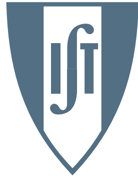

INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Practical Parallel Nesting for Software Transactional Memory

Nuno Miguel Lourenço Diegues

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Doctor Luís Eduardo Teixeira Rodrigues
Supervisor:	Prof. Doctor João Manuel Pinheiro Cachopo
Member:	Prof. Doctor João Manuel dos Santos Lourenço

July 2012

Resumo

Os processadores *multicore* são já comuns na maioria das máquinas. Isto significa que os programadores têm que enfrentar o desafio colocado pela exploração do potencial paralelismo destas novas arquitecturas. A Memória Transaccional (TM) é uma abstracção que promete simplificar esta tarefa.

No entanto, a TM inibe o programador de poder explorar todo o paralelismo latente na sua aplicação pois não permite que uma transacção contenha código paralelo. Este facto limita a expressividade da TM enquanto mecanismo de sincronização. Muitas aplicações contêm operações longas que têm que ser executadas com semântica atómica. Para mais, estes bocados de código podem requerer escritas em dados partilhados, o que tipicamente leva à criação de muitos conflitos em mecanismos de controlo concorrência optimistas como na generalidade das TMs. No entanto, algumas destas operações poderiam ser executadas mais rapidamente se o seu paralelismo latente fosse usado eficientemente, ao permitir que uma transacção seja dividida em partes que executem concorrentemente.

Nesta dissertação, ofereço esta flexibilidade adicional através de aninhamento paralelo. Para mais, proponho superar execuções inerentemente sequenciais, e com muitos conflitos, usando esta nova expressividade de uma TM. Mostro ainda que o uso de um escalonador de transacções é uma solução que benficia os resultados obtidos com aninhamento paralelo.

Mostro que a implementação destas ideias numa TM com suporte para múltiplas versões e progresso *lock-free* supera a performance da versão original em várias aplicações conhecidas até 2.8 vezes. Adicionalmente, mostro que esta solução é também até 3.4 vezes mais rápida que o estado-da-arte existente.

Palavras-chave: Memória Transaccional, Transacções Paralelas Aninhadas, Escalonamento, JVSTM

Abstract

Multicores are now standard in most machines, which means that many programmers are faced with the challenge of how to take advantage of all the potential parallelism. Transactional Memory (TM) promises to simplify this task.

Yet, at the same time, TM inhibits the programmer from fully exploring the latent parallelism in his application. In particular, it does not allow a transaction to contain parallel code. This fact limits the expressiveness of TM as a synchronization mechanism. Many applications contain large operations that must be performed atomically. These large sections may entail writing to shared data, which typically leads to many conflicts in optimistic concurrency control mechanisms such as those used by most TM systems. Yet, sometimes these operations could be executed faster if their latent parallelism was used efficiently, by allowing a transaction to be split in several parts that execute concurrently.

In this dissertation, I provide this increased flexibility by using parallel nesting. Moreover, I propose to overcome inherently sequential highly-conflicting workloads with the new expressiveness provided by TM. I additionally show that the use of conflict-aware scheduling provides an effective solution to maximize the benefits of parallel nesting.

I show how the implementation of these ideas in a lock-free multi-version STM outperforms the original version on several known benchmarks by up to 2.8 times. Moreover, I show that this solution is up to 3.4 times faster than state of the art alternatives.

Keywords: Transactional Memory, Parallel Nested Transactions, Conflict-aware scheduling, JVSTM

Acknowledgements

This document embodies a year of research, surviving the frustrations and commemorating the victories. But throughout this year, I was never alone, and I owe that to many people.

First and foremost, I would like to thank my thesis adviser, Professor João Cachopo, who guided me through the uncertainty that surrounds research. His availability led to countless discussions, and his critical reviews perfected much of my work. Many ideas stemmed from those moments, which were essential for me to have come this far.

I would also like to thank the members of the Software Engineering Group (ESW) at INESC-ID. In particular, to all my colleagues in room 635 and to Sérgio Fernandes. They all contributed to fruitful discussions and helped improve my presentations.

The Portuguese FCT (Fundação Ciência e Tecnologia) kindly granted me with a scholarship in the scope of the RuLAM project (PTDC/EIA-EIA/108240/2008), for which I am grateful.

To my mother, Luísa, and my grandmother, Laurinda, I am deeply thankful for all the support that allowed me to get this far. Your help and guidance started many years ago; it was defining and laid the foundations for making me capable of surpassing this task.

Finally, I would like to thank Beatriz Ferreira for her ever-lasting patience. Many times did your wise words shed light over my difficulties. I will never forget your unwearying support, making it easier to surmount this year.

Lisboa, July 2012
Nuno Diegues

A designer knows he has achieved perfection
not when there is nothing left to add, but
when there is nothing left to take away.
-Antoine de Saint-Exupéry

Contents

1 Introduction **1**

1.1 Thesis Statement . . . 2

1.2 Notation . . . 2

1.3 Publications . . . 2

1.4 Outline . . . 3

2 Motivation and Objectives **5**

2.1 Synchronization of Concurrent Operations . . . 5

2.2 Transactional Memory . . . 6

2.3 Seeking better performance . . . 8

2.4 Goals and Contributions of this Work . . . 10

2.5 Validation . . . 11

2.5.1 STMBench7 . . . 12

2.5.2 Vacation . . . 12

2.5.3 Lee-TM . . . 13

3 Related Work **15**

3.1 Transactional Memory Theory and Guarantees . . . 15

3.1.1 Correctness Criteria . . . 16

3.1.2 Operation level liveness . . . 17

3.1.3 Progressiveness . . . 18

3.1.4 Permissiveness . . . 18

3.2 Transactional memory design choices . . . 19

3.2.1 Update Policy 19

3.2.2 Conflict detection and resolution 19

3.3 Achieving Nesting by flattening 20

3.4 Linear Nesting 20

3.5 Parallel Nesting in TMs 22

3.5.1 NeSTM 22

3.5.2 HParSTM 23

3.5.3 PNSTM 24

3.6 Discussion of existing Parallel Nesting Implementations 25

4 A naive algorithm 27

4.1 JVSTM 27

4.1.1 Optimizations to the read-set and write-set 28

4.1.2 Nesting in the JVSTM 30

4.2 Parallel Nesting Model 30

4.3 An initial approach towards parallel nesting 31

4.3.1 Data-structures and auxiliary functions 32

4.3.2 The `Naive` algorithm 33

5 A lock-free algorithm 37

5.1 Data-structures 37

5.2 Reading from a `VBox` 39

5.3 Writing to a `VBox` 41

5.4 Committing Parallel Nested Transactions 42

5.5 Lock-free commit 44

5.6 Abort procedure 46

5.7 Correctness in the Java Memory Model 47

5.8 Discussion of the Shared write-set design 48

6 A practical algorithm 51

6.1 Data-structures and auxiliary functions 51

6.2 Reading a `VBox` . . . 54

6.3 Writing to a `VBox` . . . 55

6.4 Fallback mechanism . . . 57

6.5 Committing a parallel nested transaction . . . 59

6.6 Correctness in the Java Memory Model . . . 60

6.7 Progress guarantees . . . 60

6.8 Maintenance of read-sets . . . 61

6.9 Discussion of the `InPlace` design . . . 62

7 Scheduling for Profit **65**

7.1 Scheduling transactions . . . 66

7.1.1 A scheduler for the JVSTM . . . 67

7.1.2 `Serial` scheduler . . . 67

7.2 Using the `Serial` scheduler . . . 70

8 Evaluation **73**

8.1 Evaluating the different JVSTM-based implementations . . . 73

8.1.1 Vacation . . . 73

8.1.2 STMBench7 . . . 75

8.2 Relinquishing the overhead of parallel nesting . . . 77

8.3 Comparison against the state of the art . . . 79

8.3.1 Worst-case complexity bounds . . . 79

8.3.2 Practical comparison . . . 80

8.3.3 Discussion . . . 82

8.4 Summary . . . 84

9 Conclusions **85**

9.1 Main Contributions . . . 85

9.2 The problem of finding latent parallelism . . . 86

9.3 Future research . . . 87

9.3.1 Parallel TM 88
9.3.2 Threads and Transactions 88

A Using parallelism within transactions **91**

A.1 Interfacing with threads and transactions in Java 91
A.2 Providing support for the API 94

List of Figures

2.1 Execution of an application both sequentially and parallelized using TM 9

2.2 Execution of an application with conflicts 10

3.1 Nesting tree in the NesTM 23

3.2 Nesting tree in the PNSTM 25

4.1 A transactional location in the JVSTM 27

4.2 Representation of the actions regarding the read-set maintenance in the JVSTM 29

4.3 Nesting tree and respective state in the original JVSTM 30

4.4 Nesting tree and respective state in the `Naive` design 32

5.1 Nesting tree corresponding to an execution in the `SharedWS` design 38

5.2 Committing strategies in the `SharedWS` design of the JVSTM 43

5.3 Structures used for nested commit in the `SharedWS` design 45

5.4 Profiling data of the transactional operations in the `Naive` and `SharedWS` designs of the JVSTM 48

6.1 Representation of the state maintained in the `InPlace` design of the JVSTM 52

6.2 Profiling data of the transactional operations in all three designs of the JVSTM 63

7.1 Speedup in `STMBench7` using parallel nesting 65

7.2 Comparison of speedups obtained in `STMBench7` with three scheduling strategies 68

7.3 Speedup in `STMBench7` with and without scheduling 70

7.4 Distribution of time spent executing transactions among the working threads in `STMBench7` 71

8.1 Speedup obtained in the `Vacation` benchmark using the three parallel nesting designs . 74

8.2 Speedup of each parallel nesting design in `Vacation` 74

8.3 Speedup obtained by parallelizing the read-write long traversals of the `STMBench7` with the three parallel nesting designs 75

8.4 Speedup of each parallel nesting design in `STMBench7` 75

8.5 Speedup obtained in `STMBench7` with and without parallel nesting, and with and without scheduling 76

8.6 Speedup obtained in `Lee-TM` with embarrassing parallelization of transactions 78

8.7 Throughput in `Vacation` with parallel nesting comparing three STMs 81

8.8 Throughput in `STMBench7` using parallel nesting comparing three STMs 81

8.9 Overhead of each STM in `STMBench7` 82

List of Tables

4.1 Number of reads and read-after-writes in several benchmarks 35

6.1 Number of writes and write-after-reads in several benchmarks 58

8.1 Complexity bounds for the worst-case of transactional operations in parallel nested transactions in the JVSTM, NesTM and PNSTM 79

8.2 Possible conflicts that may lead to abort in the JVSTM, NesTM and PNSTM 80

8.3 Occurrence of each type of read in `STMBench7` using the JVSTM, NesTM and PNSTM . 82

8.4 Occurrence of conflicts in `STMBench7` using the JVSTM, NesTM and PNSTM 83

8.5 Percentage of transactions that failed in their first attempt in two benchmarks. 83

Listings

2.1 Concurrent class representing a course whose enrollments are protected with transactions. 7
2.2 Representation of an enrollment in multiple courses as an atomic action. 8
3.1 Example of code that may not be executed properly due to the lack of an appropriate correctness criterion. 16
6.1 Class representing a block of read entries. 62
7.1 Interface implemented by tasks to run through the scheduler. 68
7.2 Interface implemented by the different scheduling approaches. 68
7.3 Interface implemented by the application worker threads. 69
A.1 Class representing a college course in which students may enroll. 92
A.2 Interface to be implemented by the programmer representing points of parallelism. 92
A.3 Parallelization of the method `enrollMultipleCourses` from Listing A.1. 93
A.4 `Callable` generated for parallel execution of a method identified with `@ParallelAtomic`. 95
A.5 Class representing a parallelization of Listing A.3 after being automatically rewritten by the bytecode processor. 96

List of Algorithms

1 Read and write operations of the Naive design in the JVSTM 33
2 Commit operation of the Naive design in the JVSTM 34
3 Read procedure in the SharedWS design of the JVSTM 40
4 Write procedure in the SharedWS design of the JVSTM 41
5 Merge procedure in the SharedWS design of the JVSTM 46
6 Read procedure in the InPlace design of the JVSTM 54
7 Write procedure in the InPlace design of the JVSTM 56
8 Query procedure for transaction start in the Serial scheduler 71

List of Abbreviations

TM Transactional Memory

STM Software Transactional Memory

HTM Hardware Transactional Memory

CNT Closed Nested Transaction

ONT Open Nested Transaction

RAW Read-after-write

WAR Write-after-read

CAS Compare-and-swap

VBox Versioned Box

Orec Ownership Record

Chapter 1

Introduction

Up until 2004, multi-processor computers were seen only on research laboratories or as enterprise servers. For many years, most applications running on common hardware benefited from automatic improvements in performance as processors were upgraded with increasing clock speeds. Since then, we have reached a hard physical limit dictating the decline of Moore's Law applied to processors' frequency [47]. As Sutter entitled his article in [58], "The Free Lunch is Over", in the sense that programmers can no longer expect their applications to become faster in the way they used to.

This led to a paradigm shift: The manufacturers of these chips adopted a strategy in which newer processors have more transistors rather than a higher clock speed. This had the consequence of including more cores in each chip, thus bringing parallel architectures to common devices that are now capable of executing multiple threads simultaneously. Thus, programmers can no longer use sequential programs to explore all the computing power of modern processors; for that to happen, every core must be executing code in parallel. This has spurred the interest on easing the development of concurrent programs for shared-memory multi-processors. As a matter of fact, concurrent programming has been used for many decades, but it is only now that it is becoming an increasing trend affecting the daily life of programmers beyond a niche of researchers.

The dominant approach to protect concurrent accesses to shared data has traditionally relied on locking. Yet, decades of using blocking synchronization has not made it simpler, and in fact, resulted in a body of research that identifies many difficulties in building complex applications with locks [48, 49].

Furthermore, the paradigm in which the locking approach became popular has changed. It is expected that the current trend of multi-core processors leads to scenarios with hundreds of cores available in a single machine. In this new paradigm, programmers must explore fine-grained locking to be able to take advantage of many cores. Yet, this leads to an increased complexity: The blocking nature of locking may lead to issues such as deadlocks and convoying, which are more likely to happen with fine-grained locking and high degrees of concurrency.

This motivated the exploration of non-blocking algorithms that provide stronger progress guarantees. It is possible to avoid the use of locks by resorting to constructions based on instructions that manipulate memory atomically. However, this approach has been acknowledged as complex and difficult [34, p. 420].

Alternatively, Transactional Memory (TM) was proposed as an efficient and easy to use non-blocking construction, which avoids the pitfalls of mutual exclusion locks. Yet, TM is not exempt from limitations.

In particular, the characteristics of the application's workload influence greatly the performance gains that we may obtain when synchronizing accesses to shared data with TM. The emphasis of this work lies in exploring parallelism within transactions to overcome some of these limitations. As we shall see, this may be achieved by using parallel nesting.

1.1 Thesis Statement

This dissertation's thesis is that it is possible to explore more parallelism from TM-based applications if the TM system is extended both with efficient, light-weight algorithms to support parallel nesting and with a conflict-aware transaction scheduler that helps the application programmer decide when to use parallel nesting.

In particular, I claim that it is possible to implement a practical parallel nesting algorithm that is sufficiently efficient so that its overhead does not hinder the performance gains obtained from the extracted parallelism. As we shall see, the existing state of the art parallel nested algorithms are not able to fulfill this claim in some applications.

Moreover, because parallel nesting introduces overhead and may not be beneficial all the time, it cannot be used blindly. Yet, I claim that making programmers the sole responsible for deciding when to use parallel nesting is not viable in general. Instead, I defend that it is better to assist programmers in their work by delegating the decision of using parallel nesting to an automatic scheduler embedded in the TM system.

1.2 Notation

Throughout this dissertation I refer to transactions using T_i where the value of i may vary to represent different transactions. I also refer to transactional locations using either x, y or z.

To describe an execution involving one or more transactions, I shall use the following notation for the operations occurring within the transactions, where we assume that each operation executes atomically:

- $W_t(x, k)$ means that transaction t writes the value k to the transactional variable x.
- $R_t(x, k)$ means that transaction t reads the transactional variable x and finds the value k.
- $C_t(res)$ means that transaction t attempted to commit and either succeeded (when $res = ok$) or failed (when $res = fail$).
- $S_t(t_1, t_2, ..., t_n)$ means that transaction t spawns the parallel nested transactions $t_1, t_2, ..., t_n$.

When presenting source code, I show it in the Java programming language. I use a different font when referring to `classes`, `methods` and `fields`.

1.3 Publications

Part of the contents of this thesis were also presented in the following workshops:

1. Lock-free algorithm for parallel nesting presented in Chapter 5 and in the respective subsections: N. Diegues, S. Fernandes and J. Cachopo. Parallel nesting in a lock-free multi-version software transactional memory. In the 7th *ACM SIGPLAN Workshop on Transactional Computing, TRANSACT*, 2012.

2. Overview of Chapters 5, 6 and 7: N. Diegues and J. Cachopo. Digging parallelism out of a highly-conflicting workload. Abstract and presentation in the 1st *Workshop on Transactional Memory, WTM*, 2012.

3. Comparison of the main results obtained in this thesis with the state of the art, part of which is included in Chapter 8: N. Diegues and J. Cachopo. On the design space of Parallel Nesting. In the 4th *Workshop on the Theory of Transactional Memory, WTTM*, 2012.

Other parts of this document are also available in technical reports:

1. Extended version of the related work presented in Chapter 3: Nuno Diegues and João Cachopo. Review of nesting in transactional memory. Technical Report RT/1/2012, Instituto Superior Técnico/INESC-ID, January 2012.

2. Article under submission corresponding to Chapters 6 and 7: Nuno Diegues and João Cachopo. Exploring Parallelism in Transactional Workloads. Technical Report RT/16/2012, Instituto Superior Técnico/INESC-ID, June 2012.

1.4 Outline

The remainder of this dissertation is organized as follows:

- *Motivation*: Chapter 2 addresses the challenges of locking and introduces Transactional Memory as an alternative. It also motivates for parallel nested transactions and summarizes the contributions of this work.

- *Related Work*: Chapter 3 presents a brief overview of the related work. Namely, the guarantees and properties provided by Transactional Memory, as well as various design decisions and nesting models that exist in the literature.

- *A naive algorithm*: Chapter 4 proposes an initial solution to parallel nesting. It also identifies the inherent problems that arise from such solution.

- *A lock-free algorithm*: Chapter 5 extends the initial solution with a lock-free algorithm for parallel nesting that improves over some challenges identified earlier. This entails a thorough description of each operation of the algorithm.

- *A practical algorithm*: Chapter 6 tackles the challenges that were not taken into account previously. In addition to that, it also improves over new problems that were created by the lock-free algorithm. The description of the algorithm is complemented with an evaluation in terms that compare the three alternatives proposed for parallel nesting.

- *Scheduling for Profit*: Chapter 7 introduces the interesting coupling between parallel nesting and scheduling. It comprehends an overview of the related work on scheduling in TM. Additionally, three alternatives are presented for transaction scheduling, among which one is explained in more detail along with its evaluation.

- *Evaluation*: Chapter 8 provides a comprehensive evaluation of three alternative designs for parallel nesting in the JVSTM. Furthermore, it shows results in which parallel nesting takes advantage of scheduling and surpasses a baseline result using only top-level transactions. Finally, it presents an analysis and evaluation in which the JVSTM with support for parallel nesting is compared with two state of the art TMs (also with support for parallel nesting).

- *Conclusions*: Chapter 9 summarizes the work described in this dissertation, the results achieved, and what are its main contributions. It also presents some open issues related to this work and ways to explore them.

- *Using parallelism within transactions*: Appendix A discusses the relation between threads and transactions. In particular, it describes how the parallel nesting mechanism is exposed to the programmer.

Chapter 2

Motivation and Objectives

To understand why parallel nesting is of interest to TM, it is important to address the reasons that make TM an adequate choice for synchronization over the alternatives in the first place.

So, in this chapter I begin by describing shortly the inherent difficulties of synchronizing accesses to shared data in Section 2.1. Next, I present TM as an appealing alternative to address these challenges in Section 2.2. After, I delve into the limitations of TM that motivate the dissertation statement provided earlier in Section 1.1.

I end this chapter by stating the goals of this work, as well as the challenges that make it difficult to reach those goals, in Section 2.4.

2.1 Synchronization of Concurrent Operations

The synchronization of concurrent programs has been traditionally achieved by resorting to blocking synchronization techniques such as locks, semaphores, monitors, and conditional variables. Despite their popularity, they are not free from many pitfalls.

A usual scenario, upon which I shall build the examples, involves concurrent objects, meaning that their methods may be called in such a way that the invocation intervals overlap each other. Typically such objects contain some state that is thus protected by some mutual exclusion lock. However, if we consider an operation that requires manipulating several of these objects without allowing intermediate states to be observed, it follows that we cannot simply rely on the individual lock acquisition that takes place inside the object. That is, the lock that protects each object is not enough for preventing inconsistent states for the outer operation. As a result, traditional techniques usually resort to additional locking that guarantees none of the objects may change while that bulk action takes place. This solution, however, is prone to deadlocks. Depending on the strategy used for the multiple lock acquisition, it may happen that concurrent threads acquire one or more of the locks each and remain indefinitely trying to acquire the rest.

Whereas there are techniques to avoid deadlocks, such as establishing some total order among the elements to lock, they are hard to apply in practice. In the past, when highly scalable applications were rare and valuable, these hazards were avoided by dedicating teams of expert programmers to develop

these algorithms. Today, when highly scalable applications are becoming commonplace, the conventional approach is just too expensive. As claimed by Herlihy and Shavit: “The heart of the problem is that no one really knows how to organize and maintain large systems that rely on locking” [34, p. 418]. In practice, the association between locks and data is established mostly by convention and is not explicit in the program. Ultimately, this ordering convention exists only in the mind of the programmer, and may be documented only in comments as shown in [49]. “Over time, interpreting and observing many such conventions spelled out in this way may complicate code maintenance” [34, p. 418].

The issues do not concern only deadlocks: Overall, locking, as a synchronization discipline, has many pitfalls for the inexperienced programmer [48]. Two other common problems with lock-based synchronization are priority inversion and convoying. Priority inversion occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. Convoying may also occur when a thread holding a lock is descheduled, perhaps by exhausting its scheduling quantum, by a page fault, or by some other kind of interrupt. While the thread holding the lock is inactive, other threads that require that lock will queue up, unable to progress.

However, the fundamental flaw is that locks and conditional variables do not support modular programming: The process of building large programs by gluing together smaller programs. Creating software on top of modules that synchronize access to shared data with locks may entail finding out internal locks that are acquired as well as their order. Unfortunately, not only is this impractical, but it also breaks the abstraction that was supposedly provided by the modules.

2.2 Transactional Memory

One of the alternatives for synchronization between concurrent units of work is the Transactional Memory abstraction. The programmer is responsible for identifying, in his program, atomic blocks that the TM runs within transactions. As originally proposed, a transaction is a dynamic sequence of operations executed by a single thread that must appear to execute instantaneously with respect to other concurrent transactions [33]. The purpose is that this set of operations is seen as an indivisible action, so that transactions appear to execute sequentially in a one-at-a-time order. Despite this traditional definition, there is no obstacle preventing a transaction from being executed in parallel in several threads. In this work I explore the decoupling of a transaction from a single thread due to parallel nesting.

Operations enclosed in transactions are given the illusion of no concurrency. If a transaction fails, it is as if it never ran (no partial executions). A failed transaction may be retried, depending on the nature of the failure, to achieve exactly-once execution wherever possible. Summarizing, transactions offer:

- Atomicity: Either the whole transaction is executed (when it successfully commits) or none of it is done (when it aborts), often referred to as the all or nothing property.
- Consistency: Every transaction starts from a consistent view of the state and leaves the system in another consistent state, provided that the transactions would do so if executed sequentially.
- Isolation: Individual memory updates within an ongoing transaction are not visible outside the transaction. When the transaction commits, all memory updates are instantaneously made visible to the rest of the system.

A transaction typically works in three phases:

- Start: This event may have different purposes depending on the TM implementation. A common operation is to set up data structures that are used later on for bookkeeping.
- Accessing data: During the transaction itself, accesses performed to shared data may have to resort to the TM system. Some systems may be completely transparent, as in the case of Hardware Transactional Memory (HTM), whereas others may require the use of explicit calls to the TM in use. In any case, the TM ensures that writes are registered (in the write-set) and reads are consistent (and possibly also registered in the read-set).
- Commit: Attempts to consolidate the tentative changes, recorded during the accessing phase, making them globally visible. Depending on the system, the writes may already be in-place or still be in buffers before this phase. This operation may fail in which case it discards all its tentative changes.

Additionally, two transactions are said to conflict if there is no equivalent sequential execution ordering of the two transactions that explains the result of each individual operation that is part of the transactions. At a lower level, conflicts may be detected in different ways depending on the TM characteristics. For instance, if two operations belonging to different transactions access the same base object and at least one of them is a write, this may be seen as a conflict. Another way is to ensure that a transaction's read set is disjoint from concurrent transactions' write sets.

As explained, in this paradigm, the programmer is responsible for identifying code whose result in the system must be seen as taking effect all at once. In the following examples this boils down to marking methods with some artifact that identifies that piece of code as an atomic action. This is merely illustrative: Depending on the implementation, the programmer may be left with a more burdensome task of starting and committing transactions that encapsulate the atomic actions.

In Listing 2.1, I show an example in which a university course is represented with a maximum capacity and currently enrolled students. The mutable shared state is the list of students that may be modified concurrently by multiple users enrolling in the same course. The lack of synchronization in the concurrent manipulation of the shared structure could lead to the loss of enrollments or exceeding the maximum capacity. Therefore, I have annotated the `enrollStudent` method with the `@Atomic` annotation that demonstrates a possible way of indicating to the TM system which methods must be run transactionally. The enrollment of a student is now seen as an indivisible operation so that other threads cannot see any intermediate state of the operation.

```
class Course {

  final int capacity;
  List<Student> enrolledStudents;

  @Atomic
  boolean enrollStudent(Student std) {
    if (enrolledStudents.size() < capacity) {
      enrolledStudents.add(std);
      return true;
    } else {
      return false;
    }
  }

}
```

Listing 2.1: Concurrent class representing a course whose enrollments are protected with transactions.

```
@Atomic
void enrollMultipleCourses(Student std, List courses) {
  for (Course course : courses) {
    if (!course.enrollStudent(std)) {
      TM.abort();
    }
  }
}
```

Listing 2.2: Representation of an enrollment in multiple courses as an atomic action.

Still on the same example, Listing 2.2 shows the enrollment in multiple courses, which allows the student to build his own schedule for the semester with an all-or-nothing semantics: If one of the courses is full, he will probably need to pick an alternative and rethink the whole schedule. Consequently, I marked the `enrollMultipleCourses` as an atomic action, which is successful only if all the individual enrollments succeed. More importantly, deciding on this method's atomicity did not interfere with the previous decision regarding `enrollStudent`. It is in this sense that transactions compose: The programmer need not know the internals of the method being called. In practice, when `enrollStudent` is called from within `enrollMultipleCourses`, a nested transaction may be created, as explained in Section 2.3 and further detailed in Chapter 3.

The initial proposal for transactional memory introduced it as an abstraction that programmers could use for lock-free synchronization in their applications [33]. In that work, the authors presented designs for extensions to multiprocessors' cache coherence protocols. Later, Shavit and Touitou evolved the same concept solely to software in [55]. Yet, it was very restrictive as the programmer had to identify static transactions, that is, transactions that access a pre-determined sequence of locations. These issues were first overcome in DSTM [32], a Software TM providing a slightly more relaxed progress guarantee, obstruction-freedom, which shall be described in Section 3.1.2.

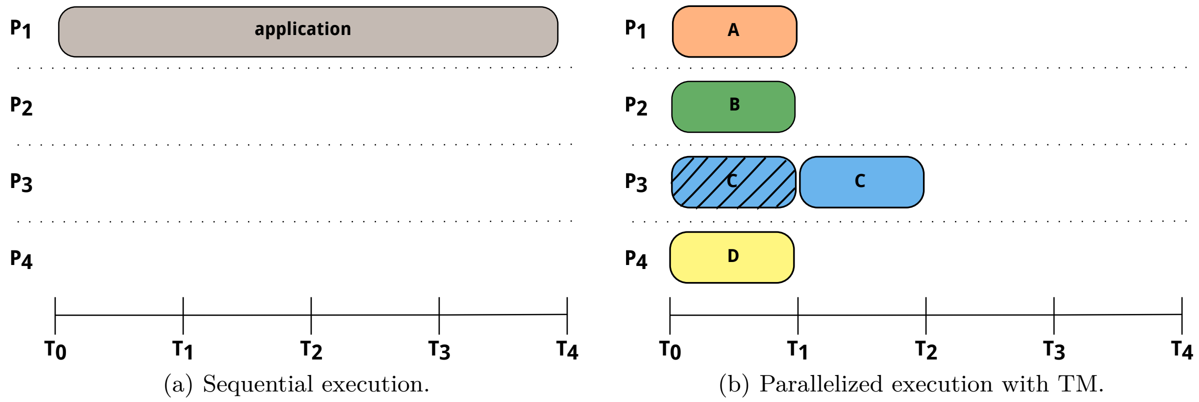
A lot of promising work has been delivered on Software Transactional Memory (STM). Although hardware implementations are more efficient, its practicality is far more complicated. As we shall see, most of the work regarding nesting models has been performed on STMs. Consequently, that is where I turn my attention to in this work.

2.3 Seeking better performance

When a programmer parallelizes an application, his sole intent is to obtain better performance than if the application ran sequentially. Yet, the synchronization of accesses to shared data by the parallel tasks, while ensuring correctness of the application, also yields difficulties in obtaining the intended improvements in performance.

When resorting to the traditional mutual exclusion mechanisms, performance may be hindered by sequential bottlenecks induced by the blocking phenomenon that takes place when the tasks contend for the same locks. Consequently, the programmers attempt to reduce the granularity of the locks, at the cost of an increased difficulty in programming the application and reasoning about its correctness.

Rather than relying on mutual exclusion to synchronize the access to shared data, TM starts out from the premise that parallel tasks seldom contend for the same data. More specifically, it explores concurrency as much as possible while preserving correctness. Figure 2.1(a) shows the sequential execution of an application in a machine with four processors labeled from 1 to 4. To take advantage of the



(a) Sequential execution.

(b) Parallelized execution with TM.

Figure 2.1: On the left, an application is executed sequentially whereas on the right it is parallelized in four transactions (A to D) to take advantage of the four processors available. Dashed transactions are aborted due to conflicts whereas non-dashed have committed successfully.

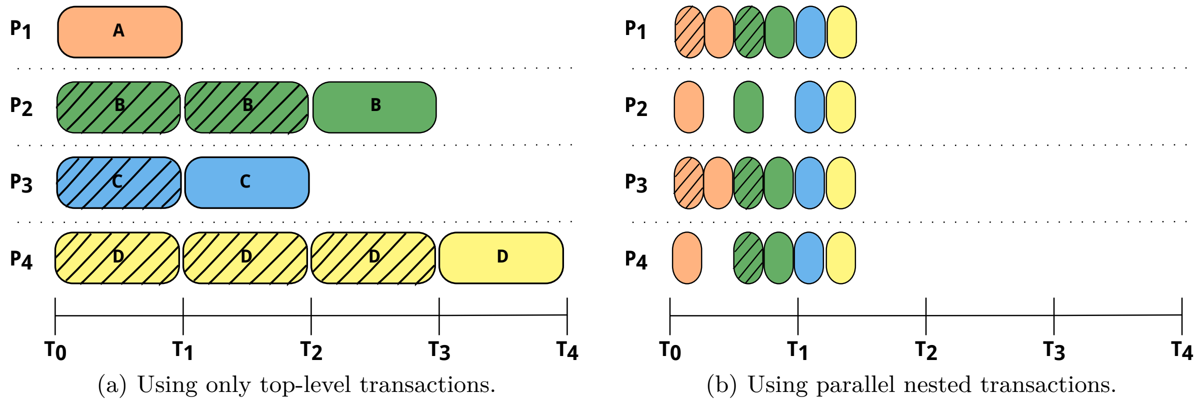
available processors, it is possible to explore the parallelism in the application, which could result in the identification of four concurrent tasks that are labeled A to D in Figure 2.1(b). In this case, these tasks are synchronized using TM, for which reason these tasks are run within transactions. For the TM system to ensure correctness, it has to abort transactions that incur in conflicts, as exemplified with the transaction in task C. Yet, the overall execution time yields improvements over the sequential execution, without the programmer being burdened with reasoning about the way the data is being accessed by these tasks.

Yet, when the application's workload is write-dominated, in the sense that most transactions perform at least some writes, this may result in a highly-conflicting execution. If that is the case, the optimistic concurrency model used by most TMs cannot overcome a logical barrier in terms of performance: Ultimately, if all the active transactions at some point conflict with each other, the time that takes to execute all of them successfully in parallel will not be less than the time it would take to execute them one at a time in a single-core machine. In practice, the single core would actually be faster due to the TM system overhead and cache invalidation concerns on the multicore. I exemplify this situation in Figure 2.2(a), in which it is visible that all the transactions are contending for the same data between time T_0 and T_1 such that they are producing conflicts and only one transaction is able to commit.

It is in this context that I attempt to answer the following question in this dissertation: Is there any way TMs can overcome this inherent limitation that challenges its optimistic concurrency control mechanism? One possible way is to reduce the amount of work that has to be repeated when a transaction restarts. This strategy has been approached by checkpointing [59] and restartable transactions [11]. In common, they attempt to make the most out of a situation in which conflicts already happened. In this dissertation I depart from those ideas and propose to explore the inner parallelism of transactions, allowing us to take full advantage of the underlying hardware.

My claim is that the parallelization of transactions can increase the performance in terms of throughput and latency. In the previous scenario of conflicting transactions, the solution would be to run one of the contending transactions at a time. But, in this case, each of these transactions would be parallelized, thus reducing their time to complete and without incurring in conflicts between any two transactions. This approach is expected to produce better results when top-level transactions conflict with high probability (such as the scenario presented) whereas the parallelization of each task does not incur in that problem.

Note that running one top-level transaction at a time in the solution proposed is a simplification: We



(a) Using only top-level transactions.

(b) Using parallel nested transactions.

Figure 2.2: Execution of four transactions (A to D) in four processors. Dashed transactions are aborted due to conflicts whereas non-dashed have committed successfully.

may very well run other concurrent transactions to fill up all the cores if needed, but the point is that there will be less of those, thus causing less conflicts.

In Figure 2.2 I show the application of this technique to the same scenario. Each task has now been further parallelized into smaller tasks. Note that each task is still meant to run with the properties of a single transaction, despite the fact that it is now executed in smaller tasks concurrently. The expectation is that the time it takes to execute the critical path of the set of parallelized transactions will be less than the time it takes to execute each of the top-level transactions sequentially as seen in Figure 2.2(a). But how can we achieve this while respecting the fact that each sub-task is still part of a transaction that itself contains other sub-tasks?

To start with, the parallelization of a transaction requires the TM structures to be thread-safe. This means that the state maintained by a single transaction must be safely accessible by multiple threads. This suffices for the case in which the sub-tasks of a transaction are disjoint-access parallel [1]. I refer to this scenario as an embarrassingly parallelization of a transaction, which I address in Section 8.2.

If the parallelization is not embarrassingly, then the parallel sub-tasks must be synchronized. This is also represented in Figure 2.2(b), where we may see that some of the smaller tasks are still conflicting and being re-executed. Using the TM model, each thread running a sub-task encapsulates in a transaction the code that it is running to allow detection and resolution of conflicts (even against other sub-tasks). Because these transactions exist in the context of top-level transactions, they are called nested transactions. This idea may be repeated to explore further parallelism at different levels of nesting. Consequently, this requires an efficient parallel nesting algorithm that supports unbounded nesting depths.

2.4 Goals and Contributions of this Work

The main goal of this dissertation is to advance the state of the art in STM research, by designing and implementing a parallel nesting algorithm that supports unlimited depth and without incurring into excessive overhead that precludes the benefits of the parallelism being explored. The starting point for this work is the current design of a lock-free multi-version STM and its implementation, the JVSTM [24], which has support for linear nesting only. Another important goal is to preserve the progress guarantee

[1]Meaning that they do not have any intersection among their footprints that may cause a conflict.

of the underlying TM.

As we shall see in the following chapter, there have been few TM implementations that address parallel nesting. So far, I have described why parallel nesting should be taken into consideration: The promise of unveiling more concurrency in scenarios where that may lead to an increase in performance is tempting. But if that is the case, then what exactly has been delaying the use of nested parallel transactions in practice?

Providing parallel nested transactions entails not only the challenges of nesting but also the need to make sure that parallel nested transactions synchronize their actions when necessary. On the first case, there is a concern regarding additional work that may need to be performed in the transactional operations (such as accesses and commit) when the nesting depth increases. The issue about the synchronization is, of course, an additional source of overheads that may entail significant costs on the use of parallel nested transactions.

The actual challenges shall be clear as I present the TM implementations that provide parallel nesting in Section 3.5 and onwards. Above all, the most important point to retain, beyond the particularities of what makes it hard, is that using parallel nested transactions should provide a performance gain. When the programmer takes specific care to identify parts of the program to parallelize, he is expecting to obtain a speedup in the execution of his program. Therefore, the difficulty is in providing a design and implementation of parallel nested transactions in which executing concurrent parts of an atomic block of the application do not end up being more costly than executing them one at a time sequentially.

To reach the aforementioned goals, my contributions are as follows:

- I provide a more flexible TM in which it is possible to parallelize transactions as opposed to the traditional perspective that has seen transactions as a sequential set of instructions. The parallel nesting algorithm proposed in this dissertation for that purpose also provides support for unbounded nesting depth.
- I show that this parallel nesting algorithm can improve the performance obtained with TM in some highly-conflicting workloads. I also present results that compare it with two state of the art parallel nesting algorithms showing considerable gains.
- I also show that the benefits obtained are substantially increased when a conflict-aware scheduler is used.
- Finally, I show that these results may be obtained without affecting the normal execution of the underlying TM when no parallel nesting is used. Moreover, the progress guarantee of lock-freedom of the underlying STM is preserved, thus making it the only lock-free TM with support for parallel nesting. Additionally, it is also the first parallel nesting algorithm with support for multi-versions.

2.5 Validation

To confirm the validity of my contributions in this dissertation, I conducted an evaluation that uses three well-known benchmarks as case studies and examples of highly-conflicting workloads. I changed those benchmarks to explore the parallelism of some of their transactions and therefore obtain better performance, as claimed in my thesis statement. In the next sections I present these benchmarks, as well as the parallelization that I performed in each one of them.

All the results presented in this dissertation were obtained on a machine with four AMD Opteron 6168 processors (48 cores total) with 128GB of RAM, running Red Hat Enterprise 6.1 and Oracle's JVM 1.6.0_24. The results were also obtained from the average of five runs, each one starting a new JVM, and executing without other significant processes in the machine.

2.5.1 STMBench7

`STMBench7` [30] is a highly customizable benchmark, with three different workloads that vary the percentage of read-only transactions: read-dominated (90%), read-write (60%), and write-dominated (10%). Besides that, we may also control a series of other parameters such as one that includes long, highly-conflicting transactions in all the workloads. This benchmark adapts the `OO7` [12] benchmark's data structure. `STMBench7` implements a shared data structure, consisting of a set of graphs and indexes, which models the object structure of complex applications.

The benchmark measures how many operations per second it executes. It supports many different operations, varying from simple to complex. Both *short traversals* and *short operations* access a small part of the graph of objects, most of the time using the indexes to short cut the path. These operations are very fast, executing in average under one millisecond on a modern processor.

On the other hand, *long traversals* sweep most of the graph of objects. The execution of one of these traversals is in the range of seconds (rather than milliseconds as in `short traversals`). I explore new parallelism in `STMBench7` by parallelizing the following read-write long traversals: *t2a*, *t2b*, *t2c*, *t3a*, *t3b*, *t3c*, and *t5*. These operations traverse the graph of objects and perform changes in some of the nodes. Therefore, I create new sub-tasks to explore multiple paths concurrently within the transaction.

There are also *structural modifications* that change the structure of the graph of objects. Contrarily to what is claimed in the paper of the benchmark, these operations degenerate significantly the structure of the graph, as they tend to delete more objects than they add. In my evaluation I always use `STMBench7` with structural modifications disabled, and long-traversals enabled.

Moreover, I modified `STMBench7` to execute a given number of operations instead of executing as many as possible for a given time. Otherwise, two executions with the same parametrization could yield very different results, because of long traversals extending the time to execute the benchmark.

2.5.2 Vacation

The `Vacation` benchmark from the `STAMP` suite [43] implements a travel agency. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. A client performs a set of operations batched in a session. Each session of a client is considered to be an atomic action, and the benchmark measures how long it takes to process a given number of sessions.

This benchmark has three different transactions that perform different clients' requests. In each case an operation is performed multiple times on (possibly) different parts of the objects in the system. Therefore, I parallelized the cycles that repeat the operation for the several requests that compose the session. This means that a transaction can now process multiple requests concurrently in each session.

The benchmark allows parametrizing the level of contention for the objects of the graph. In my

evaluation I consider two scenarios: High contention, which uses 1% of the graph of objects; and low contention, which uses 90% of the graph of objects.

2.5.3 Lee-TM

`Lee-TM` [4] implements Lee's routing algorithm for automatic circuit routing and it measures how long it takes to lay down tracks on a circuit board. Each track is laid down in its own transaction, to ensure that tracks do not overlap in the board. Consequently, this benchmark has only one atomic block identified.

In some boards, laying multiple tracks concurrently may not yield better performance because the tracks are inherently conflicting. Laying down a track consists of an expansion phase and a write-back phase. The first one takes most of the execution time of a typical transaction in `Lee-TM`. So, I parallelized the transaction that lays down the track by creating sub-tasks in the expansion phase, such that the expansion is performed concurrently in different directions.

Chapter 3

Related Work

In this chapter I provide an overview of the concepts related to this dissertation. I have selected the most relevant properties and systems to present in this overview. An extended version of this related work has been provided in a technical report [16].

In Section 3.1, I begin by presenting several properties that define a TM. It is relevant to understand and to compare them as my work shall build on some of them. Then, in Section 3.2, I describe design choices that have been addressed in the related work and that typically characterize a TM implementation.

Then I turn my attention to systems that provide some form of nesting. I divided these depending on the nesting model used. In Section 3.3, I begin with flattening as the simplest approach to achieve nesting and some TMs that use it. In Section 3.4, I continue by presenting linear nesting and implementations of that model. Then, I center the discussion in parallel nesting, for which I present the state of the art in Section 3.5.

I conclude this chapter by discussing the weaknesses of the parallel nesting implementations in the state of the art.

3.1 Transactional Memory Theory and Guarantees

The intense research on TMs has resulted in both practical implementations and theoretical assertions regarding TM. In the next sections I present some of these properties, upon which the implementations are built. In Section 3.1.1, I describe the various correctness criteria that may be used to establish what are the acceptable executions of transactions.

Then, I describe progress guarantees at the level of operations within transactions in Section 3.1.2. In Sections 3.1.3 and 3.1.4, I introduce two progress guarantees (and their variants), that may be used to characterize a TM system as a whole.

Some of these concepts are used later to characterize both existing work and the solution that I propose.

3.1.1 Correctness Criteria

From a user's perspective, a TM should provide a semantics similar to that of critical sections: Transactions should appear to execute sequentially. Yet, a TM implementation would be inefficient if it never allowed transactions to run concurrently. Reasoning about the correctness of a TM implementation implies stating if a given concurrent execution respects that correctness criterion.

Linearizability [35] was initially proposed as a safety property devised for concurrent objects. Here, linearizability means that every transaction should appear as if it took place at some single, unique point in time during its lifespan. Although it has been used for reasoning about TM correctness, linearizability does not entirely suffice as an appropriate correctness criterion for TM. Note that a TM transaction is not a black box operation on some complex shared object, but instead it is an internal part of an application: The result of every operation performed inside a transaction is important and accessible to a user. Therefore it is also important to define what exactly happens in each operation of a transaction. Yet, linearizability only accounts for the execution of the transaction as a whole.

On the other hand, **serializability** [50], which originated in the database transactions, states that the result of a history of transactions is serializable if all committed transactions in it receive the same responses as if they were executed in some serial order, i.e., without concurrency between transactions. Usually, it is said that such a serialization explains the concurrent sequence of operations of that execution. However, serializability does not state any behavior regarding accesses performed by live transactions (specifically, about transactions that may abort). As we will see, such accesses may render harmful if the correctness criterion does not safely prevent them from returning erroneous results.

In [28], Guerraoui and Kapalka argue that these previously described correctness criteria used for other purposes (databases, concurrent objects, etc...) do not fit the needs of TM. In particular, none of them captures exactly the requirement that every transaction, including not yet completed ones, accesses a consistent state, i.e., a state produced by a sequence of previously committed transactions. Whereas a live transaction that accesses an inconsistent state can be rendered harmless in database systems by being aborted, such a transaction might create significant dangers when executed within a general TM.

Suppose that in some program there are two shared variables x and y related by the invariant $x < y$, and consider the fragment of code shown in Listing 3.1. Assuming that initially $x = 5$ and $y = 10$, consider the following concurrent execution of transactions T_1 and T_2 (recall the notation introduced in Section 1.2):

$$R_{T_1}(x,5) \;\; W_{T_2}(x,0) \;\; W_{T_2}(y,4) \;\; C_{T_2}(ok) \;\; R_{T_1}(y,4) \tag{3.1}$$

In such an execution, transaction T_1 read some inconsistent state where the invariant $x < y$ is not respected and the lower bound of the cycle ends up being greater than the upper bound limit. Depending on the execution environment, the consequences may vary, but nevertheless are not acceptable.

To eliminate this problem, the **opacity** correctness criterion [28] requires (1) that transactions that commit look as if they executed sequentially (equivalent to serializability) (2) that aborted transactions

```
int lowBound = x, upBound = y;

for(; lowBound < upBound; lowBound++)
    array[lowBound] = lowBound;
```

Listing 3.1: Example of code that may not be executed properly due to the lack of an appropriate correctness criterion.

are given the illusion of no concurrency, i.e., they must also observe consistent states all the time and (3) that operations executed by an aborted transaction must not be visible to any other transaction. The algorithms that I propose in this dissertation satisfy the opacity criterion to avoid the hazards described.

3.1.2 Operation level liveness

Opacity may be trivially achieved in a TM implementation that aborts every transaction before performing any transactional read or write. Despite its uselessness, it motivates the formalization of progress conditions that capture the scenarios in which a transaction must commit or may be aborted. A simple progress condition that requires a transaction to commit if it does not overlap with any other transaction may be implemented using a single global lock. As a result, transactions will be running one at a time, thus ignoring the potential benefits of multiprocessing and yielding zero concurrency [38]. The objective is to have positive concurrency in which at least some transactions make progress concurrently.

To achieve progress at the level of transactions, it is important to formalize which guarantees should be provided at the level of operations that constitute a transaction. In [34], Herlihy and Shavit present these guarantees, which I briefly summarize next. **Starvation-freedom** states that all threads eventually progress when trying to grab some lock. **Deadlock-freedom** only requires that some thread manages to grab the lock and consequently it may happen that a specific thread never manages to do so.

A non-blocking program is said to be **wait-free** if it ensures that every thread finishes its task in a finite number of steps even if it faces arbitrary delays of concurrent threads. Such events may take place due to blocking for I/O or adverse scheduling by the operating system. **Lock-freedom** only ensures that the system as a whole makes progress: A specific thread may never make progress in face of concurrent threads progressing. The JVSTM [24], which I describe in Section 4.1 and use as the basis of my work, provides lock-freedom as the progress guarantee of its operations.

Yet, there is weaker non-blocking guarantee: **Obstruction-freedom** guarantees that one thread makes progress if it executes in isolation for sufficient time: a transaction T_k executed by thread p_i can only be forcefully aborted if some thread other than p_i executed a step (low level operation) concurrently to T_k. Although it was initially presented as a synchronization mechanism [31], obstruction-freedom has also been used to classify the progress guarantees of a TM system [32, 27]. Formally, if fits in the zero-concurrency category as a transaction is guaranteed to commit only if it faces no contention (theoretically allowing no concurrency).

Like stronger non-blocking progress conditions such as lock-freedom and wait-freedom, obstruction-freedom ensures that a halted thread cannot prevent other threads from making progress. Unlike lock-freedom, obstruction-freedom does not rule out livelock: interfering concurrent threads may repeatedly prevent one another from making progress. Compared to lock-freedom, obstruction-freedom admits substantially simpler implementations that are more efficient in the absence of synchronization conflicts among concurrent threads.

To cope with the possibility of livelock, Herlihy et al [32] proposed that modularized mechanisms could be used to enforce a given policy that seeks to avoid livelocks: Contention managers. These may be queried to decide if a transaction is allowed to abort another one or if it should abort itself instead.

3.1.3 Progressiveness

Some of the most efficient TM implementations internally resort to locking despite providing a lock-free illusion to the programmer. To capture the guarantees provided by these TMs, Guerraoui and Kapalka proposed progressiveness [29], in which a transaction encountering no conflicts must always commit:

- Single-lock progressiveness: A transaction can abort only if there is a concurrent transaction. One TM implementation providing this guarantee has been briefly addressed above (using a global lock).
- Weak progressiveness: A transaction can abort only if a conflict with a concurrent transaction arises in an access.
- Strong progressiveness: Stronger than weakly progressive as it requires that, among a group of transactions whose accesses conflict on a transactional variable, at least one of the transaction does not abort.

Strong progressiveness is the most interesting guarantee. In particular, it means that two independent transactions progress without interfering with each other. Moreover, it does not allow spurious aborts because aborts have always to be explained by some conflict. Finally, it ensures progress for transactions that perform only one transactional access. These may be relevant in TMs with strong atomicity, which wrap non-transactional accesses in single-operation transactions.

3.1.4 Permissiveness

A TM is permissive with regard to a correctness criterion C (where C may be opacity for example) if it never aborts a transaction unless necessary for maintaining safety according to that criterion. Note that a TM may be seen as an online algorithm in the sense that, on each operation that it executes, it has to decide on its influence on the overall correctness of an incomplete transaction with operations that may yet be performed. Ensuring C-permissiveness may yield a very expensive algorithm complexity wise. As a matter of fact, it has been shown that it is impractical to achieve permissiveness deterministically [26].

Therefore, an alternative notion has been suggested in the literature: Probabilistic C-permissiveness [26] in which some randomization takes place that eventually leads to the acceptance of C-safe histories by the TM. The underlying idea builds on the following example: if T_k and T_i access the same transactional variable where T_k writes and T_i reads, even if T_k commits first (but after the concurrent read took place), T_i may still commit if its serialization point is before T_k's. For this to be possible, transactions may adaptively validate themselves by maintaining a possible interval of serialization. At commit time, they randomly choose a point within that interval to serialize themselves, allowing transactions to commit probabilistically in the past, or in the future.

On a slightly different setting one may also use multi-version-permissiveness: a relaxation in which only read-write transactions may abort and in which case it has to conflict with a concurrent read-write transaction [51]. Therefore, read-only transactions must always commit. This guarantee suites TMs that maintain multiple versions of transactional variables. However, it has been shown that single version TMs may also be mv-permissive [6]. The JVSTM is an example of an mv-permissive STM.

3.2 Transactional memory design choices

Among the TM systems that have been proposed, many different design decisions were promoted. To start with, a TM can be implemented either in hardware or software, as well as in various hybrid approaches that dynamically switch between hardware and software execution modes. However, they all have common issues that have been solved very differently across the literature. Here, I present only a few of them, which are building blocks for understanding the algorithms that I describe in my work. I provide a more extensive list of these design choices, as well as how many published STM systems are characterized according to them, in a technical report [16].

3.2.1 Update Policy

The update policy establishes how the system manages both stable (valid when the transaction had started) and speculative values (attempting commit) of the transactional shared variables. The former are used when the transaction aborts whereas the latter are used in case it commits.

One possible strategy is called lazy (also known as deferred) update, in which all writes performed within a transaction are buffered until commit time. These writes may be stored as values in a set or applied to some tentative copy of an object. On commit, these buffered writes are publicized, meaning that they are written to the proper address corresponding to the transactional variable. Conversely, if the transaction aborts, it suffices to discard the local tentative writes.

On the other hand, there are eager (also known as direct) updates that are directly applied to the transactional variable instead of some shadow copy or temporary buffer. A transactional variable whose value belongs to a transaction that has not yet committed, must be somehow marked as tentative by the TM system. This bookkeeping is crucial for concurrent transactions to act correctly when accessing this tentative value. To return the global state to a consistent one when a transaction aborts, the overwritten values must be logged in what is usually referred to as an undo log. This way, upon abort, they may be retrieved and rewritten in the global state, whereas on commit it suffices to clean the undo log.

3.2.2 Conflict detection and resolution

To detect conflicts, each transaction needs to keep track of its read-set and write-set. On one hand, lazy conflict detection and resolution (also referred to as late, optimistic, or commit-time) is based on the principle that the system detects conflicts when a transaction tries to commit, i.e., the conflict itself and its detection occur at different points in time. To do so, one possibility is to have a committing transaction T_i to ensure that no write-set of a recently committed transaction T_k intersects with T_i's read-set. A transaction T_k is recently committed with regard to T_i if it committed after T_i started and before T_i committed. If there is an intersection, there is a read-write conflict, which leads the transaction attempting commit to abort. This strategy promotes more concurrency (by causing less conflicts) because a read-write conflict may not be troublesome if the reader transaction commits before the writer. However, conflicts are detected late, which may result in fruitless computation.

On the other hand, eager conflict detection and resolution is based on the principle that the system checks for conflicts during accesses, i.e., the system detects a conflict directly when it occurs. A mixed invalidation scheme has been proposed [56] to combine eager write-write detection and lazy read-write.

Despite how conflicts are managed, one may perform the conflict detection at different granularities. The most fine-grained one registers memory addresses in the read and write-sets and performs verifications by comparing the memory words. The major drawback is the extensive overhead regarding the fine-grained mapping that is created to cover all the addresses. An advantage is that this strategy avoids false sharing. There is a midterm that requires less time and space, by using cache line granularity, but risks having false sharing which may lead to unnecessary aborts. This strategy is usually applied to HTM systems that can leverage on existing cache coherence mechanisms.

There is an alternative that promotes object granularity, in which the sets maintain the objects whose field(s) are read or written. This strategy may also yield false sharing if two concurrent transactions access different fields in the same object. There is room for different granularities besides the ones mentioned as the conflict detection may employ more than one memory word or cache line.

3.3 Achieving Nesting by flattening

After going through some important properties of TM, I now delve into nesting of transactions. As we have seen, nesting of transactions is a requirement to support software composability. The simplest way to provide it is by flattening transactions into the outermost level. Although simple to achieve, some TM implementations have not provided support even for this model [25, 40, 15, 53, 22, 36, 20]. A possible implementation of such model is for a transaction to maintain a counter with the depth of nesting. This way, instead of creating a nested transaction, the counter is incremented. When a commit is reached, the counter is decremented. The actual commit is only performed when the counter corresponds to the top level.

In this setting, the code that conceptually belongs to a nested transaction is actually behaving as if it were in the top-level transaction. Therefore, all the bookkeeping performed during the accesses is maintained in the top-level transaction's structures. Both DSTM [32] and RSTM [41] are examples of TMs that provide flattening to support composability of transactions.

In DSTM, a transaction may release objects that it has read, effectively removing them from its read-set. Once an object has been released, other transactions accessing that object do not conflict with the releasing transaction over the released object. The programmer must ensure that changes by other transactions to released objects will not violate the linearizability of the releasing transaction: A transaction may thus observe inconsistent state. Clearly, the release facility must be used with care; careless use may violate the correctness criterion.

3.4 Linear Nesting

Almost all recent related work in nesting builds on the model presented by Moss and Hosking [46]. The linear nesting model imposes that a transaction may have only one nested transaction active at a given time (meaning that it is executing). Conversely to flattening, an atomic action enclosed in the control flow of an active transaction T_i will effectively create a nested transaction T_k. The parent of T_k is T_i. The definition is recursive in the sense that a nested transaction is merely a more specific term for a general transaction. Therefore, a nested transaction may also be the parent of another nested transaction. A top-level transaction may now be easily defined as a transaction without a parent.

The set of transactions created (both directly and indirectly) by a top-level transaction constitute a **nesting tree** (which includes that root transaction as well). Moreover, when T_k, a linear nested transaction, attempts to read variable x, it must obtain the value it previously wrote to x. If T_k never wrote to x, then it repeats the same procedure as if it was its parent T_i instead. Otherwise, if T_k has no parent, and thus is top-level, it obtains the globally known value.

As we shall see, we may define two types of nested transactions [46]: Closed and open. However, regardless of its type, a nested transaction accessing a variable will always obtain the most recent value known by itself (in case it has written to it) or by its ancestors. What differs between closed and open nested transactions is what happens when they attempt to commit, and when its ancestors abort.

From a high level point of view, a Closed Nested Transaction (CNT) preserves the isolation of the nesting tree, whereas an Open Nested Transaction (ONT) escapes the enclosing transaction and makes its actions globally known at the end of its execution. This means that an ONT requires possibly expensive measures if an ancestor aborts after the ONT's commit. Therefore an ONT is more suitable for situations in which its ancestors seldom abort.

In more detail, a CNT commit results in the merge of its read-set and write-set with its parent's. This type of nesting is provided in NOrec [14], LogTM [44] and McRT-STM [54]. On the other hand, ONTs, which have been provided in Atomos [13] and XModules [2], allow a committing inner transaction to release isolation immediately: The commit is partially performed as if it was a top-level transaction. This means that the writes are made globally visible and both its read-set and write-set are discarded. However, one of its ancestors may yet abort, in which case the ONT's write-set "escaped" the control of the abort mechanism and was made visible incorrectly. The workaround depends on the TM design. One example may require the nested transaction to propagate an undo-set to its parent [45].

It is not straightforward how some applications can deal with ONTs [2]. The major problem is that opacity is broken because some other transaction may read a value publicized by a transaction that aborts. Moreover, it is also difficult to deal with states in which an ONT commits values that depend on some state that was written by one of its ancestors but is not yet committed. For instance, consider that $y = 2 * x$ where x and y are shared variables in our program. Consider the history:

$$W_{T_1}(x,1) \;\; S_{T_1}(T_2) \;\; R_{T_2}(x,1) \;\; W_{T_2}(y,2) \;\; C_{T_2}(ok) \tag{3.2}$$

In execution (3.2), T_2 is an open ONT and variables are initialized to 0. By the moment that T_2 commits, any concurrent transaction that reads x and y will see an inconsistent state where $y = 2$ and $x = 0$ until T_1 commits. This leakage of uncommitted state is traditionally avoided by the following rule of thumb: An ONTs' footprint (i.e., the union of read-set and write-set) should not intersect with its ancestors' footprint. This approach was proposed as a way to increase concurrency and decrease false conflicts [44] (in the sense that, to the application logic, those conflicts were not relevant). The DSTM's eager release mechanism presented in Section 3.3 is another way of achieving a similar goal.

A common motivation for open nested transactions is based on part of an atomic action that has a very high chance of conflicting with concurrent transactions. For instance, if all transactions in some program have to increment some statistical counters, this indirectly causes one transaction only to succeed, which is the one that manages to commit without its accesses being invalidated. Therefore the usual proposal is that the changes on the counters are encapsulated in an open nested transaction.

3.5 Parallel Nesting in TMs

The linear nested transactions that I have presented in the previous section may be represented in a tree structure: Each transaction is a node; the parenthood relations are established by directed edges from the child to the parent transaction; and the root is a top-level transaction. Given this representation, in linear nesting, only one of the branches of the tree may be active at a given time. Conversely, in parallel nested transactions, we may have an arbitrary number of branches in the nesting tree with active transactions because a parent may have multiple nested transactions active at any given time.

Note that reading a variable in parallel nested transactions works in the same way as for linear nested transactions as previously described in Section 3.4. However, the fact that we may now have parallel siblings and different branches of the nesting tree active at a given time make the implementations more complex in practice: In linear nesting a nested transaction can always assume that the write-sets of its ancestors will never change during the nested transaction lifetime whereas for parallel nested transactions that is not true due to concurrent nested commits.

Next, I describe the most relevant STMs that have parallel nested transactions and in which this difficulty, and other concerns, will be addressed. I provide a more complete list and description of the existing systems in [16].

3.5.1 NeSTM

The Nested STM (NeSTM [7]) is based on the McRT-STM [54] as a blocking, eager conflict detection, word-granularity TM with undo logs for writes and a global version clock for serializability. In the original TM, each address is mapped, using a hashing function, to a variable that acts either as a lock or as a storage for a version number. The former contains the address of a structure with information about the transaction holding the variable whereas the latter contains the global clock version corresponding to the last write applied to the address that is mapped by the variable. Moreover, every transaction is uniquely identified by an identification number.

In the extension of this system to support parallel nesting, the authors argue that the most important point is that it should not interfere with the performance of workloads in which nesting is not used. They were also driven by the intent of keeping the memory footprint as close to constant as possible, regardless of the nesting depth in use. Also, the assumption that no other transaction could access a locked variable in the original system is no longer true: due to the parallel nested transactions, other transactions can correctly access the locked object as long as they are descendants of the owner. To allow this, the ownership information was always made available in the lock to query the ancestor relationship at any time. Similarly, the version number must also be visible at all times to serialize the conflicting transactions. Consequently, the lock variables now reserve some bits to identify the transaction owning it, whereas the rest is used for the version number, allowing invisible readers despite the current lock mode. This leads to two practical consequences: There is a maximum number of concurrent transactions at a given time and the transaction identifier overflows several orders of magnitude faster than normal.

At transaction start, the global clock is used to timestamp the transaction. Reading a variable X causes an abort if X was written since the transaction started. This may cause unnecessary aborts in cases such as: T_i did not perform any access, T_k commits values, T_i reads one of the values written by T_k and, thus, aborts.

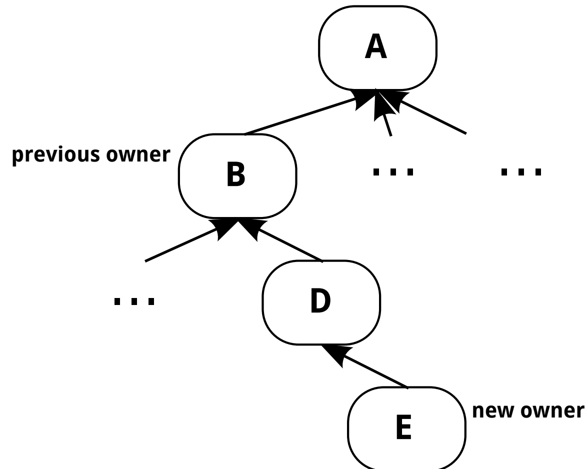


Figure 3.1: Nesting tree in which T_A is the top level transaction. In this example a transactional variable, say X, was held by T_B. When T_D attempts to acquire the ownership of X, it is able to do so because T_B is an ancestor of T_D. Some branches were omitted to simplify the example.

When writing a value, the transaction attempts to acquire the lock corresponding to the variable and then it validates the object: The transaction attempting to write, as well as its ancestors, must not have a timestamp smaller than the object's timestamp, in case they read it previously. To reduce the work needed for this validation, only transactions that were not ancestors of the previous owner of the object must go through the check. In Figure 3.1, I present an example in which the nested transaction T_D attempts to acquire the lock corresponding to a variable that was previously owned by T_D's ancestor. In this case, the validation process will be performed only for T_D and T_C due to the optimization described. Yet, this mechanism yields considerable costs in terms of computation at high depth levels. Given that the nested commit procedure requires validating the reads across the transaction and its ancestors followed by the merge of the sets into the parent, this set of actions must be atomic in the algorithm. This is meant to prevent concurrent siblings from committing simultaneously and breaking serializability. In practice it was solved by introducing a lock at each transaction and make nested transactions acquire their parent's lock in mutual exclusion with their siblings.

Moreover, NeSTM is subject to livelocks: If T_1 writes to X and T_2 writes to Y, they will both have acquired the ownership of those variables. Consider that the first transaction spawns $T_{1.1}$ and the second one spawns $T_{2.1}$. Now, if both of these nested transactions Y and X, respectively, they will abort because those variables are owned by non-ancestors in each case. However, they will have mutually blocked each other unless one of their ancestors aborts as well and releases the corresponding variable. The authors placed a mechanism to avoid this in which they count consecutive aborts and heuristically abort the parent as well.

3.5.2 HParSTM

The Hierarchy-based Parallel STM (HParSTM [37]) is based on Imbs' STM [36], thus obeying opacity and progressiveness. The novelty of this work is that it allows a parent to execute concurrently with its children nested transactions. The advantage of this approach is that it allows more nodes in the transactional tree to be active in computations and requires less depth of nesting due to useless parents standing-by.

The same protocol used for top-level transactions is extended for nesting by replicating most control data structures. The baseline STM design promotes a mixed invalidation strategy with visible readers, lazy lock acquisition and write-back at commit time. To achieve this, it uses a global structure where doomed transactions are registered: When a transaction is writing-back at commit-time, it invalidates those objects' active readers by dooming them in the aforementioned global structure. Any transaction has to check that it does not belong to the doomed transactions prior to commit.

Moreover, this information is also scattered across the shared objects, which have a forbidden set associated to them: if T_1 read X and T_2 wrote X and Y followed by commit, it not only adds T_1 to the global doomed set, but also to the forbidden set of X and Y. If T_1 attempts to read Y it will fail to do so, as otherwise, that would be an inconsistent view state. This procedure is used by nested transactions, except that they must ensure that these invalidation sets contain neither its id or any of its ancestors'. The extension performed for nesting parallel transactions also synchronizes merges in a parent transaction by concurrent siblings (and the parent's execution itself) with mutual exclusion.

3.5.3 PNSTM

The Parallel Nesting STM (PNSTM [9]) provides a simple work-stealing approach with a single global queue into which the programs' blocks may be enqueued for concurrent transactional execution. Moreover, each transactional object is associated with a stack that contains all the accesses (both reads and writes) performed by active transactions. This allows transactions to determine eagerly and in constant time if a given access to an object conflicts with a non-ancestor's access.

To achieve constant time queries for eager conflict detection, a set of transactions is represented by a memory word that has each bit assigned to a transaction (called a bitnum). This way, when T_i accesses a variable last accessed by T_j, a conflict is detected by operating on both transactions' bit vectors and deciding if one of them is ancestor of the other using bitwise operations: Assuming vec_i is the bit vector corresponding to T_i, we have a conflict when:

$$\overline{vec_i} \wedge (vec_i \bigoplus vec_j) \neq 0 \tag{3.3}$$

In Fig. 3.2 I present an example where these operations are used to validate accesses to a variable X by various transactions of a nesting tree. Each transaction in the tree has a corresponding bitnum to identify it, which refers to a position in a memory word whose bit is set to one. In this example, the access by transaction T_D creates a conflict because the last access to that variable was performed by a non ancestor of T_D. This is visible in the bit vectors in the access stack.

Note that using a memory word for this representation allows performance improvements but limits the maximum number of transactions on the system at all times. To work around that, the authors of PNSTM introduced the concept of epochs, such that a transaction identifier only has meaning when paired with the corresponding epoch. Moreover, the system would be limited to a given maximum number of concurrent transactions. The authors claim that no more parallelism would be attained over that limit if it is larger than the maximum number of worker threads. Consequently, they build on that assumption and provide some ways of reusing identifiers and making it harder to reach the limit.

When a transaction commits, it leaves behind traces in all the objects it accessed, namely the stack frames stating its ownership. To avoid having to go through all the objects in the write-set by locking

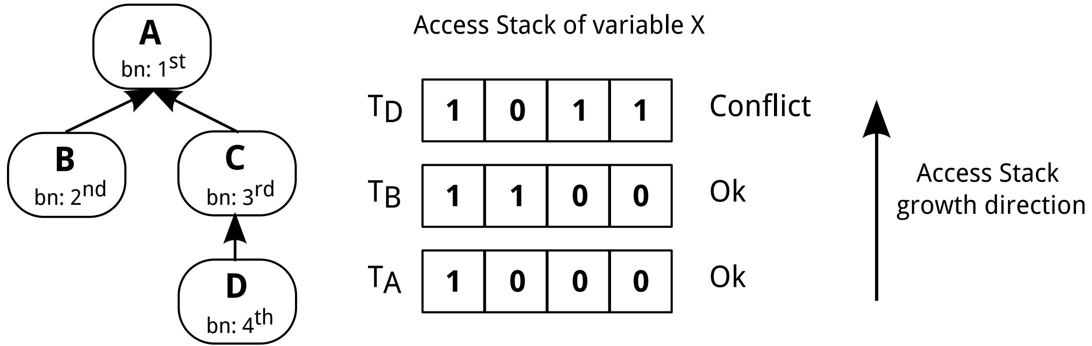


Figure 3.2: Nesting tree in which transactions are identified by bit nums according to the PNSTM. The access stack for variable x is shown when T_A, T_B, and T_D access x in that order. Note that the last access creates a conflict.

and merging the frame with the previous entry, PNSTM does that lazily. This may lead to false conflicts when some transaction accesses an object and finds an entry in the stack that corresponds to an already committed but not yet reclaimed transaction. The authors show that it is possible to avoid it by resorting to a global structure maintaining data about all committed transactions and some lazy cleaning up.

3.6 Discussion of existing Parallel Nesting Implementations

Providing nesting models along with inner parallelism may unveil yet more concurrency in our programs. We have seen that the NeSTM presented many of the difficulties that come up when providing nested parallelism, but some of them (which may break opacity) were solved only heuristically. On the other hand, the HParSTM design informally proved some guarantees that I have described but their authors did not present any evaluation. It is likely that some of the global structures they used inhibit scalability as it breaks the disjoint-access parallelism property and are intensively used for conflict detection. Finally, the PNSTM provided an efficient algorithm but it regards all accesses as writes, thereby precluding some read-read potential concurrency. Moreover, their algorithm did not take into account any way to shrink the stack of accesses in transactional variables.

There are also some parallel nesting implementations that center their attention into how parallel nesting will actually be used by the programmer with seamless integration of thread creation and transaction nesting. For instance, the SSTM [52] explored a unique perspective in which nested transactions may interfere with each other's outcome. However, their algorithm is not provided in a detailed manner and is more interested on how to make use of the underlying runtime of choice. The CWSTM [1] also took into account the composition of atomic and parallel blocks in the language. In addition to that, it was the first one to show an algorithm that was independent of the nesting depth, but it did not provide any implementation or evaluation.

All these STMs are lock-based and single-version. In this dissertation I propose a parallel nesting algorithm for a lock-free multi-version TM, the JVSTM. Moreover, the JVSTM uses a lazy write-back strategy. This strategy has been pointed out as an obstacle for a parallel nesting algorithm independent of the nesting depth [1]. However, the lazy write-back nature also allows for interesting gains by permitting more concurrency. Consequently, there is a trade-off as we necessarily have to perform work with

complexity proportional to the nesting depth, if we want to preserve the benefit of the lazy write-back nature. In Section 4.3 I explain why we cannot escape that complexity. In the final solution provided in Chapter 6 I tackle the overheads of the parallel nesting algorithm so that the constants behind the complexity bound are small enough to make it practical for use.

Chapter 4

A naive algorithm

In this chapter I lay out an initial algorithm for parallel nesting. Given that it is based on existing work, I begin by presenting the underlying STM used in this dissertation, the JVSTM, in Section 4.1. As a consequence of this existing work, all the implementations that I used in this dissertation were written in the Java programming language.

To understand fully the algorithms that provide parallel nesting, I introduce its model in Section 4.2. Then I provide an initial solution to the problem in Section 4.3. This solution gives an insight into the design decisions that arise when creating a parallel nesting algorithm. In the end of this chapter, I identify some challenges that I address in Chapter 5, where I improve over this initial algorithm.

4.1 JVSTM

The Java Versioned STM [24] is a word-based, multi-version STM that was specifically designed to optimize the execution of read-only transactions: In the JVSTM, read-only transactions have very low overheads, and never contend against any other transaction. In fact, once started, the completion of read-only transactions is wait-free in the JVSTM. To achieve this result, JVSTM uses the concept of Versioned Box (VBox) to represent transactional locations. Each `VBox` holds a history of values for a transactional location, by maintaining a list of bodies (`VBoxBody`), each with a version of the data. The access to `VBoxes` is always mediated by a transaction, which is created for that sole access if none is active at that moment.

I show a representation of these concepts in Figure 4.1. The `VBox`, corresponding to a transactional location, points to a history of three values, among which the value 2 is the most recent. This value was

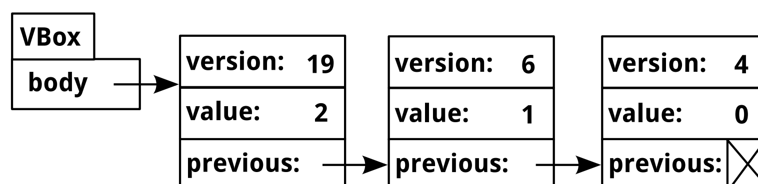


Figure 4.1: A transactional location representing a counter and its versions in the JVSTM.

written to the VBox during the commit of a top-level transaction, in a process called write-back. During this process, a global clock is incremented and used to timestamp the new versions being written-back.

A read-only transaction always commits successfully in the JVSTM because it reads values in a version that corresponds to the most recent version that existed when the transaction began. Thus, all reads are consistent and read-only transactions may be serialized in the instant they begin, i.e., it is as if they had atomically executed in that instant. This means that, in the previous example, a read-only transaction that started on version 17 and attempts to read that VBox, will obtain the version 6 with value 1.

Read-write transactions, however, must be serialized when they commit. Therefore, they are validated at commit-time to ensure that values read during their execution are still consistent with the current commit-time, i.e., that values have not been changed in the meantime by another concurrent transaction.

Transactions mediate all accesses to VBoxes because they need to record each transactional access in their local logs: Reads are logged in a transaction's read-set, whereas writes are logged in a transaction's write-set. Both logs are used at commit time: If the read-set is still valid, the tentative writes logged in the write-set are written back, producing a new version for each of the boxes written and effectively publicizing the new values.

There is a global queue of ActiveTransactionsRecords in which transactions enqueue to obtain their order of commit. A transaction only reaches this point if it validated against all past committed transactions as well as against transactions enqueued before it but not yet committed. After the enqueue, a transaction is guaranteed to commit and that happens with the help of other transactions waiting for their turn to commit, resulting in a lock-free algorithm [24].

4.1.1 Optimizations to the read-set and write-set

The JVSTM is thoroughly described in [24]. Yet, the version that I started with contains further optimizations. In particular, some optimizations were performed to avoid the overheads of maintaining read-set and write-set, which is of great relevance to the algorithms that I provide in this dissertation.

Read-write transactions have to register their reads for late validation, and writes for write-back. The general idea is that this should be made as lightweight as possible, as it accounts for the bulk of the work that is performed during the execution of a read-write transaction.

A read-set was previously maintained as a HashMap that mapped VBoxes to the VBoxBody read at that time. Consequently, each read operation entailed an insertion in the map that caused allocation of memory in the inner workings of the HashMap. Therefore, the new approach reuses the read-sets in transactions that execute in the same thread. This strategy is represented in Figure 4.2. Each thread that executes transactions in the JVSTM keeps a thread-local pool of arrays. These arrays are available for reuse as part of the read-set of the transaction. When the transaction starts, it fetches one of these arrays, and populates it with the VBoxes read during the execution. If it runs out of space, it fetches another array from the pool. When the pool is empty, it allocates a new array and uses it.

At the end of the transaction, the arrays used are returned to the pool. As a result, the most frequent case is that no allocation is required for read-sets during the execution of a transaction. Finally, only references to the VBoxes read are registered: The validation simply checks if the most recent VBoxBody of a VBox read has a version that is smaller or equal to the starting timestamp of the transaction. Therefore it is not needed to maintain a mapping of VBox to the VBoxBody read.

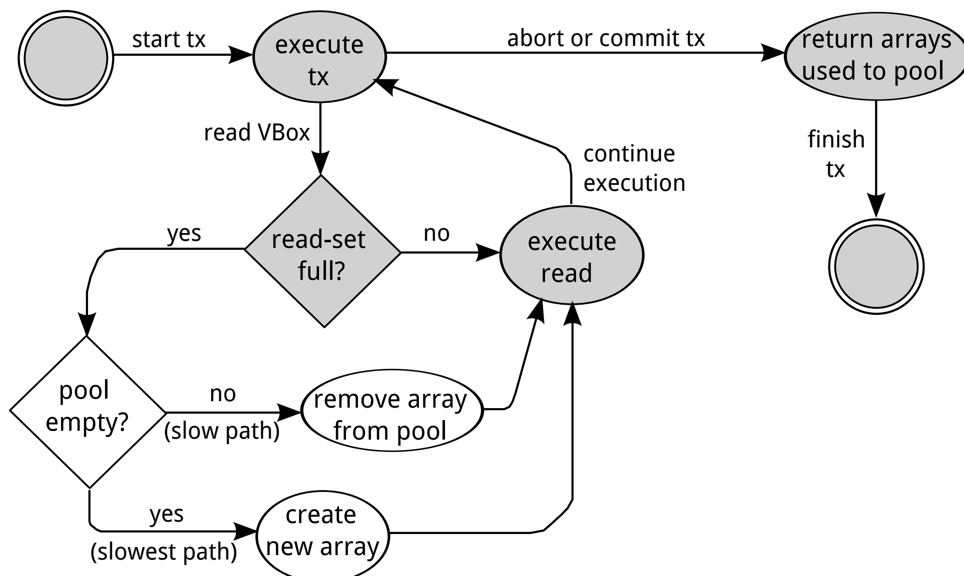


Figure 4.2: Partial representation of the states representing the actions related to the read-set maintenance throughout an execution of a transaction. The states in the fast-path are shadowed.

The write-set was similarly maintained as a `HashMap`, but in this case it mapped `VBoxes` to `Objects`, which represented the values tentatively written. Once again, the concern is that a write would entail allocations and resizes of the map. However, the same strategy used before is no longer adequate to the write-set: When a transaction reads a variable, it has to check if it ever wrote to it previously, in which case it is in the presence of a Read-after-write (RAW). If the write-sets were optimized the same way as the read-sets, this would require traversing the arrays of the write-set on every read. Consequently, a different strategy was required.

This time the rationale for a change is to find a way to avoid looking up the write-set on as many reads as possible. The solution is to perform the writes in the `VBoxes` themselves, for which reason a new field (`tentVal`) was added to them, representing a tentative value. The idea is that in the normal (and fast) case, a transaction is able to write to the `VBox` in the `tentVal` slot. To do so, another field was added to the `VBox`: The `owner` field contains an Ownership Record (Orec) that represents the transaction controlling the `VBox`. This way, a transaction has to acquire ownership of the `VBox` first to be able to write in the `tentVal`. If a transaction fails to write in this fast manner, it resorts to the traditional write-set with the `HashMap`.

This optimization to the write-sets is shown in Figure 4.3, which corresponds to a partial execution of transactions T_A and T_B:

$$R_{T_A}(x,10)\ \ W_{T_A}(x,20)\ \ R_{T_B}(y,3)\ \ W_{T_B}(y,4)\ \ R_{T_B}(x,10)\ \ W_{T_B}(x,20) \tag{4.1}$$

Initially, T_A performs its write to `VBox` x in the tentative slot, because it is able to change the `owner` of that `VBox` to its `Orec`. Similarly, T_B writes to y in the `tentVal` slot. However, once it attempts to write to x, it is not able to do so in-place, because `VBox` x is currently owned by a transaction indicated as alive by its `Orec`. Therefore T_B performs its write in the plain old write-set. The transaction that commits successfully will perform the write-back that entails creating the new body versions, including the writes performed in-place. Consequently, `VBoxes` written in-place are registered in a simple list, referred to as `ws-inplace` in the transaction structure.

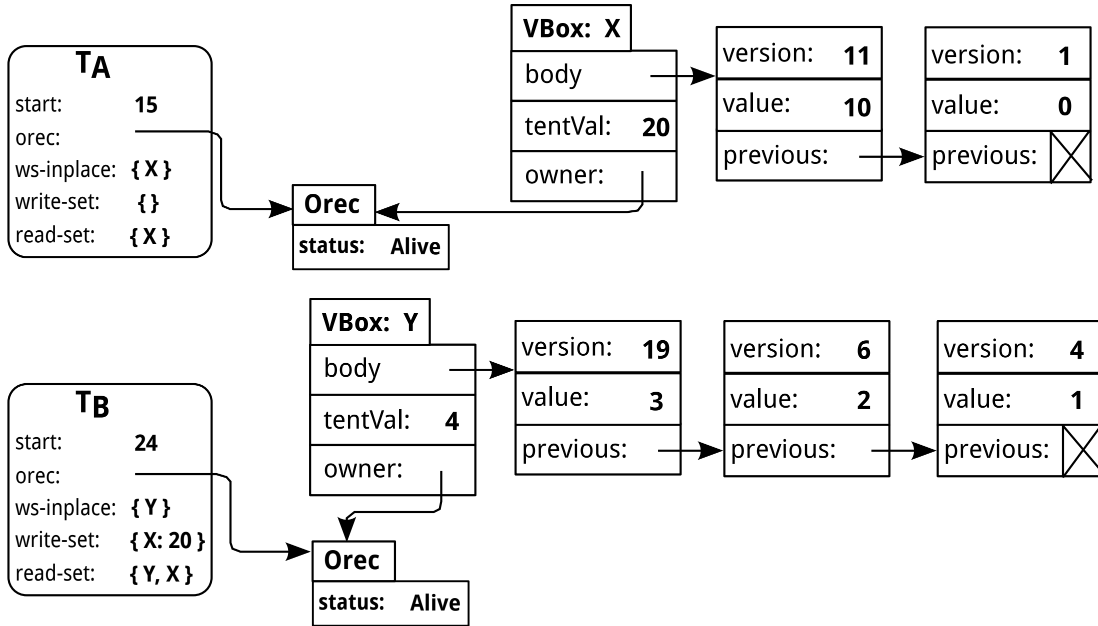


Figure 4.3: Representation of the state maintained in the JVSTM with the optimizations.

4.1.2 Nesting in the JVSTM

The original design of the JVSTM implements a linear nesting model, in which a thread that is executing a transaction may start, execute, and commit a nested transaction (which itself may do the same), effectively forming a nesting tree with only one active branch at a time. The leaf of that active branch represents an active nested transaction that is guaranteed to be the only one accessing and modifying the read-set and write-set of that nesting tree.

Overall, the existing approach was meant to simplify the algorithm and make it easy for the JVSTM to obtain nesting with the least overheads possible and still coping with the optimizations described above. Yet, this simple model does not allow the decomposition of long transactions into concurrent parts, which I provide as one of the major contributions of this dissertation in Section 4.3 and improve in Chapter 5.

4.2 Parallel Nesting Model

I have briefly introduced parallel nesting in Section 3.5 based on previous work regarding TM implementations that provided it. Yet, I provided mostly intuitions because there has not been a specific attempt to define them more formally for parallel nested transactions. Consequently, I now present those intuitions more carefully so that they serve as the basis for my work, with the model of closed nesting described by Moss [46] underlying it.

Two nested transactions are said to be **siblings** if they have the same parent. In parallel nesting, siblings may run concurrently. In this model, each top-level transaction may unfold a nesting tree in which a transaction performs transactional accesses only when all its children are no longer active.[1]

[1] This restriction simplifies the model and does not impose any significant limitation to the expressive power of the parallel nesting model, because a transaction that needs to execute concurrently with its children may spawn a nested transaction to execute its own code.

In a closed nesting model, a nested transaction maintains its own read-set and write-set, much in the same way of a top-level transaction. Yet, given the compositional nature of transactions, reading a transactional location within a nested transaction must always access the value that was most recently written to that location among the following: (1) the sequence of operations performed by the transaction reading, (2) by all of its ancestors, and (3) by its siblings that committed before the read.

However, each read must take the following in consideration: When a nested transaction T_i finds out a write in its ancestor T_k private write-set it is not necessarily guaranteed that it is safe to read it. Consider the following execution:

$$W_{T_k}(x,1) \;\; S_{T_k}(T_i,T_j) \;\; R_{T_i}(x,1) \;\; W_{T_j}(x,2) \;\; C_{T_j}(ok) \;\; R_{T_i}(x,?) \tag{4.2}$$

In this example, the last read performed by T_i would find the value 2 for x in T_k but returning it would break the correctness criterion (assuming opacity described in Section 3.1.1). The alternatives are to return the value 1 if the TM is multi-version or to abort T_i. This example unveils another important point in parallel nesting: In addition to the requirements imposed by opacity (described in Section 3.1.1), the concurrent executions of a set of siblings must be safe with regard to their parent, in the sense that those executions must be equivalent to some sequential ordering. This concept may described as opacity on a level-by-level basis (similarly to level-by-level serializability [60]).

The **set of ancestors** of a transaction with parent T is composed by adding T to the set of ancestors of T. A top-level transaction has no parent, and, therefore, its ancestor set is empty. A closed nested transaction commits by publicizing its footprint into the parent. This may be achieved by merging its read-set and write-set with its parent's read-set and write-set, respectively. The merge of the write-set overwrites possible duplicates when the nested transaction commits writes to locations that its parent had also written to. If it aborts, it may rollback only the atomic action corresponding to itself rather than the whole top-level transaction, depending on the conflict that caused the abort.

4.3 An initial approach towards parallel nesting

A simple design for implementing closed nesting is to maintain a read-set and a write-set on each nested transaction. Then, when a nested transaction commits, both its read-set and write-set are merged into the parent's respective sets, overwriting any previous writes of the parent for the same transactional locations. Overall, this is the design used by the original JVSTM to implement its linear nesting model, and a similar approach may be used for implementing parallel nesting together with synchronization of siblings and validation of commits to respect the correctness criterion.

Using this idea, I created an initial algorithm that provides support for parallel nesting in the JVSTM. In the remainder of this dissertation, I shall refer to this algorithm as the `Naive` algorithm. Next, in Section 4.3.1, I describe some data-structures and functions used in this algorithm. Then, I present the transactional operations in `Naive`, in Section 4.3.2.

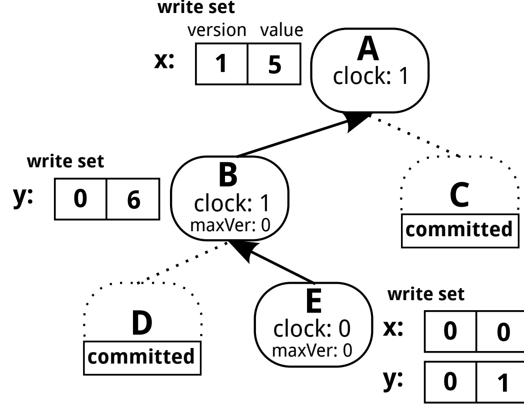


Figure 4.4: Representation of a nesting tree where some parallel nested transactions have performed transactional writes on `VBoxes` x and y with `Naive` write-set design.

4.3.1 Data-structures and auxiliary functions

To better understand the algorithm, I begin by presenting the data-structures that it uses. In Figure 4.4, I show a representation of the nesting tree that is generated by the following execution:

$$S_A(B,C) \;\; W_B(y,6) \;\; W_C(x,5) \;\; C_C(ok) \;\; S_B(D,E) \;\; W_E(x,0) \;\; C_D(ok) \;\; W_E(y,1) \tag{4.3}$$

This nesting tree shows that each transaction that is alive has a corresponding private **`writeSet`**. A `writeSet` maps each `VBox` written to a **`WriteEntry`**, which contains: (1) the **`version`** in which the write exists; and (2) the tentative **`value`** written.

The version that exists in each write is used to enforce consistent reads of tentative values in the nesting tree. For that, every transaction maintains a **`nestedCommitClock`** (shortened to `clock` in the Figure), which is a number representing the latest version that has been committed into the transaction by its children. This means that, when a transaction starts, its `nestedCommitClock` is set to zero, and it is incremented whenever one of its children commits. That is why the clocks of A and B have been incremented in Figure 4.4.

Nested transactions also keep the **`maxVersionOnParent`** (shortened to `maxVer` in the Figure), which is a timestamp obtained when a transaction starts by reading the `nestedCommitClock` of its parent. This timestamp is used to restrict a nested transaction from reading tentative writes committed in the nesting tree after the transaction started.

Moreover, all transactions maintain a **`startingVersion`** that limits the versions that they can read from globally consolidated bodies in `VBoxes`. A nested transaction always uses the `startingVersion` of its root ancestor.

Finally, a nested transaction also contains a mapping of `VBox` to `VBoxBody` called **`bodiesRead`**, similarly to a top-level transaction. This is possible because a `WriteEntry` extends the structure of a `VBoxBody`, and therefore nested read-after-writes and normal reads are registered both in the `bodiesRead`.

In the following description of the algorithm, I also resort to the `abortUpTo(tx, conflicter)`

function, which causes the execution of the algorithm to stop and leads to the abort of `tx` as well as its ancestors up to `conflicter` (including).

4.3.2 The **`Naive`** algorithm

In Algorithm 1, I present the read and write operations of the `Naive` parallel nesting algorithm. The `Write` operation registers the new `WriteEntry` in the private `writeSet`, timestamping it with its current `nestedCommitClock` (lines 30-31).

The `Read` operation starts by using `Lookup` to check for a possible RAW (line 18). If none is found, it reads a globally consolidated body, which may cause an early abort due to a W-R conflict with another top-level transaction (line 23). In that case, the whole nesting tree has to abort.

`Lookup` verifies if `tx` owns a write to `vbox`. Note that the `Lookup` function is recursive: In its first invocation $tx = requester$, meaning that it is checking for a RAW in the write-set of the transaction performing the read. If none is found, it invokes `Lookup` again if `tx` has a parent (lines 13-14).

Algorithm 1 Algorithms for the read and write operation of the `Naive` design.

```
 1: Lookup(tx, vbox, maxVersion, requester):
 2: writeEntry ← tx.writeSet.get(vbox)
 3: if writeEntry ≠ NONE then
 4:    if writeEntry.version > maxVersion then
 5:       // eager W-R conflict detection; abort up to given ancestor
 6:       abortUpTo(requester, tx)
 7:    end if
 8:    if tx ≠ requester then
 9:       requester.bodiesRead.put(vbox, writeEntry)
10:    end if
11:    return writeEntry
12: end if
13: if tx.parent ≠ NONE then
14:    return Lookup(tx.parent, vbox, tx.maxVersionOnParent, requester)
15: end if
16: return NONE

17: Read(TX, VBOX):
18: writeEntry ← Lookup(tx, vbox, tx.nestedCommitClock, tx)
19: if writeEntry = NONE then
20:    body ← vbox.body
21:    value ← body.value
22:    if body.version > tx.startingVersion then
23:       abortUpTo(tx, TOP)
24:    end if
25:    tx.bodiesRead.put(vbox, body)
26:    return value
27: end if
28: return writeEntry.value

29: Write(TX, VBOX, VALUE):
30: newWriteEntry ← (value, tx.nestedCommitClock)
31: tx.writeSet.put(vbox, newWriteEntry)
```

When a write entry is found, a check must be performed to ensure a consistent read (line 4). For that, the argument of the function, `maxVersion`, is used: The `WriteEntry` being read should not have a version larger than `maxVersion`, otherwise the `requester` is aborted, as well as every ancestor of `requester` up to `tx` (including). Note that `Lookup` is invoked with the `maxVersionOnParent` that matches the `tx` passed in the first argument (line 14). Line 8 avoids registering the RAW for later validation in the case of the first recursion of `Lookup`.

Finally, the commit operation (shown in Algorithm 2) ensures that the read-set has no stale entries (lines 3-12), and then it merges the footprint into the parent (lines 13-20). Reads performed over writes of the parent need not be propagated anymore, and thus are removed (line 6). Moreover, every write propagated has its version changed to match the timestamp acquired from the parent (lines 16 and 18). Note that this procedure is performed in mutual exclusion with concurrent siblings (lines 2 and 22).

In such a naive approach, however, the commit of a nested transaction performs work proportional to $O(R+W)$, where R and W are the sizes of the read-set and write-set of the committing nested transaction, respectively. Whereas the propagation of the write-set may be implemented in a more efficient way, a TM with a lazy write-back cannot avoid the cost of validating its read-set, as claimed by [1]. Thus, the component of the cost corresponding to R cannot be eliminated in this approach.

Yet, in this design there is a more important challenge to address, which is the cost of the read operation. To read the value of a `VBox`, the read operation not only needs to check the write-set of the current transaction, but it must also check recursively the write-set of each of the transaction's ancestors until a write is found, returning a globally committed value from the `VBox` if no tentative write is found. This means that a read made on a nested transaction at depth d has, in the worst case, to check the private write-sets of d transactions. Only when we are in the case of a read-after-write (i.e., we are reading from a `VBox` that was written to before) may the cost be lower, because the algorithm stops

Algorithm 2 Algorithm for the commit operation in the `Naive` design.

```
1: Commit(TX):
2: tx.parent.lock()
3: for (vbox, entryRead) in tx.bodiesRead.entries() do
4:    newEntry ← tx.parent.writeSet.get(vbox)
5:    if newEntry = entryRead then
6:       tx.bodiesRead.remove(box)
7:       continue
8:    end if
9:    if newEntry ≠ NONE then
10:      abortUpTo(tx, tx.parent)
11:   end if
12: end for
13: for (vbox, entryRead) in tx.bodiesRead.entries() do
14:    tx.parent.bodiesRead.put(vbox, entryRead)
15: end for
16: commitVersion ← tx.parent.nestedCommitClock + 1
17: for (vbox, writeEntry) in tx.writeSet.entries() do
18:    writeEntry.version ← commitVersion
19:    tx.parent.writeSet.put(vbox, writeEntry)
20: end for
21: tx.parent.nestedCommitClock++
22: tx.parent.unlock()
```

looking as soon as a write is found. So, to have a better idea about the average cost of a read operation in this design, we need to know how often reads correspond to read-after-write operations.

To assess this, I ran several benchmarks with varying workloads, and counted the total number of reads performed and, of those, how many were read-after-write operations. In these results I do not take into account the reads performed by read-only transactions. In Table 4.1 I show the results that I obtained, where the last column shows the percentage of read operations that are read-after-writes. The majority of the workloads tested contain very few read-after-writes—that is, a read operation will almost always have to pay the cost of doing $O(d)$ operations, where d is the depth of the nesting tree. So, the worst case of the read operations is also its average case.

[benchmark]-[workload]	reads ($*10^3$)	raws ($*10^3$)	%
bench7-r-notrav	8000	31	0
bench7-rw-notrav	9000	34	0
bench7-w-notrav	5000	19	0
bench7-r	83000	45	54
bench7-rw	109000	65000	59
bench7-w	127000	71000	56
lee-mainboard	507000	3	0
lee-memboard	281000	2	0
lee-sparselong	67000	0	0
lee-sparseshort	1000	0	0
vac-reservations	84000	0.2	0
vac-deletions	33000	600	1.8

Table 4.1: Total number of reads and read-after-writes (raws) performed in `STMBench7` (with various workloads), in `Lee-TM` (with various boards), and in `Vacation` of the `STAMP` suite (with various workloads).

Chapter 5

A lock-free algorithm

In the last chapter I described some of the challenges that may prevent parallel nesting from yielding performance benefits. One of the major challenges is the read operation in a parallel nested transaction, which may turn out to be an expensive operation in the average case.

To tackle the problem of the high cost of the read operation on a nested transaction, I propose a different design: Make all the transactions within a nesting tree maintain their write entries in a single shared structure, stored at the top-level transaction. The underlying motivation for this design is that this allows for a one-time, depth-independent check to answer the question raised when a nested transaction attempts to read a `VBox`: Does any ancestor have a private write entry for that `VBox`?

In Section 5.1, I present the data-structures used in this algorithm. Then, I describe the operations that compose this algorithm from Section 5.2 to 5.6. After that, I explain the implementation of this algorithm in the Java Memory Model, in Section 5.7. Finally, I discuss the challenges solved and new ones posed by this algorithm, in Section 5.8. I also described this algorithm in an article accepted in *TRANSACT 2012* [18].

5.1 Data-structures

This algorithm reuses some of the structures presented in `Naive`. In particular: Every transaction maintains a write-set (now called **`privateWriteSet`**); the `nestedCommitClock`; the `startingVersion`; and the `bodiesRead`. Moreover, the reification `WriteEntry` is used once again, but with more contents.

In the previous `Naive` design, putting a value in a `VBox` entailed buffering it in the private write-set. In this new design I perform an additional step to ensure that it is possible to answer the read-after-write question in a single operation regardless of the nesting depth.

Besides inserting the new tentative value in the private write-set of the transaction, writing to a `VBox` also entails making this write available to all transactions that belong to the same nesting tree. The shared data structure that allows this is the **`sharedWriteSet`** (a `ConcurrentHashMap`), which, similarly to the **`privateWriteSets`**, allows associating write entries to `VBoxes`. But, whereas the private write-sets of each transaction map a `VBox` to a single tentative write, the `sharedWriteSet`

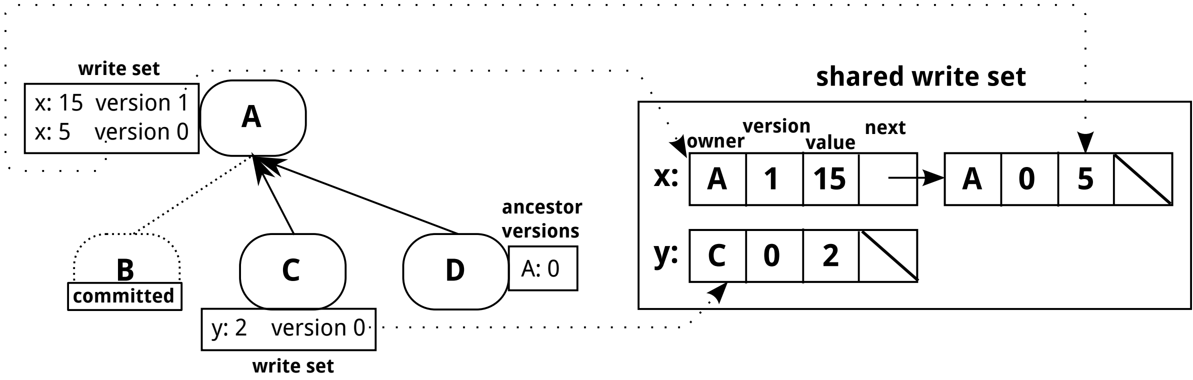


Figure 5.1: Nesting tree corresponding to the execution (5.1).

maps each VBox that has been written in the nesting tree to all the write entries that were tentatively written to it in that nesting tree.

Figure 5.1 details the sharedWriteSet structure by representing the state corresponding to execution (5.1). In this example there are two VBoxes written, mapped to their respective write entries.

$$W_A(x, 5) \quad S_A(B, C, D) \quad W_B(x, 15) \quad W_C(y, 2) \quad C_B(ok) \quad R_D(x, 5) \tag{5.1}$$

Figure 5.1 also shows that transaction D has a structure called **ancestorVersions**,[1] which is used to ensure consistent reads of the versions available in the sharedWriteSet. A nested transaction creates its ancestorVersions as follows: When a nested transaction is spawned, it computes the new map by adding the parent's current nestedCommitClock to the parent's ancestorVersions to obtain its own ancestorVersions. The ancestorVersions represent the restrictions that a nested transaction has on the view of the nesting tree's write entries that it may read.

To provide an overview of how the ancestorVersions is used to ensure consistent reads, consider once again execution (5.1). Assume that D is a read-only parallel nested transaction. When D attempts to read VBox x, it must obtain one of the write entries present in its ancestor's private write-set. In this event, the write entry obtained is restricted by the maximum version that D may ever read from A, given by the mapping contained in its ancestorVersions: It states that D can read entries owned by A with version up to 0 (including it). Therefore, in this example, it reads the entry corresponding to version 0, with value 5.

On the other hand, when the read-write transaction C tries to read VBox x, it will behave differently. Being a read-write transaction means that its linearization point takes place at the time of commit and not at the time of start, which is what happens for read-only transactions. But, given that C's ancestorVersions is equal to D's (because they were spawned simultaneously), C can read at most version 0 also. As version 0 is no longer the most recent for VBox x on the write-set of ancestor A, that means that C will not be able to commit and thus aborts eagerly.

All of this is possible only because the sharedWriteSet contains reifications of write entries with

[1]Every parallel nested transaction has its own ancestorVersions but Figure 5.1 omits some of them for simplicity of presentation.

some metadata associated to them. In addition to the `version` and `value`, the `WriteEntry` now also contains:

- **`owner`**: The transaction that currently owns the write.
- **`next`**: A pointer to a previous version written to this `VBox` in the same nesting tree.

5.2 Reading from a VBox

As we have seen before, the read operation may return a tentative write performed by a transaction in its nesting tree (thus being a RAW), or a globally consolidated value committed by some top-level transaction. The previous section described that the tentative writes of a nesting tree are now all placed in the same structure, the `sharedWriteSet`. This means that, in this algorithm, a read has to first look into the `sharedWriteSet` (instead of several private write-sets), and only then will it read some `VBoxBody` (if it was not a RAW).

The set of write entries applied to a `VBox`, available in a `sharedWriteSet`, forms a linked list in which the most recent write entry is at the head of the list. The key point in this structure is that we can rely on the following invariant: While traversing the write entries of a `VBox` during a read operation, as soon as we reach a write entry that may be read, then it is guaranteed that no other entry further down the list has to be read instead of that one.

Instead of having a depth-dependent lookup, this new design has a lookup that depends on how many writes contend for the same `VBox` in a given nesting tree. This means that this design moved from a worst-case $O(d)$ lookup, where d is the depth of the nesting tree, to $O(n)$ where n is the number of transactions in a nesting tree and $d \leq n$. This worst-case applies to a lookup to a `VBox` x by a nested transaction T when all other transactions in T's nesting tree also wrote to x after T did so.

Yet, I claim that the use case for parallel nested transactions is that the sub-transactions that compose it must have somewhat disjoint accesses so that the work may be effectively parallelized. Typically, a transaction does not write arbitrarily to transactional variables without performing some reads.[2] Therefore, it is reasonable to assume that, to parallelize a transaction without having many conflicts among its parallel nested transactions, the nested transaction's read-set and write-set must not intersect that often. This relationship between the read-set and write-set leads to the conclusion that if the nested transactions do not conflict that much and are, therefore, able to run in parallel efficiently, then each `VBox` will have very few writes contending for it. Consequently, the length of the linked list of write entries for a given nesting tree is often very short despite the depth of the nesting tree, making the average-case $O(k)$ where $k \ll d, n$.

As I explained in Section 4.3, I claim that looking up the value of a `VBox`, for which there is no write in the nesting tree, is the common case. Contrarily to the `Naive` design, this operation is very cheap in this new design: The list of write entries is empty and the lookup returns immediately. There is also the case when there are some writes, but these still required looking into different write-sets in the `Naive` design. Now, the traversal of those writes may also be considerably faster because the most recent write entry is at the top of the list, and therefore most lookups in a read-after-write scenario are satisfied with that entry and end quickly because they need not check the rest of the list due to the invariant.

[2] I shall address this point again, in Section 6.1

In Algorithm 3, I present the functions used to perform a read in a parallel nested transaction. The `GetBoxValue` function is the starting point, which first checks for a possible RAW (using `getAncestorWrite`). If there is none, then it uses `readFromBody` to obtain a consolidated value from the `VBox`.

This algorithm differs from `Naive`'s read operation in the `getAncestorWrite` when the `sharedWriteSet` is used. While iterating over the write entries for the given `VBox`, the transaction may find entries that belong to other branches of the nesting tree. In other words, some of those write entries will not be owned by an ancestor of the nested transaction and, therefore, cannot be read.

The iteration over the write entries stops when a transaction T attempting the read finds a previous write entry belonging to an ancestor of T in lines 12-21. Line 14 checks that the version of the write entry cannot be greater than the maximum version that T can read from the ancestor that owns that entry. Recall that this information is stored in the `ancestorVersions` that is created for each transaction

Algorithm 3 Algorithm to retrieving the value of a `VBox` in the `SharedWS` design for a read-write transaction.

```
 1: GetBoxValue(tx,vbox):
 2: value ← getAncestorWrite(tx, vbox)
 3: if value = NONE then
 4:    value ← readFromBody(tx, vbox)
 5: end if
 6: return value

 7: getAncestorWrite(tx,vbox):
 8: iterWriteEntry ← tx.sharedWriteSet.get(vbox)
 9: while iterWriteEntry ≠ null do
10:    // is the owner an ancestor of mine?
11:    ancestorVersion ← tx.ancestorVersions.get(iterWriteEntry.owner)
12:    if ancestorVersion ≠ NONE then
13:       // it is an ancestor, but can I read that version?
14:       if iterWriteEntry.version ≤ ancestorVersion then
15:          tx.bodiesRead.put(vbox, iterWriteEntry)
16:          return iterWriteEntry.value
17:       else
18:          // eager W-R conflict detection; abort up to given ancestor
19:          abortUpTo(tx, iterWriteEntry.owner)
20:       end if
21:    end if
22:    iterWriteEntry ← iterWriteEntry.next
23: end while
24: return null

25: readFromBody(tx,vbox):
26: body = vbox.body
27: if body.version > startingVersion then
28:    // eager W-R conflict detection; abort up to top-level
29:    abortUpTo(tx, TOP)
30: end if
31: tx.bodiesRead.put(vbox, body)
32: return body.value
```

upon its start.

Note that, once the `getAncestorWrite` loop reaches a readable write entry in line 15, i.e., one that respects the restrictions stated earlier, the algorithm never iterates any further. This is based on the invariant described in this section. This is the algorithm for read-write transactions, and therefore it uses eager conflict detection (lines 18-19).

Read-only transactions have a slightly different behavior: The iteration does not stop necessarily, either successfully or with an abort, if a write entry belonging to an ancestor is found; instead, the iteration stops only when an entry respects both restrictions of ownership and versioning and never aborts because ultimately it reads a `VBoxBody` of a globally committed version.

5.3 Writing to a `VBox`

At the start of function `setBoxValue` in Algorithm 4, a check is made for a write-after-write situation, in which the transaction overwrites a value it had previously written (lines 2-7). That represents the fast path, which requires only an update to the value of the already existing write entry. If that is not the case, the algorithm proceeds to create a new write entry containing the aforementioned data (owner, version and value) in line 9. An additional parameter is added that serves the purpose of connecting the write entries of a given `VBox` in a linked list as described above.

Algorithm 4 Algorithm to perform a write of a value to a `VBox` in the `SharedWS` design.

```
 1: SetBoxValue(tx,vbox,value):
 2: writeEntry ← tx.privateWriteSet.get(vbox)
 3: if writeEntry ≠ null then
 4:    // fast path if the transaction already owns a write entry
 5:    writeEntry.value ← value
 6:    return
 7: end if
 8: previousWriteEntry ← tx.sharedWriteSet.get(vbox)
 9: newWriteEntry ← (tx, tx.nestedCommitClock, value, previousWriteEntry)
10: tx.privateWriteSet.put(vbox, newWriteEntry)
11: // 1st write on this vbox on this tree?
12: if previousWriteEntry = null then
13:    // attempt CAS to enqueue the 1st write
14:    previousWriteEntry ← tx.sharedWriteSet.putIfAbsent(vbox, newWriteEntry)
15:    if previousWriteEntry = null then
16:       // succeded
17:       return
18:    else
19:       // another write succeeded
20:       newWriteEntry.next ← previousWriteEntry
21:    end if
22: end if
23: // try to place the new entry on the head, CAS until succeed
24: while tx.sharedWriteSet.replace(vbox, previousWriteEntry, newWriteEntry) = false do
25:    previousWriteEntry ← tx.sharedWriteSet.get(vbox)
26:    newWriteEntry.next ← previousWriteEntry
27: end while
```

The insertion of the new write entry is performed at the head of the list of write entries. This is effectively performed by setting the next field of the new write entry to what is expected to be the first entry until the atomic insertion on the `ConcurrentHashMap` returns positively. Between lines 11 and 22 the insertion is attempted as if this write is the first one making it into that list. Lines 23-27 perform the insert when another one has already taken place previously, to the same `VBox`, in the nesting tree of the transaction.

5.4 Committing Parallel Nested Transactions

The commit procedure in nested transactions is responsible for validating the execution and for propagating, into the parent, the sets of records collected by the nested transaction during its execution, effectively making them visible to the siblings of the committing transaction and serializing it.

Conceptually, the validation requires that the reads performed during the execution (read entries) still correspond to the most recent versions of the `VBoxes` read. A nested transaction may collect two different types of reads in its `bodiesRead` structure: (1) top-level reads, which are obtained via the `readFromBody` method and return a value of a `VBoxBody` that has been committed by some top-level transaction; and (2) nested reads, which are obtained via the `getAncestorWrite` method and return a value of a `WriteEntry` that represents a tentative value of an ancestor. In this sense, both `VBoxBody` and `WriteEntry` represent a value that was written to a `VBox`, but whereas the former is consolidated, the latter is tentative. Therefore, the `WriteEntry` extends a `VBoxBody` so that a transaction collects both in the same `bodiesRead`, which represents its read-set. Note that read-only transactions do not have to perform any validation, neither in top-level nor in nested. Yet, nested read-only transactions may need to collect the read-set, when they have a read-write ancestor. Only if the nesting tree is entirely composed of read-only transactions can we avoid collecting read-sets as in top-level read-only transactions. Also note that it makes no sense for a read-only transaction to spawn nested read-write transactions.

The validation of a nested transaction T iterates over its read-set. For each read entry r on some `VBox` x it verifies that, if T's parent, A, has a write entry w for x, then it must be that $r = w$. Otherwise, T fails its validation because r was outdated by w. If w already existed at the time of r then the invariant of the shared structure states that the obtained r would have been exactly w, in which case this validation would have succeeded. This is once again a consequence of the fact that the read lookup has to return the write entry belonging to the closest ancestor, if any exists.

After validating, the commit procedure has to merge both the read-set and write-set into the parent's corresponding sets. But not all of the records collected in the read-set have to be propagated. Considering a read entry r, obtained from an ancestor A, r will have to be validated upon every commit only until the commit that merges the records into A (including). Using the execution in Figure 5.1, if B had read `VBox` X prior to writing to it, it would obtain the write entry owned by A with value 5. Then, upon B's commit, the read entry associated with that access would not have to be propagated to A. On the other hand, if B had read `VBox` Z that was never written in this nesting tree, the access would result in reading a top-level `VBoxBody` that was already consolidated by a top-level commit. Therefore, this read entry would have to be propagated to A upon B's commit.

Merging the write entries entails updating the owner of the entries as well as its version. To obtain the new version during a commit procedure, nested transactions proceed in a similar fashion to what

happens at top-level, by grabbing their commit order on the parent queue of `NestedRecords` rather than on a top-level queue of `ActiveTransactionsRecord`. This procedure is described in more detail in Section 5.5. In the end, the write entries that belonged to the committing transaction will be owned by its parent, and have a new version. This version corresponds to the order obtained in the enqueue on the parent.

Now, recall that the structure presented to manage the write entries of a given `VBox` in the nesting tree was associated with the aforementioned invariant: Whenever a transaction iterates over the write entries of a `VBox` looking for a potential read-after-write, as soon as it finds a readable entry, it is guaranteed not to have to look down any further in the list. Consider once again the execution used earlier (5.1) but where B also writes to `VBox` y:

$$W_A(x, 5) \;\; S_A(B, C, D) \;\; W_B(x, 15) \;\; \boldsymbol{W_C(y, 2)} \;\; \boldsymbol{W_B(y, 17)} \;\; \boldsymbol{C_B(ok)} \;\; \boldsymbol{C_C(ok)} \;\; R_D(x, 5) \tag{5.2}$$

If we assume the simple merge of write entries in which the committing transaction updates the owner and version of each entry that it owns, the result of the execution will break the invariant. This is illustrated in Figure 5.2, where we may see the state of the write entries before any of the nested transactions commit: The entry at the head of the list of y belongs to B as it was the one that most recently wrote to it with value 17. When applying a naive merge procedure, the resulting shared structure for that `VBox` y ends up with version 1 on top of version 2. Therefore, if transaction A now attempted to read y, the algorithm would retrieve the value 17 rather than 2.

The problem in this case is that the order in which the concurrent writes are performed is not the same order in which the transactions commit. Therefore, we have to consider the merge in a more careful way. Suppose that some parallel nested transaction T is committing into its parent A. The merge of a write entry w for `VBox` y, belonging to the committing transaction T, iterates over the list of write entries for y in that nesting tree, stopping when it finds w. In this case, it updates the entry version and owner of w. However, if the iteration finds an entry w' owned by A (parent of T), then w' will be conceptually overwritten by w: Any sibling of T that starts after T's commit will always read w instead of w'. If during the merge of w, the iteration finds w' before w, we are in the case in which the invariant would be broken. That is avoided by updating the contents of w' to correspond to the merge of w, by changing its value to the value of w and the version to the commit version obtained. Note that it is not needed to change the owner as it is already A. The next section shall delve into the details of how these changes are performed in a lock-free manner.

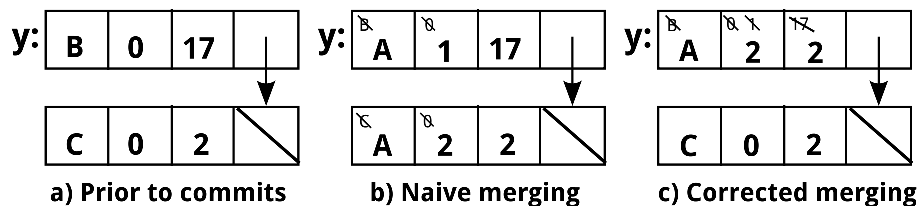


Figure 5.2: Representation of the `sharedWriteSet` prior to commit of B and C, and after, according to execution (5.2). The merge in the commit is shown when using a naive merge procedure as well as when using the corrected merge procedure.

A possible problem with this merging strategy is that it could be erasing some versions from the nesting tree, as the algorithm is physically overwriting entries that are logically overwritten by commits. In that case, it could be breaking the semantics of nested read-only transactions that resort to the logically overwritten values (older versions) to ensure that they always commit. However, I can demonstrate that the versions physically overwritten in the merge procedure are guaranteed to no longer be read ever again.

Consider once again the execution that led to Figure 5.2, but now using the corrected merging strategy. Let us assume that some other read-only nested transaction children of A, named D, needed the write entry committed by C with value 2 and version 1 on A and could never be satisfied with the write entry committed by B with value 17 and version 2. Then, D must necessarily have version 1 for A in its `ancestorVersions`. This means that D was spawned after the commit of B but before the commit of C. However, this is a contradiction because between both commits A was never in execution: B and C were active at the same time, which means that A would proceed its execution only after both its children (B and C) committed successfully. This renders impossible the chance of D being spawned between the commits of B and C and reading 1 on A's `nestedCommitClock`.

5.5 Lock-free commit

So far I have omitted how the actual synchronization of the commit of sibling parallel nested transactions to their parent is performed. I adapted the algorithm for top-level commits to be also used in nested commits: Now each transaction contains a queue that provides a committing order to concurrent siblings. In the original JVSTM, this idea was used for top-level commits and therefore there existed only one global queue.

Each committing nested transaction T performs the validation described in the previous section and proceeds to obtain its order in the queue of the parent by using a Compare-and-swap (CAS) to add a new entry to it. If that fails, then at least some other sibling of T, let us name it S, did so, and therefore T must check that S's write-set does not intersect with T's read-set (in which case T aborts by having failed the incremental validation). Note that when a transaction obtains its place in the parent's queue it is guaranteed to be valid to commit.

Consider the following execution:

$$W_A(x,1) \;\; S_A(B,C) \;\; W_C(x,0) \;\; W_B(x,14) \;\; C_B(?) \;\; C_C(?) \tag{5.3}$$

I show the structures used to commit transactions in Figure 5.3, corresponding to execution (5.3). In particular, I show the `NestedRecords` enqueued in A's queue, representing the commit of both B and C. As hinted in the description of the JVSTM in Section 4.1, there is a helping mechanism to make the top-level commit procedure lock-free. I also used this idea so that sibling nested transactions that are concurrently attempting to commit, first attempt to help commit previously enqueued siblings. This ensures that there is always global progression: As long as there are active transactions attempting to commit, there will always be one committing at a time, even if the underlying threads are allowed to fail silently. This happens because a transaction will always ensure that other siblings that are first in the commit order have already committed successfully before performing its own commit.

The `NestedRecords` enqueued for commit contain the information required for the helping mech-

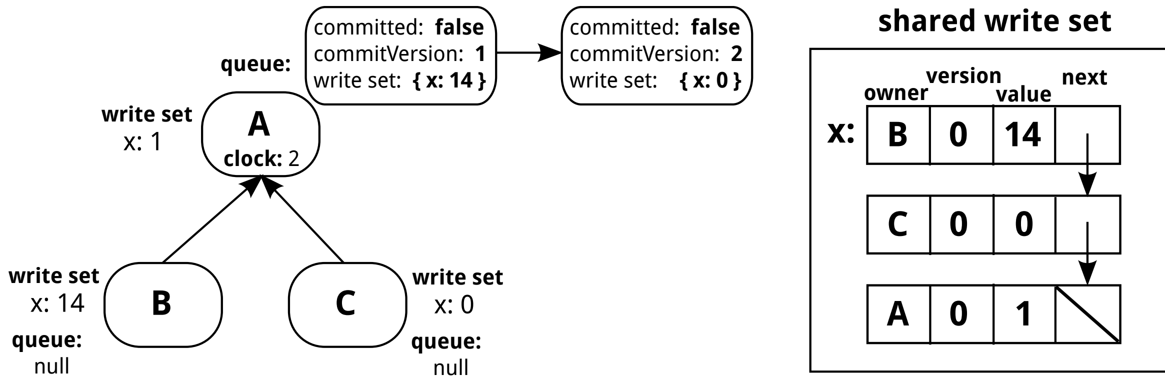


Figure 5.3: Structures used for nested commit in the `SharedWS` design corresponding to execution (5.1). Note that **clock** in A is the `nestedCommitClock`.

anism. Namely, the `write-set` to be merged into the parent, the `commitVersion` obtained when the object was enqueued successfully, and the `committed` flag stating whether this commit has been performed. Using Figure 5.3 as an example: C will only execute its own commit once B's commit has been flagged as `true`. B's commit may be performed by itself, as well as by any of its siblings attempting to commit concurrently (such as C). In this case, I say that C is a helper.

As described in the previous section, committing a transaction entails merging the write-set, whose procedure I have explained. Now, consider that C helped the commit of B before B managed to do it by itself. Therefore, B's write to x with value 14 has been merged into A successfully by C. After this, C proceeds to its own commit, which will merge a write of 0 to x.

However, B may still be conducting his own commit concurrently to the commit of C because B may not have noticed yet that its commit was finished by C. Therefore, both B and C may manipulate the contents of the same write entry in x, with different intents. Recall that the merge procedure may entail changing w, a write entry that is being committed, or w', a write entry that belongs to the parent and is about to be overwritten by the merge of w. In this specific case, the merge made by C would fall in the latter case and thus change the contents of the write that was originally produced by B, which B itself would be trying to change.

To ensure correct execution of the lock-free helping in the merge procedure, changes to the version and value of write entries have to be performed atomically. Therefore, the actual structure of the `WriteEntry` contains a reference to another object (called `VersionValue`), which contains the version and value. This way, the merge procedure uses a CAS to change a `VersionValue` object to another one, when performing changes in those fields.

Algorithm 5 shows the merge procedure of a single write-entry during the commit of a parallel nested transaction. Note that this algorithm may be executed by the transaction committing as well as by a sibling that is awaiting its turn for commit and thus performs helping as explained earlier. In any case, the `tx` argument represents the transaction whose writes are being merged and the `commitNumber` is the version acquired when `tx` enqueued in its parent for commit. The merge iterates through the entries associated to the `VBox` that was written (lines 2-3). Lines 4-18 perform the normal merge of the write entry in which only the owner and version are changed. Lines 19-27 take care of the case in which an entry belonging to the parent of the committer is found. In this case, the entry found must be overwritten.

To avoid unnecessary failed CASes of late helpers, in both cases a CAS is attempted only if the version of the entry is smaller than the version that the commit will produce.

Algorithm 5 Algorithm to perform the merge of a `WriteEntry` to a `VBox` in the commit of a parallel nested transaction.

```
1: MergeWrite(tx,commitNumber,vbox,writeEntry):
2: iterWriteEntry ← tx.sharedWriteSet.get(vbox)
3: while iterWriteEntry ≠ null do

4:    if iterWriteEntry = writeEntry then
5:       // the committing writeEntry was found first, thus apply the normal write-back
6:       oldVersionValue ← iterWriteEntry.versionValue
7:       if oldVersionValue.version < commitNumber then
8:          iterWriteEntry.owner ← parent
9:          newVersionValue ← (commitNumber, oldVersionValue.value)
10:         iterWriteEntry.CASversionValue(oldVersionValue, newVersionValue)
11:         previousWriteEntry ← tx.parent.privateWriteSet.get(vbox)
12:         if previousWriteEntry = null then
13:            tx.parent.privateWriteSet.putIfAbsent(vbox, iterWriteEntry)
14:         else if previousWriteEntry.versionValue.version < commitNumber then
15:            tx.parent.privateWriteSet.replace(vbox, previousWriteEntry, iterWriteEntry)
16:         end if
17:      end if
18:      break

19:   else if iterWriteEntry.owner = tx.parent then
20:      // found entry that is meant to be overwritten, thus re-order the write entries
21:      oldVersionValue ← iterWriteEntry.versionValue
22:      if oldVersionValue.version < commitNumber then
23:         newVersionValue ← (commitNumber, writeEntry.versionValue.value)
24:         iterWriteEntry.CASversionValue(oldVersionValue, newVersionValue)
25:      end if
26:      break
27:   end if

28:   iterWriteEntry ← iterWriteEntry.next
29: end while
```

Note that writes to `VBoxes` performed by other nested transactions cannot delay the commit procedure, which changes only already existing write entries. The helping procedure of the commit is wait-free: Each helper is guaranteed to finish in a finite number of steps regardless of concurrent actions in the same nesting tree because the operations performed during the helping never have to retry and are bounded by the size of the read-set and write-set to merge.

5.6 Abort procedure

When a transaction aborts, every write that it performed will still be in the `sharedWriteSet`. This could lead to transactions reading write entries of a transaction that was no longer active and had actually aborted. However, the algorithm guarantees that no transaction T can ever read the write entries of an aborted transaction A, unless T itself is doomed to abort. Consider the following partial execution where A is a top-level transaction:

$$W_A(x, 1) \;\; S_A(B) \;\; R_B(x, 1) \;\; ... \;\; C_B(ok) \;\; C_A(fail)$$

When A aborts, its write to x remains in the `sharedWriteSet`. Suppose now that, when re-executing, A does not write to x because its control flow was different. Yet, in practice, the write of A to x is still in the `sharedWriteSet`. Thus, when B re-executes, it will find the stale entry belonging to the aborted A. However, that entry cannot be read by B.

The implementation of my algorithms ensures that never happens. The `ancestorVersions` maintains a mapping of objects representing the transactions to versions. Therefore, when B re-executes, his ancestor A will be represented by a different object than the one in the write left from the aborted execution of A. Thus, when B checks if that write belongs to an ancestor, the comparison between both objects representing A will return false.

Moreover, there is a guarantee that no `ABA` problem can ever happen in which the address of the old A would be released and reused in some different transaction which would get this ghostly write entry out of the blue. This is provided by the garbage collector of the Java runtime: As long as one of those aborted write entries exists, the aborted transaction is still referenced and therefore its reference can never be re-used by some other transaction.

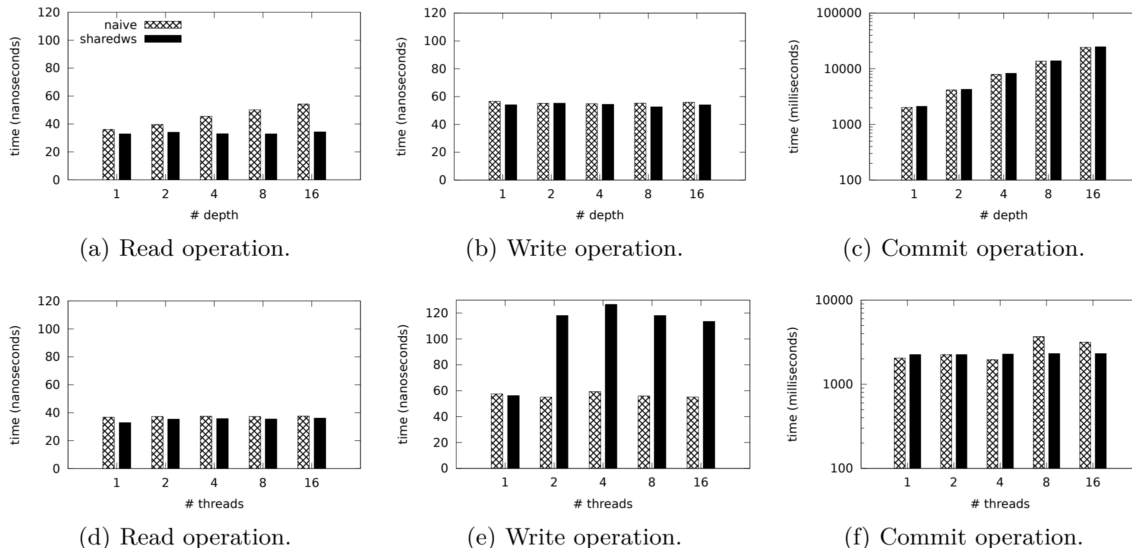
Due to memory usage concerns, the algorithm actually cleans the aborted transactions to allow the garbage collector to release the memory. Regarding the write entries, I attempt a single physical delete from the lock-free linked list that is associated with the `VBox` of the write entry w in the shared structure of the nesting tree. To do so, I find the entry b in the list that is placed before w and change its next pointer to point to the next of w. There is no need for a compare and swap as the only transaction that is ever going to change b's next, is the transaction that owns w. It may happen that b now points to another w' (that was the next of w) that was concurrently deleted by another aborting transaction. Consequently this action actually reverts that delete and, instead of having 2 successful concurrent deletes, we end up with only 1 actual physical delete. However, as shown, having aborted entries in the list is guaranteed to never harm correctness. Moreover, these structures are private to each top-level transaction (and its nested transactions), meaning that upon its completion, they are all discarded. This results in a best-effort strategy of saving memory while remaining correct.

5.7 Correctness in the Java Memory Model

The description done so far of the algorithms is not enough to assess their correction when implemented in Java, because of the relaxed memory model of Java: There need to exist some visibility guarantees of the memory operations to ensure that they behave as expected. In the following I explain how I used the Java Memory Model [39] synchronization primitives to ensure the correct behavior of the algorithms. In particular, which variables are `volatile`.

I use `volatile` in the `nestedCommitClock`, available in every transaction, so that a nested transaction that starts with version v, is guaranteed to view all changes to the writes that were performed up to, and including, the commit with version v. This happens because the committing transaction writes to `nestedCommitClock` after propagating the write entries to the parent, and a new transaction reads `nestedCommitClock` before starting.

I also use `volatile` in the `version` of the `VersionValue`. When a write entry is merged into the parent during a nested commit, both its `value`, `owner` and `version` may change. This `volatile` guarantees that a concurrent nested transaction always has consistent views of those three fields. Changes



(a) Read operation. (b) Write operation. (c) Commit operation.

(d) Read operation. (e) Write operation. (f) Commit operation.

Figure 5.4: Average time for transactional operations in a conflicting workload in the `Vacation` benchmark using the `Naive` and `SharedWS` designs. In the upper row an increasing nesting depth is used in every transaction, whereas the lower row uses always one level of nesting but with an increasing number of siblings. Note that the labels in the first figure apply to all the others.

to the `value` and `version` are made by CASing a new `VersionValue` in the `WriteEntry`. Because the `value` is written before `version` (in the constructor of `VersionValue`), and `version` is always read before `value`, then those two fields are always seen consistently. The algorithm allows a transaction to see a state in which the `version` and `value` are new, but the `owner` is stale. But this view can never cause a problem: A transaction that sees that view is concurrent to the committer transaction (or else the `nestedCommitClock` would have ensured visibility of the `owner`), and therefore it will not be able to read the write entry because the stale `owner` will never be its ancestor. When a concurrent transaction sees an up-to-date `owner`, it is guaranteed to see the corresponding (or a more up-to-date) `version` and `value`, because of the happens-before relation between the write of `version` (during commit) and its read.

5.8 Discussion of the Shared write-set design

To provide objective data about the effectiveness of this design, I profiled the average time that a parallel nested transaction takes to read, to write, and to commit during the execution of the `Vacation`. In Figure 5.4, I present the results obtained. Similar results were obtained when profiling several workloads both in the `STMBench7` benchmark and in the `Lee-TM` benchmark. To obtain these results, I forced each top-level transaction to spawn a child and to execute all of its code within the child transaction. I also profiled the execution using the `Naive` design presented in Section 4.3.

Figure 5.4(a) shows that the performance of the read operation in the `SharedWS` design is independent of the nesting depth, leading to a better performance relatively to the `Naive` design. This happens even at one level of depth, as in that case the `Naive` design already has to check two different write-sets.

On the contrary, as shown in Figure 5.4(c), the time to commit still increases with the depth in both approaches. In Figure 5.4(f), the `SharedWS` obtains slightly better results due to the parallel commit of the lock-free algorithm that hides some of the overheads of the commit.

Where the SharedWS design stops being on par with the expected results is on the performance of the writes, as shown in Figure 5.4(e). When the number of siblings increases, the cost of writing increases as well. The result in this case was expected to be similar to what is obtained with a single sibling and increasing depth, as shown in Figure 5.4(b). The difference observed is a consequence of the concurrent nature of the data structure underlying that design:[3] With multiple siblings, their actions have to be correctly synchronized, which is especially costly for modifications. Moreover, these structures may entail resize operations that are particularly expensive when paired with the sychronization costs. When we have just one nested transaction, however, the optimizations made by the Java runtime remove all the synchronization costs, leading to a performance on par with the Naive design.

The SharedWS solved the challenge regarding the cost of the read operation. In particular, it made the read independent of the depth in the average case. However, this was accomplished at the expense of making the write operation more costly. Because a transaction typically reads more than it writes, I claim that the SharedWS design is better than the Naive design with regard to parallel nesting. This shall be more clear in Chapter 8, where I evaluate both designs in known benchmarks.

[3] I tested both with Doug Lea's ConcurrentHashMap and with Cliff Click's NonBlockingHashMap, but the results were similar.

Chapter 6

A practical algorithm

Although the SharedWS improved over the initial solution, there are still some open issues. Namely, the write operation is more costly than before, and the commit procedure is still expensive. In the following work in this dissertation I seek to improve on this design and present an alternative that does not suffer from the problems identified for the SharedWS design.

Recall that transactional locations, represented by VBoxes, point to a history of all the versions for that location that may still be readable by some active transaction in the system. I propose to extend this design such that transactions perform their writes on these locations rather than having to maintain some private mapping of each location written to its new value (regardless of being a private write-set per transaction or per nesting tree). This idea is similar to the optimization described in Section 4.1.1 to the write-sets of top-level transactions. This new design is called InPlace.[1]

Next, I present the structures used in this algorithm. Then, I describe in detail the transactional operations from Section 6.2 to Section 6.5. I explain the implementation of this algorithm in the Java Memory Model, in Section 6.6. In Section 6.7, I discuss the progress guarantees of the InPlace algorithm. Section 6.8 explains how parallel nesting can deal with the read-set optimizations in the JVSTM. Finally, in Section 6.9, I compare this algorithm with the other two alternatives proposed, with regard to the challenges identified in parallel nesting.

6.1 Data-structures and auxiliary functions

In this new design a VBox contains both permanent and tentative versions. The permanent versions have been consolidated via a commit of some top-level transaction, whereas the tentative versions belong to an active top-level transaction or any of its children that form a nesting tree.

To illustrate this new design, consider the following execution:

$$W_A(y, 7) \;\; S_A(B, C) \;\; W_B(x, 10) \;\; S_B(D, E) \;\; W_D(y, 8) \;\; C_D(ok) \;\; W_E(x, 15) \tag{6.1}$$

[1] Its implementation is available in the branch *jvstm-lock-free* of the following Git repository: http://groups.ist.utl.pt/esw-inesc-id/git/jvstm.git/

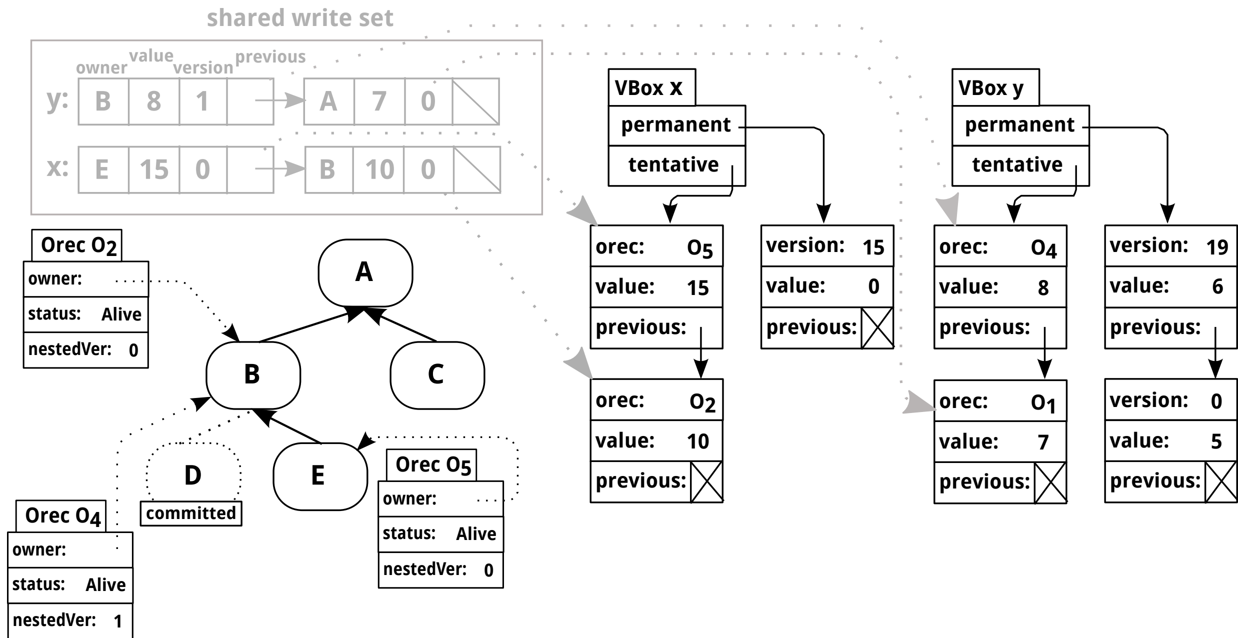


Figure 6.1: Representation after the execution (6.1) when using the InPlace design. The structures used in the SharedWS design are also shown, to illustrate what changes with the new design. To simplify the presentation, I omit the ownership record O_1, which would point to transaction A, as well as O_3 that would point to C.

Figure 6.1 shows the state after this execution when using the new algorithm. I also depict the sharedWriteSet structure that I was using in the previous design, and how the new algorithm changes it to accommodate the tentative writes in-place. In this case, the tentative writes of the represented nesting tree have been moved from the shared write-set of that nesting tree to the tentative values of the corresponding VBoxes. But the shared write-set continues to exist as a fallback mechanism to store tentative writes. The difference is that I expect it to be empty most of the time. This strategy is detailed in Section 6.4.

Figure 6.1 also depicts that the tentative writes no longer point directly to their current owner. Instead, I use the concept of **Orecs** (short for ownership records [57]) that provide a level of indirection to a reification of the ownership. As we will see in Section 6.5, this allows the algorithm to propagate the write-entries at commit time independently of the size of the write-set, thus addressing another challenge identified earlier. As shown in Figure 6.1, an Orec contains:

- **owner**: The transaction that is being represented by this Orec.
- **status**: Identifies whether the transaction is still running, has aborted, or has committed. In the latter case, it contains the timestamp acquired when incrementing the global glock in the top-level commit of the owner of this Orec.
- **nestedVer**: Timestamp acquired in a nested commit into some parent by incrementing the parent nestedCommitClock. This value only makes sense when paired with the owner. It is used to ensure that reads on tentative entries are consistent. A similar field already existed in the WriteEntry of the SharedWS design. Looking back into Figure 6.1, we may see that O_4, originally owned by D, now represents B. This happened because D already committed into B. Consequently, O_4's nestedVer field now contains the timestamp 1 obtained from the clock in B.

A `VBox` always points to a tentative write object (even if it is a placeholder). The reification of a tentative write is an **`InPlaceWrite`** and it resembles the `WriteEntry` of the `SharedWS` design. It contains: (1) the `orec` that owns this write; (2) the tentative value; and (3) a pointer to the previous tentative write to the same `VBox` in that nesting tree.

Some of the structures maintained in each parallel nested transaction are also similar to what was described for `SharedWS` in Section 5.1. Namely, the `ancestorVersions`, the `startingVersion`, and the `nestedCommitClock`. Additionally, a parallel nested transaction holds the following fields:

- **globalReads**: Read-set that maintains the `VBoxes` from which the transaction read permanent values.
- **nestedReadSet**: Read-set used for reads that obtain values from tentative writes in the same nesting tree. For each `VBox` read in this manner, a reification of the in-place tentative write that was read is registered.
- **rootAncestorWriteSet**: Shared write-set that maintains `VBoxes` and the values written to them by a top-level transaction. This structure is allocated by a root transaction, and the nested transactions in the same nesting tree only hold a pointer to it. In this design, this structure is used in the fallback mechanism when the transaction cannot write in-place to some `VBox`.
- **boxesWritten**: Maintains the set of `VBoxes` that were written in-place by the transaction. This is used in the commit of the root top-level transaction, which is responsible for the write-back that consolidates the new versions in the `VBoxes`.
- **orec**: The ownership record that this transaction uses when it gains control over `VBoxes` to write to their `tentative` slot.
- **committedChildren**: The transactions that have committed into this transaction (and thus are its children). This field is used at commit time of a transaction, to access the structures of its children.

Given this description of the data structures used by the new parallel nested transactions, I may now describe in detail, in the following sections, the transactional operations performed by a nested transaction.

A key aspect of this implementation of parallel nesting for the JVSTM is that the algorithms described affect only the nested transactions. This means that the operations of a top-level transaction remain the same as in [24], and that the few changes in the original algorithm of the JVSTM affect its performance only if parallel nesting is being used. Thus, the original performance is not affected by the extension that I performed. In particular, read-only transactions preserve their optimizations and fast paths.

Finally, in the following descriptions of the algorithms, I resort to the aforementioned `abortUpTo(tx, conflicter)` function. I also use:

- `lowestCommonAnc(tx, other)` that returns the deepest transaction in the nesting tree that is ancestor of both `tx` and `other`.
- `sameNestingTree(tx, other)` returns true if `tx` and `other` are transactions of the same nesting tree.

- `executeAsTopLevel(tx)` has the same effect as calling `abortUpTo(tx, TOP)`. In addition to that, it repeats the code encapsulated in `tx`, but in the scope of the root ancestor of `tx`.

6.2 Reading a **VBox**

Algorithm 6 Reading from a `VBox`.

```
1: GetBoxValue(tx,vbox):
2: wInplace ← vbox.tentative
3: status ← wInplace.orec.status
4: // check if it is possible that this is a RAW
5: if status = COMMITTED ∧ status ≤ tx.startingVersion then
6:    value ← readFromPermanent(vbox)
7:    tx.globalReads.add(vbox)
8:    return value
9: end if

10: if sameNestingTree(tx, wInplace.orec.owner) then
11:    // there may exist a previous write to be read
12:    while wInplace ≠ null do
13:       orec ← wInplace.orec
14:       nVer ← orec.nestedVer
15:       owner ← orec.owner
16:       if owner = tx then
17:          return wInplace.value
18:       end if
19:       if tx.ancestorVersions.contains(owner) then
20:          if nVer ≥ tx.ancestorVersions.get(owner) then
21:             abortUpTo(tx, owner) // eager validation
22:          end if
23:          tx.nestedReadSet.put(vbox,wInplace)
24:          return wInplace.value
25:       end if
26:       wInplace ← wInplace.previous
27:    end while
28: end if

29: // no InPlaceWrite may be read, check the fallback write-set
30: value ← rootAncestorWriteSet.get(vbox)
31: if value ≠ NONE then
32:    tx.nestedReadSet.put(vbox,value)
33:    return value
34: end if
35: value ← readFromPermanent(vbox)
36: tx.globalReads.add(vbox)
37: return value
```

Just like in the `SharedWS` algorithm, in this case the read operation may also obtain a value from a tentative value, or a global value from a `VBox`. I show the pseudo-code for the read operation in Algorithm 6.

In lines 1-9 the algorithm tests for a fast path that allows it to decide if there is any chance that a write may have been done to the `VBox` in this nesting tree. If it finds that to be impossible, then a globally consolidated value is returned. To reach such conclusion, it needs to verify the following condition: The transaction controlling the `VBox` has committed before this transaction started. The information required

for this is present in the Orec, namely, the status field. If there was a write in the VBox, performed in this nesting tree, then there would be an Orec in the write whose status was Alive.

If the fast path is not used, then the algorithm checks if the VBox is being controlled by the nesting tree (line 10). In that case, it iterates over the writes in the tentative list of the VBox until one of the following conditions is verified:

1. The owner of the write entry being iterated is the transaction attempting the read, in which case no further verification is needed to read it (lines 16-18).
2. The owner of the write entry being iterated is an ancestor. When this happens, the transaction may read that entry if it was made visible to it after this nested transaction started (lines 19-25). This is required this because, if this nested transaction ever attempted to read a stale version, it would be guaranteed to fail the validation at commit-time, as there is already a more recent version. This happens because the serialization point of the transaction takes place at commit-time (assuming it performs writes). If the maximum version readable from that ancestor is outdated, this transaction causes a chain abort such that the nested transactions up to that ancestor restart with the most recent versions on their ancestorVersions map.

The algorithm presented here dictates that once a nested transaction finds a tentative write that it may read, it stops and returns that value. This relies on the same invariant described for the read operation of the SharedWS design in Section 5.2.

6.3 Writing to a VBox

Algorithm 7 presents the pseudo-code of the write operation under parallel nesting. When accessing the VBox, it fetches the tentative write at the head (line 2). By reading the Orec in the first field of the tentative write entry, it is able to tell whether that VBox is currently owned by the transaction. Otherwise, after line 7, the algorithm repeats until one of the following conditions is verified:

- The VBox owner has finished before this transaction started, in which case this transaction attempts to acquire ownership of the tentative write at the head of the list of that VBox (lines 11-21). To do so, a CAS is attempted, to change the ownership of the first tentative write. If the CAS fails, the algorithm repeats. If the previous owner had finished, but after this transaction started, then no transaction in this nesting tree (and particularly the one attempting the write) will ever be able to write to that VBox in-place. In that case the algorithm proceeds to lines 33-38, in which this write operation causes a conflict and aborts the transaction. This restriction is what allows me to maintain the fast path presented in line 4 of the read operation in Algorithm 6.
- The VBox is owned by an ancestor of this transaction (lines 22-32). This is true if the current owner is present in the ancestorVersions of this nested transaction. In this case, the transaction attempts to enqueue a new in-place tentative write by performing a CAS on the head of the list of that VBox. If this CAS fails, then some other transaction in this nesting tree succeeded, in which case the algorithm is repeated.
- We are left with two alternatives: Either the VBox is controlled by another nesting tree, or by a different branch of this nesting tree. In the former case, the algorithm proceeds to lines 37-38, to

Algorithm 7 Writting a value to a VBox.

```
1: SetBoxValue(tx,vbox,value):
2: orec ← vbox.tentative.orec
3: // check for a write-after-write
4: if orec.owner = tx then
5:    wInplace.value ← value
6:    return
7: end if

8: while true do
9:    wInplace ← vbox.tentative
10:   orec ← wInplace.orec
11:   if orec.status ≠ ALIVE then
12:      if orec.status ≤ tx.startVersion then
13:         // attempt to acquire ownership
14:         if wInplace.CASowner(orec, tx.orec) then
15:            wInplace.value ← value
16:            tx.boxesWritten.add(vbox)
17:            return
18:         end if
19:         continue // a concurrent tx acquired the ownership, start over
20:      end if
21:      break  // not enough version, use the fallback mechanism

22:   else if tx.ancestorVersions.contains(orec.owner) then
23:      // belongs to an ancestor
24:      newW ← (value, tx.orec, wInplace)
25:      if vbox.CASinplace(wInplace, newW) then
26:         tx.boxesWritten.add(vbox)
27:         return
28:      end if
29:      continue // another tx in same nesting tree succeeded, start over
30:   end if
31:   break  // out of options, fallback
32: end while

33: if sameNestingTree(tx, orec.owner) then
34:   // will eventually be able to write in-place
35:   abortUpTo(tx, lowestCommonAnc(orec.owner))
36: end if

37: // will never be able to write in-place
38: executeAsTopLevel(tx)
```

the fallback mechanism. In the latter case, this transaction aborts due to a write-write conflict (lines 34-35).

Even though it would be possible to support concurrent writes in-place from different top-level transactions (or their nesting trees), that would require polluting the fast path with additional verifications. This is particularly relevant for the algorithm for top-level transactions [24]. Moreover, as pointed out earlier with regard to Table 6.1, I do not expect write-write concurrency to be very frequent without yielding read-write conflicts that would preclude that concurrency in the first place.

I additionally inhibit concurrent writes in the same nesting tree, as a nested transaction is able to acquire ownership of a `VBox` only if the `VBox` belongs to an ancestor (or to no active transaction at that time). The reason behind this decision is that allowing concurrent writes in the same nesting tree would force the algorithm to do additional work at commit-time, so that the invariant mentioned in Section 6.2 for faster reads is maintained. This additional work was responsible for much of the complexity of the commit procedure in the `SharedWS` design.

Note that this does not preclude any practical concurrency as there should not be any significant contention for writing to the same `VBox`. The rationale presented above for top-level transactions applies also to nested transactions in the same nesting tree. Moreover, recalling the motivation provided in Section 2.3, one of the reasons behind using parallel nesting in the first place is that we are expecting to move to a less conflicting workload by exploring a different scheduling approach. Consequently, the inner parallelization of a transaction should be such that we do not incur in write-write contention for the same variables, thus making the costs of a possible trade-off of not having concurrent writes in the same nesting tree, negligible.

Finally, there is one theoretical concern that was not addressed in Algorithm 7. Consider the following execution:

$$S_A(B,C) \;\; W_B(x,1) \;\; W_C(y,5) \;\; W_B(y,2) \;\; W_C(x,6) \tag{6.2}$$

Siblings B and C are writing to the same variables alternately. Accordingly to what I presented, this execution is impossible: When B writes to y, it aborts because that `VBox` is controlled by C. Thus, in practice B releases the ownership of x and re-executes.

However, C may attempt to write to x before B releases the ownership of x. This leads to C aborting as well due to a second conflict. This scenario may be repeated indefinitely leading to a livelock. Therefore the algorithm has to safeguard from such situation. For that, the algorithm uses an exponential back-off when a nested transaction restarts due to a W-W conflict, such that the livelock is ruled out probabilistically.

6.4 Fallback mechanism

In the event of write-write concurrency in the same `VBox`, a fallback mechanism is used, allowing transactions to proceed in an alternative way to writing in-place. At this point it is important to recall briefly how top-level transactions address this same problem. Their writing procedure is a lightweight version of what was presented for parallel nested transactions: A top-level transaction attempts to control the `VBox` (in case it was not a write-after-write) if the current owner has committed before it started; otherwise, it resorts to the private write-set that maintains a mapping from transactional locations to their tentative values in the transaction. Being able to write in-place in the `VBox` is cheaper than maintaining the mentioned mapping, for which reason this algorithm also yields benefits for the top-level transactions. More importantly, they also benefit from the fast path in the read procedure. This same idea was also implemented in the top-level transactions of the early designs presented in this dissertation, thus making the comparison between the various designs for parallel nesting fair.

Therefore, when a parallel nested transaction traverses all the tentative writes in a `VBox` to read

it, without success, it still needs to check the private write-set of the root (top-level) transaction of its nesting tree (lines 29-34) in Algorithm 6. It may have happened that the root transaction, prior to the execution of its children, attempted to write to that `VBox` without success, having thus had to fallback to the private write-set. Otherwise, the nested transaction is sure that it is not in the presence of a read-after-write and fetches the global value.

So, this mechanism is needed because only one nesting tree may use the in-place tentative location of a box at a given time. Thus, if more than one nesting tree writes to the same box, only one of them will be able to use the box; all the others will have to use their shared write-set to store their tentative values. Consequently, when the write procedure determines that the `VBox` is currently controlled by another nesting tree (or simply by another top-level transaction), a more drastic measure is taken. In that case, the `executeAsTopLevel` function is called (line 38 of Algorithm 7). In that event, the nested transaction is aborted (as well as any of its nested transaction ancestors) and the code that it encapsulates is re-executed in the scope of the root top-level ancestor. In particular, the specific write that caused this fallback will necessarily be performed by the top-level transaction in its private fallback write-set (the `rootAncestorWriteSet`).

I expect that this fallback mechanism is rarely used, and so the the `rootAncestorWriteSet` will be empty in the normal case. So, as long as different nesting trees do not contend for writing to the same `VBox`, transactions will be able to follow the fast path of the new write algorithm, thereby making writes faster. In practice, I expect this to be the common case.

Table 6.1 presents the total number of writes and, of those, the total number of Write-after-read (WAR) operations that were collected from several benchmarks. These results show that, typically, a transaction writes to transactional variables that it read before. Therefore, if two top-level transactions (and their possibly existing underlying nesting trees) write to the same variable, they will most likely have already read it. If that is the case, they will cause a read-write conflict and only one will be able to commit. Because conflicts should be rare, or otherwise the parallelization is not effective, I claim that successful transactions will most of the time have executed the fast path.

test	writes ($*10^3$)	wars ($*10^3$)	%
bench7-r-notrav	6158	6158	100
bench7-rw-notrav	6646	6646	100
bench7-w-notrav	3967	3967	100
bench7-r	46267	46267	100
bench7-rw	65107	65107	100
bench7-w	74744	74744	100
lee-mainboard	160	160	100
lee-memboard	148	148	100
lee-sparselong	16	16	100
lee-sparseshort	8	8	100
vac-reservations	0.3	0.3	100
vac-deletions	1684	1526	91

Table 6.1: Total number of writes and write-after-reads (wars) performed in various workloads of the `STMBench7`, the `Lee-TM`, and the `Vacation` benchmarks.

6.5 Committing a parallel nested transaction

At commit-time we are left with two tasks: To ensure that all the reads are still up-to-date, and to propagate the read-set and write-set of the transaction to the parent. This design tackled the challenges identified in the `SharedWS` with regard to the write operation. However, the new nested write operation may lead to a stalled thread preventing the system from progressing. Because the JVSTM with the `InPlace` design is no longer lock-free, I now present a commit procedure that avoids the complexities of the lock-free commit in the `SharedWS` and thus is more efficient in general.

Before validating and merging the footprint, the committing transaction acquires a commit lock associated with the parent. After validating with success, it increments the parent's `nestedCommitClock`, and reads it to obtain the new timestamp for its writes. This timestamp is useful once its writes are propagated to the parent and visible to other transactions.

The validation of a transaction T_i is performed by accessing directly the `VBoxes` that were read and looking into their tentative writes. Assume T_i read a global value from `VBox` x because there was no tentative write from its nesting tree at the time that it could read. Then, validation of T_i at commit time fails if there exists a tentative write in x that T_i is able to read, unless it is owned by T_i. Such fail is a read-write conflict. Similarly, this verification is extensible for reads that obtain values tentatively written in the nesting tree. In this case, the verification stops when it finds the tentative write that was read, which means that there is no newer value for the `VBox`. Once again, this is correct only because I ensure the invariant in the order of tentative writes that was described earlier.

Regarding the merging of the write-set, I sought to make it as lightweight as possible. In this design, every write of a nested transaction is performed in-place. This means that each of the `VBoxes` written has an `InplaceWrite` pointing to the `Orec` of the nested transaction that did the write. To merge the write-set, a transaction T that is committing changes the `nestedVer` of its `Orec` to have the timestamp read from the parent's clock. Finally, it suffices to change the `owner` in T's `Orec` to point to the parent of T and to propagate that `Orec` to the parent. Because commits may take place at different nesting depths, in practice T has a list of its committed children (the `committedChildren` field). This way, T can access every `Orec` propagated to it, in addition to its own, so that it updates and propagates them recursively to its parent when it commits.

The reads must also be merged into the parent. To do this, I use the `committedChildren` field, because it allows accessing the reads performed by those children. Consequently the read-set ends up being scattered among the nesting tree. At commit-time, this results in the validation procedure of a transaction traversing different read-sets of the children in addition to its own. The slight overhead of having a one-time level of indirection for accessing the different read-sets is far overcome by the benefit of not having to perform work proportional to the size of the read-set at each commit.

The complexity of the commit procedure is now bounded by $O(\#children + size(readSet))$ rather than $O(size(writeSet) + size(readSet))$. The component regarding the read-set is also less expensive in this new algorithm: It consists of validating it only, contrarily to also propagating it to the parent during the commit. Therefore, there is less room for contention to the same commit locks given that the time within those has been largely shortened. Recall that the lazy write-back nature of the underlying STM, which allows further concurrency, also dictates that some work proportional to the size of the read-set must be executed at each nested commit [1].

In the event of an abort, the transaction simply changes the `status` field of the `Orecs` that it

controls to an `Aborted` value. After this, the write entries at the head of the `VBoxes` controlled by the transaction are effectively ignored, because concurrent transactions see that their owner has aborted. There is, however, a caveat. Consider the following execution:

$$W_A(x,1) \;\; S_A(B) \;\; W_B(x,2) \;\; ... \;\; C_B(fail) \tag{6.3}$$

After the abort of B, if some concurrent top-level transaction (or a nested transaction in its nesting tree) attempts to write to x, it will find that an aborted transaction is controlling it. Therefore, it will be able to acquire control over x. However, A was still supposed to control x, except that its child B wrote to it and then released the control, due to its abort. Thus, the aborting nested transaction must delete its writes from the tentative lists, but only when those writes were performed when an ancestor was already controlling the `VBox`.

In practice, this entails iterating over the `VBoxes` written and checking, for each `VBox`, if there exists an `InplaceWrite` w that is not owned by the aborting transaction, and that is owned by a transaction that is alive. If that is the case, w's `value` and `orec` are assigned to the first `InplaceWrite` in the tentative list. Moreover, between changing the `value` and the `orec`, the `orec`'s `owner` field is written with its own contents, for reasons that shall become clearer in the next section. Finally, the aborting transaction changes the `status` of its `Orecs`.

6.6 Correctness in the Java Memory Model

Similarly to what I presented in Chapter 5, I describe now the details in the implementation of this design in Java that depend on its Memory Model. Once again, I use `volatile` in the `nestedCommit-Clock`, for the same reasons as in the `SharedWS` algorithm.

Then, I also use `volatile` in the `owner` of the `Orec`. This ensures that concurrent nested transactions always see a `nestedVer` of the `Orec` consistent with its `owner`: The commit is the only operation that changes those fields in the `Orec`, and it always changes the `nestedVer` before the `owner`, whereas another transaction reading or writing to a `VBox` always reads the `owner` before the `nestedVer`. This latter read creates a happens-before relation with the write of the `owner` during commit or abort. This same reasoning applies for consistent views of `value` and `owner`.

6.7 Progress guarantees

The STM underlying this work, briefly presented in Section 4.1, is lock-free, which means that a thread cannot prevent all other threads from progressing, as the system as a whole must make progress.

This is no longer true when we take into account the parallel nested transactions spawned in the scope of a nesting tree. Suppose that T_i, a nested transaction, writes to x and stalls indefinitely. Now, if T_k, a sibling of T_i, attempts to write to x, it will create a write-write conflict and restart. In this scenario T_k will never be able to write to x and is therefore prevented from progressing. Consequently, the nested write operation is no longer lock-free in this design.

For this reason, I simplified the commit procedure by using locking. Therefore, the commit operation of a parallel nested transaction may delay its siblings. Note, however, that the commit of parallel nested transactions at different nesting levels or nesting trees proceed independently. Additionally, the read operation of a parallel nested transaction executes in a bounded number of steps and independently of concurrent transactions, thus being wait-free.

Therefore, the JVSTM with support for parallel nesting is no longer lock-free, because both the write and commit of parallel nested transactions may prevent the system from progressing. This is motivated by the fact that the blocking algorithm provides better performance, and still allows top-level transactions to continue to execute in a lock-free manner and to proceed with their validation in parallel. In particular, this means that there is no lock that must be acquired at commit-time, and for which top-level transactions contend. A parallel nested transaction may never prevent other transactions (nested or not) from progressing unless they have a common ancestor, meaning that they belong to the same nesting tree.

6.8 Maintenance of read-sets

The algorithms that I described are tightly coupled with the way the JVSTM was designed in the first place. In Section 4.1.1, I explained how the latest version of the JVSTM deals with the read-set and write-set. Yet, so far, I have not delved into the details of how the parallel nesting algorithms may take advantage of the read-set optimizations. Briefly, the optimization relied in the maintenance of a pool of arrays per thread that would be reused throughout different transactions. The benefit is that this decreases drastically the repeated allocation of new memory for read-sets in each transaction.

Adapting this idea directly to parallel nested transactions is not straightforward: The read-set must survive the execution of a parallel nested transaction so that its ancestors may validate it during their commit procedure. Consequently, the root top-level transaction ends up being responsible for cleaning up the read-sets used by its children. However, that top-level transaction is executing in a different thread and has no access to the thread local storage of the threads that executed its children. So, the problem is that the read-sets are meant to be reused in the same threads, but now more than one thread may be using them, and more importantly the thread that cleans up a read-set may be different from the thread that is responsible for it. One way to work around this would be to use a global pool of arrays for read-sets. Yet, there is an alternative that maintains the decentralized nature of the optimization described earlier.

For this, I created a new reification for a block of read entries, the `ReadBlock` class that I present in Listing 6.1. In its essence, a `ReadBlock` encapsulates an array of `VBoxes`. Therefore, a parallel nested transaction now maintains a set of `ReadBlocks` instead of a set of `VBox[]`.

Each thread also maintains a thread-safe counter that is publicly accessible through its `ReadBlocks` (because it is passed to their construction). This counter plays two roles: (1) it ensures a happens-before relation between the release of used blocks and their reuse; and (2) it helps create a fast-path when a thread attempts to reuse arrays to augment its read-set because it states how many free blocks there are in the pool.

With this solution we may have a thread with a pool of several blocks, each one with a reference to the counter of free blocks in the pool of that thread, and the blocks of this thread may also be in use by

```
public class ReadBlock {

  protected boolean isFree;
  protected final VBox[] readEntries;
  protected final AtomicInteger numberFreeBlocks;

  public ReadBlock(AtomicInteger numberFreeBlocks) {
    this.isFree = false;
    this.readEntries = new VBox[BOXES_PER_BLOCK];
    this.numberFreeBlocks = numberFreeBlocks;
  }

}
```

Listing 6.1: Class representing a block of read entries.

other threads. To augment a read-set, a transaction checks the counter of the thread:

- If it is over 0, it seeks through the pool for a `ReadBlock` whose `isFree` variable is true and uses it by setting that variable to false. Lastly it decrements the counter of the thread.
- If it is 0, the transaction creates a new `ReadBlock` pointing to the counter of the thread, adds it to the pool of the thread and uses it.

Note that only the thread that owns the pool may set `isFree` variables to false and add new `ReadBlocks` to the pool. This last detail is relevant because this way the pool need not be thread-safe.

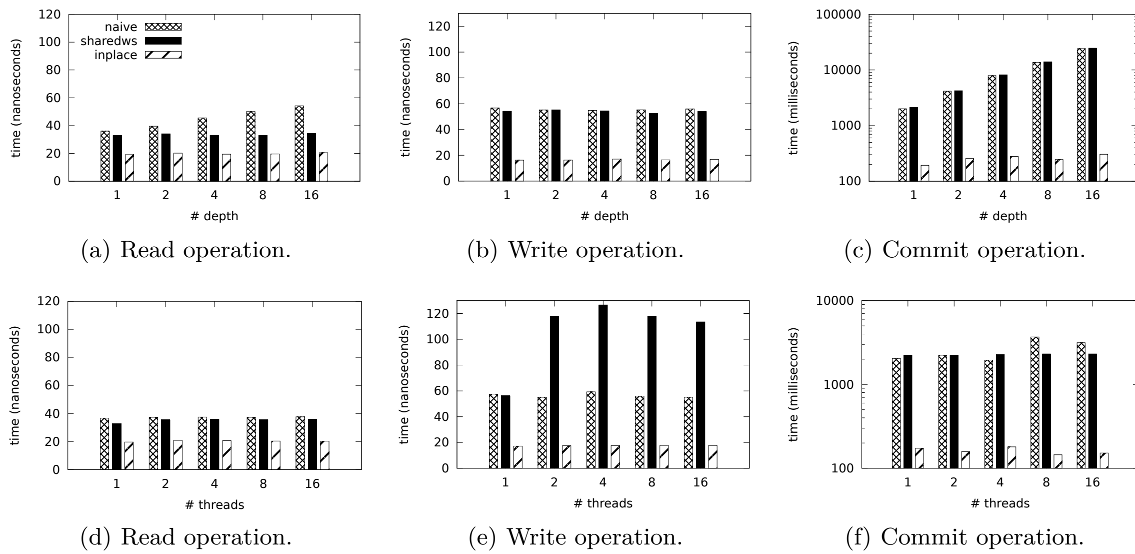
To allow the thread owner of a set of `ReadBlock` to reuse them, a transaction (running either in that thread or another one) iterates through the blocks and sets their `isFree` variables to true. Then it increments the counter of the thread owner (accessible through any of the blocks) with the number of blocks freed. The only `volatile` write performed is the increment in the counter, which guarantees visibility of the `isFree` field of the blocks when the thread owner reads the counter in the next augment procedure.

6.9 Discussion of the `InPlace` design

In this section, I look into the performance of the transactional operations of all the algorithms presented. In Figure 6.2 I present the profiling data of the transactional operations for the three designs. Namely, in Figures 6.2(a) and 6.2(d) we may see that the read operation with the `InPlace` design is able to achieve better performance while remaining independent of the nesting depth. This is an improvement over the `SharedWS` design due to the fast path enabled by the metadata stored in-place in the `VBox`.

Let us now look into Figures 6.2(b) and 6.2(e) regarding the write operation. This operation was the most important driver for the creation of a new algorithm beyond the `SharedWS`. We may see that this new design allows much cheaper write operations that are independent of both the number of siblings and the depth.

Finally, Figures 6.2(c) and 6.2(f) show that the commit procedure benefits from the new complexity bound, particularly with the increasing depth. Although the time to commit still increases as the depth increases, it happens at a much slower rate. This is an important fact given the blocking nature that I employed in the nested commits in the `InPlace` design.



(a) Read operation.

(b) Write operation.

(c) Commit operation.

(d) Read operation.

(e) Write operation.

(f) Commit operation.

Figure 6.2: Profiling data of the transactional operations updated from Figure 5.4 with the `InPlace` design.

Chapter 7

Scheduling for Profit

So far, I have addressed the main objective of this dissertation: The development of an efficient algorithm for parallel nesting that allows a programmer to explore parallelism within atomic actions. This is useful in the case of highly-conflicting workloads, in which top-level transactions conflict with high probability, whereas sub-tasks exploring parallelism in each transaction have fewer conflicts.

However, the parallel nested algorithms are more complex (and costly) than the top-level algorithms. Thus, there are some overheads inherent from exploiting parallel nesting, meaning that it may not always be profitable to use that approach. In particular, parallel nesting is useful only if we have processors available to execute the sub-tasks. These sub-tasks are identified by the programmer, but deciding when to use them is a more difficult responsibility to handle.

In Figure 7.1, I show an example of this situation, where I use all three workloads in `STMBench7`. In the dashed lines, we may see the normal execution with top-level transactions, which obtains very small performance gains and most of the time even slowdowns, due to the disruption created by read-write long traversals.

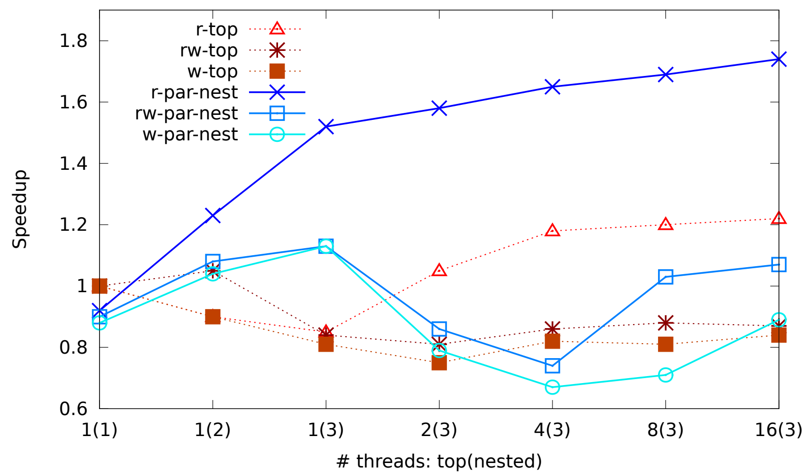


Figure 7.1: Speedup of the three workloads available in `STMBench7` relative to the corresponding sequential execution of a top-level transaction. The dashed lines use only top-level transactions and thus the thread count is the multiplication of the threads available in the horizontal axis.

I also show the executions that resort to parallel nesting (`InPlace` algorithm) in those traversals. This means that I parallelized the top-level transactions used in some of the operations of the benchmark. The labels in the thread count show the number of nested transactions spawned by each top-level transaction in the case of the parallel nested approach. Parallel nesting starts out with improvements while using a single top-level transaction (and an increasing number of children), but after that it drops considerably in the most conflicting workloads. This exemplifies the claim that parallel nesting is not always a better approach. For this reason, I propose to use a transaction-aware scheduler coupled with the TM, which allows deciding automatically when to use parallel nesting. In this chapter I further motivate its need and show the advantages obtained with its use.

In `Vacation` and `Lee-TM` all transactions are parallelizable, spawning the same number of children each. In such cases, it is easy for the programmer to reason about how many threads he wants executing top-level transactions, because he knows how many threads each one of those will require (for the nested parallelism). `STMBench7` differs from both of the previous benchmarks because it has several types of operations, of which I parallelized only some long traversals. Moreover, each of these operations spawns a different number of children. So, it becomes much more difficult for the programmer to decide how many threads he wants for top-level transactions in a given machine.

Regardless of this particularity, I regard `STMBench7` as an example of a more general application, for which it makes sense to use parallel nesting with transaction-aware scheduling. Next, in Section 7.1, I briefly present related work in transaction scheduling. Then, I describe some alternatives for an embedded scheduler in the JVSTM in Section 7.1.1. I continue by explaining in more detail the inner workings of the scheduling strategy used in Section 7.1.2. Finally, I explain how to use the scheduler to improve the results obtained with parallel nesting, in Section 7.2.

7.1 Scheduling transactions

There has been some work that explored scheduling of transactions as opposed to the use of Contention Managers [32] as a way to handle conflicts. The latter are merely reactive, in the sense that they act upon a conflict detection, and decide on the outcome of the conflicting transactions by declaring which one must abort. On the other hand, scheduling acts pro-actively, by attempting to run concurrently transactions that do not conflict with each other.

Yoo and Lee [61] proposed Adaptive Transactional Scheduling, in which threads maintain a notion of contention: When a threshold is reached in a thread, it is declared to be under high contention, and the transaction that it is executing is placed in a queue of transactions, which are then executed sequentially. All transactions end up being placed in this queue as long as the system is considered to be under contention.

CAR-STM [19] uses a serialization technique that guarantees that if two transactions conflict with each other, one of them is executed by the thread of the other such that it is guaranteed not to conflict with the same transaction in its re-execution. Also similarly, Steal-on-Abort [5] allows a thread to steal conflicting transactions from concurrent threads.

On a different fashion, Shrink [21] provides a technique that uses the previous accesses in the same thread to predict the footprint of the next transaction. Thus, it is able to estimate heuristically if the next transaction will conflict with an active concurrent transaction.

7.1.1 A scheduler for the JVSTM

In this section I describe the conflict-aware scheduler that I created, based on the aforementioned related work. I do not seek any novelty in the scheduler itself, but rather, combine the related work to create a scheduler within the STM to further improve the results obtained with parallel nesting.

I used and compared three alternatives for scheduling transactions, which I briefly describe next. In all of them, the general idea is to register the conflicts that lead to aborts, such that for each transaction we know which other transactions may create conflicts when ran concurrently. Take into consideration that in the JVSTM, when a conflict takes place, then one of the transactions in the conflict has necessarily committed, given the lazy write-back nature. The three alternatives follow:

1. **Wait**: This approach was based on earlier work [8] that proposed to use a conflict-aware scheduler together with the JVSTM to increase the performance when there were more tasks than physical cores. Its name comes from the fact that a worker thread that requests a task from the scheduler may have to block and wait even if there are available tasks. This happens because the scheduler concluded that all the available tasks had conflicted in the past at least once with the tasks currently being executed.

2. **Wait-relaxed**: In this alternative I changed the `Wait` approach slightly. Whenever the `Wait` scheduler was queried for a transaction start, it acquired a lock over the list of available tasks, as well as over the list of tasks being executed. This could potentially represent a sequential bottleneck in the execution of transactions, particularly when the length of transactions is short. Therefore, I provided a more relaxed scheduler in which a transaction could be allowed to execute even if there was a conflicting transaction already in execution. Note that this does not harm correctness in any way, but may ultimately cause aborts. This relaxation is the consequence of the lack of synchronization that I employed to relinquish some of the potential overhead of the `Wait` scheduler. In practice, I expected that most of the time the scheduling decisions were similar to the ones of the `Wait` approach, while paying less for synchronization.

3. **Serial**: This last alternative departs more radically from the previous two. Here I used some of the concepts mentioned in the related work, such as the serialization technique. The general idea is that, instead of waiting in the scheduler until an available task is executable without predictable conflicts, the worker threads offer their tasks to each other as they predict the conflicts in the scheduler. This way I create a serial execution of tasks in queues in the workers.

In Figure 7.2 I show the results obtained with each of the three alternatives for scheduling in a write-dominated workload in `STMBench7` and compare them to the sequential execution. I also show a normal execution without any scheduling, whose performance we have seen in Figure 7.1 to be very poor. From this data we may see that the `Serial` scheduler is the one performing better. Even though the difference is small most of the time, it is more consistent. Moreover these results were quite similar with the other workloads of the `STMBench7`. Next, I shall explain in more detail the inner workings of this approach and its integration with the JVSTM.

7.1.2 `Serial` scheduler

The previous work [8] that I used to create the `Wait` scheduler, was dependent on the application: The scheduler was built in the application being parallelized and synchronized with the TM. However, in

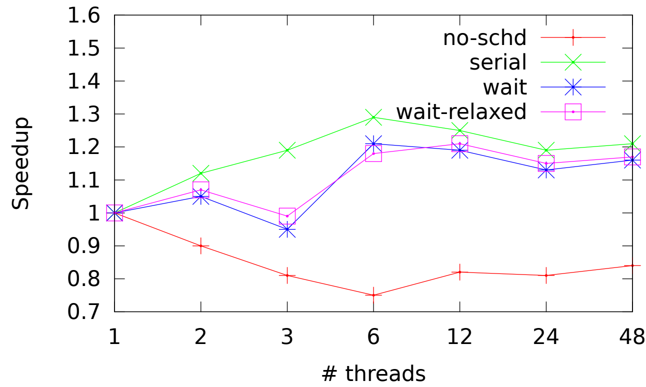


Figure 7.2: Write-dominated workload of the `STMBench7`. The results shown use either plain execution (*no-schd*) or one of the scheduling alternatives proposed. The speedup is computed relatively to the sequential execution without scheduling.

my work, I embedded the scheduler in the TM system, such that it need not be aware of the particularities of each application. In this section, I describe the integration of the scheduler in the JVSTM as well as the specific approach employed in the `Serial` variant.

I assume that the application has several different tasks that it may execute concurrently and that these are distinguishable from each other. That is, a runtime task that corresponds to some source code is different from a runtime task that corresponds to other source code. Two runtime tasks are equal if they correspond to the same source code. For example, in the specific case of `STMBench7`, all the tasks are mapped to unique identifiers provided by an enumeration.

Being able to identify the tasks is what allows the scheduler to maintain the conflict history between them. Therefore, I created an interface that must be implemented by tasks that are to be executed through the scheduler, shown in Listing 7.1. This simple interface merely requires the task to provide the aforementioned identifier.

```
public interface SchedulerTask {

    public int getTaskId();

}
```

Listing 7.1: Interface implemented by tasks to run through the scheduler.

The scheduler needs to act upon three different moments of the control flow of a transaction. These are represented by the `Scheduler` interface shown in Listing 7.2.

```
public interface Scheduler {

  public SchedulerTask getNextSchedulerTask();

  // Returns true if the transaction may run with inner parallelism
  public boolean startTx(int startingTx, Object work, SchedulerWorker runner);

  public void conflictTx(int abortingTx, int commitNumberOfConflicter);

  public void finishTx(int finishingTx);

}
```

Listing 7.2: Interface implemented by the different scheduling approaches.

With this interface, we may easily implement the different scheduling approaches as well as one in which the tasks are executed normally without any scheduling. When the transaction starts, the `Serial` scheduler implementation registers the identifier of the task in the set of executing tasks. This query to the scheduler takes place during the start of a transaction in the JVSTM. When a transaction in the JVSTM finishes (either by committing or aborting), the scheduler is informed and removes the transaction from the set of executing tasks. Note that the JVSTM calls these operations in the scheduler only when it is dealing with read-write transactions. Read-only transactions never conflict and consequently are always profitable to run concurrently, thus requiring minimal scheduling.

If a conflict happens, the JVSTM also informs the scheduler. In this case, the identifier of the transaction that caused the conflict is necessary because the scheduler maintains a table of conflicts between the uniquely identified transactions of the application. This table is populated when a conflict happens between two transactions. But we only have easy access to the application identifier of the transaction that is aborting; the identifier of the transaction that caused the conflict is not readily available. This happens because the versions of the `VBoxBody` in which conflicts are detected only map to identifiers of runtime transactions in the JVSTM and not to the transactions of the application in terms of source code.

To work around that difficulty, I extended the metadata kept by the JVSTM, which already allows the garbage collector to keep track of all the versions and timestamps used in the past. When a transaction commits (thus producing versions of `VBoxBodies`), it stores in the metadata maintained for the garbage collection, not only the global timestamp acquired in the JVSTM, but also the identifier provided in the start of the transaction that came from the application. This way, the scheduler is able to use that information to map the timestamp of the conflicting version of a `VBoxBody` to the identifier of the task that created it. After that, it uses the identifiers of both transactions to increment their likelihood of conflict in the metadata maintained by the scheduler.

There is room for improvement in terms of the maintenance of the conflict likelihood between two transactions. In particular, some of the related work has already studied some possible strategies. Yet, I did not explore those any further as it was outside the scope of my work.

Before, I did not detail completely the implementation of the `startTx` method in the `Serial` scheduler. Actually, the scheduler only registers that the transaction is running if it is able to tell that there is no conflicting transaction running at that moment. For that, it uses the set of executing transactions and the conflict table. Otherwise, it puts the task, attempting to execute, in a thread-safe queue of the worker thread that is running the conflicting transaction. For that to be possible, the set of executing transactions actually contains a representation of the worker threads that are carrying those executions. This enqueue effectively serializes the transaction and avoids the conflict. It is for this reason that I refer to this scheduler as `Serial`.

For this to be possible, the worker threads in the application must implement the following interface:

```
public interface SchedulerWorker {
    public void acceptTask(Object work);
}
```

Listing 7.3: Interface implemented by the application worker threads.

Finally, when the worker threads query the scheduler for the next task, the `Serial` implementation first verifies if the thread doing the request was given any work in the meantime (by using the `Thread.currentThread()`). In that case, it returns that task for execution. Otherwise, it fetches the immediate next task in the set of tasks to execute. This entails that the application provides the JVSTM with the tasks to be executed, so that its scheduler can distribute them to the worker threads.

7.2 Using the **Serial** scheduler

In this section I provide a more detailed evaluation of the strategies used for scheduling. This is important so that, later in Chapter 8, it is possible to understand which benefits come from the scheduler and which come from the inner parallelization of transactions, separately. Then, I conclude by explaining how to take advantage of the scheduler to further explore parallel nesting.

I used the `Serial` scheduler in the workloads of `STMBench7`, and show in Figure 7.3 the speedup relative to a sequential execution. Note that I am not yet using any inner parallelism; this experiment is meant to assess the benefits obtained from the scheduler itself.

The scheduling itself provides an increase of 12% of performance in the read-dominated workload, and 20% in both read-write and write-dominated workloads. These values refer to the best performance obtained without scheduling, which in the read-write and write-dominated workloads actually means using only 1 thread. The most important conclusion is that the performance obtained with the scheduler is more or less stable on 1.2 speedup beyond 6 threads, conversely to the normal execution that generally yields slowdowns.

Considering the behavior of the scheduler, its *give-away* mechanism is inherently creating some sequential critical paths composed of the transactions that would otherwise conflict with each other if ran concurrently. The predictable consequence of having under-usage of processors is visible in Figure 7.4, where I present how much time each thread executed transactions during a write-dominated workload in the `STMBench7`. In particular, the right Figure contains a much more erratic distribution due to the usage of the scheduler when compared to the execution in the left figure that did not use scheduling; note that the vertical axis is significantly different in both graphics. As a matter of fact, it is relevant to

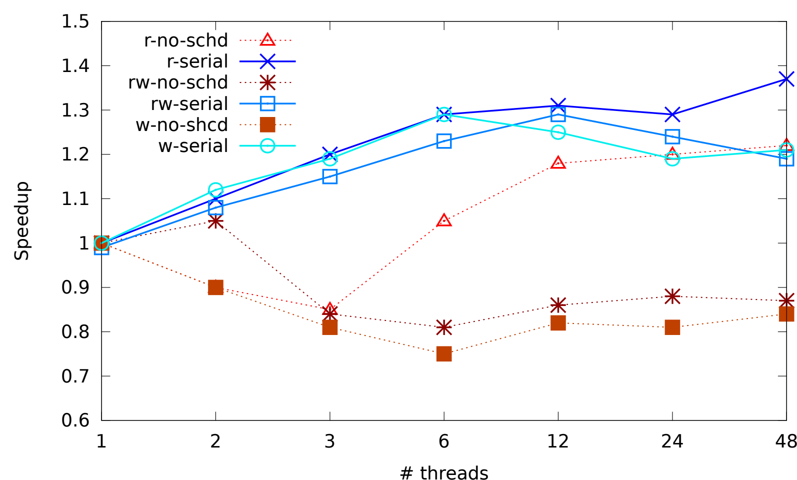
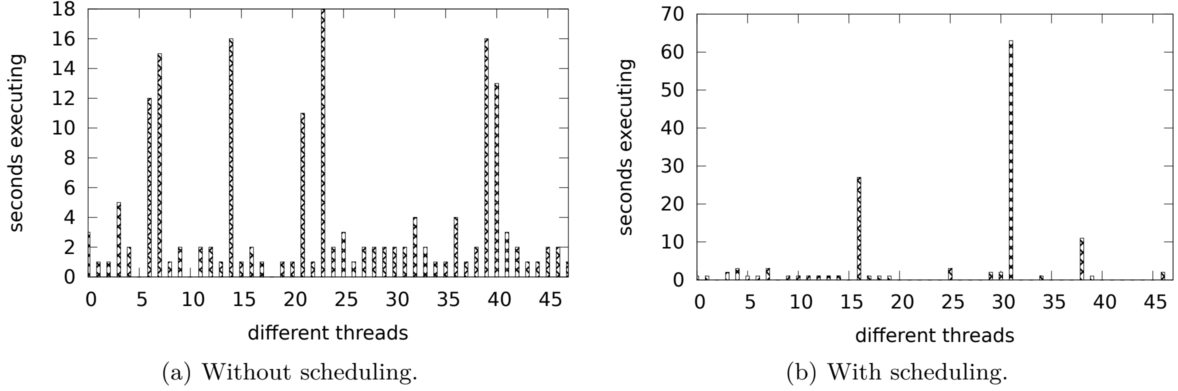


Figure 7.3: Performance obtained in various workloads of the `STMBench7` with and without scheduling.



(a) Without scheduling.

(b) With scheduling.

Figure 7.4: Distribution of the time spent executing transactions among the working threads in a write-dominated workload of the STMBench7 with long traversals.

take into account that there is more execution time spent in transactions that conflict and abort in the scenario without scheduling.

So far, we have seen that parallel nesting cannot always provide positive results, as shown in Figure 7.1 in the beginning of this chapter. On the other hand, we now have a scheduler that was able to improve the performance obtained in STMBench7. Yet, this resulted in having part of the available processors being idle for the execution of those workloads. This result was expected: If the workload is inherently conflicting, the transactions will contain some sequential critical paths free of conflicts that will lead to the successful execution. These critical paths necessarily impose an under-usage of the available processors if they are fewer than the processor count, which is by far the case in highly-conflicting scenarios. Therefore, the idea is that I may now explore these available resources to apply parallel nesting and improve the results even further.

I describe the general idea in Algorithm 8. When the scheduler is queried for the start of a new transaction, it iterates through the transactions that it has scheduled and that have not yet finished. For each transaction under execution, the scheduler estimates if it is predictable that the new transaction creates a conflict (lines 3-8). To do so, it uses the data collected so far regarding conflicts.

If a possible conflict is predicted, the scheduler forces the current thread to drop this transaction and serializes it in the queue of the thread that is executing the transaction that was likely to enter in conflict

Algorithm 8 Algorithm to schedule a transaction in Serial.

```
 1: RequestStart(TX):
 2: txStats ← conflicts.get(tx)
 3: for scheduledTx ← scheduledTxs do
 4:    if txStats.conflictsWith(scheduledTx) then
 5:       scheduledTx.executingThread.acceptTask(tx)
 6:       return false
 7:    end if
 8: end for
 9: tx.useParNest ← (scheduledTxs.size() - 1) < MAX_THREADS
10: newExecution ← makeExecution(txStats, currentThread())
11: scheduledTxs.add(newExecution)
12: return true
```

(line 5). Next, the scheduler returns false, meaning that it did not schedule the transaction in the current thread.

Otherwise, the transaction is accepted for execution. In this case the scheduler estimates if it is worth to use parallel nesting in this transaction. Line 9 establishes whether the scheduler allows parallel nesting in the current transaction, or if it should execute only as a sequential top-level transaction. Note that giving permission to execute with parallel nesting does not mean the transaction will actually do so. For that to happen, the transaction source code must have been augmented with parallel nesting by the programmer who identified the inner parallelization. This heuristic retrieves the number of transactions currently executing, which is an estimate of the current usage of the underlying processors. However, this is an under-estimate: If each execution taking place was allowed to use parallel nesting, and effectively does so, then the number of threads in use would be above the number of executions. This simple estimate was sufficient to prove the point that I sought when coupling the scheduler with parallel nesting, regardless of the room for improvement.

Chapter 8

Evaluation

In this chapter I present a twofold evaluation. First, in Section 8.1, I evaluate the implementation of the three parallel nesting algorithms that I presented in this dissertation and show that `InPlace` is the one that performs best. I also extend my contribution by addressing the particular scenario of embarrassingly parallelization of a transaction in Section 8.2. Then, in Section 8.3, I compare the performance of the `InPlace` implementation with two other implementations representing the state of the art in parallel nesting algorithms. A more detailed comparison is available in an article accepted in *WTTM 2012* [17].

8.1 Evaluating the different JVSTM-based implementations

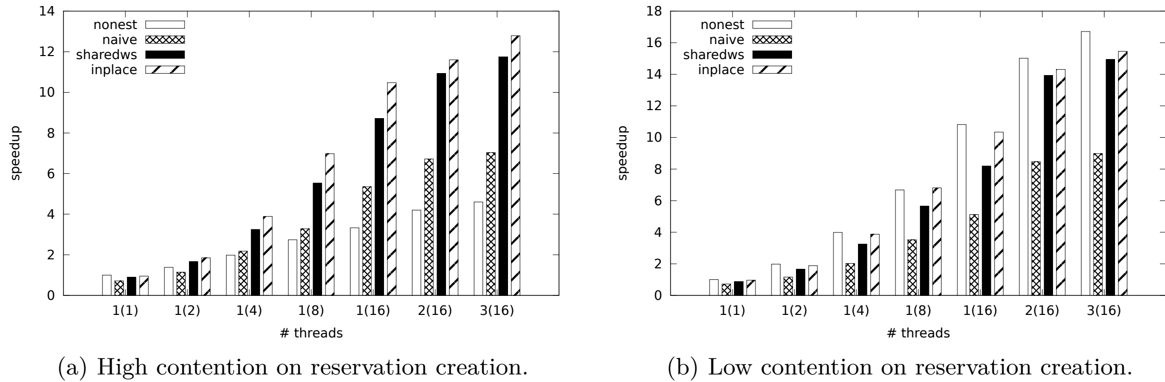
I begin by comparing implementations of the three algorithms proposed for parallel nesting in the JVSTM. We have already seen some promising data in Figure 6.2 in Section 6.9, regarding the performance of the corresponding transactional operations. Here, I complement that evaluation by assessing the behavior of each approach in terms of throughput in representative benchmarks.

The objective is to confirm the theoretical expectation that the `InPlace` variant is able to outperform both the `Naive` and `SharedWS` variants. The next two sections address that by providing an evaluation in `Vacation` (Section 8.1.1) and in `STMBench7` (Section 8.1.2).

8.1.1 Vacation

The `Vacation` benchmark of the `STAMP` suite represents the typical scenario that I motivated for earlier: Under high contention, it becomes increasingly hard to obtain improvements in terms of performance by adding more threads (and having a corresponding number of available processors). That is clear in Figure 8.1(a), where we may see that the `nonest` approach with only top-level transactions is unable to scale properly.

In this case, however, I was able to parallelize the transactions that compose the workload of the benchmark, and, therefore, use parallel nesting to run fewer top-level transactions at a time with each one spawning an increasing number of parallel nested transactions. When comparing the results of the



(a) High contention on reservation creation.

(b) Low contention on reservation creation.

Figure 8.1: Speedup of the various parallel nesting designs relative to the sequential execution in the `Vacation` benchmark. The approach of using only top-level transactions is also shown as `nonest`. The threads used are shown as the number of top-level transactions and number of nested transactions that each one may spawn. In the `nonest` approach, the number of top-level transactions used is the multiplication of those two numbers, so that the overall number of threads used is the same in all approaches.

different proposals described in this dissertation, we see that `InPlace` is able to obtain the best results with up to 2.8 times better performance than top-level transactions.

On the other hand, Figure 8.1(b) exemplifies a workload with low contention. In this case, the `nonest` is already achieving reasonable performance as the thread count increases. Thus, applying the parallel nested transactions does not yield any extra performance. As a matter of fact, we may actually see that there is some overhead from executing the transactions with some nesting.

In Figure 8.2, I specifically evaluate the overhead of executing with parallel nesting in the `Vacation` benchmark. The workload was executed with an increasing nesting depth with the various parallel nesting designs, and I measured the speedup relative to an execution without nesting. The nesting trees formed used a branching factor of one, thus taking no advantage of parallel nesting, and enabling me to assess the overhead of the algorithms. It is visible that the `Naive` design increases significantly its overhead as the depth is increased. On the other hand, both `SharedWS` and `InPlace` maintain their performance independently of the depth, with the latter presenting on average 5% of overhead in performance relative

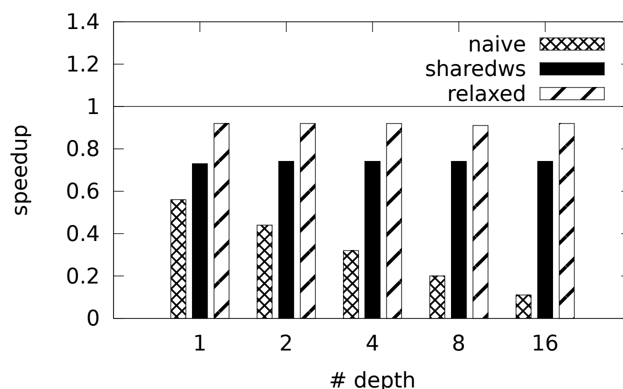


Figure 8.2: Speedup of each parallel nesting design in `Vacation` with a single thread. Each plotted execution uses an increasing nesting depth in which each top-level transaction creates a nesting tree with a branching factor of one. The speedup is computed relatively to the execution without nesting, thus making the overhead of nesting visible.

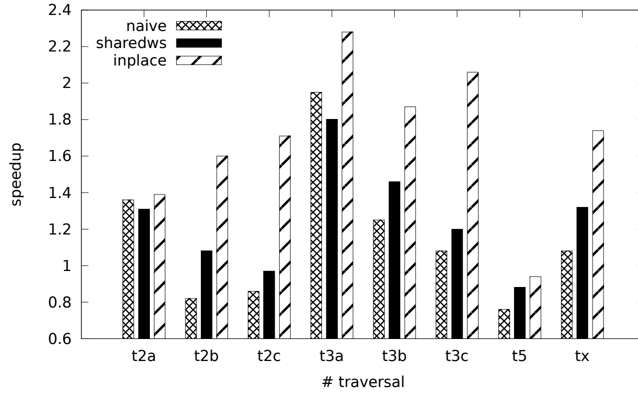


Figure 8.3: Speedup of the read-write long traversals of the STMBench7 relative to its sequential execution. The three parallel nesting designs were used to explore the inner parallelism of the traversals with up to three threads each. The *tx* execution refers to a combined mix of all the read-write long traversals.

to top-level execution. These results were similar in an experiment using 16 threads, and varying the nesting depth.

Of course, the overhead of using parallel nesting is not typically only 5%. We have other sources of overhead besides the extra complexity of the parallel nesting algorithms. For instance, the overhead of creating tasks and synchronizing on those tasks' completion, which depend largely on the granularity of the tasks. I shall address this with more detail in Section 9.2.

8.1.2 STMBench7

I now explore the performance of each design in STMBench7. Conversely to Vacation, not all transactions in STMBench7 have latent parallelism. Thus, in Figure 8.3, I start by presenting the speedup obtained for each one of the transactions that I parallelized.

The read-write long traversals available in this benchmark are particularly troublesome for obtaining an increase in performance in workloads that contain them. As explained in Section 2.5, these traversals access most of the object graph of the benchmark, both with read and write accesses. This precludes

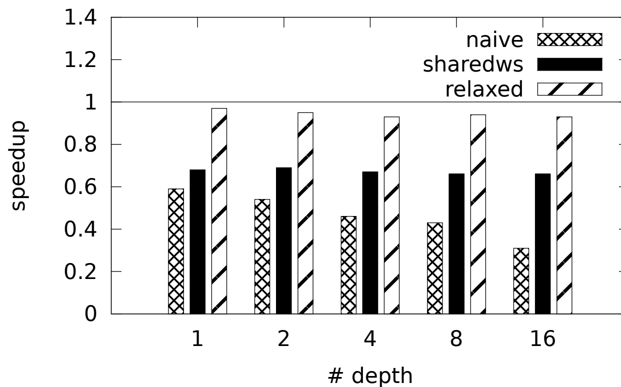
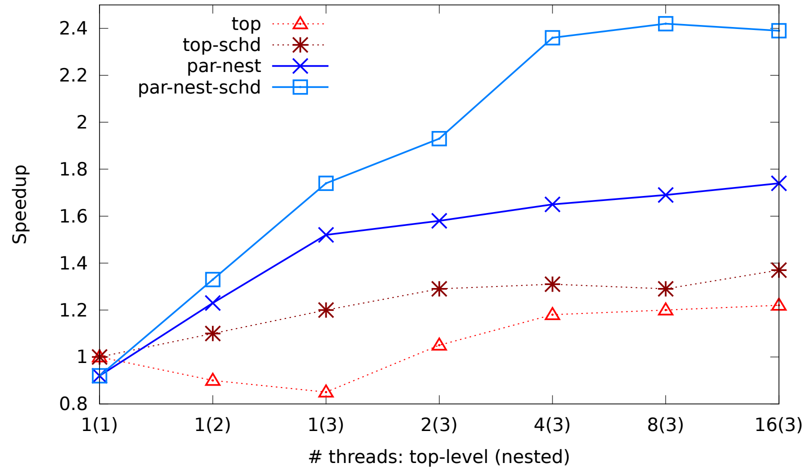
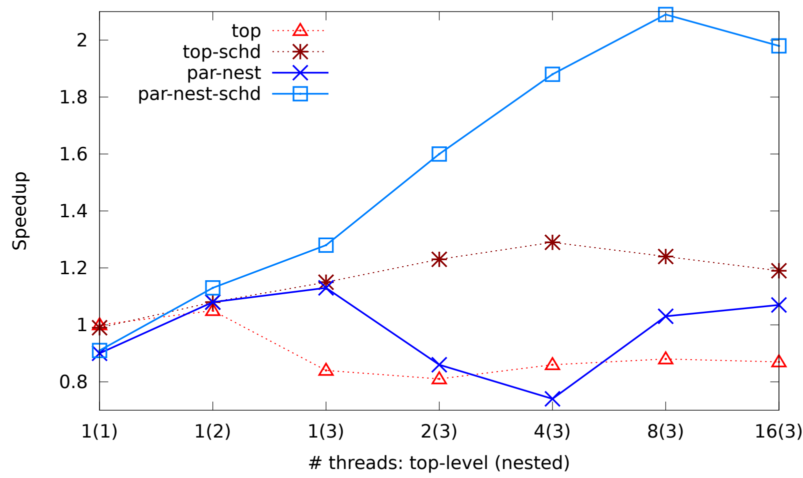


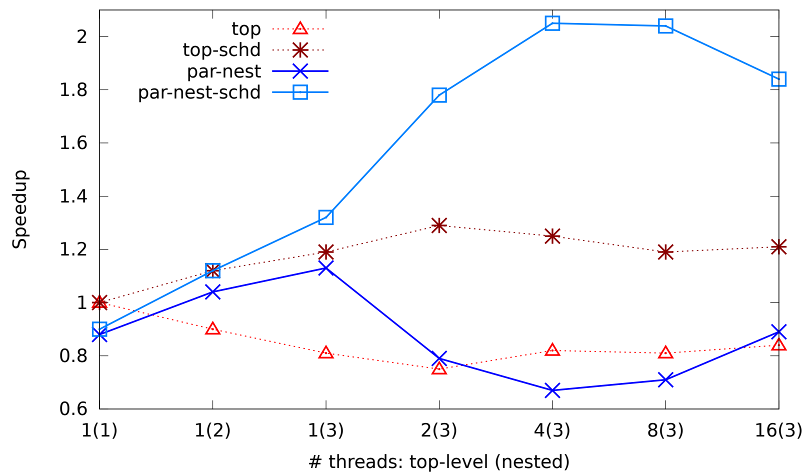
Figure 8.4: Each plotted execution uses an increasing nesting depth in which each top-level transaction creates a nesting tree with a branching factor of one in STMBench7 with write-dominated workload. The speedup is computed relatively to the execution without nesting, thus making the overhead of parallel nesting visible.



(a) Read-dominated workload.



(b) Read-write workload.



(c) Write-dominated workload.

Figure 8.5: Different workloads of the `STMBench7` with long traversals with four combinations: (1) normal execution with top-level transactions; (2) the first case with the serial scheduler; (3) using parallel nesting; and (4) using parallel nesting and the serial scheduler.

much of the possible concurrency, as TMs have to detect conflicts in most concurrent accesses with these traversals. Therefore, I exploited the inner parallelism in each of those traversals and show the resulting performance relatively to an execution of those traversals with a top-level transaction only. Traversals $t2b$, $t2c$, $t3b$, and $t3c$ are particularly large in terms of their footprint, which allows the `InPlace` design to widen the benefits attained relatively to the other two designs. In particular, some slowdowns are visible with both `Naive` and `SharedWS`. Traversal $t5$ proved to be difficult to parallelize: None of the designs was particularly helpful in this case. Besides the individual results, I also present the results for a combined mix that I created with these traversals only (under the label tx).

Furthermore, I repeat the same experience as in the last section to assess the overhead of each design. For that, I executed the write-dominated workload with long traversals in `STMBench7` and present the results in Figure 8.4. Recall that, similarly to Figure 8.2 in the last section, the benchmark is executed with a variable nesting depth that I created by modifying the benchmark. In this experience the `InPlace` design has an average of 8% overhead.

Finally, I present the results corresponding to the execution of all the workloads in `STMBench7` with long traversals enabled. Here, I use the `Serial` scheduler and the `InPlace` design, which has been shown to perform best. Each plot in Figure 8.5 corresponds to a different workload and shows the speedup obtained in different situations. The intermediate steps of either the scheduler and parallel nesting alone serve to assess how each component is contributing to the final result. Then, we may see that the best results obtained when comparing to the baseline approach with top-level transactions only is approximately 2 times better throughout all the workloads. These results confirm the thesis posed in this dissertation in yet another known benchmark.

8.2 Relinquishing the overhead of parallel nesting

Despite the overhead identified and quantified in the previous section, my expectation was that the newly explored parallelism (with fewer conflicts) could mask that overhead. That expectation proved to be correct for both `Vacation` and `STMBench7` benchmarks.

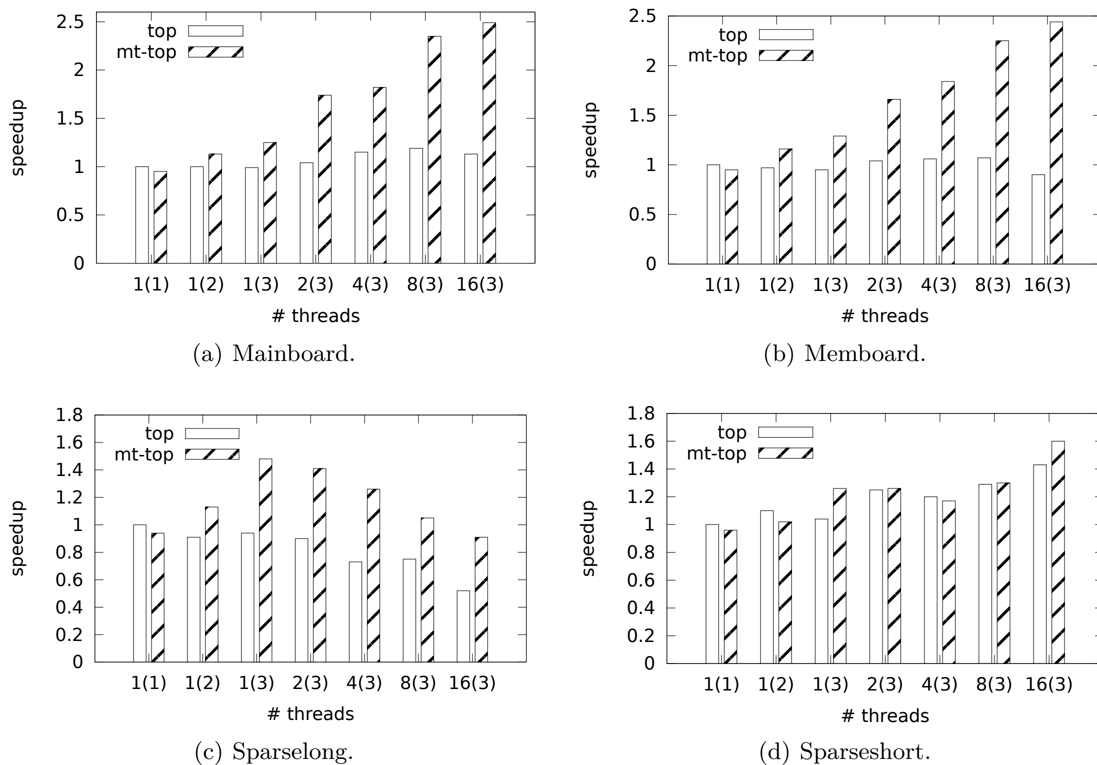
When applying the same strategy to the `Lee-TM` benchmark, however, it generally yielded slowdowns: I parallelized the only transaction identified in `Lee-TM` by creating parallel transactions in the expansion task, but that code was too quick and executed many times. Therefore the overhead of managing, validating and committing parallel nested transactions was supplanting the benefit that could be achieved with that parallelism.

But while doing that parallelization I realized that I was in the presence of embarrassingly parallelism. More specifically, I could guarantee that the parallel nested transactions that I was creating would never conflict with each other in the same nesting tree. This is a case of embarrassing parallelism. In this case, I can use several threads running in the context of the same top-level transaction. The benefit of this is that the transactional operations follow the same algorithm as a top-level transaction, rather than the more complex version presented in this dissertation for nested transactions. Additionally, when they finish their work, these threads do not need to execute any validation or commit.

In Figure 8.6, I show the application of this strategy compared with the usage of single-threaded top-level transactions. Once again, we may see that adding more top-level transactions does not necessarily translate into better scaling. In turn, allowing each top-level transaction to run faster yields a better

performance, being slightly over 2 times better at 48 threads in both `mainboard` and `memboard`. In Figure 8.6(d), this multi-threaded top-level transactions approach does not yield benefits as consistently as in the other boards. This happens because the duration of each track laid in the workload is much smaller than in the other cases. The parallelization that I identified in the benchmark takes place only when the tracks have a minimum size to make it attractive to parallelize. Therefore, in the `sparseshort` board, a top-level transaction execute most of the time without more than one thread.

The expansion phase of the `Lee-TM` benchmark collects many transactional reads, which induce an increased likelihood of conflict as the transactions get larger. This can be overcome by applying an early release technique [32, 23] to obtain better scaling with top-level transactions. Yet, it was an interesting case to illustrate embarrassingly parallelism that may arise in top-level transactions. The solution of running more than one thread in the context of a single transaction is possible only if the structures of the transaction are thread-safe. To minimize the costs of synchronization, I use a strategy similar to what was described in the `InPlace` algorithm: Both read-set and write-set are scattered across the threads executing in the same transaction, and accessed by the thread that is running the top-level transaction when it commits globally. For this to be possible, these threads use the `commmittedChildren` field of the top-level transaction in the same way as described for the `InPlace` design.



(a) Mainboard.

(b) Memboard.

(c) Sparselong.

(d) Sparseshort.

Figure 8.6: Usage of embarrassingly parallelized top-level transactions in `Lee-TM` with several test boards. The `top` executes single-threaded top-level transactions only whereas the `mt-top` uses more threads per top-level transaction. The threads used are shown as the number of top-level transactions and number of threads that may run in each top-level transaction. In the `top` approach, the number of top-level transactions used is the multiplication of those two numbers.

8.3 Comparison against the state of the art

In this section I compare the `InPlace` algorithm proposed in this dissertation against two STMs with support for parallel nesting in the related work, the NesTM [7] and PNSTM [9]. Both these alternatives provided a detailed description of their algorithms (contrarily to most of the other related work). Furthermore, the authors of NesTM were the only ones providing an evaluation beyond micro-benchmarks with fair performance gains, which strengthens its position as representative of the state of the art. On the other hand, PNSTM represents the algorithm for parallel nesting with the lowest worst-case complexity bounds for all the STM operations.

Next, I compare the worst-case complexity bounds of each of the three STMs, in Section 8.3.1. Then, in Section 8.3.2, I present the results of experiments that I conducted to assess the practical impact of the design choices of the STMs. I implemented both NesTM and PNSTM in Java according to the algorithms in their publications. This was the only alternative given that they are not publicly available nor was I able to reach the corresponding authors. To make the comparison fair, I also defined an API for these implementations that allows the applications to specify which locations are transactional. Moreover, it is important to take into consideration that this evaluation is necessarily comparing also the underlying TM designs. Because of that, I also executed the workloads using only top-level transactions, so that we may have an idea of how the baseline TMs compare with each other.

8.3.1 Worst-case complexity bounds

I have briefly described both STMs in previous sections: NesTM in Section 3.5.1 and PNSTM in Section 3.5.3. Both provide opacity [28]. Yet, in the rest, they explore different design choices. As a consequence, they provide different bounds and guarantees to applications using them. Table 8.1 lists the worst-case complexity bounds in the operations of a parallel nested transaction in the mentioned STMs, where: k is a constant value defined statically in NesTM; r and w are the size of the read-set and write-set, respectively; $children$ is the number of descendants of a transaction; $maxDepth$ is the maximum depth of the nesting tree; and $txDepth$ is the depth of the transaction executing the operation.

JVSTM is the only one where the read operation depends on the maximum depth of the nesting tree. This worst case may happen if the variable being read has been written by all the transactions in the deepest branch of the nesting tree (and necessarily different from the branch of the reader transaction). The read in NesTM may need to be repeated k times, where k is defined statically. The access has to read the value and lock separately, and thus uses a double read pattern to ensure a consistent reading. A repetition takes place when a concurrent abort occurs between the double read. Finally, the PNSTM only needs to look at the top of the access stack to decide in constant time if the access is conflict-free.

Regarding the write operation, both JVSTM and PNSTM need only to insert the new value in a single

	JVSTM	NesTM	PNSTM
read	$O(maxDepth)$	$O(k)$	$O(1)$
write	$O(1)$	$O(txDepth)$	$O(1)$
commit	$O(r + children)$	$O(r + w)$	$O(1)$

Table 8.1: Complexity bounds for the worst-case of transactional operations in parallel nested transactions.

location (reachable in constant time). On the other hand, NesTM may need to validate the variable being written in all the ancestors' read-sets.

The commit operation is also performed in a constant number of operations in PNSTM. On the other hand, NesTM needs to validate the read-set and to propagate the ownership of the writes to the parent. JVSTM also requires the validation, which cannot be avoided given its lazy update nature [1], but has a different bound for the propagation of the write-set: It depends on the number of committed children of the committer.

The bounds presented thus far are clearly in favor of the PNSTM but, in practice, what affects more the performance of an STM is the complexity of the common case, rather than the complexity of the worst case, if they are rarely the same. For instance, the read operation of JVSTM has a worst case that seems very unlikely to happen in practice because it requires that all the transactions in a nesting branch write to the same variable. This single fact is of great importance as the read operation is typically predominant in applications. Futhermore, as explained in Section 6.3, the common case for the read operation in the JVSTM ends up being $O(k)$ where $k \ll depth$.

Moreover, providing constant time operations in parallel nesting does not come for free. The design choices of the STMs are reflected in the types of conflicts that may lead to abort, which I summarize in Table 8.2.

As we may see, there is a relation between the complexity bounds and the conflicts detected: The cheaper the worst case complexity bounds are, the more conflicts the STM has to detect to guarantee correctness. There is no perfect solution as we are being faced with a trade-off. Despite having constant-time worst-cases, PNSTM limits severely the concurrency of transactions whose footprints intersect with each other.

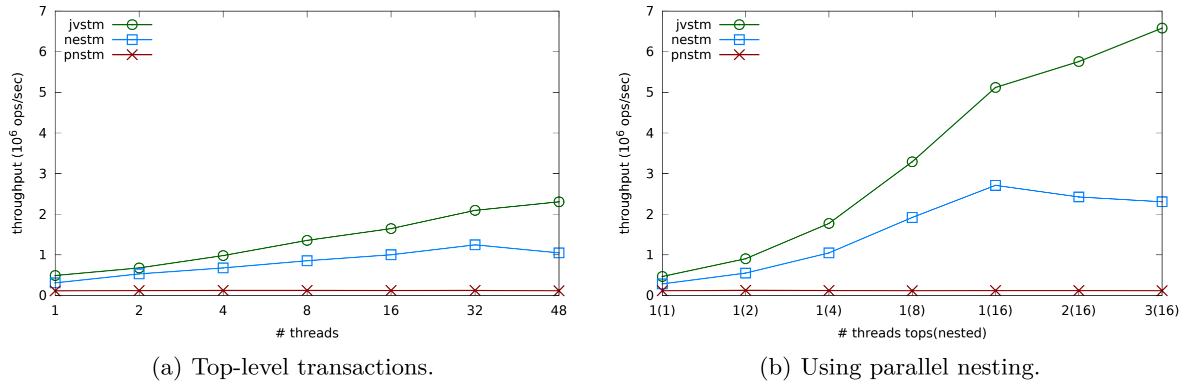
	JVSTM	NesTM	PNSTM
r-r	-	-	yes
r-w	yes	yes	yes
w-w	yes (if nested)	yes	yes

Table 8.2: Possible conflicts that may lead to abort in each STM. Note that, in the case of the JVSTM, this refers to read-write transactions only, because read-only transactions never abort.

8.3.2 Practical comparison

I begin by comparing the three STMs in the `Vacation` benchmark, for which Figure 8.7 presents the throughput of each STM in a scenario with high-contention. Looking at Figure 8.7(a), we may see that JVSTM is considerably faster than the alternatives when using only top-level transactions. In particular, it achieves 1.58 speedup over NesTM and 5.41 speedup over PNSTM with a single top-level transaction. Given that the baseline JVSTM is already faster than NesTM, it is expected that using parallel nesting is also faster in JVSTM. We can see that in Figure 8.7(b). Still, the actual improvement of using parallel nesting instead of top-level transactions (for 48 threads) is greater for JVSTM (2.8 times) than for NesTM (2.2 times) and PNSTM (no improvements).

I now present similar experiments to compare the three STMs, but using a write-dominated workload of `STMBench7`. The results are shown in Figure 8.8, where we may see that JVSTM is again faster already with only one thread: It is 2.6 and 3.4 times faster than NesTM and PNSTM, respectively, when



(a) Top-level transactions.

(b) Using parallel nesting.

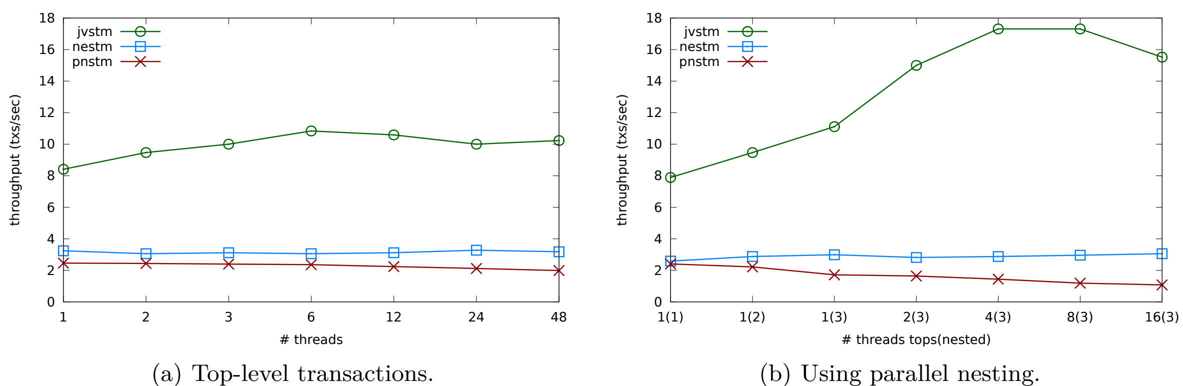
Figure 8.7: Throughput in a contended workload in the `Vacation` benchmark. When using parallel nesting, the number of top-level transactions used is followed in parentheses by the number of nested transactions that each one may spawn.

using only top-level transactions. Moreover, even though I do not show them, the results are very similar across the other workloads of the `STMBench7`.

Yet, unlike the results obtained in the `Vacation` benchmark, in this case the parallel nesting algorithm of NesTM is unable (together with PNSTM) to obtain improvements over its execution with only top-level transactions, whereas JVSTM more than doubles its throughput when using parallel nesting.

Finally, in Figure 8.9, I present results that show how each STM performs when the nesting depth increases. To obtain these results, I modified both benchmarks so that they execute all of their transactions entirely within a single nested transaction at a certain depth. This yields a nesting tree with a single branch that is increasingly deeper as the nesting increases up to a level of 128. I also performed this experiment using 16 threads, and obtained similar results (not shown here).

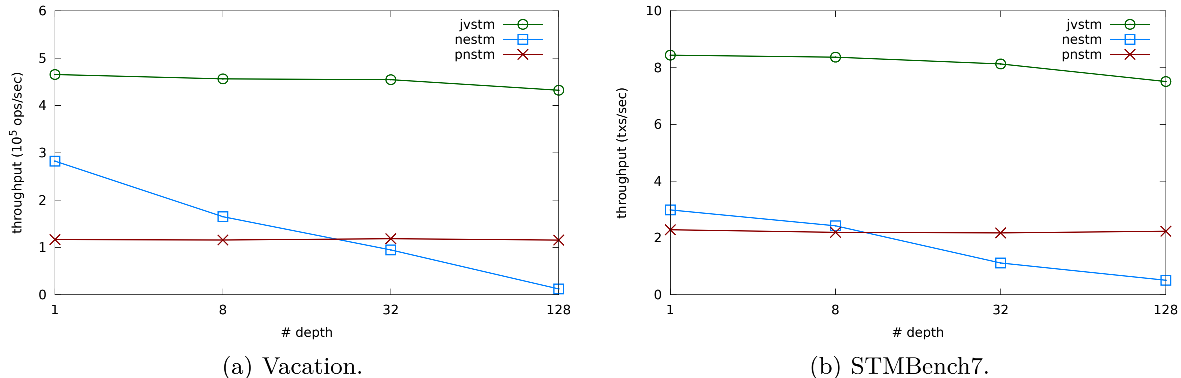
These results are consistent with the theoretical complexity bounds of each STM. Namely, PNSTM performs independently of the nesting depth, whereas the other two degrade their performance. However, JVSTM not only performs significantly better, but it also degrades at a much slower rate than NesTM. This behavior is similar in both benchmarks shown and is representative of the data that I collected in several other workloads.



(a) Top-level transactions.

(b) Using parallel nesting.

Figure 8.8: Throughput in the write-dominated workload of the `STMBench7` benchmark. When using parallel nesting, the number of top-level transactions used is followed in parentheses by the number of nested transactions that each one may spawn.



(a) Vacation. (b) STMBench7.

Figure 8.9: Throughput obtained with a single nested transaction at an increasing nesting depth in `STMBench7`.

So, just as PNSTM gets better results than NesTM for a sufficiently high depth, I expect the same to happen also, at some depth, with regard to JVSTM. Yet, given the slow decay of the JVSTM, that will require a much higher nesting depth (note that the horizontal axis is growing exponentially). In fact, I argue that such depth would seldom, if at all, be seen in real applications.

8.3.3 Discussion

A naive interpretation of the worst-case complexity bounds for the STMs presented in Section 8.3.1 could lead us to conclude that PNSTM should obtain the best performance (followed by NesTM). Yet, the results shown in the previous section contradict that conclusion. This happens for two reasons: (1) the execution of a transactional operation does not always fall under the worst-case; and (2) the amount of conflicts detected influences greatly the resulting performance. In this section I look at both reasons to understand why JVSTM obtained the best performance despite the analysis in Section 8.3.1.

I first look at the usage of fast paths in the algorithms that avoid the worst-cases. For that, I used an execution with 48 threads in the write-dominated workload of the `STMBench7`. For JVSTM, I verified that the in-place metadata was used in 43% of the reads, corresponding to read-after-writes. Moreover, the remaining 57% of the reads were able to avoid checking the in-place metadata in over 99% of the times. This means that nearly all the time is spent in the fast paths of JVSTM.

Table 8.3 presents this data together with the corresponding data for the other two STMs. I considered that the fast path in both of them is the read-after-write, because, given their eager in-place nature, such operation is considerably cheaper than a normal read. JVSTM has two possible fast paths: one when it reads in-place, and another when it reads a globally consolidated version without having to check the in-place slot. As we may see, JVSTM is able to execute nearly all of the times in either of these two modes, whereas the other two STMs go through their fast path only 39% of the times. Moreover, when reading

JVSTM			NesTM		PNSTM	
Fast Path	Slow Path	Read In-Place	Fast Path	Slow Path	Fast Path	Slow Path
0.56	0.01	0.43	0.39	0.61	0.39	0.61

Table 8.3: Fraction of occurrence of each type of read per STM in an execution of a write-dominated workload in `STMBench7`.

JVSTM			NesTM			PNSTM
R-W (eager)	R-W (commit)	W-W (nest)	R-W (commit)	Spurious	R-W (eager)	R-W (eager)
465	177	203	39	1465	123	84496

Table 8.4: Occurrence of conflicts per STM in an execution of a write-dominated workload in STMBench7.

in-place, JVSTM was able to obtain a write entry immediately without incurring in the worst-case of the read operation in over 99% of the time.

In practice this means that the complexity of the read operation in the JVSTM is $O(1)$, the same as PNSTM. Thus, what differs between them is how efficient each implementation is. The design decisions of the JVSTM make its fast path considerably simpler than PNSTM's. This fact is visible in the time that it takes to execute a transaction (measuring only executions that commit): JVSTM 1046μs; NesTM 5200μs; PNSTM 7357μs.

But having a more efficient read operation in the common case is not the only reason for the better performance of the JVSTM. The second reason is the degree of concurrency that each STM allows. To assess the importance of this second reason, I show, in Figure 8.4, the conflicts registered in an execution with 48 threads in the write-dominated workload of the STMBench7. These results are consistent with executions with a decreasing number of threads and show that JVSTM had the least number of conflicts detected. In fact, NesTM detected approximately twice more conflicts, and PNSTM detected a hundred times more conflicts.

Adding to this, I also measured the percentage of transactions that failed in their first attempt in an execution with 48 cores. The results for Vacation and STMBench7 are presented in Table 8.5.

The general trend of these statistics follows the expectation according to the conflicts that are detected in each STM (as presented in Figure 8.4): The less conflict types detected, the smaller the percentage of unsuccessful transactions. Note that the number of conflicts in the execution of STMBench7 is much smaller because of a serialization imposed by a scheduler. Nevertheless the relative degree of conflicts between each STM is consistent with the other results and withstands the point raised.

The data presented in this section backs up the two reasons pointed out for the higher performance of JVSTM. Even though this comparison was performed against my implementation of the other STMs (rather than implementations provided by their authors), I believe that the conclusions still hold, as we may single out some aspects that explain why JVSTM obtained the best results: Its multi-version property allows more concurrency (by detecting less conflicts), which results in less aborts and re-executions. Moreover, JVSTM has simpler fast paths, which are also used more frequently than in the other two STMs. That is a consequence of the different design decisions, which seems to indicate that the JVSTM represents a sweet spot in the design space of STMs due to its superior performance, while at the same time providing stronger progress guarantees.

Conversely, NeSTM suffers mainly from its costly validation procedure, naive merging at commit-

	JVSTM	NesTM	PNSTM
STMBench7	0.1%	0.4%	6%
Vacation	27%	36%	98%

Table 8.5: Percentage of transactions that failed in their first attempt.

time, and spurious aborts (which serve to avoid livelocks heuristically). On the other hand, in the case of PNSTM most of the overhead comes both from the reads that have the same costly path as the writes, and from its read-read conflict detection.

This also strengthens the idea that we cannot reason about the performance of each parallel nesting approach independently of the underlying TM used, because the quite different design choices of the underlying TM may have a significant effect on the performance of the parallel nesting algorithm. In particular, PNSTM was designed specifically with the purpose of providing a parallel nesting algorithm that performed completely independent of the nesting depth. Unfortunately, that decision makes it extremely inefficient in practical applications such as those mimicked by these benchmarks. A similar conclusion applies to NesTM because it uses single versions and, thus, read-only transactions may abort.

8.4 Summary

In this chapter I began by confirming the expectation that the JVSTM `InPlace` design would perform better than the `Naive` and `SharedWS` designs when exploring parallel nesting. This is visible not only in the fact that it produces the least overhead, but also in the fast paths that it offers even under high concurrency and contention.

Then, I compared two other STMs that support parallel nesting with the JVSTM. That comparison suggested that the JVSTM would behave worse due to the complexity of the worst cases of its operations. Yet, it is also important to take into account the design decisions that enable those complexity bounds.

To assess the consequence of those decisions, I provided an evaluation in two well-known benchmarks. Although the parallel nesting algorithm of the JVSTM is theoretically worse, it contributed efficiently to the improvement of the performance over its baseline implementation. On the other hand, the alternatives either obtained weaker results or could not improve at all. In particular, PNSTM was tailored to suit better parallel nesting but failed to be adequate to a wide variety of workloads.

These results support the thesis statement of this dissertation both in terms of alternative designs for the JVSTM as well as by comparing the chosen design against state of the art alternatives in the related work.

Chapter 9

Conclusions

The widespread growth of parallel computation power has unveiled the concern for the development of scalable applications. In the context of synchronizing the access to shared data in these applications, I addressed the pitfalls of mutual exclusion resorting to locks, and how the Transactional Memory abstraction may solve those issues.

Transactional Memory supports the composability of different software components via nesting of transactions. This nesting capability provides more flexibility to an application using a TM. However, it is still limited when code that runs within the scope of an atomic block makes use of multi-threading.

In this dissertation I explored parallel nesting, which is a mechanism that overcomes that limitation in the expressiveness of TM. I additionally used it in a novel approach to overcome the problems caused by the optimistic concurrency control mechanisms used by TM in highly-conflicting workloads. Next, in Section 9.1 I elaborate on the contributions of my work. I discuss the challenge of finding latent parallelism, in Section 9.2. Then, in Section 9.3, I conclude with some future directions for research.

9.1 Main Contributions

To accomplish the goals of this dissertation, I presented three algorithms with different design choices for parallel nesting. These incrementally improved over previous designs by addressing challenges identified in earlier stages. The `InPlace` algorithm is my main contribution: It is a low-overhead parallel nesting algorithm for a multi-version STM that successfully tackled all those difficulties. With that algorithm we can obtain the following benefits:

- *More expressiveness*: I provided a more flexible TM in which it is possible to parallelize transactions as opposed to the traditional perspective that has seen transactions as a sequential set of instructions. It is perfectly acceptable that an application programmer identifies some code as meant to run within a transaction, while at the same time willing to parallelize that same code. In that sense, I provided the tools that allow him to do so. In particular, the reason behind wanting to do that may very well be to overcome the limitations of TM in conflicting workloads such as those that I identified in this dissertation. Yet, I point out that it is also possible to envision automatic parallelization tools that may create parallel tasks within the control flow of parallel code. In fact,

this has been attempted by using TM to synchronize the access to shared data [3], in which case the work presented in this dissertation is once again crucial.

- *Better performance*: The algorithms that I proposed and the challenges overcome during their creation constitute an important understanding of parallel nesting: As shown with the initial approach, it is not trivial to design a parallel nesting algorithm that supports unlimited depth without incurring into excessive overheads that preclude the benefits of the parallelism being explored. I showed that parallel nesting can be used to improve the performance obtained with TM in some highly-conflicting workloads. For that, I presented the first complete implementation and evaluation that took advantage of parallel nesting to overcome highly-conflicting workloads in known benchmarks.
- *Unaffected performance for normal transactions*: The solutions that I proposed never put at risk the performance of the underlying TM. This means that the performance of top-level transactions remains unchanged even though the TM now supports parallel nesting. This happens because the changes that I described are either negligible or are outside the normal path of execution for a top-level transaction. This fact is of great relevance, as much of the related work that provided parallel nesting either created a new TM from scratch that suited the needs of parallel nesting (such as eager update and conflict detection) or affected the performance of normal transactions. Typically these hand-crafted TMs provided elegant or theoretically efficient parallel nesting algorithms, but were not adequate for general use in various workloads as a TM should be. Another important contribution is that my algorithm is the first to support multi-versions.

Moreover, I contributed with the creation of a scheduler that can be used to improve the results of parallel nesting. The `Serial` scheduler has the following benefits:

- *Take advantage of scheduling*: I showed that the benefits obtained with parallel nesting are substantially increased when a conflict-aware scheduler is used. This is interesting as I did not directly explore the benefits of performance that a scheduler may yield due to the reduction of conflicts. Instead, I explored a side-effect of scheduling to decide when to use inner parallelism in a transaction. This is, in itself, another benefit that may be obtained from using scheduling, beyond the more obvious reason that usually motivates for it.
- *Automatized decision to use parallel nesting*: Another important benefit concerns the integration of the scheduler with the STM. I embedded the `Serial` scheduler in the JVSTM and provided an API that the programmer may use for his applications to take advantage of the scheduler. This enriches the set of tools that I give to a programmer that seeks to parallelize transactions: He no longer has to decide when to use parallel nesting, because the scheduler (within the TM) can decide that with runtime information about the conflicts.

9.2 The problem of finding latent parallelism

In Chapter 8 I showed the performance benefits obtained with parallel nesting. Namely, I paralellized the read-write long-traversals of the `STMBench7` and presented the gains over the sequential execution. Generally, the use of the `InPlace` variant improved over the performance without parallel nesting.

Yet, even if a programmer uses the tools that I provide in this dissertation, he is not guaranteed to obtain improvements. A key observation in my evaluation is that the best results are attained when two

conditions are met: (1) top-level transactions fail to deliver significant improvements with the increase of parallel threads, because of contention among the transactions, which inhibits the optimistic concurrency severely; and (2) each top-level transaction contains some substantial computation that is efficiently parallelizable.

Yet, traversal $t5$ was a different case, as its parallelization did not result in any improvements. This happened for two reasons: (1) the identified parallelization caused conflicts between the siblings; and (2) the traversal is 10 to 100 times smaller than the other traversals that were successfully parallelized.

The first issue has been a cause of concern in a wide variety of topics. This particular example may be seen as a more general challenge of how to parallelize code. Generally, the difficulty is in doing so in such a way that the synchronization of the parallel tasks does not overwhelm the benefits of the parallelism. In the example, this is happening in the form of the conflicts created between the parallel nested transactions. Therefore, the performance gains that may be obtained with the strategy that I proposed in this dissertation are tightly related with the way the code is parallelized. That has been in itself a source of many research efforts, but nevertheless relevant to this work.

To evaluate my work, I parallelized three different benchmarks. Yet, doing that parallelization was not a particularly easy task. The difficulty of identifying parallelism is yet another well known challenge in the research community, which led to some research efforts that explored automatic parallelization and programming language constructs to ease the job of the programmer. Consequently, the difficulty of exploring nested parallelism is contradictory to the idea that TM is meant to be simple to use. Of course, nested parallelism may be delegated to experts, but then the general programmer may be losing performance gains that would be possible to obtain with TM. In practice it could almost be considered a process of fine-tuning the performance for specific workloads or transactions. This open issue falls in the same category of many other constructs built over TM: There is a constant trade-off between preserving its simplicity, or making it more powerful but at the same time harder to use.

The second point takes into consideration the amount of work being parallelized. Although tightly related to the previous point, it also has other implications. In this case, the loss of performance is due to the overhead of the management of transactions. Such overhead results from the burden of going through the start, validation and, commit of transactions that encapsulate code that would otherwise be executing in single thread within a top-level transaction. Even more, transactional reads and writes are also necessarily more complex in this setting. Consequently, this boils down to the challenge of providing parallel nesting such that this overhead does not overwhelm the paralellization obtained. Of course, as the size of the code to parallelize gets smaller, the easier it is for the overhead to be noticeable. Even though I was able to reduce this overhead substantially, as shown in several benchmarks with the `InPlace` design, ultimately it is always an open issue unless the algorithm has no overhead at all.

9.3 Future research

In this section I discuss directions that can be followed in future research related to the work described throughout this dissertation. None of these issues is directly related to the goals that I proposed to attain and, thus, were left out for future exploration as they would constitute a considerable amount of work. Nevertheless, I think that the following directions are topics worth of research effort.

9.3.1 Parallel TM

This dissertation showed that the existing parallelization in some workloads may not efficiently take advantage of all the available processors. Namely, because blindly using all of them to run top-level transactions may yield many conflicts that preclude concurrency and thus inhibit performance gains. My solution to this problem is for the programmer to identify new parallelism to be explored within transactions.

An interesting topic for future research would be to parallelize the TM itself. This would imply studying and changing the algorithms used in the operations of TM to allow them to be executed by several threads. One trivial example is the validation procedure at commit-time: Iterating the read-set to ensure that each read is still consistent is a parallelizable operation. I also envision other possibilities such as parallelizing the lookup for a RAW in the read operation.

Moreover, this idea would still benefit from the scheduling that I described in this dissertation: A transaction would only use more threads if the scheduler had granted it permission to do so.

9.3.2 Threads and Transactions

The best practices in software development encourage programmers to modularize their code and to abide to well defined interfaces [42]. As we have seen, this explicitly contradicts lock-based concurrency, in which the programmer has to be aware of modules' internal locking conventions. However, that is not the case for transactions, which provide composability using nesting. Unfortunately, this is not so simple when we take into consideration that multi-threaded code may exist in the control flow of transactions. Here, I shall refer to threads to designate some form of execution of parallel code without loss of generality.

In the literature there has been an implicit coupling between a transaction and the thread in which it is running. In particular, if a thread starts a transaction already in the context of another transaction, the new one will be nested in its outer enclosing transaction. Moreover, it will run in the same thread. This is considered to be the normal case, in which the code within the transaction is sequential and transactions compose naturally.

Consider now an application that uses TM and some of the transactions identified are wrapping code that internally resorts to threads. The consequence is that the multi-threaded code that is executed within those threads will escape the control of the TM system. This is a concern that crosscuts programming languages, although in this work I turn my attention to Java. A transaction is managed at a given thread, but nothing should forbid other threads from being included in that transactional context as well. The underlying problem is that threads are completely agnostic of transactions. Therefore, there seems to exist a need of identifying their relation and to make threads aware of transactions so that both may be used transparently in a composable manner.

It has been consensual that creating a new transaction should always mean that it executes in the current thread and in the context of a possibly already executing transaction. What is left to define is what happens when a thread is spawned by a thread that is executing a transaction. In this case, I identify three possible scenarios. To simplify the discussion, let us consider that when a thread Th_i spawns another thread Th_j, Th_i is the father of Th_j:

- Thread Th_j executes in the context of a new transaction. This new transaction is a child of Th_i's

transaction. This scenario is representative of the parallel nesting model presented in Section 4.2: There is a one-to-one relation between transactions and threads. Moreover, the childhood relationship created between threads is mimicked in transactions as well. This is the safest alternative in terms of accesses to shared data as there is no danger of breaking the correctness criterion with the new threads being executed in parallel.

- Thread Th_j executes in the context of the transaction that was running in its father. Ultimately, this means that a single transaction may have multiple threads concurrently running in its context. That may happen if Th_i spawns more than one thread for concurrent execution. I addressed how this scenario may be implemented and applied to in Section 8.2. In short, it makes sense when the concurrent threads spawned are completely independent from each other, but may not be independent from other unrelated threads. Eventually, if Th_j contains code that starts a new transaction, then this transaction composes with the transaction of Th_i.

- Thread Th_j escapes the transactional context of its father. This happens if Th_j encapsulates its execution in a new top-level transaction or none at all. In any case this may be seen as an irrevocable action: If the transaction running in Th_i aborts, the actions performed within Th_j will remain unaffected.

I only present and discuss this issue briefly in this dissertation. A future research direction would be to provide a new language construct that unifies the concepts of transaction and thread to take into account the scenarios described. The main objective is to ease the exploration of parallelization and the correct synchronization of the code, which are concerns that are related to parallel nesting as well.

Still, I partially tackled some of the issues discussed, by providing a specific approach that requires the programmer to consciously create threads aware of possible transactions that may be executing. I present this API in Appendix A.

Appendix A

Using parallelism within transactions

Throughout this dissertation, we have seen how the inner workings of a TM may be adapted to support and take advantage of parallelism within transactions. Yet, it is a mechanism that has to be exposed for it to be useful in any way. It may be used directly by the programmer or by an automatic/speculative parallelization tool. In either way, it is important to address its interface and usage.

In the next section I describe the interface that I provide in the JVSTM to allow parallelism within transactions, starting with the next section. Then, in Section A.2, I explain how I implemented the inner workings of the exposed API by means of Java bytecode rewriting.

A.1 Interfacing with threads and transactions in Java

In this section I describe how the programmer may use parallel nesting in the JVSTM. The API hides the details of the underlying algorithm, which may be any of the three alternatives presented in this dissertation.

This API represents the intent of a programmer who wants to parallelize part of its program that uses TM. This means that the parallelization may occur within the control flow of transactions. Yet, nothing forbids its use outside transactions, thus allowing the programmer to easily compose its program: A certain method that the programmer parallelizes may be reachable from both transactional and non-transactional contexts. Next, I revisit the examples from Chapter 2 to drive the description of the API.

In Listing A.1 I use some abstractions provided by the JVSTM to ease readability. Namely, the `@Atomic` annotation is processed at compile time to produce code to start, handle and finish a transaction in the annotated method. Moreover, the `VList` is a transactional class that implements the `java.util.List` interface. For that, it is using `VBoxes` in its internals.

Let us now look into the parallelization of the method `enrollMultipleCourses`. The idea is to enroll the student in each course concurrently. A given point of parallelization entails the following stages: (1) identification of the parallel tasks; (2) transactional execution of each of those tasks; (3) retrieval of the results and continuation of the execution flow. To represent one point of parallelization in the program, I created the `ParallelSpawn` interface shown in Listing A.2.

```java
class Course {

  final int capacity;
  final VList<Student> enrolledStudents;


  Course(int capacity) {
    this.capacity = capacity;
    this.enrolledStudents = new VList<Student>();
  }


  @Atomic
  boolean enrollStudent(Student std) {
    if (enrolledStudents.size() < capacity) {
      enrolledStudents.add(std);
      return true;
    } else {
      return false;
    }
  }


  @Atomic
  void enrollMultipleCourses(Student std, List courses) {
    for (Course course : courses) {
      if (!course.enrollStudent(std)) {
        TM.abort();
      }
    }
  }

}
```

Listing A.1: Class representing a college course in which students may enroll.

```java
public interface ParallelSpawn<T> {

  public static final Object RETURN_FROM_METHOD = new Object();

  public void exec();

}
```

Listing A.2: Interface to be implemented by the programmer representing points of parallelism.

The idea is that each point of parallelization is reified in an implementation of that interface. Listing A.3 shows the parallelization of the method `enrollMultipleCourses` as explained before. In particular, an inner class was created that implements the `ParallelSpawn` interface. This way, the point of the parallelization now entails instantiating this reification and executing it, thus abstracting the handling of the parallelization and transactional context inside it.

Next, I describe what the programmer is required to implement in these points of parallel task spawning. Note that each of these points is related to the three stages identified above:

- Parallel tasks: The code that corresponds to each parallel task must be identified by the programmer with the `@ParallelAtomic` annotation. The semantics of this annotation are twofold: It means that the code within the method will execute in parallel as well as within a transaction. The latter is agnostic to whether the control flow is already within a transaction or not. This means that the transaction created, for each concurrent task, may be either top-level or nested. In the example provided, the only parallel task presented is the one in method `enroll`, which delegates its logic to the code within the `Course` class.
- Method `exec`: This requirement is imposed by the interface being implemented. This method repre-

```
class Course {

  (...)


  @Atomic
  void enrollMultipleCourses(Student std, List courses) {
    if (new ParallelEnrollment(std, courses).exec() == false) {
      TM.abort();
    }
  }


  class ParallelEnrollment implements ParallelSpawn<Boolean> {

    Student std;
    List courses;


    ParallelEnrollment(Student std, List courses) {
      this.std = std;
      this.courses = courses;
    }


    @Override
    Boolean exec(){
      for (Course course : this.courses) {
        enroll(this.std, course);
      }
      return ParallelSpawn.RETURN_FROM_METHOD;
    }


    @ParallelAtomic
    Boolean enroll(Student std, Course course) {
      return course.enrollStudent(std);
    }


    @Combiner
    Boolean combine(List<Boolean> results) {
      return !results.contains(false);
    }

  }

}
```

Listing A.3: Parallelization of the method `enrollMultipleCourses` from Listing A.1.

sents the spawn of the parallel tasks. Each invocation to a method annotated with `@ParallelAtomic`, within the class, corresponds to a parallel task being launched. For this to be possible, I rewrite this method at compile time using the ASM bytecode library [10]. In particular, the `exec` method in Listing A.3 shall be rewritten in such a way that the invocation of `enroll` is changed. I explain the details of how that is achieved in the next section.

- Combining the results: The `exec` method calls one or more methods annotated with `@ParallelAtomic`, which produce a set of results. Consequently, the execution of a `ParallelSpawn` has to combine those results into some result that represents the execution of the point of parallelization. The type of this final result is defined by the parametrization of the `ParallelSpawn`, which in Listing A.3 is a `Boolean`. The programmer is responsible for creating a method, annotated with `@Combiner`, that produces that `Boolean`. The arguments of the combiner method are defined by the return types of the parallel tasks that the programmer calls in `exec`. My tool limits the combiner method to receive only one argument, which means that all `@ParallelAtomic` methods called in the `exec` of a `ParallelSpawn` must have the same return type. This sufficed for all the

benchmarks tested in this dissertation. In particular, for the example used above, the set of results of the parallel tasks is an arbitrary number of `Boolean` instances because the parallel method identified returns a `Boolean`. Consequently, the programmer has to create a method that receives a `List<Boolean>` to combine.

Summarizing, to allow changing from a sequential to a parallel execution, a `ParallelSpawn` implementation comprises a decoupling between the identification of tasks, its invocation, and the collection of a single result that represents that computation.

So far this applies to the case in which a new transaction is always created in threads spawned in transactional contexts. Yet, I also described the scenario in which a new transaction may not be required, and it is enough to execute the new threads in the context of the transaction of their parent (Section 8.2). Once again, it is a decision that the programmer has to take consciously. Therefore, it suffices to have a different interface similar to what was described so far. In this case, the `ParallelUnsafeSpawn` is interpreted by the bytecode rewriter as a point of parallelization in which a transaction is only created within each callable if the parent did not have any transaction.

A.2 Providing support for the API

The set of requirements described to use parallel nesting is far from elegant and simple. The main reason behind it is that I refrained from changing the language constructs: Everything is compatible with a common Java compiler and runtime. Next, I shall briefly explain the changes that I perform automatically in the code. To simplify the description, I shall provide examples with the equivalent code in Java despite the fact that all the modifications are performed in Java bytecode.

To allow executing each of the invocations concurrently, I encapsulate those calls in implementations of `Callables`, as shown in Listing A.4 for the method `enroll`. This way I turn those invocations into first class citizens that may be manipulated in the program. In particular, it allows collecting all the calls to methods annotated with `ParallelAtomic` and only then submitting them for parallel execution in a thread pool.

Therefore each method annotated with `ParallelAtomic` will have a corresponding `Callable` automatically generated. This `Callable` receives an instance of the class that encapsulates the annotated method to allow calling the method. Additionally, its constructor also captures a possible transaction that is executing at the time. Recall that the constructor is executed in the parent thread of this parallelization point. Lastly, the call to the annotated method is wrapped with code to handle the beginning, commit and abort of a transaction. I omitted it in Listing A.4 for simplicity. The fact that a possible parent transaction is made known in this thread leads to the aforementioned composability of transactions when the new transaction is created.

After this transformation, we just need to modify the `exec` method. Listing A.5 demonstrates the equivalent Java code of those modifications. I create a new list to hold the parallel tasks in the beginning of the `exec` method. I also append code to hand-off that list of tasks to the JVSTM, which I extended to handle the execution of parallel (and potentially nested) transactions in a thread pool. Those tasks, reified by `Callables`, are submitted to an `ExecutorService`. Finally, the result of the parallel execution is passed to the combiner method.

```java
class ParallelEnrollment$enroll$1 implements Callable<Boolean> {

  ParallelEnrollment arg0;
  Student arg1;
  Course arg2;
  Transaction parent;


  ParallelEnrollment$enroll$1(ParallelEnrollment arg0,
          Student arg1, Course arg2) {
    this.arg0 = arg0;
    this.arg1 = arg1;
    this.arg2 = arg2;
    this.parent = Transaction.current();
  }


  Boolean call() {
    Transaction.setCurrent(this.parent);
    // Code for handling transactions omitted for simplicity
    (...)
    Object result = execute();
    (...)
    Transaction.setCurrent(null);
    return result;
  }


  Boolean execute() {
    return this.arg0.enroll(arg1, arg2);
  }

}
```

Listing A.4: `Callable` generated for parallel execution of a method identified with `@ParallelAtomic`.

To populate the list of tasks, I have to replace calls to methods annotated with `@ParallelAtomic`. In its place, I need to create the corresponding generated `Callable` and to add it to the list of tasks. It may be difficult to do so, given the intrinsic difficulties of manipulating arbitrary code that may be involved in computing and pushing the arguments of the method call to the stack. Therefore, I chose to create a static method responsible for instantiating each generated `Callable`. This allows me to replace the call to the parallel method with the corresponding static method, followed by its addition to the list of tasks.

```java
class ParallelEnrollment implements ParallelSpawn<Boolean> {

  Student std;
  List courses;


  ParallelEnrollment(Student std, List courses) {
    this.std = std;
    this.courses = courses;
  }


  @Override
  Boolean exec() {
    List<Callable<Boolean>> tasks = new ArrayList<Callable<Boolean>>();
    for (Course course : this.courses) {
      ParallelEnrollment$enroll$1 var = create$enroll$1(this, std, course);
      tasks.add(var);
    }
    return combine(JVSTM.manageParallelExecution(tasks));
  }


  static ParallelEnrollment$enroll$1 create$enroll$1(ParallelEnrollment arg0,
              Student arg1, Course arg2) {
    return new ParallelEnrollment$enroll$1(arg0, arg1, arg2);
  }


  @ParallelAtomic
  Boolean enroll(Student std, Course course) {
    course.enrollStudent(std);
  }


  @Combiner
  Boolean combine(List<Boolean> results) {
    return !results.contains(false);
  }

}
```

Listing A.5: Class representing a parallelization of Listing A.3 after being automatically rewritten by the bytecode processor.

Bibliography

[1] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 163–174, Salt Lake City, USA, 2008. ACM.

[2] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 110–112, Munich, Germany, 2008. ACM.

[3] I. Anjo and J. Cachopo. Jaspex: Speculative parallel execution of java applications. In *1st INFORUM*, Faculdade de Ciências da Universidade de Lisboa, 2009.

[4] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 196–207, Agia Napa, Cyprus, 2008. Springer-Verlag.

[5] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 4–18, Paphos, Cyprus, 2009. Springer-Verlag.

[6] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN'11, pages 83–94, Bangalore, India, 2011. Springer-Verlag.

[7] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 253–262, Thira, Santorini, Greece, 2010. ACM.

[8] D. B. Baptista. Task scheduling in speculative parallelization, November 2011.

[9] J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 91–100, Bangalore, India, 2010. ACM.

[10] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, AECS '02, Grenoble, France, 2002.

[11] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, Dec. 2006. Elsevier, North-Holland, Inc.

[12] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD International conference on Management of Data*, SIGMOD '93, pages 12–21, Washington D.C., United States, 1993. ACM.

[13] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pages 1–13, Ottawa, Canada, 2006. ACM.

[14] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, Bangalore, India, 2010. ACM.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Stockholm, Sweden, 2006. Springer-Verlag.

[16] N. Diegues and J. Cachopo. Review of nesting in transactional memory. Technical Report RT/1/2012, Instituto Superior Técnico/INESC-ID, January 2012.

[17] N. Diegues and J. Cachopo. On the design space of parallel nesting. In the *4th Workshop on Theory of Transactional Memory, WTTM '12*, Madeira, Portugal, 2012.

[18] N. Diegues, S. Fernandes, and J. Cachopo. Parallel nesting in a lock-free multi-version software transactional memory. In the *7th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT '12*, New Orleans, USA, 2012.

[19] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, Toronto, Canada, 2008. ACM.

[20] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, Dublin, Ireland, 2009. ACM.

[21] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, Calgary, Canada, 2009. ACM.

[22] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, Salt Lake City, USA, 2008. ACM.

[23] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd international conference on Distributed Computing*, DISC'09, pages 93–107, Elche, Spain, 2009. Springer-Verlag.

[24] S. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 179–188, San Antonio, USA, 2011. ACM.

[25] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[26] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 305–319, Arcachon, France, 2008. Springer-Verlag.

[27] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 304–313, Munich, Germany, 2008. ACM.

[28] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, Salt Lake City, USA, 2008. ACM.

[29] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, POPL '09, pages 404–415, Savannah, USA, 2009. ACM.

[30] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, Lisbon, Portugal, 2007. ACM.

[31] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, Washington, USA, 2003. IEEE Computer Society.

[32] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th annual symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, Boston, Massachusetts, 2003. ACM.

[33] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, San Diego, USA, 1993. ACM.

[34] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Mar. 2008.

[35] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. ACM.

[36] D. Imbs and M. Raynal. A lock-based STM protocol that satisfies opacity and progressiveness. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 226–245, Luxor, Egypt, 2008. Springer-Verlag.

[37] R. Kumar and K. Vidyasankar. HParSTM: A hierarchy-based STM protocol for supporting nested parallelism. In the *6th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT, 2011.

[38] P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *Proceedings of the 15th International Conference On Principles Of Distributed System*, OPODIS '11, pages 112–117, Toulouse, France, 2011. Springer-Verlag.

[39] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, Long Beach, USA, 2005. ACM.

[40] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC '05, Cracow, Poland.

[41] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[42] R. Martin. Solid design principles and design patterns. In http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, 2000.

[43] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization*, IISWC '08, pages 35–46, Seattle, USA, 2008.

[44] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 359–370, San Jose, USA, 2006. ACM.

[45] J. E. B. Moss. Open nested transactions: Semantics and support. In poster presented at *Workshop on Memory Performance Issues*, WMPI '06, 2006.

[46] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, Dec. 2006. Elsevier, North-Holland, Inc.

[47] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, Sept. 2005. ACM.

[48] A. Oram and G. Wilson. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly, 2007.

[49] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 43–52, San Jose, USA, 2011. ACM.

[50] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979. ACM.

[51] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 16–25, Zurich, Switzerland, 2010. ACM.

[52] H. Ramadan and E. Witchel. The Xfork in the road to coordinated sibling transactions. In the *4th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT*, 2009.

[53] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 284–298, Stockholm, Sweden, 2006. Springer-Verlag.

[54] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, USA, 2006. ACM.

[55] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, Ottowa, Canada, 1995. ACM.

[56] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 179–193, Stockholm, Sweden, 2006. Springer-Verlag.

[57] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 275–284, Munich, Germany, 2008. ACM.

[58] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 30(3). 2005.

[59] M. M. Waliullah and P. Stenström. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–11, Miami, USA, 2008. IEEE.

[60] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, Mar. 1991. ACM.

[61] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, Munich, Germany, 2008. ACM.