System. In *Symposium on Languages, Applications and Technologies*, pages 25–40, 2012.

[116] J. Santos and R. Rocha. On the Efficient Implementation of Mode-Directed Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 7752 in LNCS, pages 141–156. Springer, 2013.

[117] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*. Available: http://www.dcc.fc.up.pt/∼vsc/Yap.

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

[119] O. Shalev and N. Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *Journal of the ACM*, 53(3):379–405, 2006.

[120] Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer, 2006.

[121] A. Stivala, P. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth. Lock-Free Parallel Dynamic Programming. *Journal of Parallel and Distributed Computing*, 70(8):839–848, 2010.

[122] Hakan Sundell and Philippas Tsigas. NOBLE: Non-blocking Programming Support via Lock-free Shared Abstract Data Types. *SIGARCH Comput. Archit. News*, 36(5):80–87, 2009.

[123] T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite Programs. In *International Logic Programming Symposium*, pages 633–652. The MIT Press, 1994.

[124] T. Swift and D. S. Warren. Tabling with Answer Subsumption: Implementation, Applications and Performance. In *European Conference on Logics in Artificial Intelligence*, number 6341 in LNAI, pages 300–312. Springer, 2010.

[125] T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1 & 2):157–187, 2012.

[126] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *International Conference on Logic Programming*, number 225 in LNCS, pages 84–98. Springer, 1986.

[127] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[128] Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein. *Data Structures Using C*. Prentice Hall, 1990.

[129] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *USENIX Annual Technical Conference*, pages 11–11. USENIX Association, 2011.

[130] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 29(1):320–321, 2001.

[131] Nilsson Ulf and Jan M. *Logic, Programming and Prolog*. John Wiley and Sons, Sweden, 1995.

[132] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222. ACM, 1995.

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

[134] D. H. D. Warren. Implementation of Prolog. In *5th International Conference and Symposium on Logic Programming*, 1988.

[135] J. Wielemaker. Native Preemptive Threads in SWI-Prolog. In *International Conference on Logic Programming*, number 2916 in LNCS, pages 331–345. Springer, 2003.

[136] Jan Wielemaker. *SWI-Prolog Reference Manual*. Available: http://www.swi-prolog.org.

[137] G. Yang and M. Kifer. Flora: Implementing an Efficient Dood System using a Tabling Logic Engine. In *Computational Logic*, number 1861 in LNCS, pages 1078–1093. Springer, 2000.

[138] Neng-Fa Zhou. The Language Features and Architecture of B-Prolog. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):189–218, 2012.

[139] Neng-Fa Zhou and Agostino Dovier. A tabled prolog program for solving sokoban. *Fundam. Inform.*, 124(4):561–575, 2013.

[140] Neng-Fa Zhou and Jonathan Fruhman. *Picat User's Manual*. Available: http://www.picat-lang.org/.

[141] Neng-Fa Zhou and Christian Theil Have. Efficient Tabling of Structured Data with Enhanced Hash-Consing. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 12(4 & 5):547–563, 2012.

[142] Neng-Fa Zhou, Y. Kameya, and T. Sato. Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In *IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 213–218. IEEE Computer Society, 2010.

[143] Neng-Fa Zhou, T. Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.

[144] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*, number 1753 in LNCS, pages 109–123. Springer, 2000.

[145] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Inf. Process. Lett.*, 80(6):271–277, 2001.

[146] Y. Zou, T. W. Finin, and H. Chen. F-OWL: An Inference Engine for Semantic Web. In *International Workshop on Formal Approaches to Agent-Based Systems*, volume 3228 of *LNCS*, pages 238–248. Springer, 2004.