HIGH PERFORMANCE OPTIMIZATIONS IN RUNTIME SPECULATIVE PARALLELIZATION FOR MULTICORE ARCHITECTURES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2013

By
Paraskevas Yiapanis
School of Computer Science

Contents

Word Count: 29934

List of Tables

List of Figures

Abstract

HIGH PERFORMANCE OPTIMIZATIONS IN RUNTIME SPECULATIVE PARALLELIZATION FOR MULTICORE ARCHITECTURES

Paraskevas Yiapanis

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, March 2013

Thread-Level Speculation (TLS) overcomes limitations intrinsic with conservative compile-time auto-parallelizing tools by extracting parallel threads optimistically and only ensuring absence of data dependence violations at runtime.

A significant barrier for adopting TLS (implemented in software) is the overheads associated with maintaining speculative state. Previous TLS limit studies observe that on future multi-core systems it is likely to have more cores idle than those which traditional TLS would be able to harness.

This thesis describes a novel compact version management data structure optimized for space overhead when using a small number of TLS threads. Furthermore, two novel software runtime parallelization systems were developed that utilize this compact data structure. The first one, `MiniTLS`, is optimized for fast recovery in the case of misspeculations by parallelizing the recovery procedure. The second one, `Lector`, is optimized for performance by using lightweight helper threads, along with TLS threads, to establish whether speculation can be withdrawn avoiding that way any speculative overheads.

Facilitated by the novel compact representation, `MiniTLS` reduces the space overhead over state-of-the-art software TLS systems between 96% on 2 threads and 40% on 32 threads.

`MiniTLS` and `Lector` were applied to seven Java benchmarks performing on average 7x and 8.2x faster, respectively, against the sequential versions and on average 1.7x faster than the current state-of-the-art in software TLS for 32 threads.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First of all, I would like to express my sincere gratitude to my principal supervisor Mikel Luján. With his immense knowledge and interesting research ideas he guided me on choosing a very interesting path to follow and to writing this thesis. He has been very patient with me and extremely supportive. Always trying to fit me into his busy schedule, especially when I was appearing out of the blue in his door without appointment.

I would also like to thank my co-supervisor Gavin Brown. Especially the first year of my PhD when my direction was still not clear, both Gavin and Mikel have been very supportive providing me with the right tools in order to pursue research. Without them, this thesis would not have been possible.

I would like to express my very great appreciation to John Keane for introducing me to the world of research and parallel computing, and for encouraging me to pursue a PhD.

During my PhD I was lucky to work with two great guys, Isuru Herath and Demian Rosas. We started the PhD at the same time and we spent endless hours the last 4 years "solving" the world's problems (except from computing problems). If it was not for them some of the world's problems would have been unresolved. They are very good friends as well as research partners. I feel happy the way their career continued but sad we will not be working together (at least in the near future).

Also, I would like to thank the people from the iTLS project (both Manchester and Edinburgh teams) for sharing their knowledge in their area of expertise, especially the Manchester team: Jeremy Singer and Adam Pocock.

I would like to thank Rose Date for being such a good friend. It is amazing how much energy she has and how pleasant and happy personality she always carries. I am sad she is not brewing coffee in the computer science school any more.

This thesis is dedicated to my parents for their unconditional love and moral support all those years. My brother, friends from Cyprus, and friends I met in Manchester

also deserve to be acknowledged for their moral support.

Lastly, I wish to thank Laura. She was the one closest to me during the time of this PhD, doing everything possible to support me and being very understanding, especially during the most stressful periods.

Chapter 1

Introduction

The world of computer architecture has undergone major changes over the last years. Multi-core computers have dominated the market as quad-core availability can be found from personal computers [Inc12] to mobile devices [Sam12]. Although this imposes major challenges in terms of software development, researchers in software that now live the *multi-core revolution* are faced with the chance to exploit intriguing ideas to adapt to those changes. A sequential software application alone will not realize any performance improvements unless changes are introduced to the code, new language constructs are build (or new languages), or novel underlying mechanisms that automatically transform the code into something more meaningful to the current architectures.

This chapter answers the following questions:

Why parallelism is important?

Why computer architectures have gone multi-core?

Can automatic parallelization be a solution to parallel programming?

What solutions this thesis provides?

Certain sections of this chapter include a paragraph in italic type (such as this one). While various concepts become more clear, such paragraphs explain where the discussion is headed.

1.1 There is Something about Parallelism

Even though the last few years terms such as "dual-core" and "quad-core" have become very popular, parallelism *per se* was always important for computer architects.

Evidence of parallelism can be traced back to the early 1960s when computer architects James Thornton and Seymour Cray designed one of the first supercomputer known as *Control Data Corporation (CDC) 6600* [Tho70]. A typical machine of that period would use a single Central Processing Unit (CPU), to operate the entire system. The CDC 6600 designers took a different approach from that. They designed simpler and faster CPUs that could only perform specific tasks instead of driving the entire system. For example, some CPUs would handle arithmetic and logic operation, others would handle memory or Input/Output (I/O) operations. Such a design allowed the machine to run faster not only because of the simpler and faster processors but also because operations from different CPUs could run in parallel with each other. Subsequently Cray Research, a supercomputer manufacturer found by Seymour Cray, popularized a special type of processor architecture known as the *Vector* processor by introducing a series of supercomputers starting with the `CRAY-1` in 1976 [Rus78]. Vector processors contain special instructions that can operate on a set of data elements (as opposed to the traditional instructions that operate on a single data element). For instance, obtaining the sum of two vectors could be accomplished using only one such vector instruction.

Vector instructions were very efficient for vector or array operations but they also had several drawbacks such as significant processor state requirements and complicated instruction sets which resulted in higher costs. In the early 1990s advances in CMOS technology, the common technology used for integrated circuits, allowed more transistors to fit in a single die. Furthermore, the die could be clocked faster as feature size shrank. Rapid changes in CMOS technology allowed higher price-to-performance ratio, displacing slowly the widespread use of vector architectures that mainly continued to use Bipolar technology[1].

Another form of parallelism in processor designs can also be observed since 1985; nearly all of them since then are using *pipelining* to overlap the execution of instructions and improve performance [HP11]. In *pipelining* each instruction is divided into different stages and a set of instructions are in various stages at any given time. This

[1] A processor could be implemented on a single chip in CMOS. The same processor would require multiple chips in old vector machines.

concept allows instructions to be overlapped in a simultaneous fashion forming an example of *Instruction-Level Parallelism (ILP)*. ILP is a fundamental idea used by processors in order to take advantage of available parallelism in a sequential application.

There are a number of hardware techniques that extend the basic concept for pipelining such as dynamic multiple-issue (superscalar) execution, out-of-order execution, register renaming, speculative execution and branch prediction. In *superscalar* execution, the hardware components are replicated allowing multiple instructions to be launched in one clock cycle. *Out-of-order* execution allows instructions to be executed out of order as soon as their operands are available. Possible hazards can be avoided by register renaming. *Speculative* execution overcomes control dependencies by speculating on the outcome of branches and proceeds execution as if the guesses were correct. Underlying mechanisms are necessary in order to support incorrect speculations. *Branch prediction* allows the processor to fetch and execute instructions without waiting for the resolution of a branch condition.

The fact that transistors preserved Moore's trend (known as *Moore's Law* [Moo65]), that is, getting smaller and more numerous, enabled architects to add more memory, deeper pipelines, and promote in general techniques such as the ones explained above. Consequently single-thread performance was continuously improving.

1.2 The Rise of Multi-core Architectures

While more transistors enabled the evolution of more complex processor designs and higher clock rates, another factor was waiting its turn to put an end to this pattern: *power consumption*. The main source of power dissipation in CMOS technology is *dynamic power*. Power is calculated using the following formula [PH08]:

$$P = CV^2 f$$

where P is power dissipation, C is Capacitive Load, V is voltage, and f is the frequency the transistors switch. Frequency is a function of the clock rate and higher frequencies imply more power consumption. One obvious way to minimize power is to lower voltage. However, lowering voltage sufficiently for today's demands appears to cause static power dissipation due to leakage current that flows even when a transistor is off. A way to remedy this problem was to increase cooling or turn off parts of the chip not currently in use. Consequently, due to the high cost of power and cooling

mechanisms, computer designers decided to explore different avenues in microprocessor performance. In order to effectively utilize the increasing number of transistors computer manufacturers were forced to turn to *Chip Multiprocessors (CMPs)*, where multiple *cores*[2] exist on the same chip. With the main reason being the *Power Wall* along with diminishing returns from complex superscalar designs the hunt for parallelism has shifted from instruction level into thread level (Thread-level Parallelism - TLP). Software developers cannot rely any more on the latest processor with higher clock speeds to improve their application performance. The parallelism token has now passed to the developers themselves. Applications must be structured in a specific way in order to take advantage of the multiple cores. Programmers must break down the application into independent parts in order to keep the hardware as busy as possible.

There are different ways parallelism can be expressed. Predominantly there are two models namely *shared-memory* and *message passing*. Message passing may be implemented by combining the power of multiple machines together forming a *cluster* and programming the application in a way so that data are communicated between different machines using messages. Famous libraries for programming that model include the *Message Passing Interface (MPI)* [For93]. Shared memory model usually refers to the multi-core machines where every core is sharing the same address space (apart from any local caches) and shared data must be protected by some *locking* primitive. Notable libraries for shared memory programming include `Java`™ threads and `POSIX` threads.

The rest of the discussion will be focused on shared-memory models. Threads will be treated as the main execution unit. Parallel programs are divided into parts that are in turn mapped to threads in order to execute in parallel on the available cores. The terms 'processor' and 'core' will be treated equally and used interchangeably throughout the discussion.

1.3 Challenges in Parallel Programming

Since the computer industry is turning its attention towards multi-core architectures and clock speeds have reached an upper bound, parallel programming is gaining more momentum as it is the most prevalent way of speeding up an application. However,

[2]Core is used more commonly nowadays to refer to a processor on the chip.

writing parallel applications, also known as *concurrent programming*, is a very challenging task compared to the sequential paradigm [OB96, MG99]. The following sections review some of the major challenges in concurrent programming.

1.3.1 Discovering Available Parallelism

It is not always possible to express an application in parallel and even when it is the speedups are not guaranteed to grow linearly with the number of cores. A well known computer scientist noted that the maximum expected improvement depends on the fraction of the application that can be parallelized. For a program to enjoy linear speedups, the parallel fraction should comprise nearly 100% of the total application. If only 30% of the total application can be parallelized, then only that portion can observe speedups. The remaining 70% will still execute sequentially. This observation is commonly known as *Amdahl's law* [Amd67] and is expressed using the following formula:

$$speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

where P is the fraction of the job that can be executed in parallel and N the number of cores. Assuming that 1 is the time for a single core to complete the job, the parallel part takes P/N and the sequential part takes $(1-P)$. Therefore speedup is obtained as the ratio between the sequential (single core) time and parallel time.

1.3.2 Reasoning about Shared Mutable Data

Even when the parallel fraction is relatively large, the lunch is usually not given for free. In many cases programs rely on processing elements from a data structure such as an *array*, a *list*, or a *tree*. When the program is expressed as a parallel implementation, it is likely that a given data structure can be accessed by multiple threads simultaneously. Thus, the programmer must reason and decide how shared data are going to be accessed in a predictable way to maintain correctness of the results at all times. Usually this is accomplished by the use of *locks*, which are language synchronization primitives used to provide mutual exclusion of a critical section accessed by multiple threads.

1.3.3 Optimizing for Performance

Even when a programmer is able to reason how to write the parallel code to execute correctly, there is still the performance issue. Below are some considerations that need to be taken into account if one wishes to optimize for performance.

Lock Granularity

The *granularity* of which locks are used can play a vital role on that. *Coarse-grained* locking granularity is simpler to use but offers limited concurrency whereas *fine-grained* locking offers better performance but sometimes at the cost of higher complexity [HS08]. Imagine for example a program where multiple threads access a shared list to add or remove elements from it. Under coarse-grain locking a thread might lock the entire list using a global lock (a lock that is responsible to lock the list and also it is shared among all threads) before modifying it. This is an extreme scenario and clearly it removes all concurrency. In fine-grain locking usually more than one lock is associated with different portions of the code (the list in this example) so threads must ensure that the correct locks are acquired and released to avoid *deadlocks*[3] or *livelocks*[4].

Task Granularity and Load Balancing

The minimum execution time of the parallel portion of the algorithm is the length of the longest running parallel task (assuming n tasks running on n processors) [YHM+08]. While this implies formulating tasks as small as possible, they should be broken down to a minimum size that at least amortizes the cost of creating and managing them. When an application is broken down into multiple parts, an efficient scheduling mechanism is required that prevents situations where some cores are busy while others are idle. Such a situation can lead to performance losses due to inefficient utilization of the architecture. In this context, this phenomenon is known as *load imbalance*.

Architectural Considerations

To complicate even further the life of the programmer, cases where response time is critical or cases where all possible performance must be exploited, require the data structures to be designed in a way that takes into account the processor's cache. If a processor operates on a given memory location, it is also likely to access the nearby

[3] In a deadlock, threads are waiting for each other to release a lock so neither can progress.
[4] Livelock is similar to deadlock but threads involved might change in regard to one another.

location(s) as well. This is known as *spatial locality*. In order to take advantage of the spatial locality observation processors operate at larger portions of the cache (a group of neighboring words than just a single word) called *cache lines*. On the one hand spatial locality is desirable since the cache may hold a significant portion of the data accessed by a processor minimizing the accesses to main memory (which are slower than cache accesses). On the other hand placing data that might be accessed by multiple processors/threads nearby increases the possibility of *false sharing*. This occurs when processors access distinct data but still triggering a conflict because the data happens to lie in the same cache line. Furthermore, the size of the data objects accessed by a thread at a regular basis could be designed to reflect the size of the cache lines or local cache in general. This may prevent any performance losses due to *misaligned* accesses. Data objects could be aligned by a technique known as *padding* which adds some meaningless bytes between data objects so that they can fit at a memory offset equal to some multiple of the cache line. Moreover, since cache size changes from architecture to architecture, portability is lost (performance-wise due to optimization mismatch).

1.4 Automatic Parallelization

The challenges explained above emphasize the difficulties in understanding and developing parallel programs, let alone debugging them since their runtime behavior might be non-deterministic.

1.4.1 Manual and Automatic Parallelization

Despite the difficulties, this is the prevalent way to program in order to take advantage of what current architectures have to offer. Apart from having a programmer with parallel programming knowledge to perform the transformation from sequential to parallel manually, there is also the possibility of using a tool that can transform a sequential program into a parallel one automatically. Such tool is referred to as a *parallelizing compiler*. Although the easy way sounds appealing both methods, manual and automatic, have their advantages and disadvantages.

Manual Parallelization is usually superior than using an automatic tool [OB96]. The programmer not only can apply domain knowledge to the problem but also can

benefit from using profiling to guide the optimization. Profiling can help to find places that optimizing is more beneficial, or explain why some optimization might not work as efficient as expected. The obvious downside of manual parallelization is the time and effort required. The programmer may spend hours, days, or even months trying to parallelize the code and get the most out of it [OB96].

Automatic parallelization aims to automate the process of transforming a sequential application into a semantically equivalent parallel one. Automatic parallelization is still difficult but the burden moves from the developer to the parallelizing compiler engineer. One of the problems that arise from using a parallelizing compiler is that the output of the code might be almost unrecognizable from the sequential code and difficult to read [OB96]. That makes it difficult for the programmer to influence compiler decisions. Nevertheless, there has been work where *machine learning* was applied in order to replace traditional target-specific and inflexible heuristics for task scheduling with more sophisticated and adaptive decisions [TWFO09].

1.4.2 Runtime Automatic Parallelization

Traditionally automatic parallelization is performed *offline* (at compile time). This entails some form of data dependence analysis to decide whether the portions wished to be parallelized are independent, in terms of data accesses, from each other. Some notable parallelizing compilers include `Polaris` [BEF+95], `SUIF` [WFW+94], and `Intel® C++` compiler.

Offline automatic parallelization is very effective with counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Although promising and inexpensive, in terms of application performance, offline automatic parallelization is sometimes limited mainly due to insufficient runtime information or the inability of the parallelizing compiler to perform the transformation due to complex inter-procedural relationships. For example, it is difficult to perform static dependence analysis on code that makes extensive use of pointers, which is typical for modern languages such as C++ or Java. More sophisticated memory dependence analysis (such as points-to analysis [NKH04]) can help, but parallelization often fails due to unresolved memory accesses. Also when subscripted subscripts are used to access array elements, the actual memory locations may not be available until runtime. Furthermore, loops with unknown number of iterations make it hard to parallelize since there is no information on how to schedule the loop

(*e.g.* the loop might terminate abruptly due to a runtime condition). For these reasons *Speculative Parallelization* also known as *Thread-Level Speculation (TLS)*, a runtime technique for automatic parallelization has gained attention over the last few years. Speculative parallelization is designed to overcome the offline parallelization shortcomings at the expense of introducing additional runtime overheads. There is more detail regarding those overheads in the next section.

Both manual and automatic parallelization have their advantages and disadvantages as explained earlier. The focus of the discussion will be on automatic runtime parallelization and specifically using speculation.

1.5 Speculative Parallelization

1.5.1 Phases of Speculative Parallelization

There are two main phases in speculative parallelization, as illustrated in Figure 1.1. The first phase takes place offline and the sequential program is optimistically (without knowing if the program will actually be parallelizable) transformed into a parallel one, assuming that the parallel execution will maintain the sequential semantics of the original program; *i.e.* no data dependencies.
Also, some extra code is inserted mainly to maintain meta-data that will facilitate in runtime tests that provide correctness. The second phase takes place at runtime. While the application executes, each parallel running thread collects and maintains information regarding all of its memory accesses. For example, any updates (stores) from a given thread are kept locally to that thread, instead of written immediately back to memory, until proven to be correct. Since those threads have not proven to be successful while still executing, they are called *speculative threads*. During, or at the end of speculative execution, an inspection phase takes place to ensure that there were no violations of the sequential semantics of the application (referred to as *test* in Figure 1.1). If a thread did not conflict with another, then it is safe to propagate its modifications back to memory, an action which is called *commit* in this context. Otherwise the offending thread has to *squash*, that is, discard any temporary (local) modifications and re-execute its code. When threads squash, a procedure initiates to ensure that those threads will undo their modifications properly and leave the memory state as it was before the squash occurred. This procedure is called *rollback*.

The focus of this discussion is on the runtime phase of Speculative Parallelization

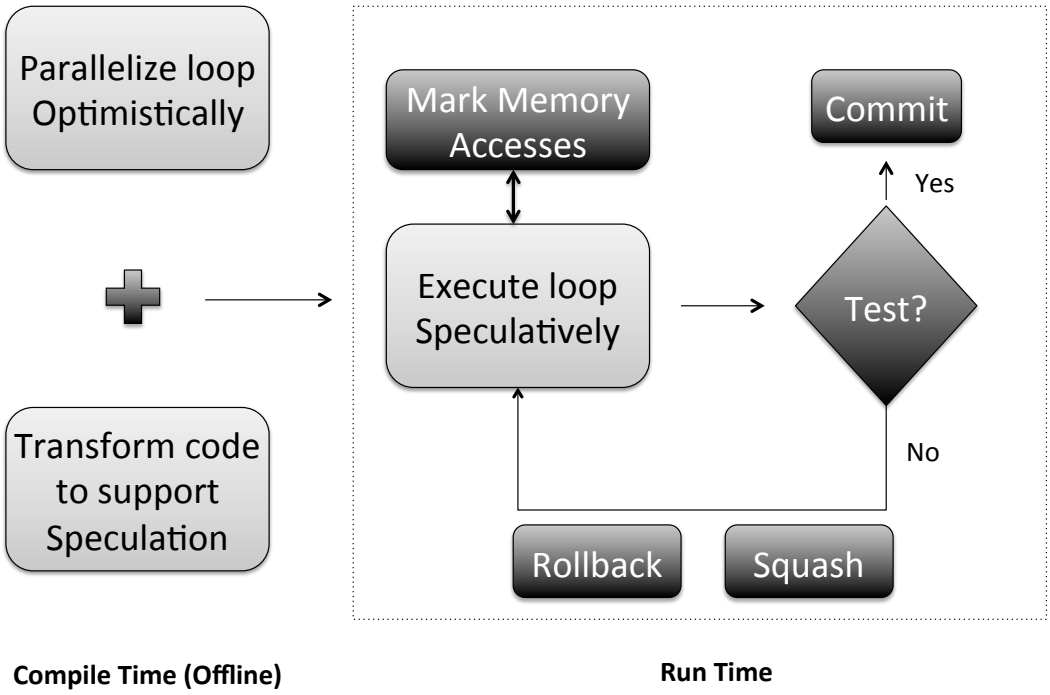

Figure 1.1: Speculative parallelization phases.

and specifically on looking how to reduce the main overheads arising from such a system. The main overheads lie in the phases depicted with darker boxes in Figure 1.1. That is, marking memory accesses, testing for correctness, committing, squashing, and rollbacking.

1.5.2 Performance Potentials of Speculative Parallelization

The performance potentials of speculative parallelization were assessed in a recent study [ISK+10] conducted in collaboration between the University of Edinburgh and the University of Manchester, work in which the author of this thesis was involved. The study evaluated a mixture of different design aspects of speculative parallelization in a simulation environment in order to establish an upper bound on performance. The goal was to offer an architecture-agnostic characterization of the potentials of speculative parallelization. Applications were tested from a variety of application domains (*e.g.*

scientific and business domains) and programming styles (*e.g.* procedural and object-oriented styles). Implementation details regarding this study can be found in Appendix B.

Findings

Figure 1.2 shows results from speculative loop parallelization (*i.e.* loop iterations are mapped into threads and execute speculatively) extracted from the limit study [ISK+10]. The *x* axis shows applications from four important benchmark suits. The *y* axis shows the average improvement percentage of speculative runtime parallelization over sequential runtime. That is, how much an application may improve when speculative parallelization is applied.

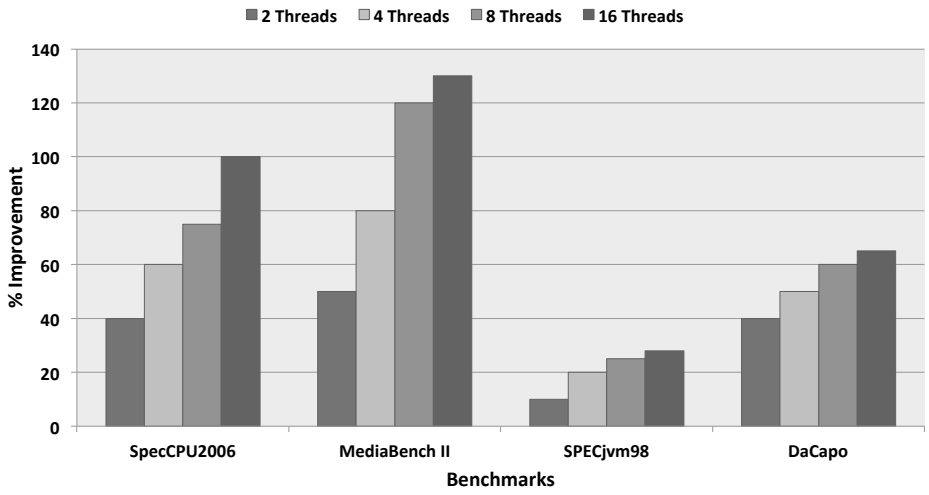

Figure 1.2: Speculative parallelization of various applications from four important benchmark suites. The graph shows the improvement percentage from speculative parallelization over sequential application runtime.

What is notable in the graph of Figure 1.2, is that the improvement does not seem to grow linearly with the number of threads. For instance, `MediaBench II` shows 50% improvement with 2 threads but only 130% improvement with 16 threads. Therefore, instead of showing about 8x more improvement for 8 threads, it is nearly 3x. A similar

pattern is observed with the other benchmarks as well. The key conclusion drawn from these experiments is that *most of the speculative parallelism can be exploited with a small number of threads.*

The topic of discussion is implementing software speculative parallelization systems optimized to reduce the overheads associated with this paradigm for a small number of threads.

1.6 Motivation

Previous work on *Speculative Parallelization (Thread-Level Speculation - TLS)* looked at hardware implementations [HWO98, OHL99, CMT00, MG02, CO03, SCZM05, PO05, JEV07, LPH+09, MLC+09, KRL+10] as well as in software [DYR02, CL05, OMH09, MHHM09, TFNG08, TFG10, RKM+10, KJL+12], just to name a few. However, no widely available architecture or compiler has incorporated Speculative Parallelization. Nonetheless, the published limit studies suggest that it could be profitable to use Speculative Parallelization [PZH+09, ISK+10], although research should focus on optimizing for a small number of threads. Motivated by this potential, experiments were conducted to verify whether this applies in a real machine, besides limit studies. Figure 1.3 shows the speedup results, up to 32 threads, of one of the new software TLS proposed in this thesis (`MiniTLS`; described in Chapter 4) for the Sparse Matrix Multiplication benchmark part of the `SPECjvm2008` suite on a `UltraSPARC T2` machine.

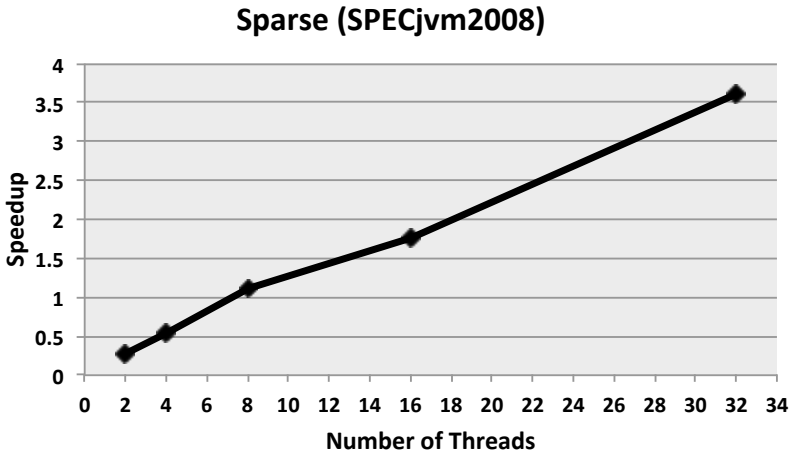

Figure 1.3: Speculative parallelization speedup over sequential for `Sparse` benchmark (`SPECjvm2008`).

Further details of the experiments appear in Chapter 6.

To this end, this thesis proposes two software speculative parallelization systems optimized to reduce the overheads associated with the TLS paradigm for a small number of threads. The next section elaborates further on the contributions of the thesis.

1.7 Contributions

At a high level, there are two main design approaches that have been followed by the software speculative parallelization literature, in terms of how a system maintains its speculative state. These two approaches are known as *lazy* and *eager* version management. Chapter 2 explains both in detail. Two novel speculative parallelization systems are presented, one for each of these design directions. The contributions (Chapters 4, 5, and 6) are the following:

- A compact data structure to represent the dependency tracking for a speculative parallelization system (Chapter 4). The experiments show space overhead reduction of 5x on average compared to state-of-art approaches (Chapter 6).

- `MiniTLS`, a software speculative parallelization system for Java applications, that relies on eager memory data management, is presented (Chapter 4). Speculative threads modify directly data which needs rollback when misspeculation occurs. This eager treatment provides faster execution in the absence of data dependencies. Speculatively parallelized applications are able to run more than 4x faster on average than their sequential versions (Chapter 6).

- `MiniTLS` outperforms state-of-the-art software speculative parallelization systems by being nearly 2x faster and it is the first to parallelize the rollback process under Eager Version Management (Chapter 6).

- A second novel software speculative parallelization system, `Lector`, using lazy version management (Chapter 5). Compared with a state-of-the-art lazy speculative parallelization, `Lector` shows performance improvements on average of 1.7x faster (Chapter 6).

- A novel algorithm for accelerating TLS systems applicable to any type of implementation (Chapter 5). The results show that applying this technique, improves speedup on average 1.7x for 2 threads increasing close to 8.2x speedups with 32 threads (Chapter 6).

1.8 Publications

The material of this thesis from Chapters 4, 5, and 6 appears in the following Journal publication:

- OPTIMIZING SOFTWARE RUNTIME SYSTEMS FOR SPECULATIVE PARALLELIZATION. **Paraskevas Yiapanis**, Demian Rosas-Ham, Gavin Brown, Mikel Luján. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 9 (4), 39, January 2013.

Other Related Publications

- ARCHITECTURAL SUPPORT FOR EXPLOITING FINE GRAIN PARALLELISM. Demian Rosas-Ham, Isuru Herath, **Paraskevas Yiapanis**, Mikel Luján, Ian Watson. In *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications*, June 2012.

- TOWARD A MORE ACCURATE UNDERSTANDING OF THE LIMITS OF THE TLS EXECUTION PARADIGM. Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, **Paraskevas Yiapanis**, Adam Pocock, Gavin Brown, Mikel Luján, Ian Watson, Marcelo Cintra. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, December 2010.

- ONLINE NONSTATIONARY BOOSTING. Adam Pocock, **Paraskevas Yiapanis**, Jeremy Singer, Mikel Luján and and Gavin Brown. In *Proceedings of the International Workshop on Multiple Classifier Systems (MCS)*, LNCS 5997, pp 205-214, 2010.

- STATIC JAVA PROGRAM FEATURES FOR INTELLIGENT SQUASH PREDICTION. Jeremy Singer, **Paraskevas Yiapanis**, Adam Pocock, Mikel Luján, Gavin Brown, Nikolas Ioannou, Marcelo Cintra. In *Proceedings of the 4th Workshop on Statistical and Machine Learning Approaches to Architecture and Compilation (SMART)*, January 2010.

- MINING STATIC FEATURES FOR SQUASH PREDICTION IN THREAD-LEVEL SPECULATION. **Paraskevas Yiapanis**, Jeremy Singer, Adam Pocock, Mikel Luján, Gavin Brown. *In the 5th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES)*, July 2009.

- FUNDAMENTAL NANO-PATTERNS TO CHARACTERIZE AND CLASSIFY JAVA METHODS. Jeremy Singer, Gavin Brown, Mikel Luján, Adam Pocock, **Paraskevas Yiapanis**. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA)*, March 2009.

1.9 Thesis Structure

The rest of the thesis is organized as follows: **Chapter 2** provides an introduction to runtime and speculative parallelization. The main design options when implementing such systems in software are also discussed. **Chapter 3** elaborates, through related work, on advanced techniques for runtime parallelization with focus on speculative parallelization. **Chapter 4** describes the first system, `MiniTLS`, as well as the novel parallel rollback algorithm. It is illustrated how advantage is taken of the information provided by speculative parallelization limit studies to optimize `MiniTLS`. Oancea *et al*. [OMH09] introduced a top performing software speculative parallelization work using eager memory management. Their contribution was how to eliminate any associated synchronization when accessing the data structures holding the dependency information. Oancea *et al*. [OMH09] traded-off increasing the size of the eager memory management data structures to remove synchronization and optimize performance. Hereafter, it is referred to the speculative parallelization software developed by Oancea *et al*. [OMH09] as `SpLIP` (`SpLIP` is described in detail in Appendix A). In the experiments the performance delivered by `MiniTLS` vs. `SpLIP` is compared directly (the experiments appear in Chapter 6). **Chapter 5** describes the second system, `Lector` as well as the novel idea of using inspector threads in the role of helper threads. `Lector` is compared against `TL2` [DSS06] in the performance results. **Chapter 6** reports the speedup results as well as speculative operations and memory overhead comparisons using seven Java benchmarks, with three belonging to `SPECjvm2008`. Note that two of the seven benchmarks do include data-dependencies. Finally, **Chapter 7** summarizes the thesis and presents potential future directions. The thesis includes also two appendices. **Appendix A** provides implementation details of the two baseline systems used in the experiments presented in Chapter 6: `SpLIP` [OMH09] and `TL2` [DSS06]. **Appendix B** provides implementation details for the limit study on speculative parallelization mentioned in Chapter 1 (current chapter) from Ioannou *et al.* [ISK+10] conducted in collaboration between the University of Edinburgh and the University of Manchester.

Table 1.1 shows concisely the structure of this thesis.

CHAPTER 1	Introduction, Motivation, Contributions, Publications
CHAPTER 2	Fundamentals of Runtime Parallelization
CHAPTER 3	Advances in Runtime Parallelization (including related work)
CHAPTER 4	`MiniTLS`: In-place Speculative Parallelization with Parallel Rollback
CHAPTER 5	`Lector`: Reducing Speculative Overhead via Inspector Threads
CHAPTER 6	Experimental Results from `MiniTLS` and `Lector`
CHAPTER 7	Conclusions, Future Work
APPENDIX A	Baseline Implementation Details (`SpLIP` [OMH09] and `TL2` [DSS06])
APPENDIX B	TLS Limit Study [ISK+10] Implementation Details

Table 1.1: Thesis structure.

Chapter 2

Fundamentals of Runtime Parallelization

2.1 Introduction

In recent years multicore chips became the standard configuration in commercial computing. In order to harness the power they have to offer, applications need to be structured in such a way that will yield efficient utilization of the available resources. Parallel programming accomplishes that by dividing the computation across the available processors (or threads), yet this process involves experienced software engineers in this type of programming model. A promising idea is to have the compiler automatically restructure a sequential program into a parallel version. Sometimes this is not possible due to insufficient information during offline compilation. A solution is to proceed with parallel execution speculatively until sufficient information is collected, providing mechanisms to maintain sequential program correctness. Such a solution is known under the names of *Speculative Parallelization*, *Thread-Level Speculation*, or *TLS* in short. All three terms will be used interchangeably throughout the rest of the chapter as well as the rest of the thesis.

This chapter explains speculative parallelization and its mechanics as well as the major design requirements for such an execution model. However, before examining speculative parallelization, the chapter begins with an overview of an earlier model for runtime parallelization. This model is identified as *Inspector/Executor*. A brief discussion to areas of similar interest as TLS, such as *Transactional Memory* and *Database Systems Transactions*, is also provided.

2.2 Inspector/Executor: An Overview

2.2.1 Inspector/Executor

Early work on Runtime Parallelization involved a technique known in the literature as *Inspector/Executor* [ZY87, SMC89, SM91, SMC91, CTY94, RP94b]. As the name implies this method involves the generation of two versions of the loop to be parallelized during compilation. The first version, called *Inspector*, would simply execute a stripped version of the original loop that contains only accesses to shared mutable data, investigating whether the loop contains any data dependencies that prevent it from being parallelized. Figure 2.1a shows a candidate loop for parallelization. The assumption is that the memory accesses from variables x and y to locations of array A cannot be verified until runtime. Furthermore, the code surrounding the memory accesses to A (represented with dots in Figure 2.1a) does not perform any updates to locations shared across different iterations (it could be just because of the nature of the code or after some compiler transformation). The *Inspector* is not required to replicate all of the computation done by the original program (only memory accesses), and thus could be executed quicker. Figure 2.1b shows the stripped version of the code from Figure 2.1a that is required by the inspector. The stripped version of the loop will be replicated across multiple inspector threads and each thread will inspect a different portion of the *iteration space* (the set of loop iterations L, in this example $0 < L < 100$ and i is used to traverse the iteration space, where $0 < i < L$).

2.2.2 Inspector/Executor for Fully Parallel Loop Identification

One of the most notable *Inspector/Executor* models in the runtime parallelization literature was proposed by Rauchwerger and Padua [RP94b]. Their approach used the inspector to simply detect whether or not a loop is fully parallel. Such loops, are known as *DOALL* loops. In [RP94b] each inspector thread would be allocated its own portion (chunk of iterations) of the stripped version of the original loop (from j to P in Figure 2.1b, where from j to P is a chunk of iterations allocated to a specific thread). Each thread will also make use of some thread-local auxiliary data reflecting each memory location of A that can be accessed during parallelization (see Figure 2.1c). In this simplified version, each inspector thread will mark any loads or stores performed during the execution of its portion of iterations. A location in the thread-local auxiliary arrays $Load[]$ or $Store[]$ is marked to indicate the corresponding action by a particular thread.

At the end of the inspection phase, all inspector threads will check each others findings to ensure that the same memory location was not accessed by different threads in a way that violates the sequential semantics of the loop. For instance, if two threads perform a store at the same memory location but in the wrong order the resulting value would be unpredictable. The only valid way multiple threads can access the same memory location is if they just read from that location. Since reading from a memory location does not affect its value then threads are safe to read in any order they wish from there. The second version, called *Executor*, will execute the loop in parallel across multiple threads, given that the inspector version indicated so. Otherwise the loop will be executed sequentially.

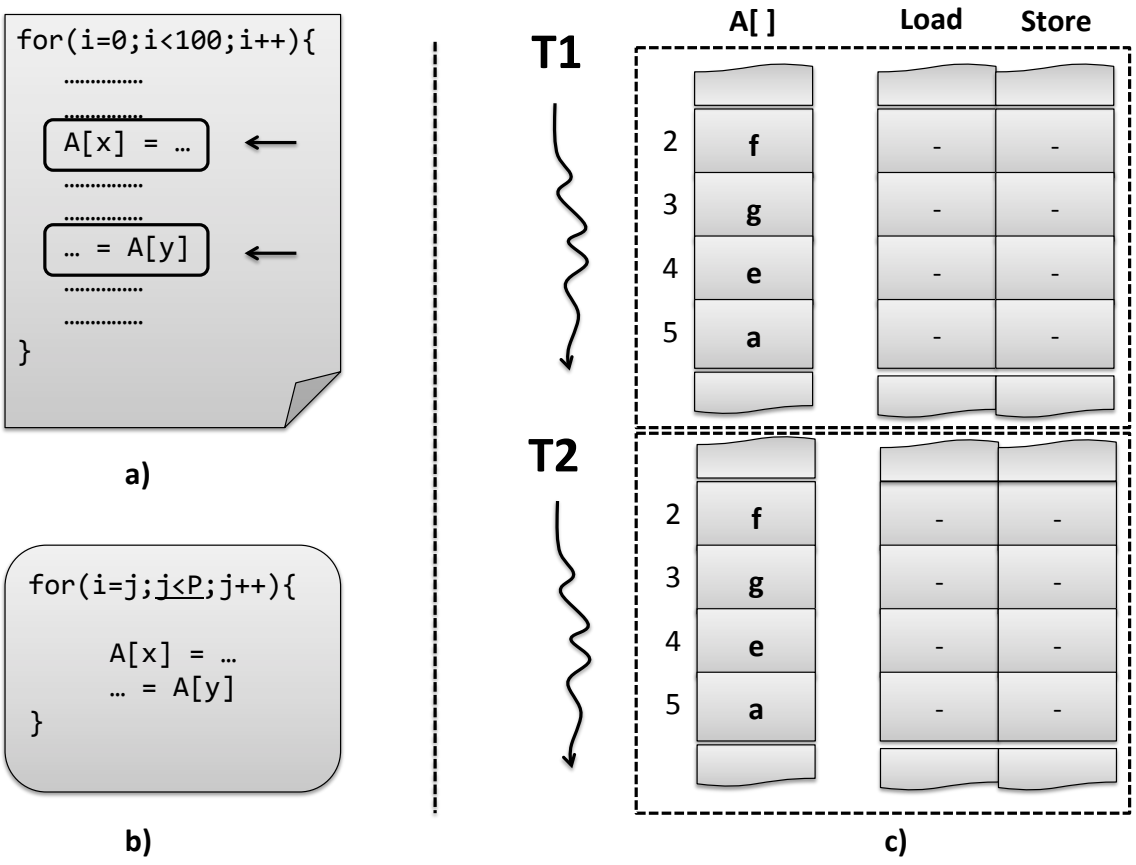

Figure 2.1: a) A loop to be parallelized. b) Stripped-down version of the loop to be executed by the inspector threads. c) Auxiliary data to facilitate inspection.

2.2.3 Earlier Work on Inspector/Executor

The focus of earlier attempts of the *Inspector/Executor*, such as the one proposed by Zhu and Yew [ZY87], was mainly on partially parallel loops and they were using the inspector phase to order the iterations in groups that contained parallel iterations. Within those groups, iterations could execute in parallel between them, but the groups themselves have to be separated using synchronization. Their scheme was divided in multiple stages with each stage including both an inspector and an executor. Every stage would gather information (inspection phase) of which iterations are allowed to execute in parallel without any conflicts and record this information into an auxiliary data structure. Then, the executor would execute those iterations in parallel with the aid of the auxiliary data structure. The next stage would do the same and the process would continue in a repetitive fashion until all iterations of the loop finish. Later work, such as Saltz *et al*. [SMC91], provide more optimized versions of the *Inspector/Executor* technique by statically partitioning the iterations of the loop among processors and then reordering the iterations within each partition at runtime.

2.2.4 Weakness

Clearly there are certain conditions that have to be met in order for this technique to work efficiently. The main drawback of this technique is that an adequate stripped version of the original loop is not always possible to be extracted. If the memory accesses to be analyzed during inspection constitute the majority of the loop's computation, the *Inspector* eventually executes nearly the same amount of computation as the *Executor* version [Rau98]. This was the main reason that led research towards speculative runtime parallelization.

2.3 Speculative Parallelization

2.3.1 Brief Description

Assume, for the sake of the argument, that one wishes to parallelize the loop shown in Figure 2.2a. Similarly, assume that the array indexes cannot be resolved until runtime. That is, there is no feasible means of performing any static analysis (manual or automatic) to prove correct parallel executions by eliminating the possibility of data dependencies across threads. This can be the case for example, where i or j are the

result of accessing an indirection array. Therefore, standard parallelizing compilers must conservatively produce sequential code for the loop in order to guarantee correct execution. Consider now the instance of sequential execution shown in Figure 2.2b. Clearly the values populated for the array indexes did not yield any data dependencies amongst them, and thus, the compiler could have generated a parallel code such as the one in 2.2c and allow the application to run in parallel.

Speculative Parallelization (Thread-level Speculation - TLS), circumvents this conservatism by executing the threads (which are formed by loop iterations in this case) in parallel assuming that the run-time values of i and j will not trigger any cross-thread conflicts. For instance, in this case TLS would execute the loop iterations in parallel, while at the same time underlying mechanisms would monitor every speculative access to ensure that the parallel execution will produce the same results as if the program was executed sequentially. In addition, any memory updates are stored locally to the thread rather than written-back to main memory. Figure 2.3a shows the case where all speculative threads executed successfully and thus are allowed to *retire* or *commit* by propagating the buffered updates back to main memory. Sometimes we have the case of a memory dependency like the one shown in Figure 2.3b. In this case, a thread (or iteration) has loaded a value that was not produced by the correct store. This action causes a *Read-After-Write (RAW)* data dependence violation. As a result, the offending threads need to *squash* by initiating the *rollback* procedure (in this case discard any buffered updates) and re-execute in the correct order (see Figure 2.3c).

2.3.2 Design Specification

Implementing the underlying mechanisms that will guarantee correct execution in TLS requires certain design decisions. From the brief description given above in Section 2.3.1 the main requirements for supporting speculative parallelization can be categorized as follows (also identified by Cintra and Llanos [CL03, CL05]):

Metadata management. A way to know which memory locations are accessed and by which threads. This will facilitate to identify whether or not any threads have accessed memory locations in a way that is not desirable.

Version management. A way to manage speculative data. When threads execute speculatively, different versions of the data are produced. A mechanism is required to manage temporary (speculative) data and maintain consistency among operations.

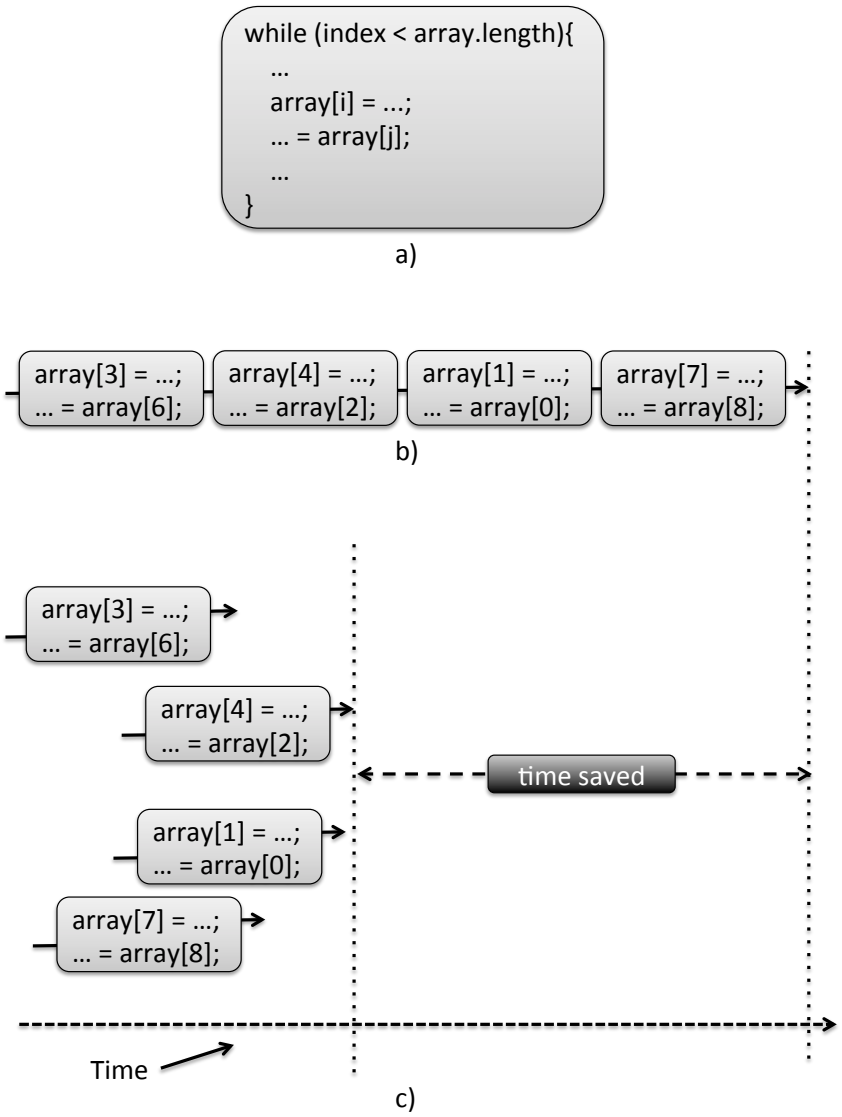

Figure 2.2: a) Code fragment of loop to be parallelized. b) Sequential execution. c) Sample parallel execution.

Detect data dependence violations. A way to identify potential data dependence violations.

Commit and rollback operations. A way to maintain main memory at a correct state. That is, to be able to commit the correct values and rollback execution to a correct state when necessary.

Scheduling speculative threads. An efficient way to schedule speculative work and threads.

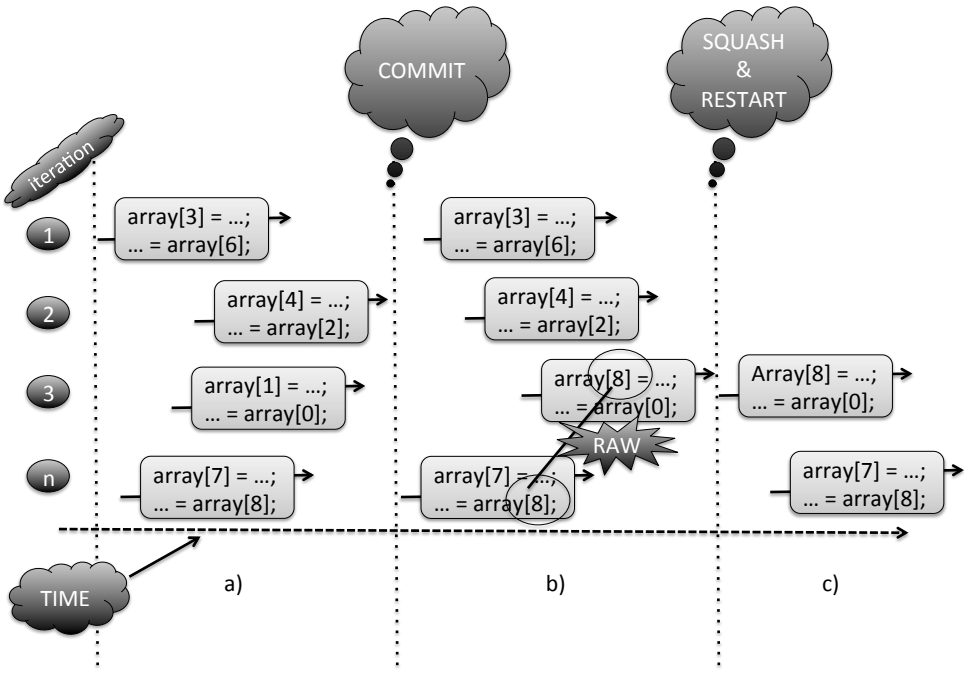

Figure 2.3: a) Speculative loop execution without dependencies. b) Speculative loop execution with dependency. c) Re-execution of offending threads.

The following sections provide an overview of each requirement and discuss the main implementation options available.

Metadata

Section 2.3.1 quoted that while speculation takes place, underlying mechanisms monitor speculative accesses. This action is what was also addressed as "marking memory accesses" in Figure 1.1 from Chapter 1. To enable a TLS system to monitor memory accesses, some auxiliary data (referred to in this work as *metadata*) are required to be associated with the user data structures as well as the speculative threads. Figure 2.4 shows one way in which speculative memory access information can be kept in the system. The various ways metadata can be exploited are explored later under the "Version Management" sub-section.

Metadata to Reflect User Data Structures. Figure 2.4a illustrates how metadata are arranged to reflect the user data that can potentially be accessed in a speculative way. Every user datum that can be potentially accessed speculatively (*i.e.* a datum that

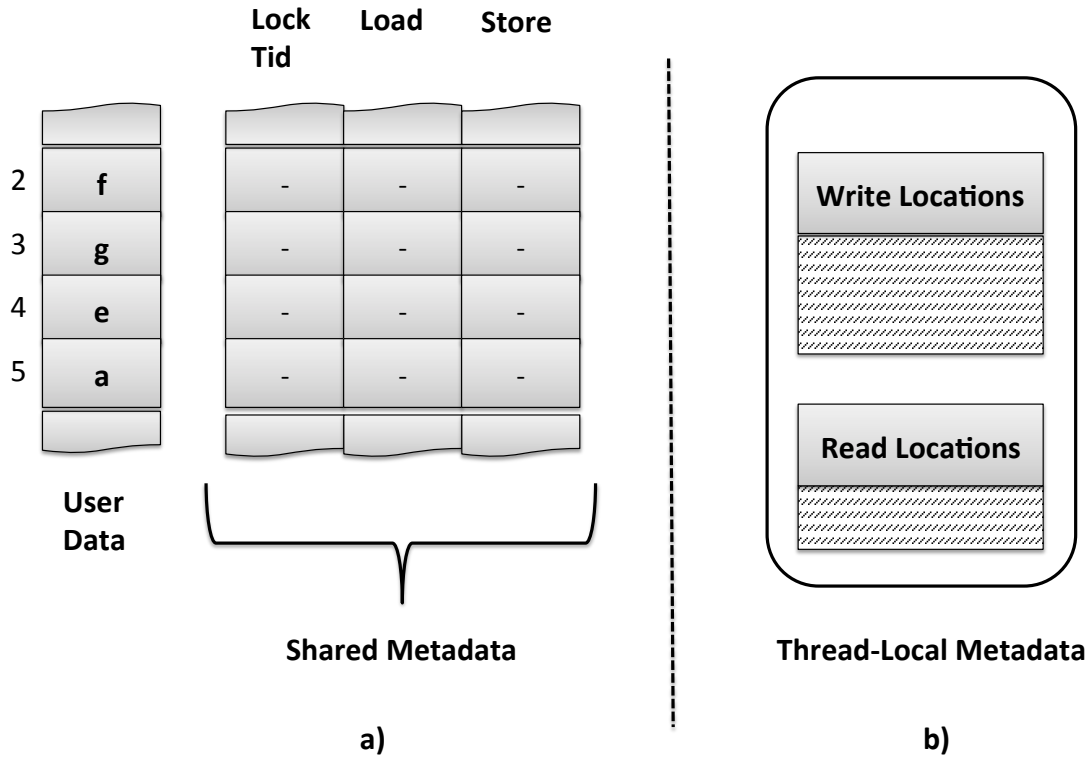

Figure 2.4: a) Metadata or shadow data associated with user data structure. b) Metadata associated with every speculative thread.

is shared among speculative threads) is associated with a sequence of memory locations. These locations represent actions on user data by a given speculative thread. For instance, a general TLS system could maintain information about which particular thread is operating, reading, or writing at a given moment in time on a given user memory location. The thread actions are denoted with the labels "Lock Tid", "Load", or "Store" in Figure 2.4a. "Lock Tid" is used by a thread in order to acquire *exclusive ownership* of a particular memory location. That is, by using a locking primitive a thread stores a unique thread identifier (Tid) in the "Lock Tid" record and proclaims ownership of the location associated with that record. This action prevents another speculative thread from performing an operation at the same memory location simultaneously with the currently owner thread allowing predictable results. The owner thread is allowed to perform a speculative operation, either a load or a store by setting the corresponding record with its unique thread identifier. The lock is released immediately after the speculative operation takes place in order to allow other threads to examine

the information on that location.

Metadata for Speculative Threads. Apart from the metadata required to reflect any user memory location, speculative threads also require some extra information regarding the locations they access during their lifetime. For example, Figure 2.4b implies that information is kept regarding which memory locations have been accessed for reading or writing. When a speculative thread performs a read operation on a memory location, the address of that location is saved in a thread-local set that contains all locations read by that thread while executing a particular speculative region. This set is known as the *read-set*. Similarly, when a thread performs a speculative write on a memory location, that location is recorded on the thread-local *write-set*. The write-set usually contains also the speculative value produced by the thread for the address accessed for writing. The reason for that is discussed later in the next sub-section ("Version Management").

Version Management

While threads execute speculative code, different versions of data are being produced. *Version Management* refers to the way those different versions are maintained by the TLS system. Typically, there are two major approaches for that: *Lazy Version Management (LVM)*, also known as *deferred updates* and *Eager Version Management (EVM)*, also known as *in-place updates*.

When LVM is used as a choice, speculative threads require at least a write buffer per thread in order to keep any tentative stores. Speculative loads search first the thread's local buffer in case they find an associated value for that location there. If not, the value needs to be loaded from memory. Speculative stores just need to add or update the corresponding value in the local buffer. At the end of a thread's speculative execution (and provided that there was no conflict for this thread) the results from the local buffer are propagated to main memory to make visible the updates to other threads. In the case of a conflict, the speculative thread only needs to discard its local write buffer since there was no modification of the actual data in memory.

EVM systems, on the other hand, update memory locations directly when the speculative store occurs rather than delaying the action. This involves having a buffer that preserves the original value of the memory location just before the update. This buffer is known in the literature as the *undo log* since in the case of a conflict the log is used to restore the memory back to a correct state. Speculative loads can use the values form

memory, since the new values are already there. Upon successful commit, the thread simply discards the undo log without requiring any value propagation as in LVM.

Conflict Detection

A conflict can occur when two or more speculative threads access the same memory location in a way that cause a data dependency violation. Depending on the version management system used, different actions may or may not cause violations. There are three type of data dependencies: *flow*, *anti*, and *output dependencies* which give rise to Read-After-Write (RAW), *Write-After-Read (WAR)*, and *Write-After-Write (WAW)* hazards respectively. A RAW violation is caused when a thread loads a value that was not produced by itself. WAR and WAW violations arise due to reuse of memory locations. A system that uses LVM does not need to worry about WAR and WAW dependence violations since the updates are buffered and speculative loads use those instead. This is somewhat similar to the *Register Renaming* action taken at the hardware level to prevent those kind of hazards. In contrast, EVM has to take precautions for WAR and WAW dependence violations since the values in memory are always up-to-date. Nevertheless, both EVM as well as LVM systems need to be observant for RAW violations.

There are two types of conflict detection: *Lazy Conflict Detection (LCD)* and *Eager Conflict Detection (ECD)*. LCD implies that threads may be allowed to run through their respective speculative code without checking for conflicts on every access. Conflict detection can occur at a later stage as long as that happens before thread commit. In this way eager checks during execution are eliminated. ECD checks for conflicts usually on every speculative access in order to catch any violations as soon as they arise. The idea here is to prevent any wasted work after a conflict has happened.

Commit and Rollback

If the TLS system confirms that a group of speculative threads had correct execution the commit phase allows those threads to provide memory with the correct values. A system using lazy version management (LVM) typically commits those values sequentially. That is, the threads propagate their speculative (currently buffered) values to

main memory one by one in order[1] of speculation (*i.e.* starting from the least and moving towards the most speculative thread). The reason is that, during speculation some threads might have triggered a conflict due to a RAW dependence violation and thus had to wait for the correct value to be produced. A less speculative thread is always more correct than a more speculative thread. If the threads are allowed to commit out of order then this assumption is lost and the wrong values may be populated. On the other hand, since on an eager management system updates are performed in-place, at the end of speculative execution the committed values are already in the correct place in memory. Thus, commit across multiple speculative threads occurs in parallel.

When a conflict arises, main memory must be restored to the last known correct state. This involves squashing the offending threads and rollbacking speculative state. With LVM this procedure is very simple, effective, and threads can operate in parallel. Since LVM systems buffer speculative updates, each thread is allowed to proceed in parallel with each other and discard their speculative values. Unfortunately for an EVM system the process is a bit more complicated since main memory already contains the "polluting" values. Speculative thread buffers contain the correct values and are iterated sequentially to restore main memory. The reason they are iterated sequentially is because not all the values are eligible for rollback. In the case that multiple threads have updated a memory location, only one thread -the one that performed the first update- has the correct value and should restore the memory location. As a result every location examined in a thread's buffer requires to check other threads' buffers to identify such a scenario.

Scheduling

Since the loop is parallelized automatically from sequential code, the execution behavior is rather unpredictable. The way iterations are scheduled to run across the available threads can have significant impact in the final performance. Traditional scheduling possibilities include *static* and *dynamic scheduling*. Static scheduling partitions the loop into equal chunks of iterations based on the number of available threads. The thread that will execute a particular chunk is decided statically. In contrast, dynamic scheduling allows those chunks to be assigned to threads at runtime. Both of these are

[1] Assume n iterations are mapped on n threads in order (thread 0 has iteration 0, thread1 has iteration 1, and so on). This specifies a speculation order in which thread k is less speculative than thread $k+1$, where $0 < k < n$.

not very well suited for TLS since they can cause load imbalance, increase the probability of dependence violation, and increase memory overhead. A different scheduling technique known as *Sliding Window* (originally introduced by [DYR02]) was evaluated amongst different scheduling techniques by [CL03, CL05] and found to be a good alternative for TLS. Under sliding window (see Figure 2.5), chunks of iterations are assigned into windows of size W. The window moves forward (slides) when all iterations in the window have finished. This allows better load balancing, decrease in likelihood of dependence violation and better decoupling of memory overhead [CL03, CL05].

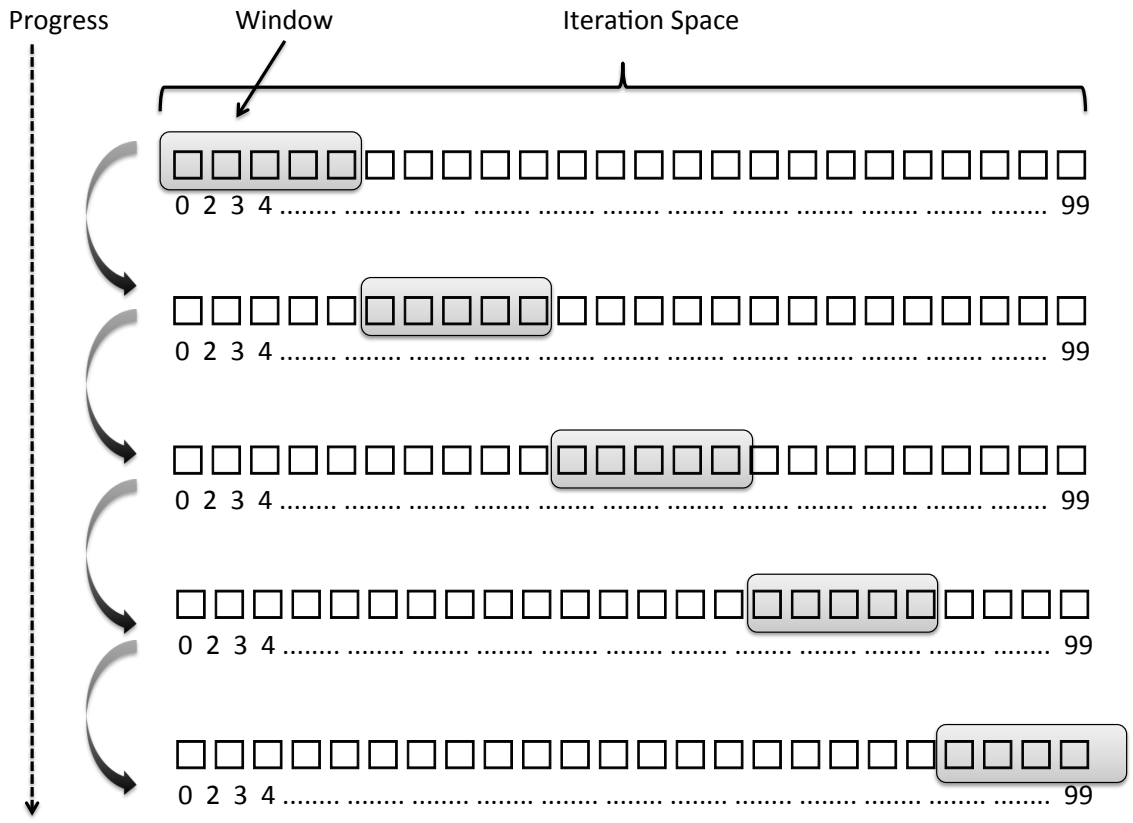

Figure 2.5: Sliding window scheduling.

2.3.3 Other Implementation Details

The main overhead in software TLS arises from maintaining the speculative state. Certain hardware solutions that apply lazy version management (LVM) can overcome this problem by taking advantage of the cache coherence protocol [GVSS98]. For example, cache lines are modified with additional information, an extra bit and a pointer, to

facilitate marking of speculative data. The bit, known as the "load bit", is set when a processor loads a value not produced by itself (*i.e.* a potential RAW violation). The extra pointer indicates the next most speculative cache line that has a version of that datum in case a value needs to be forwarded to a remote cache. Furthermore, L1 cache is normally private to the processor and thus buffering occurs at the L1 cache lines by default. Not only buffering is cheap but also marking simply needs to write a bit and change a pointer. Software approaches, on the other hand, rely on additional data structures to maintain marking information. Thus each potential speculative load or store will produce at least an extra load and store from the hardware point of view (to read and insert that item to the data structure). These data structures are normally accessed via locks or *Compare-And-Swap (CAS)* operations to avoid data races between different threads.

2.3.4 General Considerations

Generally speaking, a system that utilizes EVM might be more complicated to implement than one using LVM. Since stores are written in-place (eagerly), the designer has to also consider *Write-After-Read (WAR)* and *Write-After-Write (WAW)* dependencies, apart from *Read-After-Write (RAW)* dependencies. Nevertheless, both design choices (LVM and EVM) are advantageous in different cases. Recall from Section 2.3.2 that EVM has a cheap commit phase (updates are in-place) but an expensive rollback. To the contrary, an LVM system has a cheap rollback (just discard local buffers) but a more expensive commit. Thus, a rule that may be wise to follow is that EVM is probably more appropriate for applications with minimal runtime conflicts (since commit is cheap) and LVM more appropriate for applications with high number of conflicts (since rollback is cheaper). This pattern was also observed by Garzarán *et al*. [GPL+05] in a study where separation of task state (version management) and merging of task state (commit) were analyzed. Consider, for instance, the following scenarios:

Best Case Scenarios

Ideally, the best case scenario would be an application with very few speculative accesses and no conflict at runtime. Every time a load or store occurs to a shared mutable data item, an instrumentation call needs to be inserted to the TLS system. This call in turn will cause the system to register the action and validate whether the execution is

still correct. Clearly less instrumentation means less time spend on speculative operations. Furthermore, an application that does no trigger any conflicts will not suffer from re-executing any code path. In such scenarios eager version management will be more ideal since commits do not cause extra overhead.

Worst Case Scenarios

An application that consists of mainly accesses to shared mutable data as well as causes a large number of conflicts would be a worst case scenario. In this case it may be more beneficial to fall back to sequential execution. Maintaining speculative state is expensive as well as frequent rollbacks as Chapter 6 (experimental results) shows later thus ideally these should be minimal. Previous work [OMH09] has found beneficial the fact of reverting to sequential execution if the number of rollback exceeds 1% of the number of executed iterations.

2.4 Transactional Execution

Speculative execution closely resembles the techniques used in *Transactional Memory*, another form of optimistic execution. As a matter of fact, both speculative execution and transactional memory inherited most of their semantic details from *Database Transactions*. The following sections provide the connection between those areas.

2.4.1 What is a Transaction?

A *transaction* is a sequence of actions that appear to occur indivisibly and instantaneously to an outside observer. In the context of speculative parallelization, a transaction is the region that executes speculatively. Assuming work chunks are assigned one per speculative thread, then a speculative thread is equivalent to a transaction. A transaction serves as a nice abstraction so the user can specify the parts they wish to execute atomically and the underlying system takes care of the rest so everything works correctly. The definition of correctness may vary depending on the executing environment, as the following sections explore.

2.4.2 Database Transactions

Transactional processing has been successfully used in database systems for decades as a means of exploiting parallel architectures. Multiple queries are allowed to execute in a simultaneous fashion without the programmer having to worry about the parallelism. Database transactions must conform to the *ACID* [HR83] properties which are defined as follows:

1. **Atomicity**. The actions specified within a transaction must either all complete successfully or none of them to appear executing alone.

2. **Consistency**. Depending on the specification of consistency for a given system, a transaction must initiate from one consistent state after termination leave the database to another consistent state.

3. **Isolation**. Transactions must not interfere with each other during execution.

4. **Durability**. Any modifications done by a transaction must be stored permanently (*e.g.* on a disk) on commit and be available to other subsequent transactions.

2.4.3 Transactional Memory

In *Transactional Memory (TM)* [HM93], parallel portions of applications are executed concurrently as transactions, and access shared data simultaneously. In the TLS paradigm, a sequential program is first transformed to parallel and then execute speculatively. In TM the input program is already parallel but instead of using locks to protect critical sections, synchronization is achieved by transactions. *Transactional* threads in TM have the same guarantees as in TLS apart from the requirement to commit in a predefined total order (the sequential order). As such, transactional memory can be seen as a general form of TLS.

The main difference between transactions in databases and transactions in TLS and TM is in terms of the last *ACID* property. Traditional database operate on disks whereas TLS and TM operate only with main memory. Furthermore, work done by transactions in TM and TLS normally cease to exist after program completion unlike databases where work must be committed back to disk.

2.5 Summary

The topic of this chapter was *runtime parallelization* with focus on *speculative parallelization*.

Static approaches to automatic parallelization are generally very successful but fail to parallelize code where sufficient information is not known until runtime. Initial attempts to runtime parallelization applied the Inspector/Executor model where an inspector phase determines at runtime the applicability of a loop to run in parallel, an action accomplished by the executor phase. Overheads associated with the inspection phase of that model directed research in runime parallelization towards a more promising solution: *speculative parallelization*.

Speculative parallelization executes a loop in parallel optimistically (without knowledge whether the loop can be parallelized) and provides mechanisms to maintain correctness during runtime. This entails ways to schedule speculative threads, ways to monitor accesses by speculative threads, ways to detect undesirable actions, and prevent wrong results to be maintained in main memory. This chapter elaborated on those mechanisms and explained different ways they can be implemented.

The next chapter moves a step forward to discuss the main speculative parallelization systems in the literature and how each of those systems proposes a solution for the above design choices.

Chapter 3

Advanced Topics in Runtime Parallelization

3.1 Introduction

Chapter 2 introduced the area of runtime parallelization providing an overview of the early ideas using the *Inspector/Executor* model and the rise of *Speculation* as a way to parallelize an application at runtime. There has been numerous work in speculative parallelization over the last two decades. This chapter identifies and presents the most important ones while discussing various advances in the area of speculation. The contributions of this thesis (mentioned in Chapter 1) are the result of two speculative parallelization systems implemented purely in software. Thus, any work discussed in this chapter concerns software implementations of speculative parallelization systems unless otherwise stated. A significant portion of Chapter 2 also explored the various dimensions to implement a speculative parallelization system: metadata, version management, conflict detection, commit/rollback, and scheduling speculative threads. This is illustrated in Figure 3.1.

Those categories will lead the structure of this chapter while explaining the main work in software speculative parallelization (summarized in Table 3.1). The last column of Table 3.1, `MiniTLS` and `Lector`, is published work by the author of this thesis. Both systems are involved in the contributions of this thesis and are described in detail in Chapters 4 and 5 respectively. As in Chapter 2 the terms *Speculative Parallelization*, *Thread-Level Speculation*, and *TLS* will be used interchangeably throughout the rest of the chapter. To aid the information flow of this chapter, some categories will be

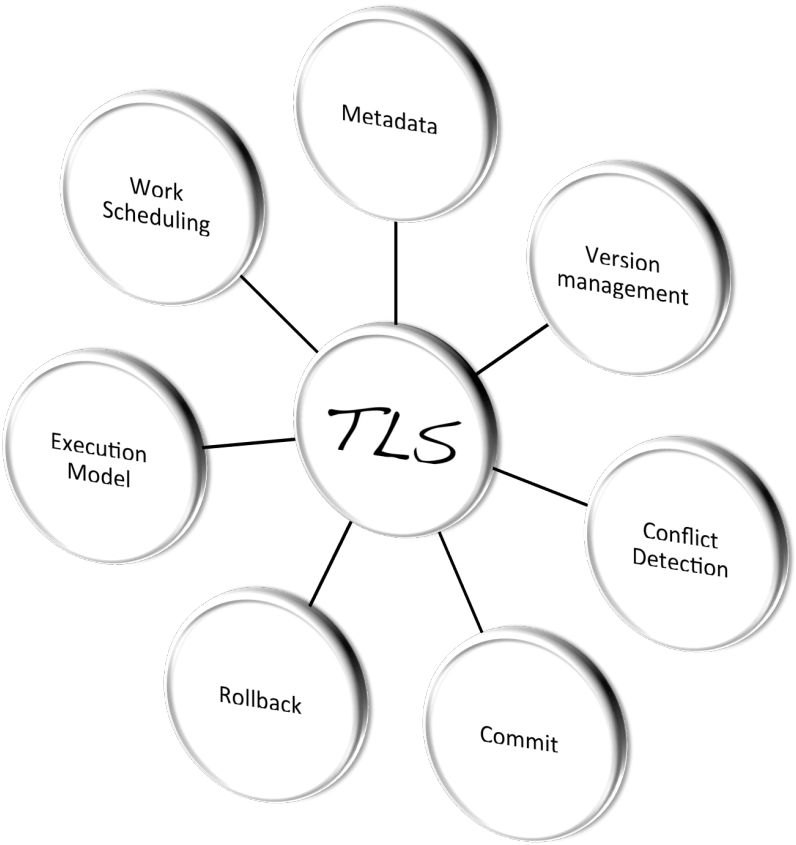

Figure 3.1: The various design points that make up a speculative parallelization system.

merged together and one more category will be added in regards to the runtime execution technique each system utilizes. For example, Chapter 2 explained two different runtime execution models: *Inspector/Executor* and *Speculative Parallelization*.

3.2 Execution Model

3.2.1 Inspector/Executor

The first serious discussions and analyses of Runtime Parallelization emerged during the late 1980s with the Inspector/Executor technique. Early work on this model [ZY87, SMC89, SM91, SMC91], focused on how the inspector phase could analyze

YEAR	WORK
1994	DOALL test [RP94a]
1995	LRPD test [RP95]
1998	Gupta and Nim [GN98]
2001	Rundberg and Stenström [RS01]
2002	R-LRPD [DYR02]
2003/2005	Cintra and Llanos [CL03, CL05]
2008/2010	CorD-based systems [TFNG08, TFG10]
2009	SpLIP [OMH09]
2009	STMLite [MHHM09]
2010	Raman *et al*. [RKM+10]
2013	MiniTLS, Lector [YRHBL13]

Table 3.1: Advances in the literature of speculative parallelization.

the runtime access patterns among loop iterations and produce a valid execution schedule that the executor phase would later materialize. Subsequent attempts of Inspector/Executor [RP94a] used the model to identify the presence of a fully parallel loop at runtime and proceed with parallel execution. Apparently the model was successful mainly in cases where an efficient inspector loop could be obtained [Rau98]. That is, if the inspector loop is not able to be decoupled sufficiently from the loop (*i.e.* small code replication), then the inspector becomes computationally expensive making it nearly equivalent to the loop itself. Mainly for this reason, research in runtime parallelization turned towards solutions employing speculation.

3.2.2 Speculative Parallelization

The first traces of speculative runtime parallelization in the literature was the DOALL test proposed by Rauchwerger and Padua [RP94a]. The DOALL test is a runtime technique for *DOALL* loop identification (*i.e.* loops without cross-iteration data dependencies) initially practiced using the *inspector/executor* model. The test later became the core of one of the earliest and most influential work on speculative parallelization known as the LRPD test (Lazy Reduction Privatization DOALL test) [RP95], proposed also by Rauchwerger and Padua.

LRPD [RP95], eliminates the overheads of a possibly poor inspector (one that contains a large potion of the original loop) by combining the *inspection* and *execution* face in a single step. Furthermore, certain types of anti and output dependencies are

removed by using a transformation known as *privatization*. Privatization transforms certain shared variables into private copies for each thread cooperating on the execution of the loop. For instance, variables that are first *defined* (written) every time before they are *used* (read) inside the same loop iteration, can be safely privatized. Sometimes, variables initialized outside of the loop could be privatized if a *copy-in* mechanism is provided (*i.e.* provide a copy of the external value for the first use of that variable in each iteration). Similarly, if a privatized variable is required after the loop execution, a *copy-out* mechanism needs to be provided to ensure that the correct value is copied out to the original (non privatized) version of that variable.

While inspection of memory accesses takes place by a thread, the actual memory values of the shared user data structure are computed as well, only instead of being placed immediately in main memory, they reside in thread-local storage until inspection completes. This introduces a notion of *speculation* on the values being produced during the loop execution. Yet another novel feature of `LRPD` is the ability to identify and handle code that fits the *reduction*[1] pattern, thus allowing for more loops to be parallelized.

The majority of subsequent speculative system implementations in the literature [GN98, RS01, DYR02, CL03, CL05, TFNG08, OMH09, MHHM09, TFG10, YRHBL13] are considered as an extension to the `LRPD` test.

3.2.3 Speculative Parallelization with Inspection Support

Yiapanis *et al*. [YRHBL13] propose a TLS system, called `Lector`, where the techniques of speculative parallelization and inspector/executor are combined together. For simplicity, assume that speculative parallelization is implemented in a similar manner to the `LRPD` [RP95]. From the inspector/executor model, only the inspection loop is manipulated. Lightweight threads, coined in `Lector` as *inspector threads*, are extracted and applied in a similar fashion as the `DOALL` [RP94a] runtime test. While inspector threads are running, speculation continues as usual. Inspector threads do not replicate the entire code from the loop and thus are expected to be faster than typical TLS threads. If inspector threads determine that the loop is fully parallel, speculation is withdrawn and parallel execution continues without the speculative overheads. If inspector threads fail due to data dependencies, speculation continues without any changes.

[1] Reduction operation of the form: $x = x \otimes exp$, where $\otimes$ is an associative operation and variable x does not occur in exp or anywhere else in the loop.

The traditional inspector/executor model would suffer performance losses in two cases: (a) When the inspector replicates a large portion of the loop, (b) When the inspector identifies data dependencies since the inspection time is completely lost. Combining the model with speculation these two drawbacks are addressed as follows: (a) Even if inspection completes at the same time as speculation, at least the loop was executed speculatively rather than serially, (b) If the inspector identifies data dependencies the inspection time is amortized by having speculation executing the loop simultaneously.

`Lector` is one of the contributions of this thesis and it is described in greater detail in Chapter 5.

3.2.4 Decoupled Software Pipelining with Speculation Support

Raman *et al*. [RKM+10] demonstrated a different approach to runtime parallelization by enhancing a technique known as *Decoupled Software Pipelining (DSWP)* with speculation support.

DSWP is somewhat similar to another earlier form of offline-based parallelism transformation known as *DOACROSS* [Cyt86]. DOACROSS targets loops with cross-iteration data dependencies and enables parallelism by scheduling parts of each loop iteration across multiple threads. In DSWP each thread executes part of the loop for all iterations and threads are scheduled is such a way to form a pipeline. To better understand the difference consider an example[2] code traversing a linked-list as in Figure 3.2a and its dependence graph (the graph showing the program's data dependence relationships). The pointer chasing load is labeled "LD" and the loop body is labeled "X".

Figure 3.2b illustrates how the DOACROSS technique would schedule the loop iterations among two threads ($T1$ and $T2$) in an alternate fashion. This way the body of the loop in one thread can be executed in parallel with the next field traversal load of the other thread. In contrast, the DSWP technique (see Figure 3.2c) schedules the iteration of the loop in a way that half of the iteration (the pointer chasing load) is in one thread and the other half (the body of the loop) in another thread. What DSWP aims to optimize over the DOACROOS technique is to keep the loop's critical path (the longest path in the dependence graph) dependence in the same thread. If the critical path has to be routed across threads as in the DOACROSS example, the total execution time of the

[2]This example was taken from [ORSA05a].

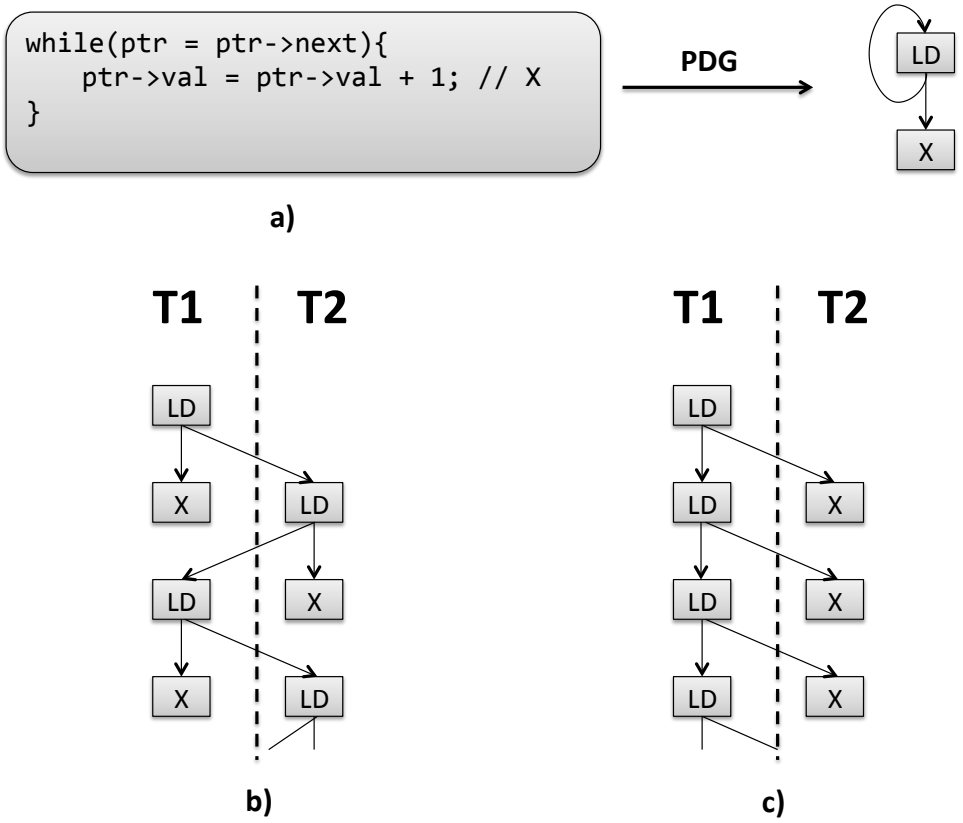

Figure 3.2: a) Linked-list traversal. b) DOACROSS scheduling. b) DSWP scheduling. This example appears in [ORSA05a]

loop may increase due to communication costs. Placing the critical path in the same thread allows decoupling from that communication latency among multiple threads. One limitation of DSWP is the requirement that the loop must be able to be broken up in such a way that all instructions from the same recurrence (strongly-connected component) in the flow graph can be placed on the same thread.

By default, DSWP is a non-speculative technique and therefore has to respect all dependencies in the loop. In an attempt to allow more loops to be parallelized, Vachharajani *et al*. [VRR+07] proposed the first system towards speculative DSWP, although, requiring specialized hardware support. Raman *et al*. [RKM+10] present a software approach for DSWP by providing a software TLS back-end to the initial idea in [VRR+07]. In their work, they coin this TLS support as *Software Multi-threaded Transactions (SMTX)*.

As with the non-speculative technique, speculative DSWP is limited to the number of strongly-connected components in each loop, which is typically much smaller than

the number of loop iterations [RVOA08].

3.3 Metadata and Version Management

Typically work on runtime parallelization manipulates additional data structures to indicate an action on a particular memory location (this idea was explained in Chapter 2, Section 2.3.2). Usually these data structures are implemented using arrays or hash tables. Since each of those array elements reflects a memory location, they are referred to as *shadow arrays*. There are three ways that have been mainly used in literature to maintain metadata: (a) Keep the shadow arrays private to threads and check for correctness only at the end of speculation, (b) Keep shadow arrays shared among threads so that each thread could see what other data threads are accessing during speculative execution, and (c) Keep shadow arrays private to threads but provide means to enable early detection in case of a conflict. In other words, allow threads to commit partial results during execution so that conflicts are not delayed until the end. The next sections elaborate more on those three types of maintaining metadata.

3.3.1 Speculation with Decoupled Shadow Data

Early work on speculative parallelization was fully optimistic in the sense that it was more effective when speculation triggered no conflicts. Speculative threads did not attempt to communicate between each other until only after the end of their corresponding speculative execution. Thus, a misspeculation was only detected after the final commit.

The `DOALL`

The `DOALL` test [RP94a] involves the manipulation of helper data structures in order to track any memory accesses performed on the user shared data structures. Loads and stores are marked during program execution based on memory accesses using those helper or *shadow* data structures. In its simplest form, the compiler analyzes the loop to be parallelized and generates shadow data structures for each user data structure under question. Figure 3.3a shows an example of a candidate loop for *DOALL* parallelization. In order for the loop to be classified as *DOALL*, no data dependencies must exist among different iterations (or chunks of iterations). The shared data structure in question for this example is an array $A[n]$, where n indicates the size of the array.

Assuming that integer arrays B and C are populated at runtime, there is no way to analyze A for data dependencies at compile time as its indexes are unknown by that time. During compilation two versions of the loop are generated. The first loop will be used to compute all the indexes and perform the `DOALL` test, without actually modifying any shared data. The second loop, provided that the `DOALL` test was a success, will use those indexes to access the actual storage and perform the computation. Two additional arrays, $Load[]$ and $Store[]$, are introduced to record the indexes of loads and stores respectively for array A (see Figure 3.3b). The two arrays have the same length as A and are used only to mark index locations. Loop iterations are distributed among multiple threads and the shadow structures are replicated for each thread to perform the marking concurrently (without the need of synchronization). At the end of the inspection/marking phase, the different copies of those shadow structures are examined. For a given index i, if $Load[i]$ and $Store[i]$ are both marked in different threads (*i.e.* chunks of iterations) then the loop is NOT a *DOALL*. If no load from one thread intersects with a store from a different thread in the same memory location, then the loop is classified as fully parallel. Several extensions to this simplified version have been proposed over time and will be discussed below. Given the test yields a success, the second loop that will perform the actual computation can be executed as a *DOALL* loop across multiple threads.

The `LRPD`

The `LRPD` test [RP95] takes the `DOALL` test [RP94a] a step forward and actually computes the loop in a speculative fashion while inspecting. `LRPD` allocates five data structures in total: four boolean-valued shadow arrays to indicate actions on memory locations and another array as temporary space for speculative values to be stored during execution. The four shadow arrays are defined as follows:

Load: An element $Load[i]$ in this array is set to "true" to indicate a speculative load performed on a memory location i in the user space.

Store: Similar to the load array, an element in this array is set to indicate a speculative store operation on a memory location.

NotPrivatizable: A thread sets an element in this shadow array if a load was performed in a location without a preceding write by the same thread. This is done

a)

```
for(int i = 0; i < n; i++){
    A[ B[i] ] = … ; // Iteration 1: B[i] = 4
    … = A[ C[i] ];  // Iteration 1: C[i] = 6
}
```

b)

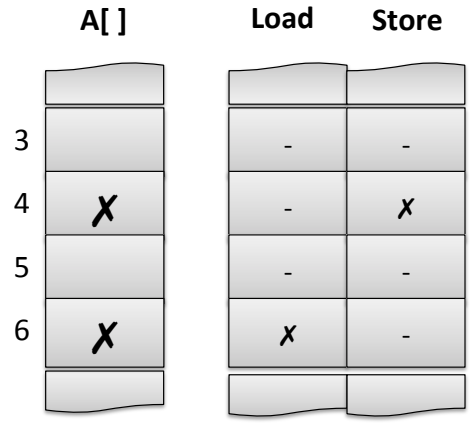

Figure 3.3: DOALL test [RP94a] basic data structures.

to prevent the possibility of the loading thread reading a value produced by a different thread in the wrong order. Thus, the array is used to indicate an element that cannot be privatized.

NotReduction: Indicates that memory location cannot participate in a reduction operation.

When the loop is parallelized speculatively, every thread is assigned a chunk of iterations to execute. Each thread has its own copy of the above data structures that reflect only the memory locations accessed by that thread. A local copy of those data structures can be accessed only by the thread that owns it. Therefore, during execution there is no need for synchronization to mark those arrays. Nevertheless, at the end of the speculative execution, all these local copies must be merged in order to check for cross-thread memory accesses.

LRPD is an example of a lazy version management system, since tentative stores reside in private storage and only become visible to memory at a thread's commit

time.

3.3.2 Speculation with Shared Shadow Data

In the work discussed above [RP94a, RP95] threads avoid communication between them until the end of speculative execution. Other proposals, such as the ones that will be discussed below, expose the metadata to multiple threads during execution but prevent data races between threads through the use of locks or CAS operations. One important reason that a TLS system designer might choose to do that is to allow threads to detect early misspeculations.

Rundberg and Stenström

Rundberg and Stenström [RS01] proposed a speculative parallelization scheme in which a thread must first secure *exclusive ownership* of a particular location before any speculative access. This is enabled by requiring a thread to acquire a lock associated with each memory location that may be accessed in a speculative manner.

Every user shared data structure is shadowed by three helper arrays. Every location of these shadow arrays is associated with an individual memory location in the user shared data structure, the same way as in the `LRPD` test [RP95]. The first shadow structure is an array of locks used to indicate ownership of a particular location by a thread. The remaining two are used to keep an identifier for a thread that has performed a speculative load or store on a particular location, respectively. Also a private storage is maintained for each thread to buffer any speculatively produced values.

Cintra and Llanos

Cintra and Llanos [CL03, CL05] use a slightly different layout for the metadata than the work from Rundberg and Stenström [RS01]. Apart from the private storage for speculative values, three other shadow arrays are used as follows (and illustrated in Figure 3.4):

AT: The AT array, short for Access Type, shadows the user memory locations. It is used to contain information about the access type upon a memory location by a thread. Access types can be in *Read* or *Mod* state (corresponding to Load or Store actions respectively), in *NotAcc* (to indicate a memory location never accessed before by different thread) state, or in *ExpLd* state (to indicate a memory

location could have potentially been written by a different thread and consumed by the current thread - this action is termed as *exposed load* in this work).

IA: Indirection Array. This array is just a summary of memory locations in state other than *NotAcc* (not accessed) for each thread. It is used to speed up checking which data a given thread has accessed. An integer variable "Tail" is used to indicate the last element of this array.

GlExpLd: An element of this array (Global Exposed Loads) reflects a memory location across multiple threads. If a given memory location *i* shadowed by GlExpLd is consumed by a speculative thread but not written first (*i.e.* exposed load), then $GlExpLd[i]$ is set to indicate a potential violation. This is another attempt to speedup the checking process. If a memory located has never been exposed loaded then there is no need to be included in any checking for violation.

Cintra and Llanos [CL03, CL05] work is also an example of lazy version management. In their work, to prevent certain memory access violations, threads are allowed to communicate values between them using a mechanism known as *value forwarding*. This will be revisited in the discussion regarding conflict detection.

`SpLIP` and `MiniTLS`

`SpLIP` [OMH09] and `MiniTLS` [YRHBL13] are two speculative parallelization examples of eager version management. `MiniTLS` is one of the contributions of this thesis and it is described in detail in Chapter 4.

`SpLIP` [OMH09] is the first TLS system supporting *in-place* updates (eager version management). The systems discussed so far employ a *deferred* update (or lazy) version management. A thread-local buffer is still required, only in this case it is used to record the original value of a memory location just before the speculative memory update is performed. Two supporting data structures are required in order to record the thread id that is currently performing a load or a store for a given location, respectively. Up to this point the data structures used are very similar to the ones used in the work by Rundberg and Stenström [RS01]. A major difference from [RS01] is the interesting way exclusive ownership is defined. `SpLIP` takes advantage of certain properties of the Intel *x86* architecture in order to abolish locks or *CAS* operations used by conventional TLS systems to protect a speculative location from multiple thread accesses. This comes at the cost of increasing memory requirements by requiring two additional

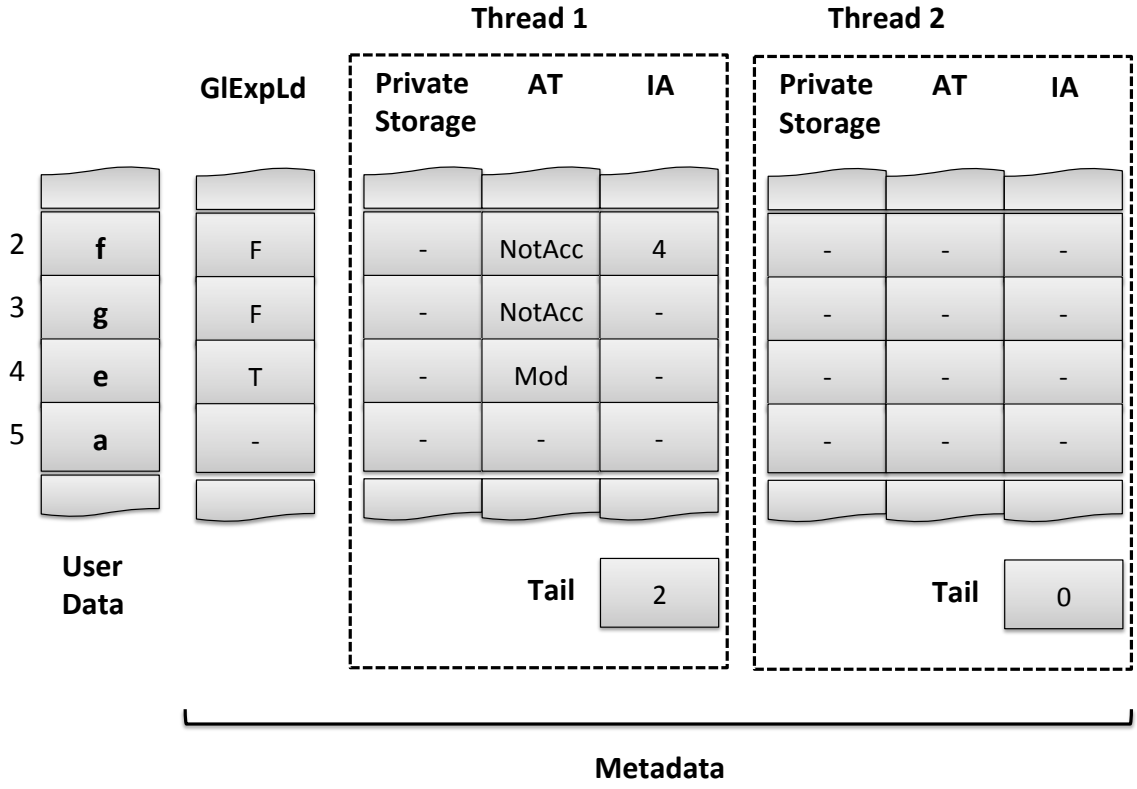

Figure 3.4: The data structures used by Cintra and Llanos [CL03, CL05]. "AT" stands for Access Type,"IA" stands for Indirection Array, and "GlExpLd" stands for Global Exposed Load. The values inside the "GlExpLd" can be either true (T) or false (F).

shadow data structures to ensure proper synchronization between multiple thread accesses (the use of the two arrays is discussed in Appendix A). Intel *x86* guarantees that read/write access to a 64-bit word occurs atomically as well as access to any subwords of the corresponding word. Exploiting this information the four shadow structures (two for read/write marking and two for read/write synchronization) are implemented as interleaved, aligned 16-bit subwords of a 64-bit word; reading any of these is replaced by reading the full word and computing the required value. As a result read/write operations from `SpLIP` are sound for Intel *x86* without the use of locks even though this hardware does not provide sequential consistency. A multi-core system is sequentially consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [Lam79]. This sequentially consistent behavior is likely due to flushing the cache line when cache

coherency detects concurrent accesses to a word and one of its contained subwords. More details on SpLIP can be found in Appendix A.

3.3.3 Speculation using a Centralized Manager Thread

The work discussed above concerned either systems that communicate their shadow arrays at the end of speculation or work that the shadow arrays are shared between multiple threads. The following paragraphs discuss a different approach in which although shadow data are distributed among threads, there exist a centralized manager unit to ensure misspeculations are detected before the end of speculation. Also if the manager is executed in a dedicated thread and it is the only thread allowed to commit values to memory then commit-time locks can be avoided. Nevertheless locks may still be required to coordinate multiple threads from accessing the manager thread and *vice versa*.

CorD

Tian *et al*. describe CorD [TFNG08], a TLS system which maintains a central manager unit that is dedicated to only one thread. The manager is the only thread that has access to user data and it is not speculative. Each speculative worker thread maintains its own private space for marking and execution. When a speculative thread is created, any value that is needed, is copied-in from the non-speculative state (the user data) to the speculative one (the worker's thread private space) and later the results are copied back if speculation was successful. This transfer of data between the user space and the speculative space is performed only through the central manager. A mapping table is maintained for each speculative thread that has entries associated with each variable's copy. *Version* numbers are used between the two states (user space and speculative space) in order to detect misspeculations. When a memory location is needed, the main thread provides a copy of the associate value to a speculative thread and stores an integer value in the "version shadow array" (shown in Figure 3.5) for that location. The version number is also provided to the speculative thread's mapping table. If that user location has changed during execution the version number will be updated. When a speculative thread finishes execution, its mapping table will be sent to the manager thread. The mapping table will be traversed by the manager thread to identify any expired versions. The "WriteFlag" is set to *true* by a speculative thread to indicate the memory locations needed to be copied back to the user space. Work from speculative

threads is committed in-order by the main thread.

Furthermore, in [TFG10] they address certain challenges specific to applications manipulating dynamic data structures. For example, the copying of objects is limited to only those that are modified by a speculative thread. Also when an object is moved to the speculative space, all other references that may point to that object must be changed to point to the speculative copy rather than the original. This causes the address translation problem. To overcome copying all those references, the authors introduce *double pointers*. Under this idea, every pointer variable in the program, is modified by the compiler and represented by two pointers. One for the non-speculative state and one for the speculative state. In this way extra copying is avoided by using the appropriate pointer for corresponding space.

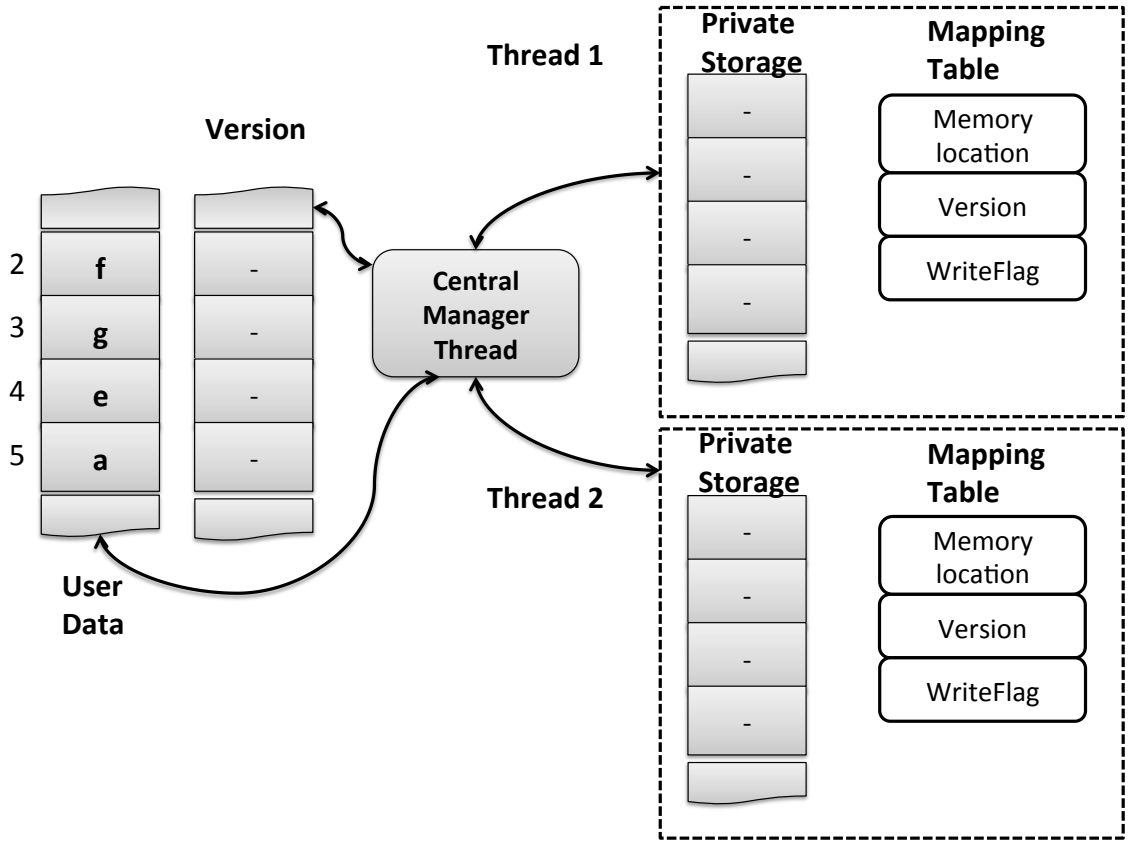

Figure 3.5: Metadata used for `CorD` [TFNG08].

One requirement in `CorD` is that the user space and speculative space remain completely separated. The manager thread supplies any value needed to a newly created speculative thread, even values to be loaded. This could cause unnecessary copying overhead since anyway if a value needs to be loaded it could be retrieved on demand

from main memory. In any case `CorD` would invalidate a copy that was updated during speculative execution. Also in many scenarios, pointer values may be undefined by the time they are required to be copied (*i.e.* when the speculative thread is created). `CorD` addresses this issue by delaying the copying of the pointer address up until it is actually read for the first time.

`STMLite`

Mehara *et al*. [MHHM09] present `STMLite`, a Software Transnational Memory (STM) model modified to support speculative parallelization. `STMLite` aims to reduce the overhead associated with validating the read-set by decoupling the conflict detection process from the main process using a Central Commit Manager (CCM). Also individual locks for copying out the write-set are avoided. The CCM, runs on a dedicated thread and allows only one transaction (or in this case a chunk of iterations) at a time to write to a particular location (note that, multiple transactions can update different locations concurrently). During execution, each transaction computes their read and write accesses in the form of software *signatures* (inspired by hardware signatures [CTTC06]), while buffering their speculative store values. At the end of their speculative execution, signatures are send for validation to the CCM. If granted by CCM, a transaction is safe to *commit* its buffered values. The relative start and commit times of transactions is tracked using a *global clock* mechanism (a shared counter used to maintain consistency between transactions), which is incremented every time a transaction commits. The clock is incremented every time a transaction *commits* and it is used to invalidate other "live" transactions with an out of date value. This is possible since every memory location is associated with a version number. Before the transaction terminates, as part of its commit procedure all write and all read accesses are re-validated. If successful, the global clock is incremented atomically, all speculative updates are propagated to memory, and all writes are amended to hold the new value of the global clock. This way the next committing transaction can ensure consistency by comparing their previously recorded local value of the global clock against the current up-to-date clock of any values read. They global clock idea for STM was first used in [SMSS06]. Loop iterations are given ordered ids and they must *commit* in that order.

Raman *et al.*

Raman *et al*. [RKM+10] also use a centralized commit unit in a similar manner as in `STMLite` [MHHM09]. Their system uses lazy version management and therefore

speculative loads must first consult their write-sets to check whether that value has been already produced there. This incurs additional costs on such systems especially for applications with long running iterations as every load operation must first search potentially large write buffers. Raman *et al*. [RKM+10] take advantage of certain properties of the operating system to optimize such a potential bottleneck. The speculative regions are executed inside UNIX processes. This allows the underlying virtual memory to transparently create private physical copies of locations being updated (at the page granularity). As a result any subsequent load operation to a memory location being updated in the same speculative thread will have the up-to-date value (with respect to that thread) available without the need to scan the write-set.

Furthermore, their execution scheme (explained in Section 3.2) allows a loop iteration to span across multiple threads. Since the unit of atomicity is still considered to be a loop iteration, the proposed system allows a speculative thread to generate sub-threads in order to cover the entire code from a single iteration across the pipeline. This complicates further the implementation of the system as the interaction between parent speculative threads and sub-threads must be handled with care. Such a situation is known as *transaction composability problem* in databases. *Nested transactions* provide suitable semantics to address composability [BN97]. For instance, a parent transaction (or thread in this case) is not allowed to commit until all of its children commit. Counters are introduced to the parent transaction which increment when a child transaction starts and decrement when one commits. Furthermore, in TLS, the speculative order among children threads is only meaningful within their parent thread. Aborting a child transaction entails aborting the parent thread entirely.

The `CorD`-based systems [TFNG08, TFG10], `STMLite` [MHHM09], and the system by Raman *et al*. [RKM+10] employ lazy version management.

3.4 Conflict Detection, Rollback, and Commit

Earlier, Section 3.3 described the methods that different TLS systems devised to enable a way of communication between the accesses from speculative threads and the rest of the system. The following paragraphs will provide the critical piece that is missing from the puzzle which is how each of those approaches maintains correctness during speculative execution. This section is structured based on the two popular ways that a TLS system can use to detect a conflict: *Lazy* and *Eager* conflict detection. While discussing conflict detection, information about different types of commit and rollback

will be explained.

3.4.1 Lazy Conflict Detection

A thread that delays the validation for correctness to the end of its speculative execution is said to employ a *lazy* conflict detection mechanism. The benefit of acting as such is to avoid the overhead of checking for correctness upon every speculative access. The downside is that if a misspeculation is detected, any work performed during speculative execution is wasted. So far in the literature of software speculative parallelization, lazy conflict detection has been used only by lazy version management systems.

`LRPD`

`LRPD` test [RP95] is a very simple and effective technique for applications that have no data dependencies. The user data structure is shadowed by helper arrays to mark the loads and stores from each thread. At the end of speculative execution the arrays are examined.

Rollback & Commit: In the simplest case, a conflict can arise *if two threads have marked the same location, one thread as a store and the other one as a load*. That is, the same memory location has been read and written in different iterations (threads in this case) indicating a RAW or WAR violation. Nevertheless, if the load is preceded by a store from the same thread then there is no conflict since speculative values are committed lazily. Furthermore, the `LRPD` keeps track of the total number of store operations performed by all threads. If that number is different from the total number of store operation markings in the shadow arrays then some locations have been overwritten causing a WAW violation. Also violation is caused if a variable was not able to be privatized but written during speculation. When a conflict is detected all speculative work is discarded and the loop re-executes sequentially. If the loop is found validly parallelized then all threads can commit their speculative results in parallel to main memory (in parallel only if no two or more threads have performed an update to the same memory location, otherwise sequentially and in-order to avoid WAW conflicts).

The `LRPD` test [RP95] is probably one of the laziest[3] TLS systems. The test for correctness is performed at the end of the entire speculative execution. While this scheme

[3]Laziest in terms of conflict detection.

is ideal for an application that contains no data dependencies, many opportunities for parallelization remain unexploited.

R-LRPD

As mentioned earlier, the main disadvantage of the LRPD [RP95] is having to waste all speculative work performed during execution, by discarding all computed values, when the test was unsuccessful. The R-LRPD (Recursive-LRPD) test [DYR02] aims to address this shortcoming. To a great extent R-LRPD is the same as its predecessor. The loop is executed speculative using the same technique as in LRPD and performing the correctness test at the end. If the test is "passed", then the process is equivalent to the LRPD test. However, in the unfortunate case where the test fails, a complete *rollback* is not necessary. Instead, all iterations of the loop under question that computed any results before the misspeculation is triggered, are allowed to *commit*, whereas the rest of the iterations are re-executed in the same fashion. This process continues in a recursive fashion until all the executions of the loop complete.

This optimized *rollback* procedure of R-LRPD is made possible by using a *sliding window* scheduling mechanism. Only a set of iterations are executed in the window at a time. Thus, successful execution of iterations inside the same window phase appears the same as LRPD. When a conflict arises, it is isolated within the current window of execution.

Gupta and Nim

Gupta and Nim [GN98] also aim to address the shortcomings of LRPD by proposing some simple extensions. The main extension concerns elements not able to be privatized. In LRPD a thread that loads a value initialized by a different thread causes a violation. In the scheme proposed by Gupta and Nim [GN98] a thread that attempts to load a value not initialized by itself can decide to stall until the previous threads have finished execution. This is done by inserting synchronizations for the threads that may require a value not yet produced. This scheme assumes that threads are totally ordered and iterations are assigned to threads in increasing order. This implies that a lower thread in the ordering will always be less speculative than a higher thread. The stalled thread can proceed only when its less speculative threads have finished and thus commit the value needed in main memory.

CorD

A system, namely CorD [TFNG08], with a centralized manager thread was introduced earlier in Section 3.3. Figure 3.5 shows the metadata organization for that system. The manager thread is the only one allowed to interact with the user data. When execution begins, the central manager creates a speculative thread copying-in to that thread's private storage any values needed from the user space. For every user memory location the manager associates a version number. When a value is required by a thread, the current version number is also given by the manager. During execution a speculative thread loads and updates values only in its local storage. The speculative thread also maintains a mapping table that contains the address of a given location, its original version number when copied-in, and a flag. The flag is set only to indicate a location that has been updated locally.

Commit: When a speculative thread completes, the manager is notified. The manager will consult the mapping table of that thread to identify which locations have the flag set and therefore be copied-out to the user space. When a value is updated in the user space, its corresponding version number also changes.

Rollback: When the manager consults the mapping table of a speculative thread the version numbers must be examined. If the thread contains a value that has an expired version then a violation is raised. The manager notifies the speculative thread to discard any work done and the values are copied-in again to restart execution.

STMLite

The process for conflict detection in STMLite [MHHM09] is similar to the one implemented in CorD [TFNG08]. Once a speculative thread completes execution, the addresses of the values read and written are send to the central thread. The central thread also maintains a log containing the values committed by previously successful speculative threads. The read and write addresses are compared against the log of committed values.

Rollback: If a conflict is identified between a value consumed by the current speculative thread and a value residing inside the log of committed values, the central manager notifies the thread to discard its local buffer and restart execution.

Commit: If the values used by the speculative thread do not intersect with any of the values in the committed log the central manager notifies the thread to start copying any updates to main memory. Since the central manager in this case (and unlike in `CorD` [TFNG08]) allows speculative threads to begin committing their values independently and in parallel, a mechanism is required to prevent concurrent memory updates by multiple committing threads. This is implemented using an additional data structure that shadows the number of active thread in the system. This data structure is used as a synchronization point which indicates when a particular thread has finish committing its values. The manager thread contains the addresses to be committed by all threads. Before the manager allows a thread to begin the commit process, it consults the write signatures from the currently committing threads. If any of the write signatures from the speculative thread waiting permission to commit conflicts with a thread currently committing, then the manager will postpone the commit for the waiting thread until the others finish.

Raman *et al.*

The speculative parallelization system proposed by Raman *et al.* [RKM$^+$10] also uses a lazy conflict detection mechanism. Like `CorD` [TFNG08] the central manager is notified when a thread completes executing its corresponding speculative region. The central manager is then responsible to identify any conflicts among speculative threads. The main difference from `CorD` is the extra complication from supporting the sub-threads. The children threads must commit within the context of their parent threads. This is required to maintain consistency among speculative threads. The authors address this issue by having the sub-threads copying on demand the values they require from previous sub-threads (or the parent thread). Consequently the parent thread as well as its children all have the same version of memory.

3.4.2 Eager Conflict Detection

Rundberg and Stenström

In the software TLS system proposed by Rundberg and Stenström [RS01] a given user memory location is protected by a lock as well as keeping information about its latest reader and writer threads. This feature allows a speculative thread to check in isolation (from other threads) whether reading or writing the value in question can cause a violation. Their implementation prevents a speculative load to cause a data

dependency violation by supporting a technique known as *value forwarding*. Using this technique, a thread performing an *exposed load* (*i.e.* a load on a value not produced by the same thread) is allowed to search backwards (in terms of speculation order) and find the latest value produced by a less speculative thread to serve that particular load. After the latest value is found, it can be forwarded from the less speculative thread's buffer to the more speculative thread's buffer bypassing any rollback related costs.

Rollback: A conflict is caused only as a result of a thread reading a value "too early". For instance, imagine a thread that is about to perform a speculative store on location x. The thread first locks x and then checks if a different thread has already loaded that location. If indeed another thread has loaded that value and that thread is more speculative, then a misspeculation is detected by the system as this causes a RAW conflict. As a result the more speculative thread that caused the conflict and all its successor threads must be squashed and discard their local buffers.

Commit: A thread that carried out its speculative execution without any conflict arising is allowed to commit its results to main memory. Typically, in a system that employs a lazy version management, such as this one by Rundberg and Stenström [RS01], speculative threads commit their buffered values in order one by one. Their scheme allows this process to be optimized by offering a *parallel commit* phase. Recall that the shadow arrays keep track of the latest thread that has written to a particular location. Even if multiple threads have performed a write on a location during speculation, the only one that must provide memory with the correct value for that location is the latest thread. Before commit, the "Store" shadow array that corresponds to the memory locations to be updated is inspected to find the latest threads recorded there. Threads can proceed committing in parallel by having each memory locations updated by the latest thread written on them.

Value Forwarding synchronization: The authors also present a more efficient implementation where Load and Store shadow array locations are represented at the byte size. Since the architecture they tested supports atomic byte operations, this enables them to perform speculative loads by issuing low level hardware atomic loads without using explicit locks. Such an optimization needs to be handled with care when the system allows value forwarding because the future thread must be able to "see" the correct value produced by the past thread. In other words the thread that owns

the value to be forwarded must first write that local copy before the forwarding thread loads it. Rundberg and Stenström [RS01] handle this case in the following way: the thread that owns the value to be forwarded is allowed first to write a $0x0F$ to its byte in the Store shadow array, then perform the update of the local copy, and after this must write $0xFF$ to that byte. A load, from the thread that requires that value, discovering a $0x0F$ to the byte of the store shadow array simply needs to wait for it to change to $0xFF$ before it performs the forwarding load operation.

Cintra and LLanos

Cintra and Llanos in [CL03, CL05] experiment with both lazy and eager conflict detection. The idea of implementing lazy conflict detection is so that the cost of checking every memory location (which usually requires synchronization) is avoided. However, delaying the conflict detection will cause wasted work if a conflict is triggered. Cintra and Llanos in [CL03, CL05] indicate that the cost of checking for violations on every speculative access is negligible compared to the cost over checking only at the end of speculative execution where significant amount of work might be wasted. The conflict detection idea is the same as in the TLS system by Rundberg and Stenström [RS01]. Rollback is only triggered by stores as loads do not cause conflicts when value forwarding is enabled. Their system differs in terms of commits from the one by Rundberg and Stenström [RS01] in two ways: (a) Cintra and Llanos further optimize their systems for cases where there were no *exposed loads* present on a given window execution. This is accomplished by introducing an extra data structure that raises a flag whenever any one thread performs the first *exposed load* for a given location (see Figure 3.4), (b) Cintra and Llanos employ a serial commit phase.

`SpLIP`

`SpLIP` [OMH09] and `MiniTLS` [YRHBL13] differ from all other TLS systems in that they are the only software TLS systems implementing an eager version management system. Eager version management systems perform speculative updates to main memory but buffer the original values before they do so. Threads update directly the main memory with speculative values and as a consequence all three types of violations are possible (RAW, WAR, WAW).

`SpLIP` uses five shadow arrays to facilitate conflict detection (For simplicity only three of them are discussed here. For more details see Appendix A):

Load: To mark the latest thread that performed a load on a memory location.

Store: To mark the latest thread that performed a store on a memory location.

TimeStamp: To mark the relative time that a thread performed a store on a memory location.

A conflict can be detected as follows:

Read-After-Write: A speculative thread attempts to perform a store to a memory location but discovers that a more speculative thread has already consumed the value from there.

Write-After-Read: A speculative thread attempts to perform a load from a memory location but discovers that a more speculative thread has already stored a value there.

Write-After-Write: A speculative thread attempts to perform a store to a memory location but discovers that a more speculative thread has already stored a value there.

Rollback: When a conflict is detected, the offending threads must go through their local buffers which include the original values and restore memory back to the latest known correct state. An issue arises when more than one thread involved in the violation has written to the same memory location. That is because only one of them must restore the correct value back to main memory - the one that has the earliest copy in terms of speculation order. `SpLIP` [OMH09] uses a "TimeStamp" shadow array to record the relative time a thread has stored to a location. Using this shadow array the system is able to recover the earliest value need to be rolled back.

Commit: Since threads update directly the speculative values in memory, if no conflict is detected then the final results are already there. Thus commit implicitly happens *in parallel*.

`MiniTLS` [YRHBL13] detects conflicts in the same manner as in `SpLIP` [OMH09]. `MiniTLS` offers an enhanced way of performing a parallel rollback and is explained in detail in Chapter 4.

3.5 Work Scheduling

Static and Dynamic Work Scheduling

The `LRPD` test [RP95] can be applied either with *static* or *dynamic* scheduling. With static scheduling the iteration space is divided evenly among the number of threads. Using dynamic scheduling large chunks of iterations are assigned at runtime to the number of threads (usually the number of threads is significantly smaller than the number of chunks).

The work by Gupta and Nim [GN98] can also use both static or dynamic scheduling. The only requirement is that threads are assigned contiguous chunks of iterations and threads are totally ordered by their speculation level. This allows the notion of less and more speculative thread. Remember that work performed by a less speculative thread can potentially invalidate a more speculative thread. The same scheduling policy is also used by many others [RS01, TFNG08, TFG10, MHHM09, OMH09, RKM$^+$10].

Sliding Window Work Scheduling

Three work scheduling strategies have been investigated using `R-LRPD` [DYR02] to recover from misspeculation. First, the *Non-Redistribution (NRD)* strategy is examined, where failed iterations must re-execute their work in the threads that were originally assigned to. A major issue with NRD is the load imbalance that can be potentially introduced when some threads finish earlier than others and have to wait sitting idle. This problem is addressed in the other two work scheduling techniques. The *Redistribution (RD)* strategy allows iterations, involved in *rollback*, to subdivide themselves and be distributed among different threads. Finally, a *Sliding Window* scheduling strategy is applied where contiguous chunks of iterations are assigned to a group (or window) of threads in a way that satisfies a speculation order. Iterations in lower indexes of the window are less speculative than iterations in higher indexes and a less speculative chunk of iterations has priority over a more speculative chunk. This ordering is important when different chunks of iterations are involved in a data dependency violation or they are ready to commit.

Two types of sliding window are evaluated by Cintra and Llanos in [CL03, CL05]. The so called *aggressive* sliding window that proceeds to retrieve a new chunk of iterations whenever the least speculative thread commits their results and the conservative sliding window that reloads only when all speculative threads on the window commit

their results. The *aggressive* sliding window was found to be superior to the *conservative* one, however, at the cost of higher implementation complexity. `MiniTLS` and `Lector` employ a *conservative* sliding window for work scheduling [YRHBL13].

3.6 Summary

This chapter discussed the advances in runtime parallelization with focus on thread-level speculation. The discussion was broken down to sections reflecting the following design choices on implementing a TLS system: execution model, metadata, version management, conflict detection, commit/rollback, and work scheduling for speculative threads. Each of these sections described how different work in the literature implements a particular design dimension. Table 3.2 summarizes the design choices that each work follows. `MiniTLS` and `Lector` (indicated in the highlighted rows), which is work published by the author of this thesis, are discussed in detail in Chapters 4 and 5, respectively. To the best of the thesis author's knowledge `MiniTLS` is the first software TLS system to combine an eager version management TLS system with parallel rollback phase and `Lector` is the first TLS system that combines the inspector/executor model with speculation.

WORK	EXECUTION MODEL	VERSIONING	CONFLICTS	COMMIT	ROLLBACK	SCHEDULING
DOALL test [RP94a]	Inspector/Executor	N/A	N/A	N/A	N/A	Static
LRPD test [RP95]	Speculative	Lazy	Lazy	Serial	Parallel	Static/Dynamic
Gupta and Nim [GN98]	Speculative	Lazy	Lazy	Serial	Parallel	Static /Dynamic
Rundberg and Stenström [RS01]	Speculative	Lazy	Eager	Parallel	Parallel	Static/Dynamic
R-LRPD [DYR02]	Speculative	Lazy	Lazy	Serial	Parallel	Sliding Window
Cintra and Llanos [CL03, CL05]	Speculative	Lazy	Eager	Serial	Parallel	Sliding Window
CorD [TFNG08, TFG10]	Speculative	Lazy	Lazy	Serial	Parallel	Static/Dynamic
SpLIP [OMH09]	Speculative	Eager	Eager	Parallel	Serial	Static/Dynamic
STMLite [MHHM09]	Speculative	Lazy	Lazy	Parallel	Parallel	Static/Dynamic
Raman *et al.* [RKM+10]	Speculative/DSWP	Lazy	Lazy	Parallel	Parallel	Static/Dynamic
MiniTLS [YRHBL13]	Speculative	Eager	Eager	Parallel	Parallel	Sliding Window
Lector [YRHBL13]	Speculative/Inspector	Lazy	Eager	Serial	Parallel	Sliding Window

Table 3.2: Design choices for main work in the literature of speculative parallelization.

Chapter 4

MINITLS: In-Place Speculative Parallelization with Parallel Rollback

4.1 Introduction

The previous chapters introduced speculative parallelization along with the latest advances in the area. Generally speaking, previous work in TLS has focused mainly on improving the execution runtime of an application.

This chapter is dedicated to a novel speculative parallelization mechanism that apart from performance improvements, also targets minimizing the memory footprint used by the supporting TLS data structures (the metadata).

A major bottleneck of software over hardware TLS systems is registering speculative state, since this requires maintaining speculative information in supporting data structures and ensuring exclusive access from different threads on them. Speculative data must be registered in such a way that when required the retrieval is as fast as possible. To allow speedy access, sufficient auxiliary data must be saved for a given speculative access (*e.g.* thread id, type of access, version number). These requirements also introduce additional demands for the data structures to be as memory efficient as possible, especially for applications with large number of speculative accesses.

The following sections present `MiniTLS`, a light, compact and memory-efficient software TLS system, that reduces space overhead as well as showing performance improvements over state-of-the-art software TLS systems. The memory-efficient side of `MiniTLS` is concerned with the overheads associated with maintaining speculative state as the main barrier for adopting TLS. `MiniTLS` relies on eager memory data management; *i.e.* speculative threads modify directly data which needs to be rollback when

misspeculation occurs. On the other hand, this eager treatment usually provides faster execution in the absence of data dependencies. The performance side of `MiniTLS` is concerned with optimizing the rollback phase (which is more time consuming for eager version systems) and it is the first TLS system with eager version management to implement a parallel rollback phase.

4.2 MINITLS: System Description

`MiniTLS` is based on eager version management and it is implemented, entirely using the Java programming language, as a library.

4.2.1 General Concept

In compiler optimization and runtime parallelization, loops are the main target for parallelization as most of the parallelism is usually found there [ALSU07]. Although `MiniTLS` can be used for any type of parallelization (*e.g.* methods, basic blocks, instructions) the discussion will focus on loop parallelization. Loop iterations run in parallel while threads are monitored during execution for potential violations. Every memory location is protected by a lock-bit (implemented using CAS operations) and as such only one thread at a time is allowed to have access to it. Threads update memory locations in-place and therefore any load performed by others is guaranteed to have the most up-to-date value written there. Before a thread updates a memory location, the original value is saved in a log (write-set), in case it is required by a rollback in the future. Marking and monitoring is facilitated by a shadow data structure (explained in the next section). When a data dependency violation occurs, memory must be recovered to the latest known correct state by restoring the polluted state using the logs from the threads involved in the violation. When memory is restored, parallel execution restarts and continues until all loop iterations complete.

4.2.2 Metadata

Shadow Data Structure

There is a shared data structure, termed a *shadow data structure* (see Figure 4.1), that maps every user-accessed address into an array of integers using a hash function. Figure 4.1 illustrates the case of sixteen running threads. This structure is an array

of Java integer values. In this case of sixteen threads, each address in the user data space is represented by two consecutive 32-bit integer memory locations in the shadow array. The first location (named "mark") is used to mark the thread(s) that performed a *load* and/or a *store* on that user address, and the second (named "owner") is used to indicate the thread(s) that are currently operating in the user address. The thread that currently performs a read or a write has to set the appropriate bit in "owner" in order to claim exclusive ownership to the location. Besides the action to be performed, the appropriate bit in "mark" is set before ownership is released.

For a thread T that needs to access memory location x, the order of operations is as follows: First, T checks that x is available by issuing a CAS operation on $h = hash(x)$. T indicates that is the owner of x by setting the appropriate bit in 'owner'. Then, T operates on h (load or store) accordingly by setting the appropriate bit in 'mark' to indicate the action performed. Finally, T resets the bit in 'owner' and releases the lock so another thread can access h if needed. Note that, the bit location in the bit sequence acts as thread id and indicates the order of speculation. That is, a less significant bit indicates a less speculative thread.

The synchronization mechanism is totally flexible to the designer's choice. Figure 4.1 illustrates how the "owner" could be used for read-write locks which allow multiple readers but only one writer per user address at a time. In this implementation for the sake of simplicity a simpler approach was taken so the same lock is used for readers as well as for writers and thus requiring only half the space of the "owner". The underlying locking mechanisms are bounded spin-locks. A thread may busy-wait (a finite number of times) for another to finish any work in the same location without blocking.

Assuming an 8-thread configuration, each location in the shadow structure requires 24 bits (although extended to 32 bits to avoid misalignments) to keep the owner-(8-bits)-reader-(8-bits)-writer-(8-bits) information. After the hash function determines a location in the shadow structure, the appropriate bits must be examined and updated. If thread 4 requires to read location x, then $hash(x)$ will be accessed in the shadow structure, and the information will be read and updated using a CAS operation. Assuming the contents of $hash(x)$ are empty 00000000_{hex}, thread 4 will check no other thread is operating there; *i.e.* the owner part is empty (*i.e.* 0000_{hex}). Thus the CAS operation will succeed and set the owner part to 0100_{hex} as well as the reader part to 0100_{hex}. This will leave the contents of $hash(x)$ being $0100-0100-0000-0000_{hex}$.

So, in essence, information for all the readers and writers of a particular user address, can be stored from as little as 6-bits for 2 threads to 96-bits for 32 threads. This solution represents a more compact way of memory representation compared to other solutions in the literature [CL03, CL05, OMH09], as Chapter 6 (experimental results) explains later. The choice to experiment with only up to 32 threads and not more, was motivated by the results in previous TLS limit studies (such as [ISK+10]) which indicate there is no significant benefit with a higher number of threads.

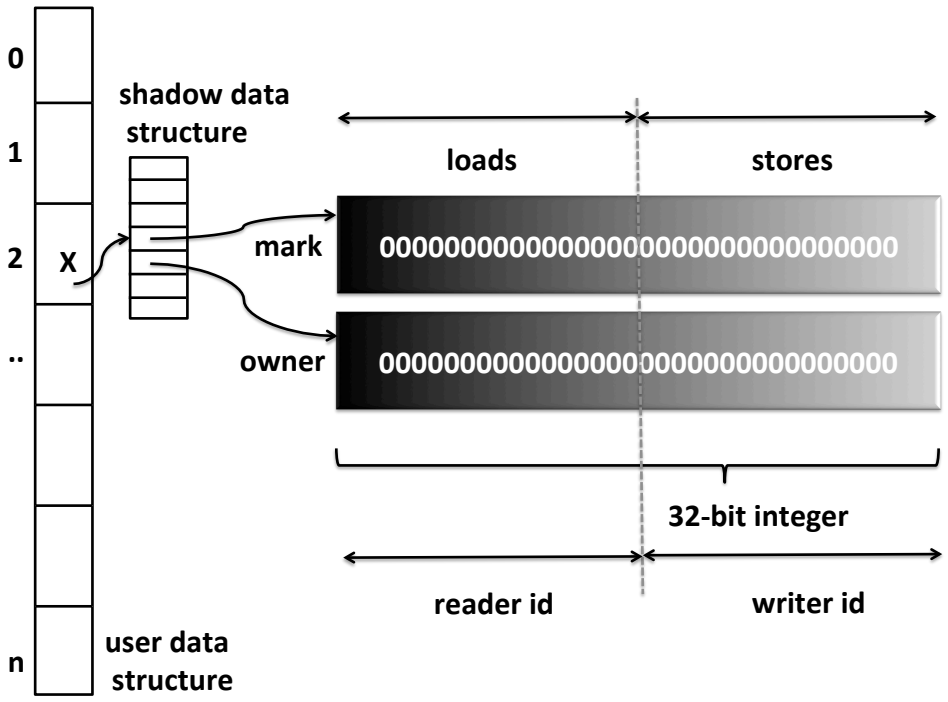

Figure 4.1: Shadow data structure in `MiniTLS`.

Data Structures Private to Each Thread

In addition to the shadow data structure, each thread has a local read- and write-set implemented using array structures. The read set consists of a set of indexes accessed in the shadow array. This is used when the thread finishes, in order to reset all its accesses in the shadow array and avoid any potential false conflicts in later accesses. The write set, apart from the indexes accessed, also keeps a record of the values that are updated in main memory. Since, this is an implementation of an in-place (eager) system, the memory values need to be recorded before being modified by a speculative thread.

The write-set is also called an *undo log* in this context, since in case of a conflict it is used to undo all the speculative operations. The only other in-place implementation [OMH09] in the software TLS literature requires a time-stamp, for each memory location, to be recorded in the undo log. This provides a sense of order between multiple writer threads in case of a rollback. When multiple threads have accessed a location that has to be reverted, only the value of the thread with the earliest timestamp is used so that the program's sequential order is maintained. Unlike [OMH09], `MiniTLS` does not need this extra time-stamp. Since the compact shadow structure contains already all the writer threads of a given location, the one with the lowest ID (lowest bit) can be used to revert the value.

4.2.3 Speculative Operations

In TLS, threads are normally organized in terms of speculation order. For instance, in loop-level speculation, where threads execute different loop iterations in parallel, the thread that executes the first chunk of iterations is known as the *least speculative thread*. A thread is always more speculative than the thread that executes the previous set of iterations. Consequently the thread that executes the last set of iterations is the *most speculative thread*. Such an order is useful in TLS, as it facilitates commit and rollback decisions in order to preserve the program's sequential order. Less speculative threads have the right to "kill" more speculative threads.

Loads

A speculative load operation by a thread T to a location x simply needs to set its corresponding bit in the shadow array. The bit is not set again if x was already accessed by the same speculative slice. Before this is done, the thread needs to check whether a more speculative thread has performed a store operation to x, as this can cause a *Write-After-Read (WAR)* conflict. The code is shown in Figure 4.2.

Stores

A store operation needs to check whether a more speculative thread has performed a store to the same location to prevent *Write-After-Write (WAW)* conflicts. Similarly it also needs to check whether a more speculative thread has performed a load to the same location. This causes a *Read-After-Write (RAW)* conflict. If none of those conflicts occurs, then the thread can safely record the current memory value in its write set and

perform an in-place update to the to-be-modified location (code not shown). Similarly to loads, the store bit is not set if it was already set earlier by the same thread. The code is shown in Figure 4.3.

```
specLoad(int hash, int threadID){

   // lock by setting "owner" bit
   // if cannot lock → throws Exception
   shadowStruct.lock(hash);

   int loads = shadowStruct.getLoads(hash);
   int stores = shadowStruct.getStores(hash);

   // (stores == 0) → no other previous (thread) stores
   // (stores <= threads) → no more speculative stores
   if( (stores == 0) || (stores <= threadID) ){

      Value v = memory.loadValue(hash);
      thread.setCurrentLoad(v);
      shadowStruct.setLoads(hash, threadID);
      thread.recordLoad(hash, threadID); //in read set
      shadowStruct.unlock(hash);
      return;

   }else { // (store > threadID) → a more speculative
           //                       store found
      thread.squash(); // WAR
   }
}
```

Figure 4.2: Speculative load in `MiniTLS`.

4.2.4 Conflict Detection

`MiniTLS` implements eager conflict detection, that is, conflicts are detected as soon as they occur. Another option would be to wait until the end of a speculative thread's execution before any conflict is detected. However, previous work identified that checking for conflicts on every speculative operation is less costly, compared to the wasted execution when the detection is delayed [CL03, CL05]. Furthermore, the `MiniTLS` system employs immediate conflict resolution. That is, a thread that detects a conflict pauses instantly any speculative execution, and initiates a rollback by notifying all the more speculative threads than itself, to squash. Since each speculative thread checks for conflicts on every speculative access, the action will take place immediately. All more

```
specStore(int hash, int threadID, Value v){

   // lock by setting "owner" bit
   // if cannot lock → throws Exception
   shadowStruct.lock(hash);

   int loads = shadowStruct.getLoads(hash);
   int stores = shadowStruct.getStores(hash);

   if( (stores == 0) || (stores <= threadID) ){

      if( (loads == 0) || (loads <= threadID) ){

         Value old_v = memory.loadValue(hash);
         memory.writeBack(hash, v);
         shadowStruct.setStores(hash, threadID);
         thread.recordStore(hash, threadID, old_v);
         shadowStruct.unlock(hash);
         return;

      }else{ //more speculative load found
         thread.squash(); // RAW
      }
   }else{ //more speculative store found
        thread.squash(); // WAW
   }
}
```

Figure 4.3: Speculative store in MiniTLS.

speculative threads than the offending thread (including the offending one), will wait until the rollback process begins. As soon as all the less speculative (than the offending one) threads finish execution and the offending thread becomes the least speculative thread, rollback is ready to begin.

4.2.5 Scheduling Policy and Ordering

For scheduling the speculative threads we have used a *sliding window* (explained in Section 2.3.2 of Chapter 2) mechanism [DYR02]. In a sliding window policy, the number of active threads depends on the size of the window. There are two reasons why this mechanism was chosen. First, due to the nature of the policy (chunks of iterations are scheduled in windows), the probability for data dependency violation is decreased and load imbalance is reduced. Second, it was found to be beneficial scheduling choice in previous experiments [DYR02, CL03, CL05]. MiniTLS uses a

conservative sliding window implementation where the window is reloaded when all threads currently occupying the window have completed.

The mapping of iterations to threads within a window is done by allocating contiguous sets of iterations of equal size. The mapping allows for the thread id to inform of how speculative each thread is; thread T_1 is always less speculative than T_3 and so on. `MiniTLS` performs a static block scheduling of the iteration space within a speculation window. Once all the threads within a window have completed, the shadow data structure is reinitialized and a new mapping of the following iterations is performed. Figure 4.4 shows how the sliding window with 4 threads works for a loop of 16 iterations.

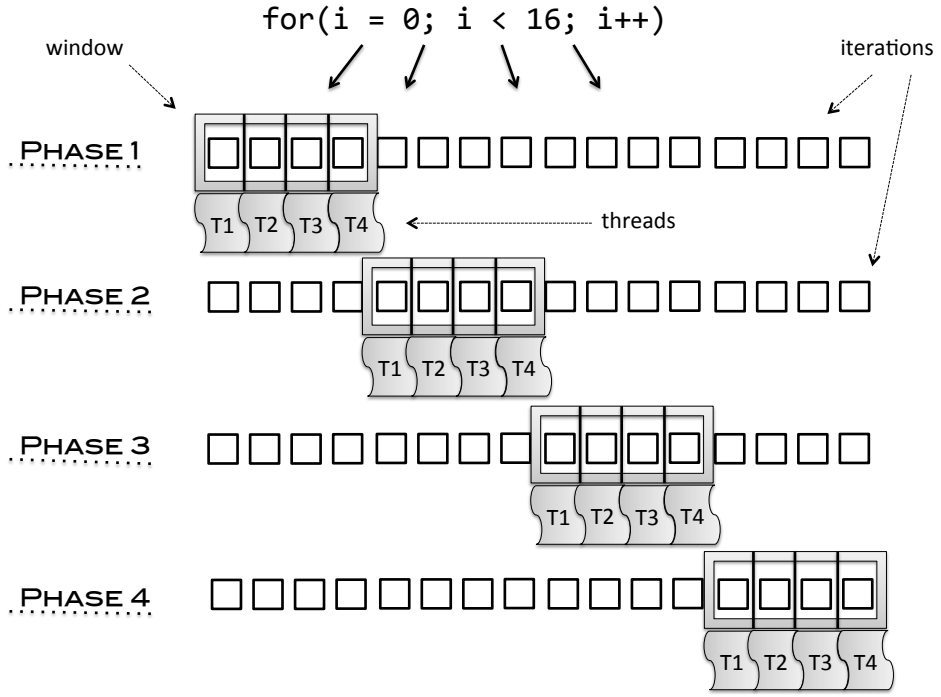

Figure 4.4: Four-thread sliding window scheduling policy for 16 iterations.

4.2.6 Rollback and Recovery

`MiniTLS` is the first software TLS eager management system to propose and implement a parallel rollback operation. This can help in reducing the overhead when misspeculation occurs. Those speculative threads that need to be squashed will take part in the rollback operation while those less speculative threads for which no data dependency has been found will wait for the rollback operation to finish. The rollback mechanism is started by `thread.squash()` in Figures 4.2 and 4.3. First we need to identify which

is the least speculative thread that modified each location. We can do this in parallel by allowing each participating speculative thread to visit its write-set data structure and for each element in the write-set check in the shadow data structure whether any other thread has modified it. To access the shadow data structure threads use CAS operations. Should more than one thread have written a given location, the least speculative thread to have modified it is identified in the following way: If the speculative thread is the first one to have modified it, the thread can go ahead and restore the value for that memory location. If the speculative thread is not the first one to modify it and aliasing on that location ($hash(x)$) is possible, the speculative thread has to check whether its memory location x is actually contained within the write-set of the less speculative threads denoted in the shadow data structure for $hash(x)$. If it is not found in these less speculative write-sets, that speculative thread will restore the value for memory location x.

Once the memory state has been rolled back, the participating speculative thread can reset the pertinent memory locations in the shadow data structure in parallel.

Performing the rollback procedure efficiently is an important issue and has also formed a concern in the past for the database community [BN97]. In log-based databases, a log is maintained by the system to record information regarding memory accesses by all executing transactions. In case of abort, the log is scanned starting from the last record moving backwards to restore the original values back to memory. Searching this log sequentially is inefficient. To avoid this sequential scan a data structure called *transaction descriptor* is used to describe each transaction's updates. Starting from the descriptor there is a pointer to the last record updated by a given transaction T. The last record updated by T contains a pointer to the penultimate record updated by T and so on. This allows transactions to proceed with rollback in parallel with one another by starting from the transaction descriptor and following the pointers.

4.2.7 Speculative Thread Lifecycle

Figure 4.5 is a state diagram illustrating the lifecycle of a speculative thread. A thread can be in one of the following states:

1. **FREE**: Thread is ready to get the next chunk of iterations and start work.
2. **RUN**: Thread has started speculative execution.
3. **WAIT**: Thread has finished execution. Note that since updates are in-place, the thread has actually committed its results (parallel commit). The thread now must

wait until it becomes the least-speculative thread. While waiting it can still be squashed by a less speculative thread.

4. **COMMITTING**: Thread is now the least-speculative thread and starts clearing its local data structures. At this stage the thread cannot be squashed since it has already finished execution and there are no less speculative threads.
5. **COMMITTED**: Thread has cleared its local data structures and indicates that is ready to become a speculative thread again.
6. **FAILED**: Thread has been involved in a data dependency violation.
7. **SQUASHING**: Thread starts rollback in parallel with any other offending threads.
8. **SQUASHED**: Thread has finished rollback. It is now waiting to be restarted.

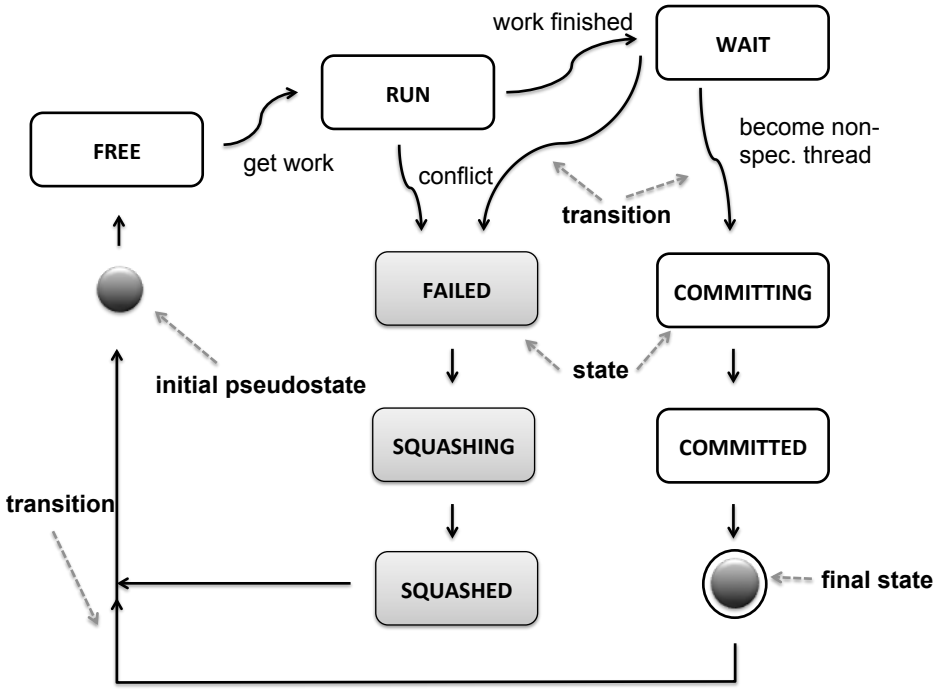

Figure 4.5: Speculative thread lifecycle in `MiniTLS`.

4.3 Summary

This chapter presented a software TLS system, `MiniTLS`, which is based on eager version management. `MiniTLS` uses a novel compact data structure for speculative marking in order to minimize space requirements. This compact data structure enables `MiniTLS` to perform the rollback procedure in parallel and thus accelerating the entire

parallelization process. `MiniTLS` is the first TLS system with eager version management to implement a parallel rollback phase. Later, Chapter 6 presents experimental results for speedup and memory usage of `MiniTLS` against state-of-the-art work.

Chapter 5

Accelerating Speculative Runtime Parallelization using Inspector Threads

5.1 Introduction

One of the main goals of parallelization is speedup of the application. This has been the main aim of Thread-level Speculation (TLS) over the last years. A major bottleneck of software over hardware TLS systems is registering speculative state, since this requires maintaining speculative information in supporting data structures and ensuring exclusive access from different threads on them. The following sections describe a software TLS runtime system, written purely using the Java programming language, aiming to relax the requirement of speculative state when possible.

Previous TLS limit studies (presented in Chapter 1) observe that on future multicore systems it is likely to have more cores idle than those which traditional TLS would be able to harness. Since idle cores will be available, the question is whether "helper" tasks can be created to determine whether speculation is actually needed without stopping or damaging the speculative execution. In `Lector`, for each conventional TLS thread running speculatively with lazy version management, there is associated with it a lightweight *inspector*. The inspector threads execute alongside to verify quickly whether data dependencies will occur. If inspector threads decide that the loop contains no data dependencies, then speculation can be switched to regular parallel execution and thus eliminating any overheads associated with marking and conflict detection.

5.2 LECTOR: System Description

There are two main components involved in the system described in this chapter: A core TLS system, and the extensions for accelerating the overall parallelization. First, the proposed TLS system is introduced and then the extensions required for acceleration are explained.

5.2.1 General Concept

The runtime speculative software system proposed in this chapter follows lazy version management and eager conflict detection. Parallel execution proceeds monitored by the TLS system. Performing a load or store, requires a thread to first acquire exclusive ownership of the desired memory location. Speculative stores are buffered locally only after ensuring that none of the more speculative threads has already loaded that location. Otherwise the more speculative threads are squashed due to RAW dependence violation. Speculative loads search the local buffer (write-set) first, before loading from memory, in case that value has already been written locally from an earlier store. This provides the illusion of memory consistency. If not found there, then the value has to be loaded from main memory. In case an earlier thread has written on that location, the current thread can either forward the most recent value, wait for the earlier thread to write-back its write-set to memory and then load, or squash. Buffered values are written-back to main memory after a thread has been proven successful.

5.2.2 Metadata

The proposed TLS system utilizes a shadow data structure as shown in Figure 5.1. The figure presents the case of 32 speculative threads. Every user memory location is mapped into this shadow table using a hash function. Each mapping in the shadow table is composed of three consecutive 32-bit integer memory locations as shown in the picture: one to represent the owner thread, one to indicate whether a thread performed a load, and one to indicate a store by a thread. A bit in any of this three locations reflects the identity of the thread that is accessing the original user's location for the appropriate action. The next section will provide more details on how that works in practice.

In addition to the shadow table, there is a local read- and write-set for each thread.

The read-set is a list with all locations read by that particular thread, whereas the write-set is a hash map with all the location/value pairs to be written to main memory when that thread commits.

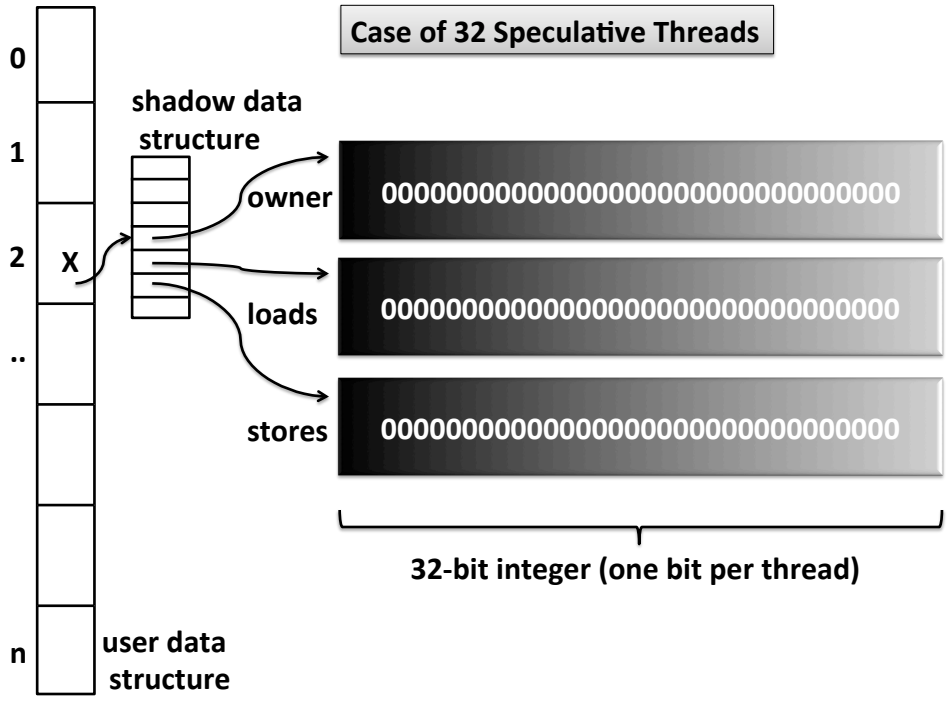

Figure 5.1: Shadow data structure of `Lector`.

5.2.3 Speculative Operations

Speculative Stores

Whenever a speculative store takes place (see Figure 5.2), the thread must successfully acquire exclusive ownership of the location to be written in order to proceed. This is done by accessing the shadow table for that particular location (using a CAS operation) and setting the appropriate bit in the "owner" to indicate that this thread is operating on this location. If the thread finds the location occupied, it spins a bounded number of times before initiating a squash. When the thread acquires exclusive ownership, it checks "loads" to see whether a more speculative thread has already loaded from that location. Such a load is called an *exposed load* in this context and requires all the more speculative threads to be squashed. There is no need to take any action if a less speculative thread has loaded that value or if there was a store by a different

less speculative thread since this is a lazy version management system. Following the check for violations, the thread releases the lock and inserts the value in its write-set.

```
01. specStore(int hash, int threadID, Value v){

02.   if(shouldSquash()){thread.squash();}

03.   shadowStructure.lock(hash, threadID);

      //may throw exception and squash() here
04.   shadowStructure.checkForViolations();

05.   if( isNonSpeculativeThread(threadId) ){
06.      writeBackToMemory(v);
07.   }else{

08.      shadowStructure.markStore();
09.      writeSet.put(hash, v);

10.   }
11.   shadowStructure.unlock(hash, threadID);
12. }
```

Figure 5.2: Speculative store in `Lector`.

Speculative Loads

Since stores are buffered, a speculative load (Figure 5.3) will first check the thread-local write-set in case the value has already been written earlier by the same speculative thread. If this is true, the load will return the latest buffered value. There is no need for such a load to consult the shadow table since, it is guaranteed that the loaded value was produced by the correct store. If the value is not present in the write-set, then the thread acquires exclusive ownership of the location in the same manner as in speculative stores, checks for violations, and loads the value from main memory. There is no need to worry about different threads loading the same value as no conflict can arise from that. Also, threads from the future (more speculative) that have written to that location are harmless for the moment. The reason is because a more speculative thread that has produced a store in that location, will buffer the value and write it back to memory when the time is appropriate. Then is when the system will worry about conflicts because it could be the case that the earlier thread was squashed. The only situation

that can cause a problem is when a less speculative thread has produced a store for that location but not yet committed. That means that the current thread requires that value in order to proceed but that value is in earlier thread's buffer and not yet in memory. One solution (that is feasible due to the novel representation) is to identify, using the "stores" in the shadow table, the latest thread that has produced a store in that location (but not yet committed) and *forward* the correct value [RS01]. This requires careful handling in case the write-set of the thread trying to forward from, is updated simultaneously. Another solution is to wait for the less speculative to write-back its results and load the value from memory [GN98]. A simpler solution, which is the one implemented in this case, is to squash the current thread and restart its execution. If no violation is present the ownership is released, the value is loaded from main memory as normally, and the hash value from the shadow table is recorded in the read-set. The read-set is used when the thread commits in order to release the marking in the shadow table.

```
01. specLoad(int hash, int threadID){

02.   if(shouldSquash()){thread.squash();}

03.   if( writeSet.contains(hash) ){
04.      return writeSet.get(hash);
05.   }else{

06.      shadowStructure.lock(hash, threadID);

         //may throw exception and squash() here
07.      shadowStructure.checkForViolations();
08.      shadowStructure.markLoad(hash, threadID);
09.      Value v = memory.loadValue();
10.      readSet.add(hash);
11.      shadowStructure.unlock(hash, threadID);
12.      return v;
13.
14.   }
15. }
```

Figure 5.3: Speculative load in `Lector`.

Scheduling and Commits

The proposed system follows a type of *scheduling window* (described in Section 2.3.2 from Chapter 2) for scheduling the threads. The window size is always the same as the number of threads; therefore, for n threads the window size will be $n-1$. Thread in slot 0 is the non-speculative thread and similarly thread $n-1$ is the most speculative thread. Although thread 0 is allowed to write its results immediately back to memory (*i.e.* no buffering), it still checks the shadow table in order to eagerly squash more speculative threads that have exposed loaded from a location the non-speculative thread updates. Threads in the window commit their results back to memory in ascending order of speculation by locking the location in the shadow table, propagating the appropriate values, and clearing the shadow table from their markings. After all threads in the window have committed, speculation restarts with all threads getting work from a work-queue.

5.2.4 Speculative Thread Lifecycle

Figure 5.4 is a state diagram illustrating the lifecycle of a speculative thread. A thread can be in one of the following states:

1. **FREE**: Thread is ready to get the next chunk of iterations and start work.

2. **RUN**: Thread has started speculative execution. Note that it can be switched to squashed, if it was involved in a data dependence violation.

3. **WAIT**: Thread has finished execution. Thread now must wait until it becomes the non-speculative thread in order to commit its results to memory. While waiting it can still be squashed by a less speculative thread.

4. **COMMITTING**: Thread is now the non-speculative thread and starts clearing its local data structures and propagating the buffered values to memory. At this stage the thread cannot be squashed since it has already finished execution and there are no less speculative threads. However, it can initiate a squash for a more speculative thread that has performed an exposed load form a location this thread is writing.

5. **COMMITTED**: Thread has cleared its local data structures and finished any memory updates. It indicates that is ready to become a speculative thread again.

6. **SQUASHED**: Thread has been involved in a data dependence violation. It must clear any marking in the shadow table and wait to be restarted.

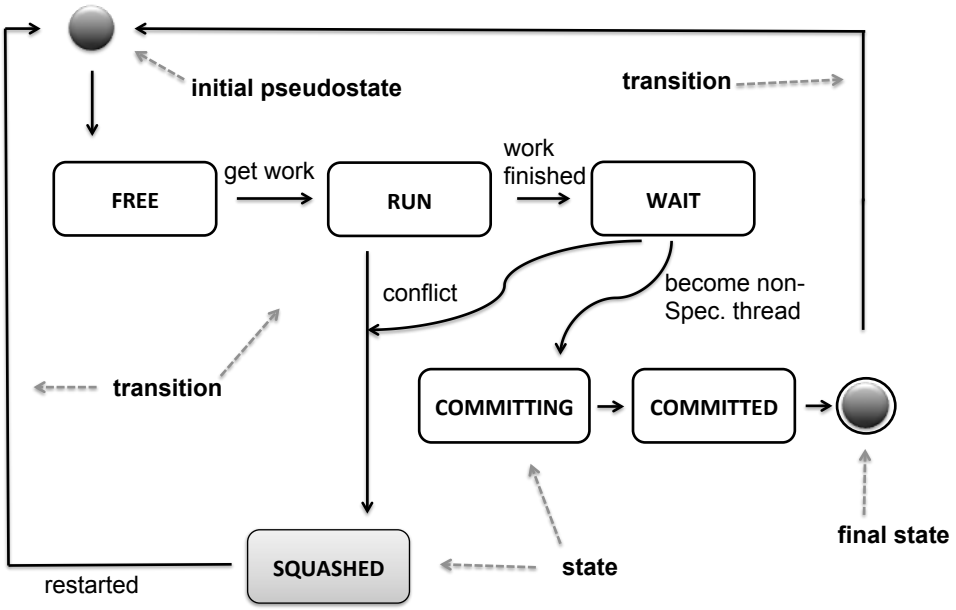

Figure 5.4: Speculative thread lifecycle in `Lector`.

5.2.5 Inspector Threads

A novel technique was investigated in this work which combines the advantages of two popular models involved in the beginning of the TLS research, namely *Inspector/Executor* model and `LPD` *(Lazy Privatizing DOALL) Test* described in [RP94a].

Inspector/Executor model

Using the *Inspector/Executor* model (described in Section 2.2.1 from Chapter 2), a simpler version of the loop under question is extracted and executed to verify whether the loop carries any data dependencies. The simpler version does not produce any side-effects as well as does not require all the code from the original loop (just the memory accesses). Therefore the inspector is expected to execute faster than the original sequential loop. If proven safe, then the executor may execute the loop in parallel. Inspectors are created by analyzing memory accesses, collecting information about their iteration number and access type (read/write) in a separate data structure, and then checked for data dependencies [SBW91]. The drawback of this model is that, in

cases where the loop cannot be stripped-down sufficiently, the inspector might end up taking the same time as the original loop.

LPD Test

The `LPD` test was known as being the heart of `LRPD` [RP95] test and `R-LRPD` [DYR02] test (both described in Section 3.3 from Chapter 3). The `LPD` test checks the loop under question for data dependencies or whether dependencies can be eliminated when privatization is used (*i.e.* buffered updates). The test flags whether dependencies exist or not, while the loop executes in parallel buffering any updates to memory. If dependencies are found, the loop discards any buffered updates and executes sequentially, otherwise the loop has been parallelized correctly using privatization. `LPD` has an advantage over the *Inspector-Executor* model, especially when the inspector cannot strip-down the loop to an adequate level. A successful parallelization of `LPD` in that case simply needs to update main memory, whereas the other model has to still run the executor. Still though, under both models, in case the loop was not found parallel, all execution and time spent while the test was running, will be wasted.

LPD meets *Inspector/Executor*

In this work the two ideas were combined in a way that their drawbacks are eliminated. A a stripped-down version of the loop to be parallelized is extracted (see Figure 5.5), like in the case of inspector/executor; however, there is no executor as such. The inspector performs the `LPD` test but since it does not replicate the program's entire code, it is expected to run faster.

The inspector threads start running ahead, as soon as the application begins. At the same time the loop is executed using the speculative parallelization runtime environment described earlier in Section 5.2. Once the inspector phase completes (end of "phase 1" in Figure 5.6), its results dictate how the speculation will continue ("phase 2" in Figure 5.6). When the inspector finds the loop to be *DOALL*, speculation is dropped, any buffered results are propagated to memory, and the loop continues to execute in parallel without any speculative overhead. The inspector terminates as soon as it finds a data dependency, allowing the execution to proceed speculatively. Moreover, even in the unfortunate case that the inspector finishes at the same time as the speculative execution, there is no need to execute the loop sequentially since it has been already executed speculatively. The same applies in case the loop contained any data dependencies. Using this scheme, *DOALL* loops that can express a "light" inspector, may

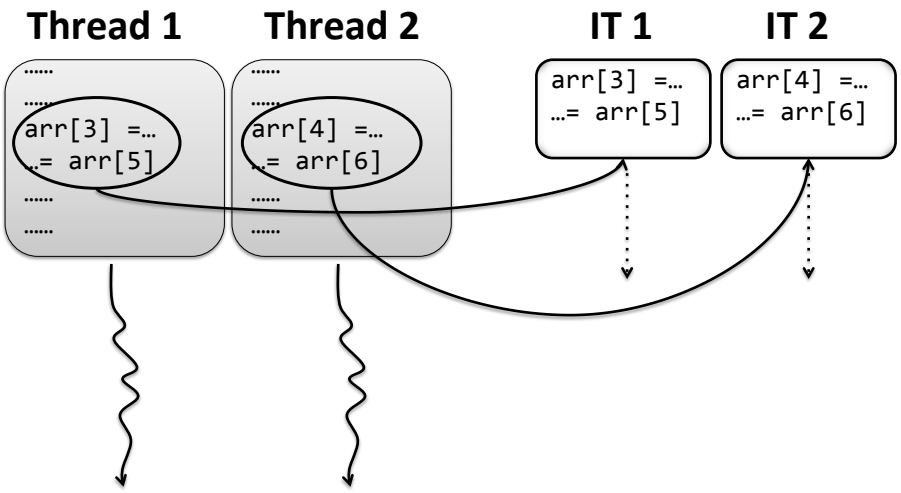

Figure 5.5: Inspector threads are created by replicating the memory accesses from speculative threads. "IT" stands for Inspector Thread.

benefit significantly from the absence of speculation-associated overhead.

The system described in this work takes advantage of the benefit from the Inspector-Executor model which is discovering whether a loop is *DOALL* or not, quickly. At the same time, its drawback of having wasted work is eliminated in case the loop was not *DOALL* since the underlying TLS model executes alongside with the inspector. The same applies in case the inspector was as "heavy" as the loop itself. For the same reasons any shortcomings related with the `LPD` test are also eliminated.

The inspector threads in `Lector` are created following the `LPD` test [RP94a].

5.3 Summary

A major bottleneck of software over hardware TLS systems is registering speculative state, since this requires maintaining speculative information in supporting data structures and ensuring exclusive access from different threads on them. This chapter presented a software TLS runtime system, written purely using the Java programming language, that is able to relax the requirement of speculative state when possible. An extended version of a lazy version management TLS for that purpose is explained.

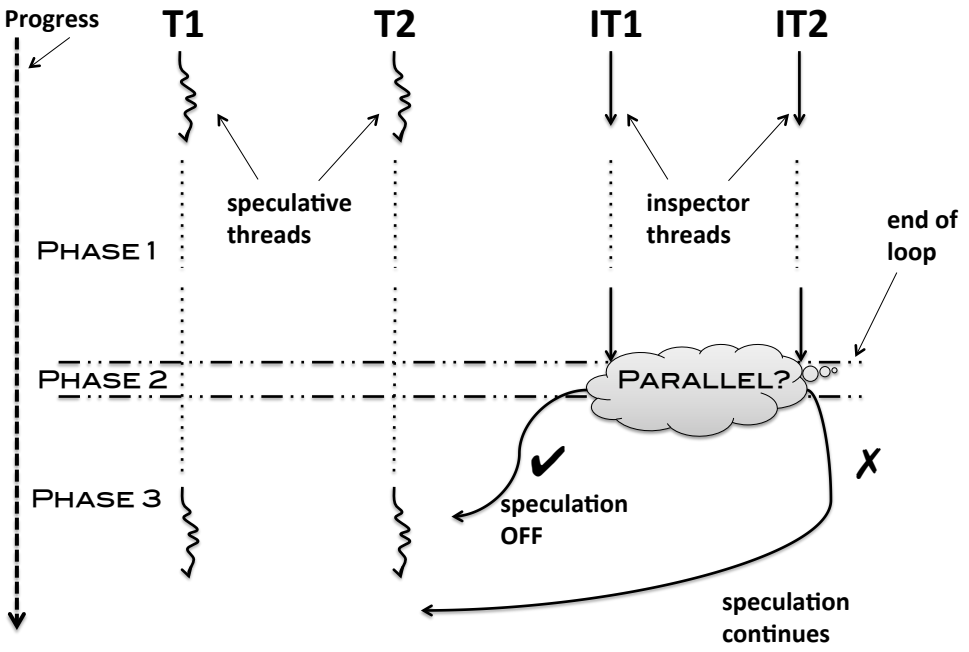

Figure 5.6: Phase 1: Lightweight inspector threads (IT) execute concurrently with TLS threads. Phase 2: ITs complete execution earlier than TLS threads and test if the loop is *DOALL*. Phase 3: Depending on the outcome of phase 2, either speculation continues as normal, or speculation is turned off (*i.e.* non-speculative parallel execution).

Lightweight inspector threads run ahead of the TLS execution with the purpose of discovering early the potential of removing any excess speculation. Should the Inspector threads uncover a DOALL loop, speculation can be relaxed. Otherwise speculation continues as if nothing happened. As it is shown later in Chapter 6, this technique is able to increase speedups over normal speculative approaches.

Chapter 6

Evaluation and Results

6.1 Introduction

This chapter provides the results obtained by applying `MiniTLS` (Chapter 4) and `Lector` (Chapter 5) against standard sequential applications used for performance comparisons in the area of speculative parallelization. These comparisons are provided in terms of performance improvements over the sequential applications when the proposed TLS systems are applied to them. Furthermore, the two systems, `MiniTLS` and `Lector`, are compared against state-of-the-art work in terms of performance.

6.2 Evaluation Methodology

6.2.1 Hardware Platform

For all the experiments a `UltraSPARC T2` system also known as `Niagara 2` was used. It has 8 processors, each of which has 8 hardware threads, making it able to process up to 64 threads simultaneously. Furthermore, it has 4-MB shared L2 cache. The Solaris®10 OS was installed on the machine. Solaris uses a "maximum dispersal" thread scheduling to assign threads to their initial processors. The kernel selects the least loaded core when placing a thread, in order to avoid resource contention among concurrently running threads. This is the default OS thread affinity and running a Java application it is not possible to change it using the standard libraries or JVM settings. `UltraSPARC T2` was selected because the cores on that chip are simpler (processors are not speculative, no out-of-order execution, pipelines are less deep, frequency is lower) compared to other chip-multiprocessor architectures such as `Intel`®`Zeon`,

AMD®Opteron and IBM®Power7. This allows UltraSPARC T2 to favor more the multithreaded performance (compared to its single thread execution) over its competitors. Nevertheless, the other architectures due to their richer instruction-level parallelism (from more complex processor design) might be able to offer better performance on very small thread configurations (2 to 4 threads) where scalability is not such a big issue in TLS. In fact, as the experiments reveal later on in this chapter, UltraSPARC T2, seems to offer suboptimal performance (even though still scalable) for 2 to 4 thread configurations (compared to higher thread configurations).

Java™SDK version 1.6.0 and Java™HotSpot VM were used with fixed 4GB maximum heap size for all executions and the System.nanoTime() timer provided by the JVM. All the results presented are the average of ten executions for each thread number for each benchmark and also the standard deviation was checked.

6.2.2 Benchmark Applications

The TLS systems proposed in this thesis are designed to be used on applications for which a (static) parallelizing compiler could not evaluate with certainty the data dependence relations of the loops under question. Having that in mind, the selected applications for the experiments originate mainly from two benchmark suits: SPECjvm2008 and JOlden. Following the methodology used by SpLIP [OMH09], applications that parallelizing their *loop-kernels* improves the total application run time were also chosen. SpLIP has been reimplemented in Java to provide a direct comparison with MiniTLS. Applications with irregular accesses (not parallelizable by static compilers) were also selected, thus making them good candidates for TLS. The following benchmarks were selected from SPECjvm2008 (large data sets used): (i) Sparse, a matrix multiplication algorithm, (ii) SOR, that simulates Jacobi Successive Over-relaxation, and (iii) Monte-Carlo, which approximates the value of Pi. From JOlden the following benchmarks were selected: (i) Barnes Hut, an implementation of the Barnes-Hut force calculation algorithm (using input *C*), (ii) Em3d, a simulation of electro-magnetic waves traveling through objects in three dimensions, and (iii) Perimeter, an implementation of Samet's algorithm for computing perimeters of regions in a binary image. Finally LeeRouter [WKL07] was selected, a circuit routing application using Lee's algorithm. For LeeRouter, the *mainboard* input dataset was used. Most of these applications are also used in previous TLS studies [QnMS+05, OMH09, TFG10, ISK+10].

As the focus of this thesis is on optimizing the software runtime system of TLS

and not on how to transform a loop into its parallel equivalent by a compiler, the applications were transformed manually into parallel speculative versions. Two benchmarks exhibit dependencies (`LeeRouter` and `Em3d`), and five have no runtime dependencies. Loop-induction variables were eliminated as they introduce false dependencies under TLS. The most time consuming loops were considered for TLS parallelization. Finding these automatically is an interesting optimization problem in itself [QnMS+05, LTC+06] (for task selection) and [JEV04, ORSA05b] (for finding suitable tasks).

6.2.3 Java Virtual Machine Implications

Implementing a software system in `Java`™ as opposed to a language that offers native execution such as `C++` can have both its advantages and disadvantages:

Compilation

The `C++` compiler translates and optimizes the application ahead-of-time before the program executes. `Java`™ code is initially compiled to a generic platform-independent representation called *bytecode*. This allows portability since any platform that has a Java™ Virtual Machine (JVM) installed can read the bytecode and produce the appropriate binaries for that specific platform. In most cases, as in `HotSpot VM`, the JVM initially performs *Just-In-Time (JIT)* compilation. That is, while the program executes, a method is compiled only when it is encountered for the first time. During the program's execution, the JVM collects statistics to identify "hot methods" (most frequently executed methods) as potential candidates for more aggressive optimizations (such as aggressive virtual method inlining). On the one hand the compilation route taken by the JVM at start-up can be slower than a statically compiled language (which has all the code compiled already before execution starts). On the other hand, run-time compilation can potentially take advantage of platform-specific information on which the program is being executed to improve code more effectively.

Garbage Collection

`Java`™ takes advantage of automatic memory management in order to relief the programmer from the burden of allocating and deallocating objects manually and the risk of causing further heap fragmentation in case of forgotten objects. However, this adds

an extra cost on performance, since when the *Garbage Collector (GC)* is triggered to free the memory from unused objects, the running application has to pause execution.

Thread Synchronization

Since Java™ 6, the JVM includes an optimization known as *biased locking*. Basically, an object that is found to be locked by only one thread during its lifetime is allowed to remove the locking constraints towards that thread. In other words that thread can subsequently lock and unlock the object without resorting to expensive atomic instructions. This is of course another benefit from run-time compilation and its achieved through a technique known as *escape analysis* [Bla99].

MINITLS

6.3 MINITLS - Experiments

6.3.1 Baseline for Experiments: SPLIP

`MiniTLS` is compared against a state-of-art TLS library, `SpLIP` [OMH09]. The reason `SpLIP` was chosen is because it also relies on eager memory version management and provides the best published performance results. Thus, `SpLIP` sets an optimized baseline against which to compare `MiniTLS`.

Although, `MiniTLS` and `SpLIP` use an eager version management technique, their implementations are fundamentally different. `SpLIP` employs two data structures for handling the iterations that load and store for each speculative access. Two additional data structures are used for imposing order to accesses to each location between reading and writing threads. Yet another data structure is required for storing a timestamp for a particular location in case of a rollback. The rollback procedure also differs from `MiniTLS`. `SpLIP` aggregates the write-sets of all speculative threads involved in the violation by comparing timestamps in case of multiple thread access the same location. This requires a hash map in order to be able to check whether a location was already written by a different thread. In contrast, `MiniTLS` requires no aggregation, and each thread involved in the violation proceeds in parallel with each other for rollback, without using timestamp comparisons. Since all the information is kept in the main shadow structure and there is no need to check the write-set entries once they have been recorded, the write-set can be implemented as an array structure. Further details about the `SpLIP` implementation can be found in Appendix A.

6.3.2 Performance Results

Figure 6.1 presents speedup results against the original sequential unmodified version, obtained from applying `MiniTLS` to the benchmark applications. The *y* axis indicates speedup, whereas the *x* axis shows the benchmarks used. The sequential application's speedup is marked by speedup = 1 in the *y* axis, thus whatever is higher than that, shows improvement. On average, speedups start to be observed when the system goes over 4 threads. The reason is due to the speculative overhead added by TLS. The parallelism starts amortizing the cost on average when more than 4, sometimes 8 threads are used. The cost introduced by speculative operations is high and thus for a small number of threads it drowns the benefits of parallelism.

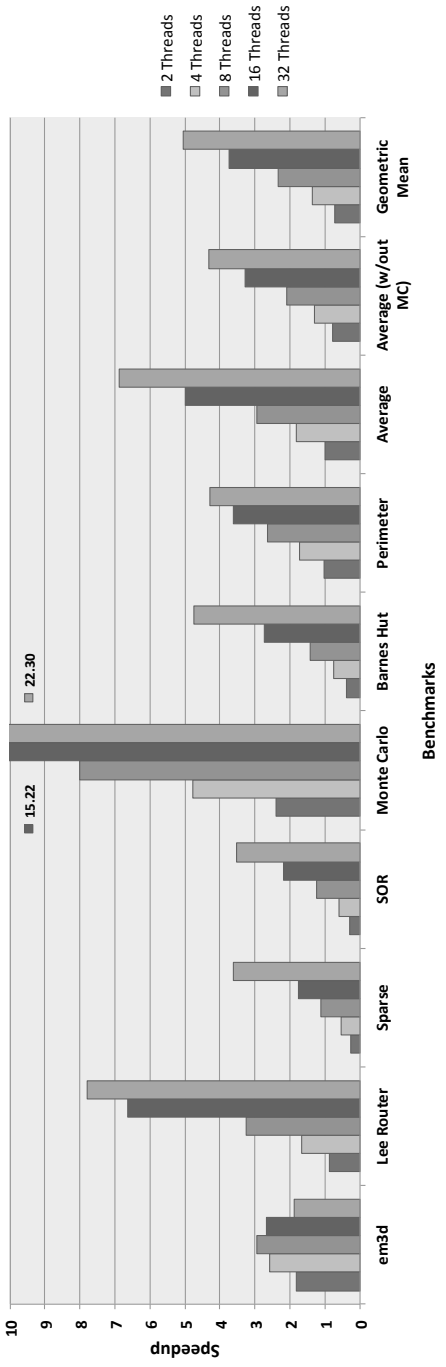

Figure 6.1: Speedup results for `MiniTLS`. Sequential execution is denoted by 1 in the y axis.

Figure 6.2 shows an example of the overhead introduced to support `MiniTLS` in terms of execution time for the `Sparse` application. The *y* axis is intersected where the sequential time is (*i.e.* the baseline). When the overhead bars grow below the *x* axis this

implies overhead less than the sequential time, and thus the application starts showing speedups (after 8 threads). With 4 threads TLS is 1.8x faster than the sequential application, for 8 threads nearly 3x faster, 16 threads 5x faster and 32 threads almost 7x faster. In order to be objective, the average speedup excluding the `Monte-Carlo` benchmark, which does not carry any data dependencies, was also included. `Em3d` shows a decline in speedup after 8 threads. The benchmark carries a large number of data dependencies which causes more frequent rollbacks when more that 8 threads are used.

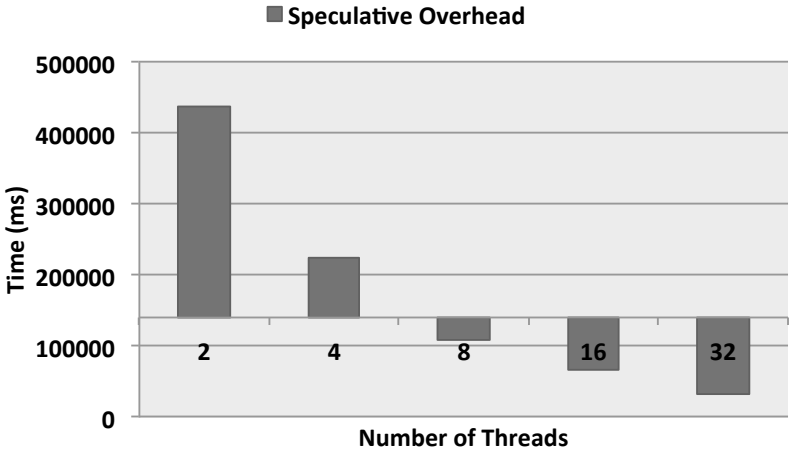

Figure 6.2: Time spent on speculation for `Sparse`. The *y* axis is intersected at the sequential time.

Figure 6.3 shows speedup comparisons between `MiniTLS` and `SpLIP`. As before, the *y* axis shows speedup against the sequential unmodified application and the *x* axis indicates the number of threads used. The same pattern is again observed in which speedups are observed after 4 or 8 threads (for the reasons explained earlier). `MiniTLS` outperforms `SpLIP` 1.33x, on average: 1.1x for 2 threads, 1.2x for 4 threads, 1.3x for 8 threads, 1.4x for 16 threads and 32 threads. The main reasons for this performance difference are analyzed in the next section where the execution overhead is presented.

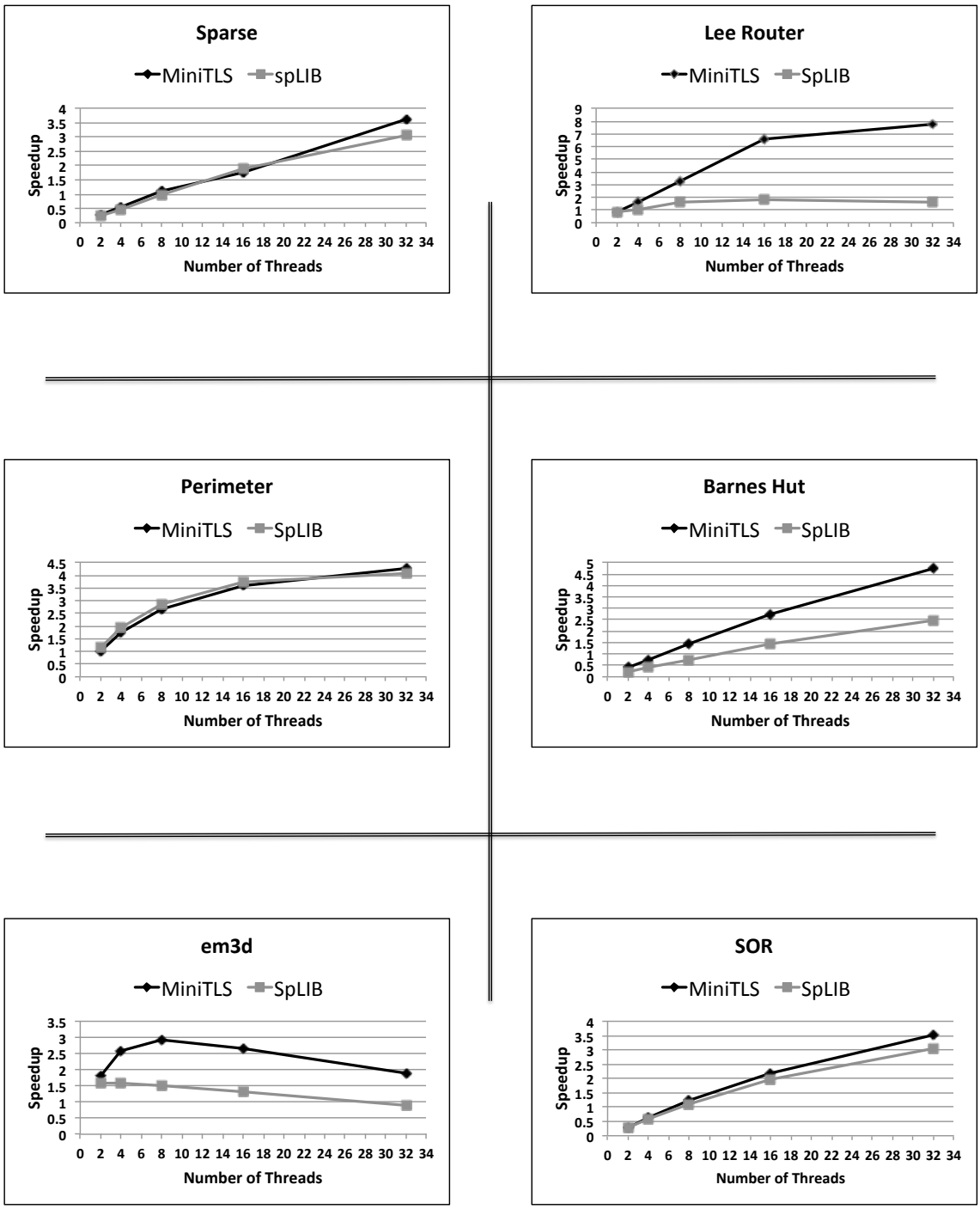

Figure 6.3: Speedup comparison of `MiniTLS` and `SpLIP`.

6.3.3 Speculative Overhead Comparison

Figure 6.3 shows how `MiniTLS` outperforms `SpLIP` in terms of speedup. The main reason why speculative systems suffer speedup losses is due to speculative overheads, for example, marking speculative loads/stores and time spent during rollback. `MiniTLS` shows performance improvements over its competitor by reducing those overheads. Figure 6.4 presents the reduction percentage of `MiniTLS` against `SpLIP` for speculative operations. The graph has two parts: The first part shows how much `MiniTLS` reduces speculative marking over `SpLIP` (`SpLIP` is the normalized baseline in all cases). The second part shows how much `MiniTLS` reduces rollback time over `SpLIP` (`SpLIP` is the normalized baseline in all cases). Both parts are independent. That is, Rollback percentage has nothing to do with the Marking percentage. For example, looking at the marking section for `Em3d` for 32 threads one can say that `MiniTLS` spends around 30% less time for marking compared to `SpLIP` (where 100% marking is the total amount of time for `SpLIP` to perform marking). `LeeRouter` and `Em3d` are shown since they are the ones that carry data dependencies and thus could see how much time is saved from rollback. Among the non-carrying dependency benchmarks the average was chosen as a representative of all benchmarks (except `Monte-Carlo`) since they all feature similar execution patterns. The graph clearly shows the effectiveness of the parallel rollback routine. This is due to two reasons: (i) `SpLIP` uses extra buffers for conservative synchronization before and after the same location is accessed by multiple threads. Due to the absence of locks the possibility of those buffers to cause a data-race is very high when a particular location is accessed by multiple threads concurrently. (ii) When a violation occurs using `SpLIP`, the write-sets of the involved threads must aggregate their values, comparing their timestamps, before write-back occurs. In `MiniTLS`, this process is accelerated since the involved threads can proceed in parallel for write-back and without the cost of the timestamps. Marking for non-carrying dependency benchmarks is very similar in both systems. The absence of rollbacks allows both systems to spend approximately the same amount of effort to book-keep their information. However, for the dependency-carrying benchmarks the case is different. Rollbacks cause a lot of code re-execution and thus more marking involved. Since `MiniTLS` performs less rollback, it benefits from having less code to re-mark. `Barnes-Hut` is the only application that does not follow the same pattern with other similar benchmarks especially after 8 threads. This application requires very minimal marking. When `MiniTLS` performs marking, it requires very little space compared to the other library as it is explained in the following section.

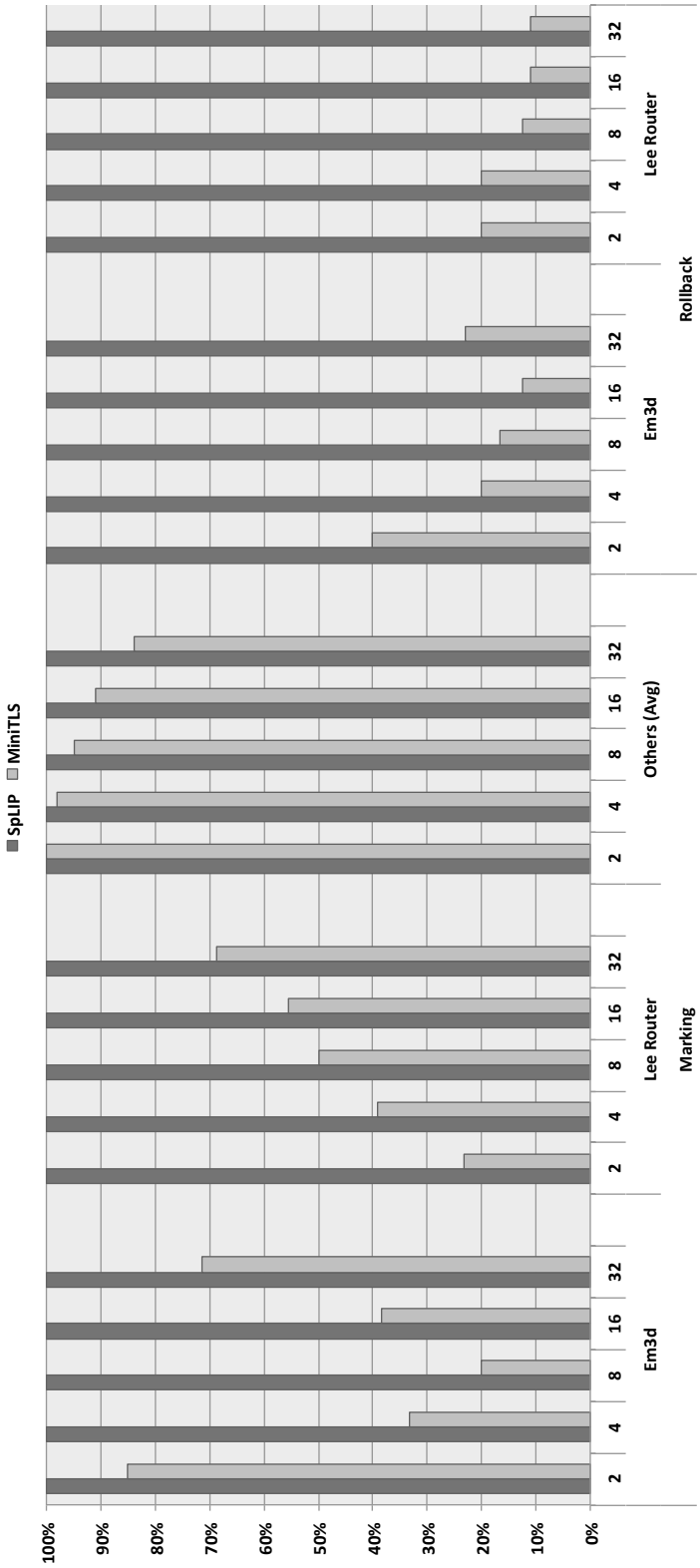

Figure 6.4: Shows the amount of overhead reduction of `MiniTLS` against `SpLIP`. The graph is normalized (baseline `SpLIP`). The first part shows reduction of speculative read/write marking. The second part reduction of rollback time.

6.3.4 Data Structures Space Comparison

To maintain the speculative state, additional space is normally required. This section compares the space overhead between the two systems: `MiniTLS` and `SpLIP`. The space referred to in this section, is the space in regard to any data structures required for version management. The space required by the undo log is not considered since there is negligible difference between the two approaches. The novel marking scheme in `MiniTLS` is designed in such a way as to require the minimal space based on the number of threads running (hence the name `MiniTLS`). A typical TLS system, unlike `MiniTLS`, will consume the same amount of speculative space regardless of the number of threads used. Figure 6.5a and Figure 6.5b illustrate the memory space required to facilitate speculative marking for eight threads for `SpLIP` and `MiniTLS` respectively. Figure 6.5a shows that for each user accessed memory location, `SpLIP` would require at most 160 bits in order to mark the iterations that will possibly perform a load or a store, the timestamp, as well as the thread ids for synchronizing loads and stores. Figure 6.5b shows that `MiniTLS` requires only 24 bits to perform the same operations as opposed to 160 bits that `SpLIP` requires.

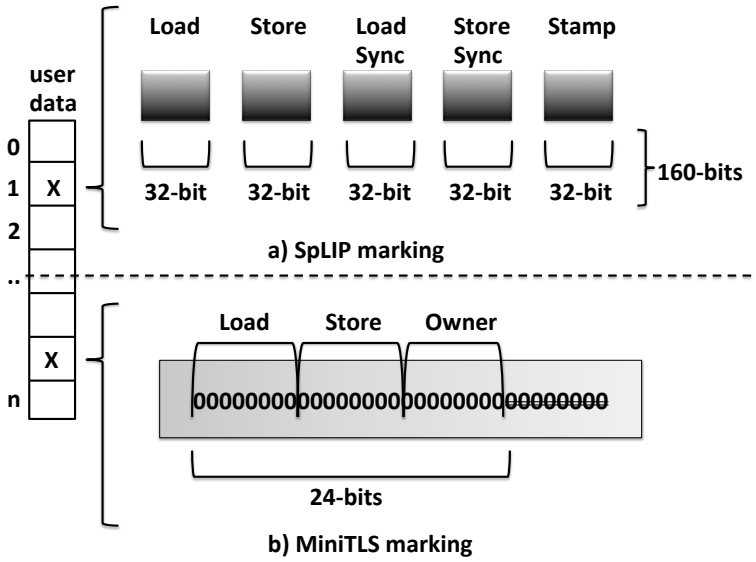

Figure 6.5: Space required for 8 speculative threads using a) `SpLIP` and b) `MiniTLS` .

Figure 6.6 shows the normalized (with `SpLIP` as baseline) Space Overhead comparison between `MiniTLS` and `SpLIP`. There is a significant space overhead reduction of 96% when 2 threads are in use, 92% reduction with 4 threads, 87% with 8 threads, 70% with 16 threads, and 40% with 32. In other words, `MiniTLS` requires on average

5x less space than `SpLIP`. This can have a great impact in performance, especially in automatic memory managed languages such as Java since there will be less garbage collection triggers than normally required and thus less interruptions of the user's application.

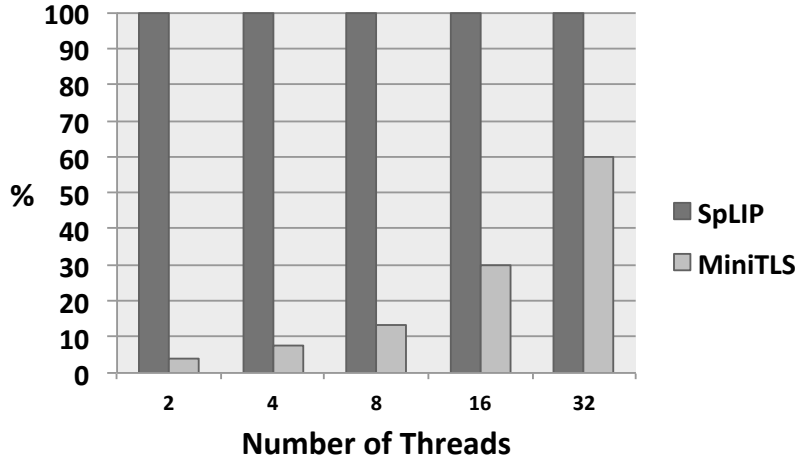

Figure 6.6: Normalized (baseline `SpLIP`) space overhead comparison between `MiniTLS` and `SpLIP` .

Memory Overhead

While TLS can decrease application execution time, more memory is required to support the additional data structures. Apart from the single shadow structure, each thread has its own copy of its read and write sets. Thus, an experiment (similar to [TFG10]) was conducted to measure the memory consumption of TLS. Two representative applications were selected for comparisons. The first one, `Sparse` could be considered as the worst case scenario as 90% of the total application accesses, are speculative. The second, `Lee Router`, is the average case application where about 50% of the total access are speculative. Figure 6.7 shows the results of comparing the additional memory required compared to the sequential application.

For the "worst case", the memory consumed is between 0.2x and 1.8x for `MiniTLS` and 2.9x for `SpLIP`, compared to the sequential version. For the "average case" it was between 0.02x and 0.4x for `MiniTLS` and 0.6x for `SpLIP`, compared to the sequential version. What is considered as memory overhead in this experiment the total amount of storage in order to support the shadow data structure compared to the sequential application. For baseline memory overhead, all the memory accesses that

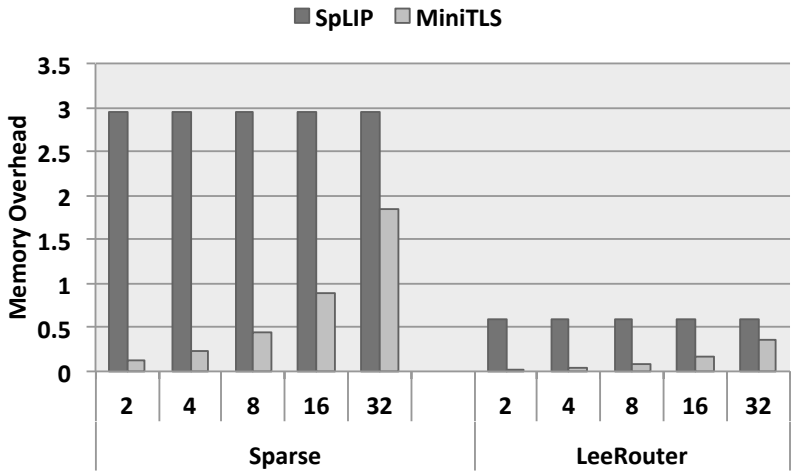

Figure 6.7: Memory overhead of `MiniTLS` and `SpLIP` compared to the sequential version.

the sequential application performs during the entire execution were taken into account. The storage required for the shadow structure based on the unique speculative accesses during the execution of the application (only the unique ones since only one instance of a read / write exists in the shadow structure) was also measured. along with how many bits are required for that information. `SpLIP` is constant across threads since the amount stored per memory location is always the same. For `MiniTLS` the storage requirements grow with the number of threads as more bits are necessary to support marking on those threads. Although not shown in the graph, there is also the extra overhead for maintaining local read/write sets. Storing these local per thread data structures requires 50% extra storage of the sequential application's memory requirements in these two benchmarks. This number is dependent on the proportion of data accessed during TLS execution, which in these benchmarks is fairly high. This is a constant overhead in TLS systems, however it can be significantly optimized by replacing read-set data structures with *bloom filters*.

LECTOR

6.4 LECTOR - Experiments

For the following experiments three systems were compared: `TL2TLS`, the baseline which is explained next, `LazyTLS` which is simply `Lector` with inspector threads disabled, and `Lector` with inspector threads enabled. When showing `Lector` for n number of threads that signifies n inspector threads and n TLS threads. Thus, for 2 threads, `Lector` uses 2 inspector threads and 2 TLS threads which are 4 threads in total.

6.4.1 Baseline TLS system: TL2TLS

In order to test the performance of `Lector`, a comparison against an established baseline used in [MHHM09] and [TFG10] is considered. The baseline is based on the state-of-the-art algorithm - Sun's Transactional Locking 2 [DSS06]. Similar to [MHHM09] and [TFG10], speculative code is added within transactions and explicit synchronization is added into transactional functions to enforce in-order commit, which is necessary to maintain sequential program semantics. The rest of this chapter refers to the baseline as `TL2TLS`. Further details about the baseline [DSS06] implementation can be found in Appendix A.

6.4.2 Performance Results

Figure 6.8 presents speedup results against the original sequential unmodified version, obtained from applying `Lector` to the benchmark applications. On average, `Lector` performs between 1.8x and 8.2x faster among multiple threads compared to the sequential application. The following paragraphs explain how `Lector`'s inspector threads help minimize the speculative overhead and allow for more useful computation to be performed.

Figure 6.9 shows speedup comparisons between `LazyTLS`, `Lector`, and `TL2TLS`. As before, the y axis shows speedup against sequential unmodified application and x axis indicates the number of threads used. For `em3d` and `Lee Router`, `LazyTLS` and `Lector` are actually identical (thus only one is shown) because the inspection phase ends shortly after the first data dependency is found which for both benchmarks is nearly at the beginning of execution. Speedup comparisons for `Monte-Carlo` are not shown since it does not carry any speculative activity and thus it has the same behavior under any system. The most interesting fact is the speedup benefits observed in certain benchmarks when using the inspector threads. When inspection is enabled, even the

minimal thread configuration produces higher speedup than running 32 threads with inspection disabled. For instance, in `Sparse`, `SOR`, and `Barnes Hut` it is more beneficial to run the benchmark with 2 speculative plus 2 inspector threads rather just with 32 speculative threads and no inspector threads. More details are explained in the following section where speculative overhead is discussed.

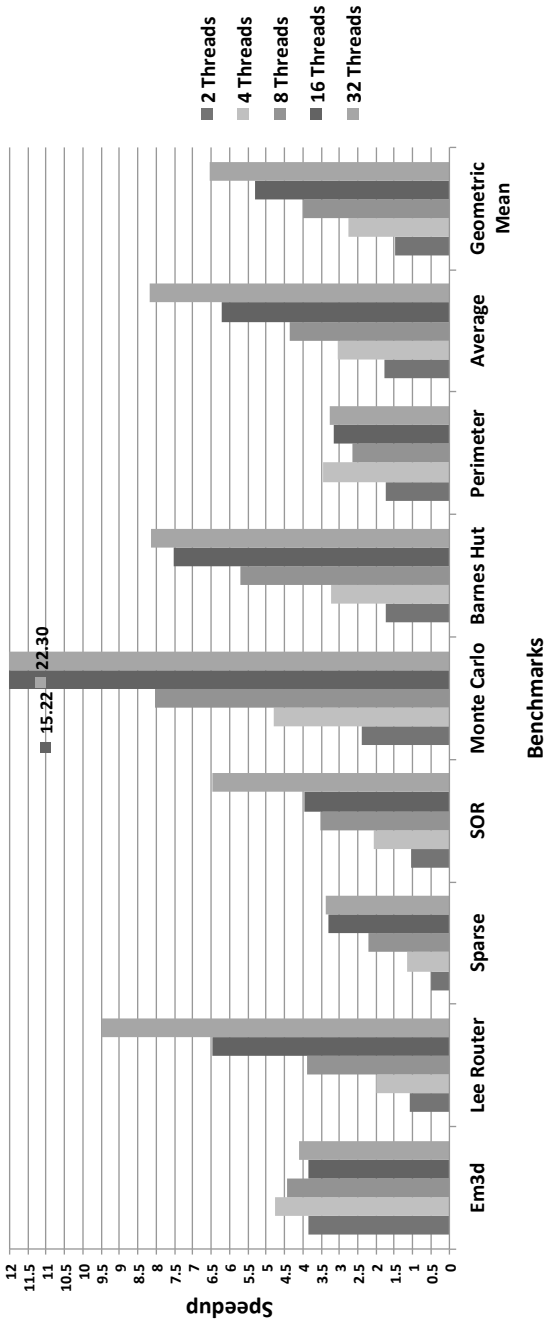

Figure 6.8: Speedup results for `Lector` against the sequential execution. Sequential execution is denoted by 1 in the y axis.

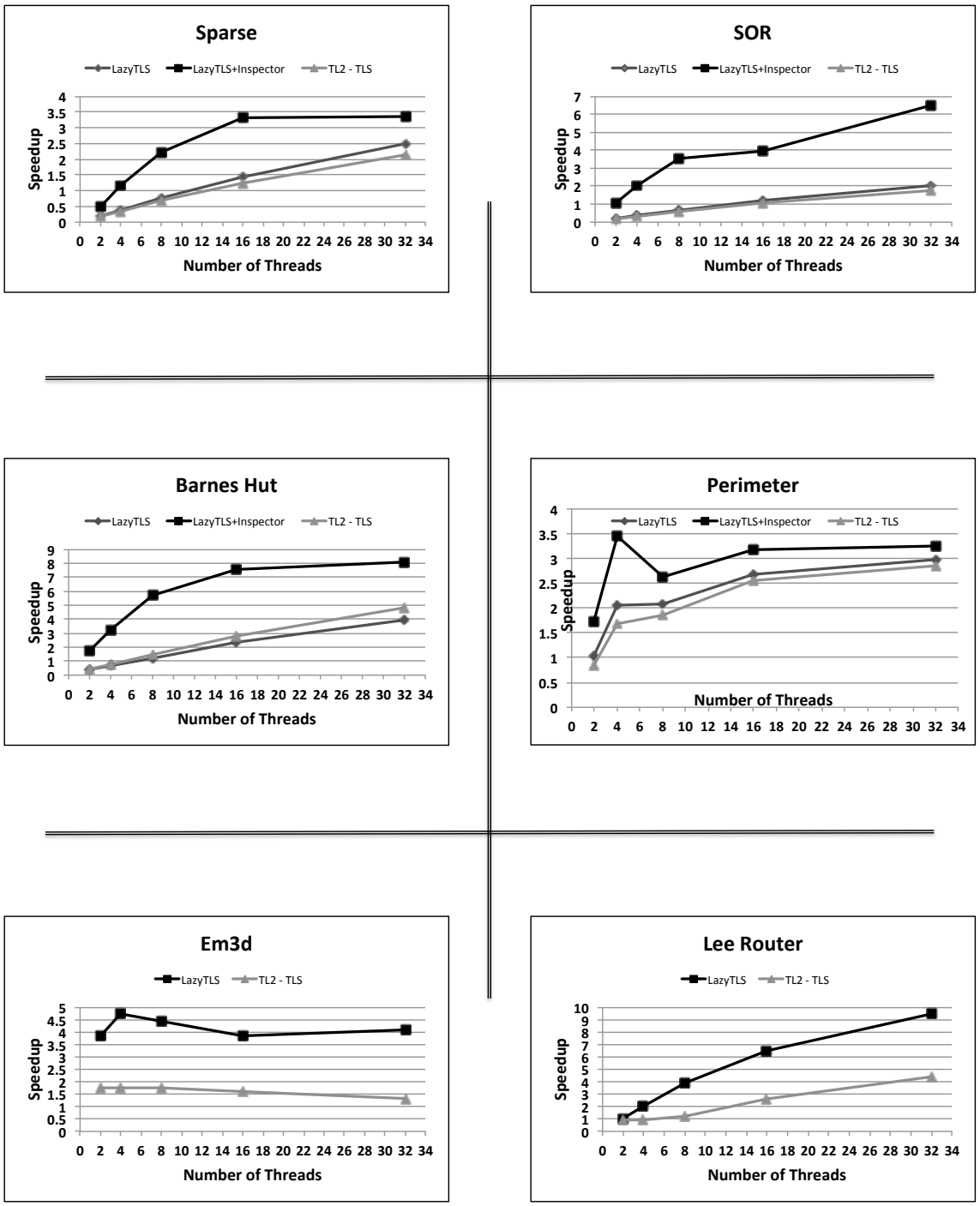

Figure 6.9: Speedup comparison between `LazyTLS`, `Lector`, and `TL2TLS`.

6.4.3 Speculative Overhead Comparison

Speculative overhead is maintaining the information in the shadow data structure as well as per thread TLS context support. `Lector` outperforms the other two configurations as it reduces that extra cost. Figure 6.10 shows the percentage of speculative overhead reduction between the three systems. The graph is normalized to `TL2TLS`. For a given amount of overhead that `TL2TLS` spends, the reduction for the other two systems is shown. For example, executing `Barnes-Hut` with 8 threads, `LazyTLS` spends about the same time in speculative overhead as `TL2TLS`. `Lector`, on the other hand, spends about 80% less time compared to `TL2TLS`. In most cases `TL2TLS` and `LazyTLS` are similar, however in the cases of `LeeRouter` and `Em3d` (which are the ones that carry data dependences) `Lector` is not only faster but also spends less time for speculative marking. This is mainly because `TL2TLS` performs lazy conflict detection (at commit time) which in case of many conflicts, produces wasted work and additional speculative marking. The idea of `TL2` is that a successful thread will change the version of a memory location at commit time notifying any thread that holds an out-of-date version of the same location, to be squashed. However, a thread with an out-of-date version will not discover that incident before its commit time. Therefore more execution and marking is done between the time a conflict occurs and the time of detection. In `LazyTLS` and `Lector` it is impossible for a thread to have an out-of-date version of a memory location and still keep executing. If a thread updates a given location it will "see" in the shadow structure that a different thread has performed an action there and thus squash that thread eagerly.

In all cases, `Lector` has the lowest cost compared to the other two systems. In the non-carrying dependency benchmarks, an adequate inspector version was extracted and managed to finish earlier than speculative execution. Abandoning speculation (*i.e.* speculative marking) with the aid of inspector threads, allows speedup increase since there is no more speculative work after that point.

`Lector` shows greater speedups as the number of threads increases. This is due to the advantages of the inspector threads during the marking phase having to check only their local speculative locations as opposed to the other two systems that use the same data structure for all thread configurations. `Perimeter` does not follow the same behavior due to the high number of speculative accesses in each inspector thread, causing the analysis phase to spend considerable time. `LeeRouter` and `Em3d` does not show any advantage using inspector threads since they carry data dependencies.

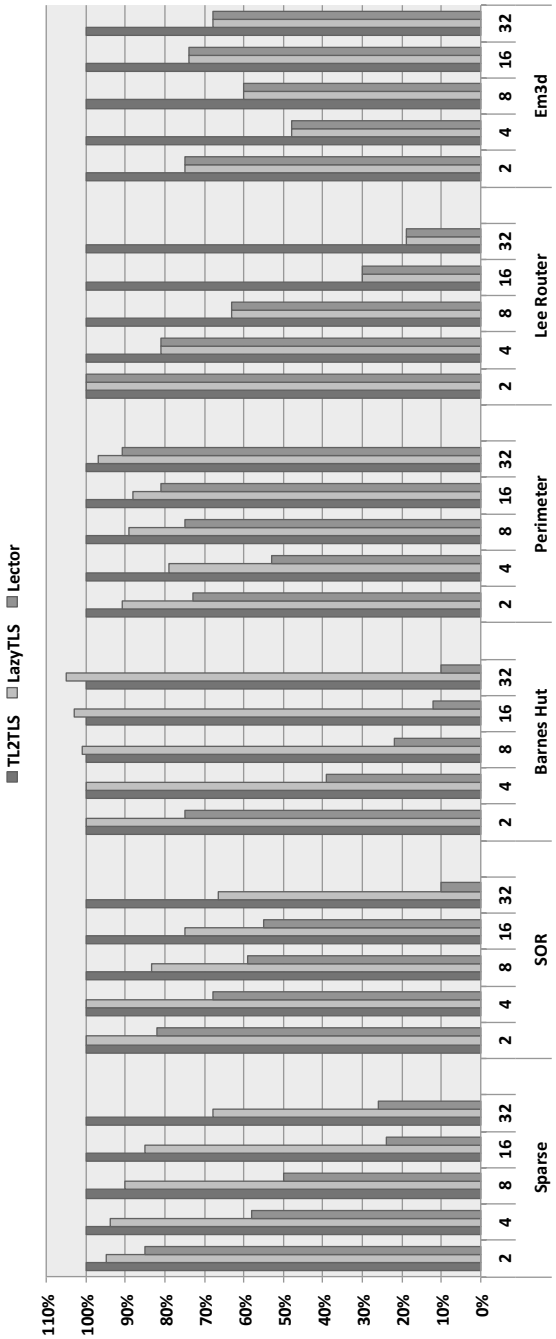

Figure 6.10: Normalized speculative overhead reduction with baseline the `TL2TLS` system.

MINITLS vs. LECTOR

6.5 MINITLS vs. LECTOR - Experiments

Finally, a speedup comparison between the two systems is presented: `MiniTLS` and `Lector`. As previously, speedups are shown over the unmodified sequential execution for each benchmark. Figure 6.11 clearly shows how `Lector` outperforms `MiniTLS` nearly in every case. Inspector threads are able to quickly identify a loop without data dependencies and notify `Lector` that speculation is no longer necessary. Thereafter the loop can execute in a fully-parallel non-speculative mode. In cases where data dependencies exist, inspector threads will not be of any additional benefit, however, they quit inspection as soon as conflicts are found to allow speculation to continue without any unnecessary overhead. Nevertheless, `Lector` is still faster in those cases. For the benchmarks experimented with, `Lector` still benefits from its lazy version management as it does not trigger as many data dependencies as an eager version management system (lazy can only trigger RAW dependencies). On average `Lector` performs approximately 2x faster than `MiniTLS` for 2 and 4 threads, 1.8x faster for 8 threads, and nearly 1.5x faster for 16 and 32 threads. `MiniTLS` appears to be faster than `Lector` on the `Perimeter` benchmark for 16 and 32 threads. The reason for this is the relatively high speculative overheads of `Lector` for `Perimeter` running with 16 and 32 threads, shown in Figures 6.9 and 6.10.

Inspector threads can only be applied to systems that buffer their memory updates. Such threads cannot be applied to eager version management systems such as `MiniTLS` as updates to main memory are performed in-place and the inspectors can load the wrong values and by extension addresses.

The results indicate that `Lector` outperforms `MiniTLS` in most cases. However, this comes with the cost introduced by the additional inspector threads in terms of memory consumption. Figure 6.12 shows the memory overhead of both systems for the `Sparse` benchmark against the sequential version. `Lector` requires 1.5x more memory than `MiniTLS`. That is, 2.7x compared to the sequential version. Nevertheless this is acceptable as it has still lower overhead compared to `SpLIP` (as shown in Figure 6.7) and it is faster.

Finally, moving a step back from TLS *per se*, it is interesting to see how speculative parallelization performance compares to manual or static parallelization. Figure 6.13 shows speedup comparison between parallelizing the `Sparse` benchmark speculatively and manually. There are two reasons why manual parallelization, and not from an automatic tool, was applied. First, no static parallelizing compiler is yet available for the Java programming language. Second, a static parallelizing compiler would

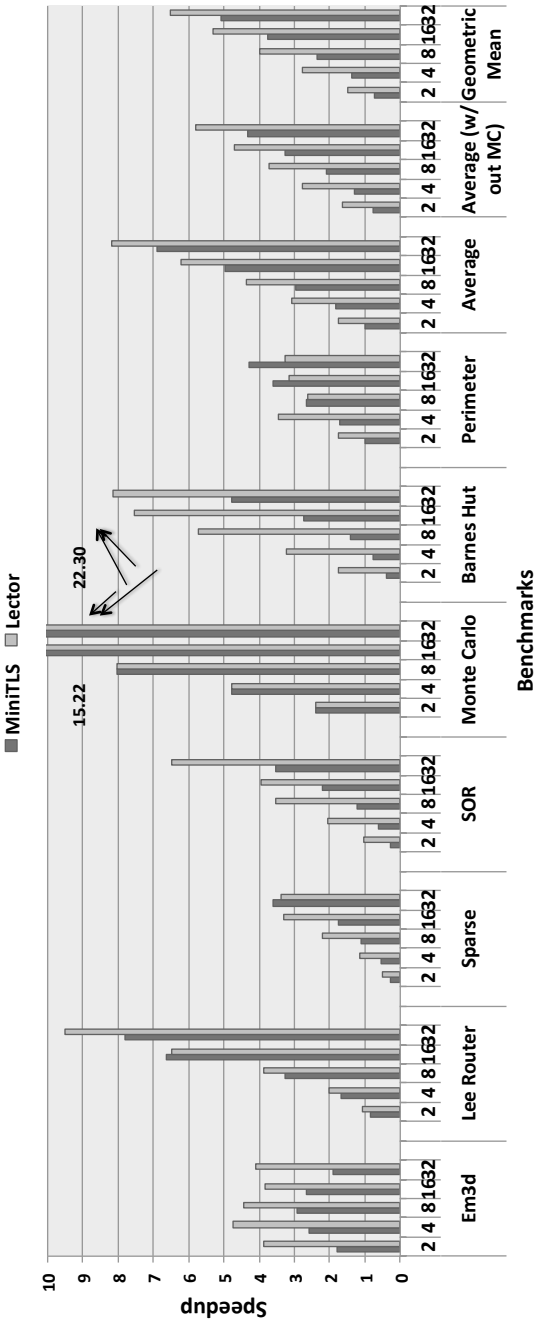

Figure 6.11: Speedup results for `Lector` vs. `MiniTLS` against the sequential execution. Sequential execution is denoted by 1 in the y axis.

not be able to parallelize the benchmark due to ambiguities from pointer aliasing and thus domain knowledge is required. It is clear that the performance difference is very large, especially for large number of threads. For example, using 8 threads speculative

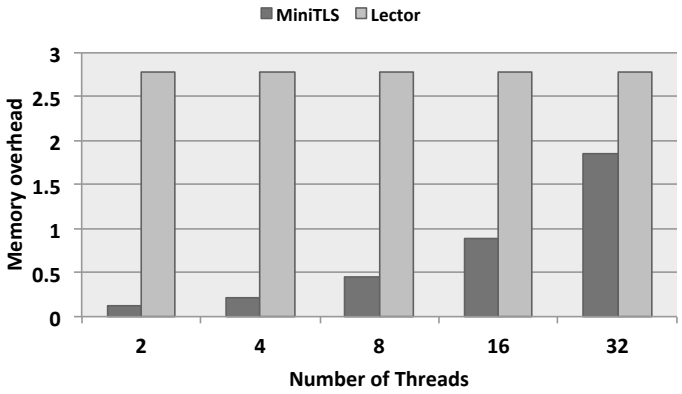

Figure 6.12: Memory overhead of `MiniTLS` and `Lector` compared to the sequential version of the `Sparse` benchmark.

parallelization is about 3x slower than the parallel equivalent but using 32 threads it is 8x slower. This indicates the impact on performance from only maintaining speculative state, since `Sparse` does not trigger any rollbacks.

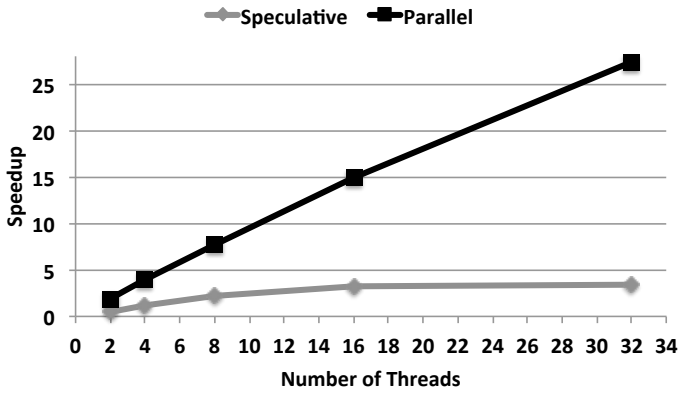

Figure 6.13: Speedup comparison between speculative and manual parallel execution for the `Sparse` benchmark (with the sequential version of `Sparse` used as baseline denoted by speedup == 1).

6.6 Summary

This chapter provided the results obtained by applying `MiniTLS` (Chapter 4) and `Lector` (Chapter 5) against standard sequential applications used for performance comparisons in the area of speculative parallelization. These comparisons were provided in terms of

performance improvements over the sequential applications when the proposed TLS systems are applied to them. Furthermore, the two systems, `MiniTLS` and `Lector`, were compared against state-of-the-art work in terms of performance.

`MiniTLS` and `Lector` were found to outperform their state-of-the-art comparison TLS systems in terms of application runtime performance. `MiniTLS` was also superior to the state-of-the-art in terms of memory requirements (*i.e.* more memory-efficient). `Lector` was also found to be superior than `MiniTLS` in a comparison against the two systems.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

Runtime parallelization is concerned with sequential applications that a static compiler cannot parallelize due to insufficient information at compile time. Speculative parallelization (or Thread-Level Speculation - TLS) is a form of runtime parallelization that transforms a sequential application to a semantically equivalent parallel one by speculating on potential data dependencies. For example, a loop is parallelized with the assumption that the iterations are independent. During execution, runtime checks ensure that any data dependencies that may arise will be handled appropriately so that parallelization continuous correctly. In order for those runtime checks to be able to discover data dependencies, information regarding memory accesses by iterations must be maintained by the runtime system. This is known as the speculative state.

Chapter 1 presented the importance of parallelization from the early days of supercomputing and vector processors until today's computer architecture market that is dominated by multicores. Details were also explained on the difficulty of producing parallel programs and the tradeoffs between different ways of parallelization. The focus of the thesis was in speculative parallelization. Chapter 2 introduced the area of runtime and speculative parallelization while briefly explaining the different parts that are required to assemble a TLS system in software. Chapter 3 discussed the different approaches followed by work in the literature in order to build and optimize a software TLS system. The discussion of Chapter 3 also subsumes the related work for this thesis.

A significant barrier for adopting software TLS is the overheads associated with maintaining speculative state. This thesis has investigated two solutions to optimize

speculative parallelization systems. Two techniques for version management (a way of maintaining speculative state) have been used extensively in the literature. Thus, for completeness two systems were proposed, one for each technique and in both cases were found to be better than the state-of-the-art.

Chapter 4 presented a software TLS system with a novel compact version management representation; `MiniTLS`. Facilitated by this representation, `MiniTLS` reduces the space overhead over state-of-the-art software TLS systems between 96% on 2 threads and 40% on 32 threads. `MiniTLS` relies on eager memory data management and, thus, when a misspeculation occurs a rollback (to restore memory) process is required. `MiniTLS` takes advantage of the novel compact version management representation to parallelize the rollback process and is able to recover from misspeculation faster than existing software eager TLS systems.

A second TLS system, `Lector` (Lazy insp`ECTOR`), is presented in Chapter 5, which uses a novel way of minimizing speculative marking (also uses the compact version management data structure). `Lector` performs on average 1.7x faster for 32 threads over an established state-of-the-art software TLS system. While the conventional TLS system is running, lightweight inspector threads are executed alongside to verify quickly whether speculative state maintenance is actually required. Those threads are highly likely to be faster than the TLS threads, as they only inspect a stripped-down version of the actual loop iteration consisting of memory accesses. Should the inspector threads discover that the loop is DOALL, speculation is abandoned allowing the application to run in a parallel speculation-free mode. On the other hand, if the inspector exposes any dependencies, then inspection is terminated and the system continues with conventional speculative parallelization.

In Chapter 6 `MiniTLS` and `Lector` were applied to seven Java sequential benchmarks (with presence of misspeculations for two benchmarks), including three benchmarks from `SPECjvm2008`. The experiments for `MiniTLS` report average speedups of 1.8x for 4 threads increasing close to 7x speedups with 32 threads. `Lector` experiments report average speedups of 1.7x for 2 threads increasing close to 8.2x speedups with 32 threads. It was shown that TLS can speedup the execution of some `SPECjvm2008` benchmarks but it was not fully explored which other `SPECjvm2008` will benefit from TLS. Similarly, “informed” scheduling was not explored for `Lector`, in order to avoid discovered dependencies.

7.2 Future Directions

There are numerous interesting directions to follow in designing speculative multi-threaded systems. This section discusses a few of them.

7.2.1 Scheduling Partially Parallel Loops

Chapter 5 explained how a software TLS system such as `Lector` can boost performance by the aid of inspector threads. In `Lector`, inspector threads execute alongside the TLS threads in order to verify whether a loop is a *DOALL* and allow non-speculative parallelization to take over. This verification is currently tied only to *DOALL* loop identification. Previous work using an *inspector/executor* model demonstrated how one can create an optimal schedule for *partially parallel* loops [RAP95]. A partially parallel loop is one that requires synchronization in order to maintain the correct execution order between iterations. Independent loop iterations are grouped together in sets called *wavefronts*. Iterations within each wavefront can execute in parallel but each wavefront must execute one after another. That is, there are no data dependencies between iterations inside a given wavefront but different sets of wavefronts depend on one another.

In order to parallelize the partially parallel loop, the inspector must first analyze at runtime the data dependence relations within statements in the loop and based on that information construct the appropriate schedule. The schedule dictates how the executor code will execute the loop iterations.

Combining the above technique with `Lector`, the inspector could identify a partially parallel loop and provide an optimal schedule such as that iterations within wavefronts execute in a speculation-free fashion.

7.2.2 Method-Level Speculation

The experiments for this thesis have considered only *loop*-level speculation, however, previous work indicates that there may be advantageous exploiting parallelism at different levels of granularity such as *method*-level [WS01, LTC+06, Pic07, ISK+10]. `Java`™ language is a suitable candidate for method-level parallelism since it enforces Object Oriented design and all executable code must be contained within a method [ISK+10]. Furthermore, since `Java`™ does not allow methods accessing each others stack, logging stack variables is avoided.

In the case of loop-level speculation, loop iterations form the boundaries of speculative regions. Any values before the loop starts that are still used inside the loop must be handled with care especially if they are updated. Similarly, any values produced inside the loop by the speculative threads must be copied appropriately if they are still used after the loop. In method-level speculation, speculative regions are formed by method boundaries. Each method is registered to a speculative thread and executes in parallel with other methods. Sometimes a method *B*, that depends on the return value of a method *A*, might be able to start early execution if the return value is predicted. This is known as *Return Value Prediction (RVP)*. There is no difference in the TLS support provided for both levels of speculation. The main difference is in the front-end translator that will take the sequential program and convert it to speculation-ready one. The loop-level translator would encounter loops, break-up the iterations to form speculative boundaries, handle the values coming in and going out of the loop, and insert calls to the TLS back-end library. The method-level translator will do the same except for method boundaries. The TLS back-end will still be exactly the same.

7.2.3 Adaptive Selection of TLS System

Generally speaking, a system under lazy version management might be more beneficial when conflicts are expected to be relatively frequent. When conflicts occur on a frequent basis, local buffers need only to be flushed before execution restarts. On the contrary, in case of frequent conflicts a system under eager version management will have to undo all values since updates appeared in-place. However, in case the conflicts are expected to be rare, lazy version management is not as suitable as eager since buffers need to be copied-out in memory after speculative execution. In this case an eager system simply continues execution neglecting any copying-out as updates are already there. This thesis discussed and experimented with `MiniTLS` and `Lector` which implement eager and lazy version management respectively.

It would be worthwhile for a system that is able to switch between the two systems during execution based on conflict frequency. For example, the system might start optimistically with `MiniTLS` maintaining some counter regarding the number of conflicts occurred. When that counter exceeds a certain threshold the system could switch into using a lazy version management such as the one employed in `Lector`.

Such an adaptive mechanism has been already evaluated in software transnational memory and found to be beneficial [Spe10]. Due to the similarities of STM and TLS, it is anticipated that an adaptive TLS system might offer superior performance to one

that uses only a single version management scheme. Furthermore this system might be able to adapt to applications with different behavior or applications that change phases during execution [GBEDB04].

7.2.4 Hardware Support

One of the main overheads of a thread-level speculation system (implemented in software) is maintaining the speculative state. Loads and stores needs to be examined globally using locks, before are inserted in their local data structures. As a consequence many clock cycles are wasted simply to synchronize and track such information. And even if that was not enough, during rollback, values may need to be locked and restored causing further clock cycles. Work in transactional memory and thread-level speculation has shown that using hardware might avoid these redundant cycles by taking advantage of certain properties of the coherence protocol. For example, local processor caches might be extended to support speculative behavior at minimal cost. Research in transactional memory has also demonstrated that it is possible to implement a hybrid system which some parts of it are controlled by hardware where others by software [DCW+11]. A future work in TLS that aims to minimize the software overheads could use `Lector` or `MiniTLS` on top of a full-system simulator (such as `Simics` [MHW02]) or a best-effort hardware (such as `ASF` [CYD+10]) in order to accelerate speculative operations.

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles Techniques and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007.

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[BEF+95] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David A. Padua, Paul Petersen, William M. Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 141–154, 1995.

[Bla99] Bruno Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *Proceedings of the 14th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, 1999.

[BN97] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[CL03] Marcelo Cintra and Diego R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[CL05] Marcelo Cintra and Diego Llanos. Design Space Exploration of a Software Speculative Parallelization Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[CMT00] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2000.

[CO03] Michael K. Chen and Kunle Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 434–446, 2003.

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 227–238, 2006.

[CTY94] Ding Kai Chen, Josep Torrellas, and Pen Chung Yew. An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops. In *Proceedings of the ACM/IEEE International Conference on Supercomputing (ICS)*, pages 518–527, 1994.

[CYD+10] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 39–50, 2010.

[Cyt86] Ron Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 836–844, 1986.

[DCW+11] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A Case

Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52, 2011.

[DSS06] Dave Dice, O. Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.

[DYR02] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 20–29, 2002.

[For93] MPI Forum. MPI: Message Passing Interface, 1993.

[GBEDB04] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level Phase Behavior in Java Workloads. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 270–287, 2004.

[GN98] Manish Gupta and Rahul Nim. Techniques for Speculative Run-time Parallelization of Loops. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 1–12, 1998.

[GPL+05] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors. *ACM Transactions in Architecture and Code Optimization (TACO)*, 2:247–279, September 2005.

[GVSS98] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 195–215, 1998.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69. ACM, 1998.

[Inc12] Apple Inc. MacBook Pro with Retina Display, 2012.

[ISK+10] Nikolas Ioannou, Jeremy Singer, Salman Khan, Paraskevas Yiapanis, Adam Pocock, Polychronis Xekalakis, Gavin Brown, Mikel Luján, Ian Watson, and Marcelo Cintra. Toward a More Accurate Understanding of the Limits of the TLS Execution Paradigm. In *Proceedings of the IEEE Inernational Symposium on Workload Characterization*, 2010.

[JEV04] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut Program Decomposition for Thread-Level Speculation. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 59–70, 2004.

[JEV07] Troy Johnson, Rudolf Eigenmann, and T.N. Vijaykumar. Speculative Thread Decomposition Through Empirical Optimization. In *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 205–214, 2007.

[KJL+12] Hanjun Kim, Nick Johnson, Jae Lee, Scott Mahlke, and David I. August. Automatic Speculative DOALL for Clusters. In *Proceedings of the Intl. Symposium on Code Generation and Optimization (CGO)*, 2012.

[KRL+10] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable Speculative Parallelization on Commodity Clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2010.

[Lam79] Leslie. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[LPH+09] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic Performance Tuning for Speculative Threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 462–473, 2009.

[LTC+06] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–167, 2006.

[MG99] N. Mukherjee and John Gurd. A Comparative Analysis of Four Parallelisation Schemes. In *Proceedings of the 13th International Conference on Supercomputing (ISC)*, pages 278–285, 1999.

[MG02] Pedro Marcuello and Antonio González. Thread-Spawning Schemes for Speculative Multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 55–67, 2002.

[MHHM09] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 166–176, 2009.

[MHW02] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-System Timing-First Simulation. In *Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 108–116, 2002.

[MLC+09] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A Fine-Grain Thread Decomposition Scheme for Speculative Multithreading. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–25, 2009.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[NKH04] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *Proceedings of 11th Static Analysis Symposium (SAS)*, pages 165–180, 2004.

[OB96] Michel O'Boyle and Mark Bull. Expert Programmer versus Parallelizing Compiler: A Comparative Study of Two Approaches for Distributed Shared Memory. *Scientific Programming*, 5(1):63–88, 1996.

[OHL99] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 303–313, 1999.

[OMH09] Cosmin Oancea, Alan Mycroft, and Tim Harris. A Lightweight in-place Implementation for Software Thread-level Speculation. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 223–232, 2009.

[ORSA05a] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[ORSA05b] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[PH08] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, Fourth Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Pic07] Christopher J. F. Pickett. Software speculative multithreading for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications companion (OOPSLA)*, pages 929–930, 2007.

[PO05] Manohar K. Prabhu and Kunle Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 142–152, 2005.

[PZH+09] V. Packirisamy, A. Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring Speculative Parallelism in SPEC2006. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 77 –88, 2009.

[QnMS+05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis Compiler: An Infrastructure for Speculative Threading based on Pre-computation Slices. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 269–279, 2005.

[RAP95] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A Scalable Method for Run-time Loop Parallelization. *International Journal of Parallel Programming*, 23(6):537–576, 1995.

[Rau98] Lawrence Rauchwerger. Run-time Parallelization: Its Time has Come. *Parallel Computing*, 24:527–556, 1998.

[RKM+10] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative Parallelization Using Software Multithreaded Transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, 2010.

[RP94a] Lawrence Rauchwerger and David Padua. Speculative Run-Time Parallelization of Loops. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, 1994.

[RP94b] Lawrence Rauchwerger and David Padua. The privatizing DOALL test: a run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS)*, pages 33–43, 1994.

[RP95] Lawrence Rauchwerger and David Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN International Conference on Programming language design and implementation (PLDI)*, pages 218–232, 1995.

[RS01] Peter Rundberg and Per Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *Journal of Instruction-Level Parallelism*, 3:1–28, 2001.

[Rus78] Richard M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[RVOA08] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance Scalability of Decoupled Software Pipelining. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(2):8:1–8:25, 2008.

[Sam12] Samsung. Samsung Galaxy SIII, 2012.

[SBW91] Joel H. Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. *Concurrency, Practice and Experience*, 3(6):573–592, 1991.

[SCZM05] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[SM91] Joel H. Salz and Ravi Mirchandaney. The Preprocessed DoAcross Loop. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 174–178, 1991.

[SMC89] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. The DoConsider Loop. In *Proceedings of the International Conference on Supercomputing (ISC)*, pages 29–40, 1989.

[SMC91] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[SMSS06] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Conference on Distributed Computing (DISC)*, pages 179–193, 2006.

[Spe10] Michael F. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 273–283, 2010.

[TFG10] Chen Tian, Min Feng, and Rajiv Gupta. Supporting Speculative Parallelization in the Presence of Dynamic Data Structures. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 62–73, 2010.

[TFNG08] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–341, 2008.

[Tho70] James E. Thornton. *Design of a Computer - The Control Data 6600*. Scott Foresman, Glenview, IL, 1970.

[TWFO09] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 177–187, 2009.

[VRR+07] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *Proceedings of the 16th International Conference on*

Parallel Architecture and Compilation Techniques (PACT), pages 49–59, 2007.

[WFW+94] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical report, Stanford, CA, USA, 1994.

[WKL07] Ian Watson, Chris Kirkham, and Mikel Lujan. A Study of a Transactional Parallel Routing Algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 388–398, 2007.

[WS01] Fredrik Warg and Per Stenström. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, 2001.

[YHM+08] Paraskevas Yiapanis, David J. Haglin, Anna M. Manning, Ken Mayes, and John A. Keane. Variable-grain and dynamic work generation for Minimal Unique Itemset mining. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*, pages 33–41, 2008.

[YRHBL13] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. Optimizing Software Runtime Systems for Speculative Parallelization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):39:1–39:27, 2013.

[ZY87] Chuan-Qi Zhu and Pen-Chung Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, 1987.

Appendix A

Baseline Systems Description

A.1 Introduction

This appendix describes in more details the baseline systems used to compare against `MiniTLS` (Chapter 4) and `Lector` (Chapter 5). The results were presented in Chapter 6.

A.2 Baseline used for MINITLS: SPLIP

Oancea *et al.* [OMH09] proposed `SpLIP` for software speculative parallelization support. To the best of the thesis author knowledge, it is the first software TLS system to implement eager version management and the only one in the literature apart from `MiniTLS`. Since both systems use the same version management implementation, `SpLIP` was the best candidate and thus selected as the baseline. The following paragraphs provide implementation details on `SpLIP`.

A.2.1 Metadata

`SpLIP` uses four shadow arrays apart from the private storage. This organization is illustrated in Figure A.1.

The data structures for `SpLIP` are used as follows:

Load: To mark the latest thread performed a load on a memory location.

Store: To mark the latest thread performed a store on a memory location.

TimeStamp: To mark the relative time that a thread performed a store on a memory location.

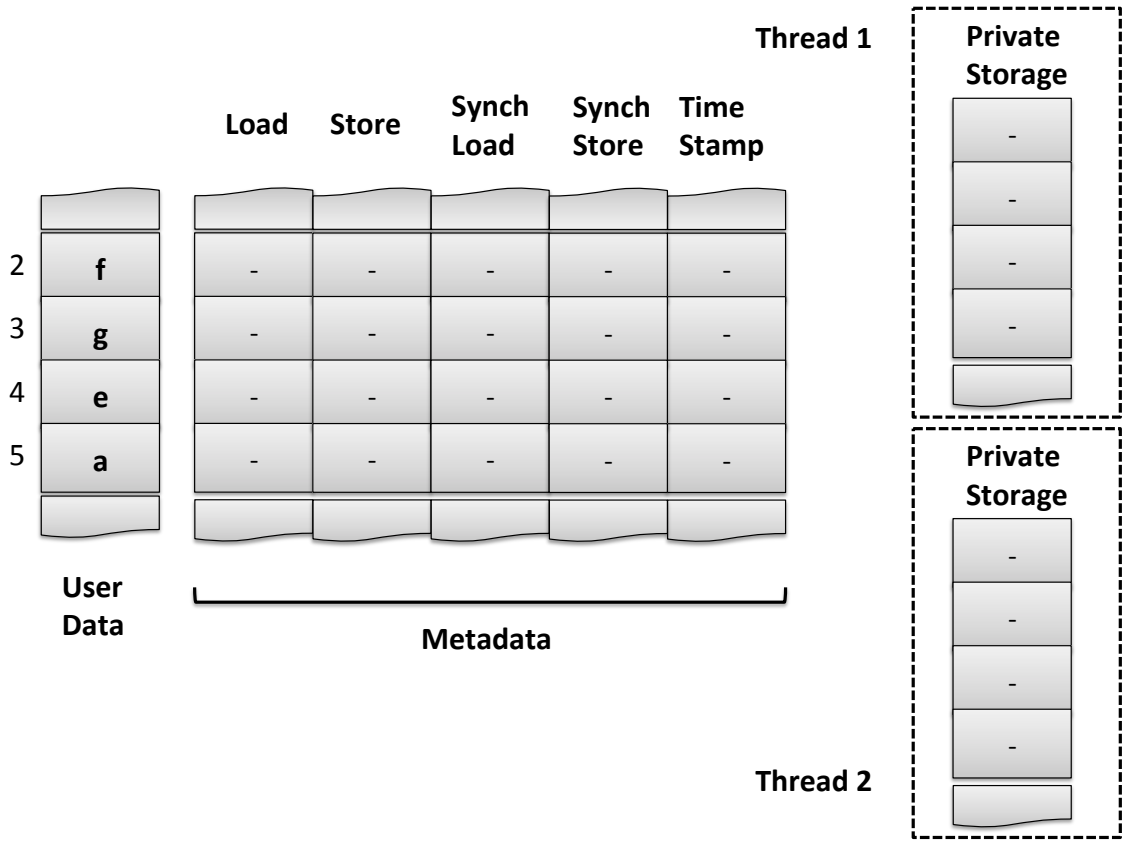

Figure A.1: Metadata organization for SpLIP [OMH09].

SynchLoad: Facilitates synchronization by tracking the current thread performing a load operation.

SynchStore: Facilitates synchronization by tracking the current thread performing a store operation.

A.2.2 Algorithm Outline

The execution flow of a speculative thread is the same as in MiniTLS. The differences between the two systems is how SpLIP performs speculative loads and stores. Since eager version management systems perform speculative updates directly in main memory they must be aware for all three types of violations (RAW, WAW, WAR). Assume a speculative thread with id T performing a speculative operation (load or store) on a memory location x.

Load

A speculative load by thread T to be performed at location x eventually needs to initialize $Load[x]$ with T's id to indicate that thread T is the latest one to load from x. Before loading the value from x, T must first ensure that the last thread to initialize x comes from a less speculative thread. Otherwise, if the last thread to write x is from a more speculative thread a WAR violation is triggered. This check is done by ensuring that $Store[x] <= T$.

Store

Similarly, a speculative store must initialize $Store[x] = T$ to indicate the latest thread that has written on x. A speculative store must check that $Store[x] > T$ in order to discover any WAW violations as well as that $Load[x] > T$ to find any RAW violations. If no violations are found the original value from x is saved in T's private storage along with the current timestamp in $TimeStamp[x]$.

Synchronization Arrays

`SpLIP` does not use any locks or CAS operations but instead uses two additional data structures to facilitate synchronization. When a speculative loads is performed all the operations are surrounded by actions on the *SynchStore* array. When speculative load starts, the *SynchStore* array is initialized with the id of the thread that performs the load (*i.e.* $SynchStore[x] = T$). At the end of the speculative load the condition $SynchStore[x] == T$ must hold for the operation to be valid. $SynchStore[x]$ may change while the load is performed only if a speculative store executed concurrently (the speculative store also sets $SynchStore[x] = T$ to indicate the action). In a similar way *SynchLoad* is initialized with the thread id that performs a load and checked within the speculative store operation.

The checks to the synchronization arrays are carefully placed before and after certain instructions and rely on the memory ordering of the architecture to be correct.

Rollback and Commit

When a conflict is detected, the offending threads must go thought their local buffers to restore memory location to a correct state. An issue arises when more than one thread involved in the violation has written to the same memory location. That is because only one of them must restore the correct value back to main memory, the one that has

the earliest copy in terms of speculation order. `SpLIP` uses the "TimeStamp" shadow array to record the relative time a thread has stored to a location. Using this shadow array the system is able to recover the earliest value need to be rolled back.

Since threads update directly the speculative values in memory, if no conflict is detected then the final results are already there. Thus commit implicitly happens *in parallel*.

A.3 Baseline used for LECTOR: TL2

Characterized by its simplicity, `TL2` [DSS06] formed the basis of several TLS and STM (Software Transactional Memory) designs. `TL2` uses a shadow data structure to reflect every memory location accessed *transactionally* (*i.e.* speculatively). The metadata organization is presented in Figure A.2. This data structure is an array in which every location combines a lock (to protect write accesses) along with a version number (that is used during commit to detect read conflicts). During execution a shared memory location is mapped and represented by a location in this shadow array. In the STM context this type of shadowing is called *versioned locks* and it is shared between threads. The system also maintains a *global clock* (a shared counter used to maintain consistency between transactions), which is incremented every time a transaction commits. Once a transaction initiates, the current value of the global clock is read atomically (using CAS) and stored locally. During execution the local clock of a transaction is used to ensure that every read operation is consistent across threads. This is possible since every memory location is associated with a version number. Before the transaction terminates, as part of its commit procedure all write accesses are locked and all read accesses are re-validated. If successful, the global clock is incremented atomically, all speculative updates are propagated to memory, and all writes are amended to hold the new value of the global clock before releasing their locks. This way the next committing transaction can ensure consistency by comparing their local value of the global lock against the lock of any values read. They global lock idea for STM was first used in [SMSS06].

The main reason that `TL2` has been selected for comparisons against `Lector` is because it has been already established as a baseline in the literature by [MHHM09] and [TFG10] in speculative parallelization and in numerous work in transactional memory.

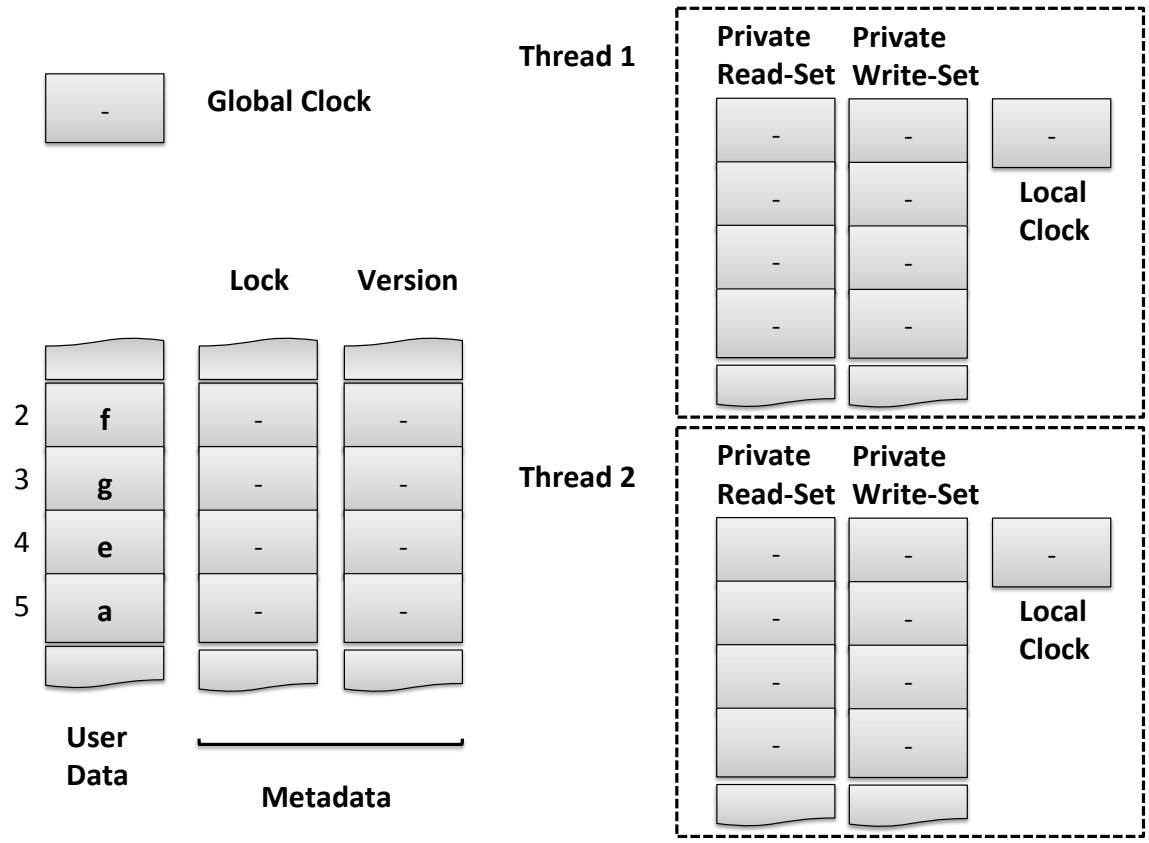

Figure A.2: Metadata organization for TL2 [DSS06].

A.3.1 Algorithm Outline

A speculative thread operates as follows:

1. **Sample global clock:** Load the current value of the global clock and store it in the thread's local clock.

2. **Execute speculative region:** Stores are buffered locally (lazy version management) in the thread's write-set along with the address to be written. Loads first check to see if the write-set already contains a store on that address to fetch the value from there (this avoids WAR and WAW dependencies). If not, the value needs to be loaded from memory in which case the local clock must be checked with the version lock of the address to ensure no other thread has changed it in the meantime (eager conflict detection).

3. **Lock write-set:** Acquire the locks of the locations to be written (based on the write-set). If not all locks are successful, the thread is squashed.

4. **Increment global clock:** When all locks are acquired the global clock is incremented using a CAS operation.

5. **Read-set validation:** The versions of locations in main memory are checked (based on the read-set) to ensure that their version did not changed (RAW violations) in the meantime by another thread (*i.e.* Their version is still the same as the local clock).

6. **Commit and release locks:** The values in the write-set are committed and the locks are released.

Appendix B

Implementation Details for TLS Limit Study

The performance potentials of speculative parallelization were assessed in a recent study [ISK+10] that was conducted in collaboration between the University of Edinburgh and the University of Manchester, a work that the author of this thesis was also involved. This fact was also discussed in Chapter 1. The study evaluated a mixture of different design aspects of speculative parallelization in a simulation environment in order to establish an upper bound on performance. The goal was to offer an architecture-agnostic characterization of the potentials of speculative parallelization.

The following section briefly explains how the study was performed.

Evaluation Details

Applications were tested from a variety of application domains (*e.g.* scientific and business domains) and programming styles (*e.g.* procedural and object-oriented styles).

The compiler was augmented with special instructions that were triggering events while the application was running. Those events were indications of the points wished to speculate. For instance, an event would be triggered at the beginning of a loop and another one at the end of the loop. Between those two events the hardware was recording every memory access performed by the sequential application. The hardware was implemented using a full system simulator known as `Simics` [MHW02] and all the events along with memory accesses were recorded in files called *traces*.

Traces were later fed to a custom-made simulation infrastructure. The infrastructure was simulating a speculative parallelization execution environment driven by the events and memory accessed recorded in the traces.