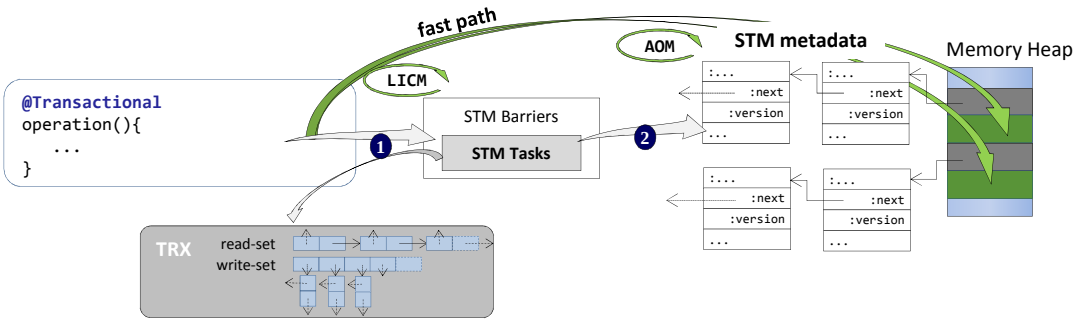
UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO



Optimizing Memory Transactions for Large-Scale Programs

Fernando Miguel Santos Gamboa Lopes de Carvalho

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in

Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury:

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor João Manuel dos Santos Lourenço

Doctor Luís Manuel Antunes Veiga

Doctor João Manuel Pinheiro Cachopo

Doctor Aleksandar Dragojevic

2014

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Optimizing Memory Transactions for Large-Scale Programs

Fernando Miguel Santos Gamboa Lopes de Carvalho

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury:

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor João Manuel dos Santos Lourenço
Professor Auxiliar
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Doctor Luís Manuel Antunes Veiga
Professor Auxiliar
Instituto Superior Técnico, Universidade de Lisboa

Doctor João Manuel Pinheiro Cachopo
Professor Auxiliar
Instituto Superior Técnico, Universidade de Lisboa

Doctor Aleksandar Dragojevic
Post Doc Researcher
Microsoft Research, United Kingdom

Funding Institutions:

Fundação para a Ciência e Tecnologia
INESC-ID
Instituto Superior de Engenharia de Lisboa
Instituto Politécnico de Lisboa

2014

To the MMMMM project

Resumo

O trabalho de investigação que descrevo nesta dissertação insere-se no âmbito do problema de sincronização de acessos a memória partilhada em programas de larga-escala. São bem conhecidas as dificuldades de desenvolvimento de sincronização baseada em locks de grão fino e por isso, muitos investigadores têm argumentado a necessidade de encontrar novas abordagens alternativas ao uso de locks. De forma sucinta, o principal objectivo do meu trabalho é fornecer uma alternativa eficiente às abordagens baseadas em locks. A minha solução utiliza memória transaccional por software (STM) e foi implementada numa framework para Java bem conhecida—Deuce STM.

Para tal, eu proponho uma nova abordagem que reduz significativamente os custos associados a uma STM em programas de larga-escala, para os quais só uma pequena parte da memória está sob contenção. A minha solução combina duas técnicas de optimização inovadoras de uma forma sinergética, que conseguiu pela primeira vez, obter um desempenho com uma STM igual ao desempenho obtido com as melhores abordagens de sincronização baseadas em locks, nalguns dos benchmarks mais difíceis de superar. A minha abordagem e os resultados apresentados mostram que uma STM pode ser a primeira alternativa eficiente ao uso de locks na sincronização de acessos a memória partilhada em programas de larga escala.

Palavras-chave: Memória Transaccional por Software, Sincronização de Memória Partilhada, Optimizações em Tempo de Execução, Programas de Larga-Escala, Programação Paralela, Programação Concorrente, Algoritmos Concorrentes não Bloqueantes, Instrumentação de Bytecodes Java, Desempenho, Benchmark

Abstract

The research work that I describe in this dissertation is concerned with the problem of *shared-memory synchronization* in large-scale programs. The difficulties of developing fine-grained lock-based synchronization are well-known and many researchers have argued for the need of alternative approaches. Simply put, the main goal of my work is to provide an efficient alternative to such approaches. My proposal is based on Software Transactional Memory (STM) and I implemented it in a well-known STM framework for Java—Deuce STM.

To that end I propose a new approach that significantly lowers the overhead caused by an STM in large-scale programs for which only a small fraction of the memory is under contention. My solution combines two novel optimization techniques in a synergistic way, allowing us to get, for the first time, performance with an STM that rivals the performance of the best lock-based approaches in some of the more challenging benchmarks. My approach and experimental results show that STMs may be the first efficient alternative to locks for shared-memory synchronization in real-world–sized applications.

Keywords: Software Transactional Memory, Shared-memory Synchronization, Runtime Optimizations, Large-scale Programs, Parallel Programming, Concurrent Programming, Non-blocking Concurrent Algorithms, Java Bytecode Instrumentation, Performance, Benchmark

Acknowledgments

"*Programming is understanding*" is a famous quotation from Kristen Nygaard. This methodology helped me to understand much about computer science and software engineering. Yet, it is much harder to follow the same approach for a PhD. Nevertheless, my first acknowledgment is for all my colleagues that taught me the art of programming, including the PROMPT team (from CCISEL) and also my adviser João Cachopo—the best programmer that I ever known.

I was happy to find an adviser like João Cachopo that is able to take a discussion from a high level of abstraction—as you expect from an adviser—to the low level details of the computer. He went much beyond the call of duty in his advising work. He was my mentor, my colleague, my reviewer and my confessor. I deeply thank you for all the support.

I would also like to thank my colleagues of ESW for their critical opinions.

I would like to thank FCT for the PROTECT grant that allowed to reduce my workload labor by 50% for 3 years and gives me time enough to embrance other professional challenges, such as launching the first post-graduation in Portugal about programming and technology—the PROMPT. Unfortunately, not many people agree with Kristen Nygaard's opinion.

Finally and the most important, Mafalda. This is the 3rd time that I write your name in a dissertation and that expresses all that you mean in my life. Again, I will finish my acknowledgments with the same quotation that you said to me in 2002, when I decided to quit my job, after 6 years of working on the information systems industry and embrace an academic career: "You want to exchange so much, for so few!".

Yet, this time I agree with you.

During this long journey we decided to have 3 kids and, today, my older daughter has almost the same age of the work that I developed for this PhD. Now, that I am close to finish this long task, I hope I can fulfil my father commitments and teach you to ride your bikes without training wheels. You three (Madalena, Margarida and Miguel) are the most successful project that we both (Mafalda and Miguel) ever done.

The MMMMM force.

Contents

Resumo iii

Abstract v

Acknowledgments vii

1 Introduction 1
- 1.1 Thesis Scope . . . 1
- 1.2 Introduction to STMs . . . 3
- 1.3 Why STMs do not perform better? . . . 5
- 1.4 Thesis Statement . . . 6
- 1.5 Main Contributions . . . 7
- 1.6 Outline of the Dissertation . . . 10

2 Motivation, Problem Statement, and Approach 13
- 2.1 Basic Terminology . . . 13
- 2.2 What is the overhead of a transparent API? . . . 14
- 2.3 How much overhead can we eliminate? . . . 17
- 2.4 Problem Statement . . . 20
- 2.5 General Approach . . . 21
- 2.6 Summary . . . 24

3 Background & State of the Art 25
- 3.1 STM design alternatives . . . 25
  - 3.1.1 Buffering mechanism . . . 26
  - 3.1.2 Ownership acquisition . . . 26
  - 3.1.3 Concurrency control and validation time . . . 26
  - 3.1.4 Transactional memory accesses . . . 28
  - 3.1.5 API decomposition level . . . 28
- 3.2 Runtime Overheads . . . 29
- 3.3 Compiler Over-instrumentation . . . 32

3.4 Benchmarks for STMs . . . 35
3.5 Debug and Profiling Tools for STMs . . . 38
3.6 Summary . . . 39

4 Annotations to Avoid Over-instrumentation 43
4.1 Deuce STM Optimizations . . . 44
4.2 Over-instrumented Tasks . . . 45
4.3 New Java Annotations for the Deuce API . . . 47
4.3.1 `@NoSyncField` annotation . . . 47
4.3.2 `@NoSyncArray` annotation . . . 49
4.4 Performance Evaluation . . . 50
4.5 Summary . . . 53

5 Lightweight Identification of Captured Memory 55
5.1 Deuce STM Overview . . . 57
5.2 Runtime Capture Analysis . . . 59
5.3 Lightweight Identification of Captured Memory . . . 61
5.4 Extending Deuce STM . . . 65
5.4.1 Filtering . . . 67
5.4.2 Storing metadata in-place . . . 70
5.5 Validation . . . 71
5.5.1 Performance evaluation . . . 72
5.5.2 Memory Consumption Evaluation . . . 76
5.6 Summary . . . 79

6 Adaptive Object Metadata 81
6.1 JVSTM Overview . . . 82
6.2 The Adaptive Object Metadata approach . . . 85
6.2.1 Reverting Objects . . . 87
6.2.2 Extending Objects . . . 88
6.2.3 Reading Objects . . . 91
6.2.4 Correctness of the AOM Operations . . . 92
6.3 Implementation Approaches . . . 96
6.3.1 Basics of the Java Object Model . . . 97
6.3.2 Integrating JVSTM with AOM in Jikes RVM . . . 98
6.3.3 Extending Deuce STM . . . 102
6.4 Validation . . . 105
6.4.1 Performance Evaluation . . . 105
6.4.2 Memory Consumption Evaluation . . . 109

6.5 Summary 109

7 Combining LICM and AOM **113**
7.1 Enhancing the JVSTM with both LICM and AOM 114
7.1.1 Vacation 114
7.1.2 STMBench7 115
7.1.3 JWormBench 116
7.2 Comparing *jvstm-aom-licm* with other approaches 117
7.3 Summary 120

8 Conclusions **123**
8.1 Main Contributions 124
8.2 Future Research 127
8.3 Concluding Remarks and Future Directions 130

A JWormBench **133**
A.1 The WormBench benchmark 134
A.2 JWormBench: A port of WormBench to Java 137
A.2.1 JWormBench applications 137
A.2.2 STM integration 139
A.2.3 Correctness test 141
A.2.4 Contention level 141

B Extending Jikes RVM's just-in-time compiler **143**
B.1 Architecture of Jikes RVM's just-in-time compiler 143
B.2 Wrapping a method call in a transaction control flow 144
B.3 Changing the `getfield` and `putfield` default behaviors 147

C Extending Deuce STM **151**

List of Figures

2.1 The STMBench7 throughput in three different workloads without long traversal operations, for three different STM algorithms: LSA, TL2 and JVSTM, and for two lock-based approaches. 15

2.2 The STMBench7 throughput in three different workloads without long traversal operations, for JVSTM and two lock-based approaches: a coarse- and a medium-grained lock. 17

2.3 Overheads incurred by an STM barrier when accesses a transactional location. 22

2.4 A fast path to access objects that are not under contention. 23

4.1 Speedup of each synchronization mechanism over sequential, non-instrumented code. 53

5.1 Three different transactions accessing a shared object `Counter`. 62

5.2 Performance improvement from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) in the Vacation benchmark, when using the TL2 STM. 64

5.3 The throughput for two workloads (low-contention and high-contention) of the Vacation benchmark, when using the TL2 STM. 64

5.4 Speedup from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) in three workloads (read-dominated, read-write and write-dominated) of the STMBench7 benchmark, when using the TL2 STM. 65

5.5 The throughput for three workloads (read-dominated, read-write and write-dominated) of the STMBench7 benchmark, when using the TL2 STM. 66

5.6 Speedup from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) for two workloads. 67

5.7 The throughput for two workloads (*N-reads-1-write* with smaller write-sets and *N-reads-N-writes* with larger write-sets) of the JWormBench benchmark, when using the TL2 STM. 67
5.8 The STMBench7 throughput for LSA and JVSTM, in the three available workloads, without long traversal operations. 75
5.9 The JWormBench throughput for LSA, JVSTM, and locks, for two different wokloads. 76
5.10 The STMBench7 memory consumption for TL2, with and without LICM, in the read dominated workload, without long traversal operations. 78
5.11 The Vacation memory consumption for TL2, with and without LICM, in the low contention workload. 78

6.1 Structure that represents a transactional object, instance of a class `Point`, using one *vbox* for each of its fields. 83
6.2 An instance of the class `Point` in the AOM's *extended layout*. 86
6.3 `Point` object in the *compact layout*. 86
6.4 Reverting an object that is in the extended layout and that stores the values 11 and 13 as the most recent, and only, committed values. 88
6.5 An example of a transaction that commits the values 22 and 26 to the fields `x` and `y`, respectively, of a `Point`, which was in the compact layout and was storing the values 11 and 13 before. 91
6.6 Object's compact layout. 98
6.7 Processor register 99
6.8 The results for Vacation with TL2 and JVSTM, in the two workloads (low and high contention). 106
6.9 The results for STMBench7 with JVSTM, in the three available workloads, without long traversal operations. 108
6.10 The JWormBench throughput for JVSTM and locks, for two different workloads. 108
6.11 The STMBench7 memory consumption for JVSTM with, and without AOM, in three different workloads, without long traversal operations. 110
6.12 The Vacation memory consumption for JVSTM with, and without AOM, in the low and high contention workloads. 111

7.1 The results for Vacation with JVSTM, in the two workloads (low and high contention). 115

7.2 The results for STMBench7 with JVSTM, in the three available workloads, without long traversal operations. 116
7.3 The JWormBench throughput for JVSTM for two different workloads. 117
7.4 The results for Vacation with TL2 and JVSTM, in two workloads (low and high contention). 118
7.5 The results for STMBench7 with LSA, JVSTM, and locks, in the three available workloads, without long traversal operations. 119
7.6 The JWormBench throughput for LSA, JVSTM, and locks, for two different workloads. 120

A.1 Example of Worms layout in JWormBenchGui application. 135

B.1 Architecture of the Jikes RVM's just-in-time compiler 144
B.2 Stack frame before, during and after arguments duplication. 148

List of Tables

2.1 Barriers suppressed for each STMBench7 operation and the corresponding speedup on the operation when we exclude the accessed classes from being instrumented. 19

4.1 Distribution of the operation execution time accessing each of the five kinds of memory locations, with Deuce configured to use the TL2. . . 46
4.2 The execution time in milliseconds of each *worm operation* with and without useless STM barriers and the corresponding speedup when we suppress those barriers. 51
4.3 Ratio between *worm operations* for each of the three workloads' configurations. 52

5.1 Configuration parameters used for each STAMP benchmark. 72
5.2 The speedup of each STM with LICM support for 1 thread and N threads in the STMAP applications. 73

A.1 13 *worm operations* provided in the WormBench implementation. . . . 136

Listings

5.1 Resulting class `Counter` after the instrumentation by the Deuce engine. 58
5.2 The `ContextFilterCapturedState` class is a context decorator that adds a transaction fingerprint to any existing STM `Context` implementation. 68
5.3 The LICM algorithm performed by the `isCaptured` function. 69
5.4 Default implementation of an STM barrier for the `int` primitive type in regular objects and arrays. 69
5.5 The code skeleton of two read barriers for both objects and arrays, using runtime capture analysis. 70
5.6 `CapturedState` class adds an extra field owner to all transactional classes. 71
6.1 Algorithm used by the JVSTM to write-back to a vbox. The `commit` method receives the new value and the transaction number corresponding to the version of the new value. 84
6.2 Algorithm to clean the `VBoxBody` objects committed by that record's transaction. 85
6.3 Algorithm to revert an object as part of the GC's clean task. I show in bold the line that was added to the `clean` method. 88
6.4 New write-back algorithm of the `VBox` class including the extension. I show in bold the code that was added to the original code of the JVSTM. 90
6.5 Wrapping `m` call in a transaction control flow 100
6.6 Behavior of `getfield` operation for atomic objects expressed in IA32. 102
A.1 Guice *module* for JVSTM configuration 138
A.2 Implementation of *step* factory for Deuce STM. 140
B.1 `genCode()` method from `TemplateCompilerFramework` 145
B.2 `genCode()`: emitting code to wrap a method call into a transaction. . . 146
B.3 `emitSTM_begin()` 147
B.4 Emit code for duplicating method's arguments. 148
B.5 Machine code for duplicating method's arguments. 148
B.6 Machine code for trying to commit a transaction. 148

B.7 `genCode()` method from `TemplateCompilerFramework` 149
C.1 Skeleton of the class `Agent`, with two different entry points: `main` and `premain`, depending on whether the instrumentation is performed in offline mode or by a Java agent. 152
C.2 `ClassEnhancer` interface. 153
C.3 Required enhancers to run Deuce with the JVSTM. 153

Chapter 1

Introduction

The research work that I describe in this dissertation is concerned with the problem of *shared-memory synchronization* [Mellor-Crummey & Scott, 1991] in large-scale programs. The difficulties of developing fine-grained lock-based synchronization are well-known and many researchers have argued for the need of alternative approaches [Sutter & Larus, 2005; Dice & Shavit, 2006; Adl-Tabatabai *et al.* , 2006; Herlihy & Shavit, 2008]. Simply put, the main goal of my work is to provide an efficient alternative to such approaches. My proposal is based on Software Transactional Memory (STM) [Shavit & Touitou, 1995] and I implemented it in a well-known STM framework for Java—Deuce STM [Korland *et al.* , 2010]. Although I developed my solution for the JVM (Java Virtual Machine), the same approach can be applied to any other managed runtime environment [Smith & Nair, 2005] based on the same principles of the JVM type system [Lindholm & Yellin, 1999].

In this introductory chapter, I start by giving a general overview of my work. Then, I introduce the main features of an STM implementation. After that, in Section 1.3, I explain some problems that result from the implementation of those features. These two elements are essential to present my thesis statement, which I shall do in Section 1.4. Then, in Section 1.5, I present the main contributions of my work. Finally, I present the outline of the dissertation.

1.1 Thesis Scope

This dissertation addresses the problem of *shared-memory synchronization* in large-scale programs. By large-scale programs I mean enterprise applications that support the

needs of the organizations and it includes the definition of Martin Fowler that "*Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data*" [Fowler, 2003]. In most cases these applications run on a managed runtime environment [Smith & Nair, 2005] and are maintained by teams of software engineers that need to answer quickly to the requirements of the customers. Thus, programmers require tools and abstractions that hide the low-level details of the software development and let them focus on the domain problem. Shared-memory synchronization is one of those low-level details that arises and today most managed runtime environments do not provide any alternative to the existing lock-based abstractions.

In this context, Software Transactional Memory (STM) [Shavit & Touitou, 1995] has been the subject of intense research as one of the most promising approaches to simplify the development of programs with shared-memory synchronization. Yet, current STM implementations fail to demonstrate applicability to real-world problems: In most cases, the performance of an STM on a real-world–sized benchmark is significantly lower than the sequential version of the benchmark, let alone its parallel version using locks. This problem was pointed out by some authors (e.g., Cascaval *et al.* [2008]; McKenney *et al.* [2010]), who raised questions about the practical applicability of STMs to real-world programs. In fact, whereas STMs have shown promising results when applied to micro-benchmarks, they typically perform worse than coarse-grained lock-based approaches in larger, real-world–sized benchmarks (e.g. Herlihy *et al.* [2006]; Dice *et al.* [2006]; Riegel *et al.* [2006]; Felber *et al.* [2008]; Dragojević *et al.* [2009]).

My goal is to provide an STM-based synchronization solution for a managed runtime environment with the same level of programming easiness of a coarse-grain lock, which means that I would like to keep the API of the STMs *transparent*, but with better scalability and performance, turning it in an efficient alternative for shared data synchronization in large-scale programs.

To that end I explore a set of alternative techniques that significantly lower the overhead caused by an STM in large-scale programs for which only a small fraction of the memory is under contention. These optimization techniques avoid the STM overheads in different scenarios and thus, they can complement each other to remove the further overheads left by the use of the other technique. My approach allows me to get, for the first time, performance with an STM that rivals the performance of the best lock-based approaches in some of the more challenging benchmarks. My approach and experimental results show that STMs may be the first efficient alternative to locks for shared-memory synchronization in real-world–sized applications.

1.2 Introduction to STMs

The idea of providing hardware support for transactions was firstly introduced in 1986 by Knight [Knight, 1986] to check the correctness of parallel execution of Lisp programs. This model was inspired by database transactions for controlling access to shared memory in concurrent computing. In database systems, the basic *correctness* condition for concurrent transactions is *serializability* [Papadimitriou, 1979], which states that the result of executing concurrent transactions on a database must be identical to a result in which these transactions executed serially. *Strict serializability* [Herlihy & Wing, 1990] is a stronger correctness criteria which respects the real-time ordering, in contrast with the *serializability* property.

Later, in 1993, this concept was extended by Herlihy and Moss to the notion of transactional memory [Herlihy & Moss, 1993] and, in 1995, Shavit and Touitou proposed a software implementation of the same idea [Shavit & Touitou, 1995]. Since then, the research on STMs has been immensely active, with many researchers proposing new STM implementations such as: McRT-STM [Adl-Tabatabai *et al.* , 2006], JVSTM [Cachopo & Rito-Silva, 2006], TL2 [Dice *et al.* , 2006], Haskel-STM [Harris & Fraser, 2003], Bartok-STM [Harris *et al.* , 2006], DSTM [Herlihy *et al.* , 2006], RSTM [Marathe *et al.* , 2006], TinySTM [Felber *et al.* , 2008], SwissTM [Dragojević *et al.* , 2009], Deuce STM [Korland *et al.* , 2010], and NOrec [Dalessandro *et al.* , 2010], among others.

One of the main advantages of STM is its programmatic API based on an `atomic` keyword that allows programmers to annotate functions, or code blocks, that should be performed in the scope of a transaction. This is the approach used by Deuce [Korland *et al.* , 2010], in the case of the Java platform, which provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior. This agrees with the generally accepted idea that STMs should be *transparent*, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations.

A transaction accesses memory locations speculatively and executes completely—*commits*—or has no effect—*aborts*—as if the transaction did not execute at all. A transaction runs in *isolation*, meaning that the effects of the writes cannot be visible outside the transaction until it commits and reads just can observe committed data. To guarantee isolation and before committing, the TM runtime must verify that none of the read locations have been modified by other transactions during its execution – this corresponds to *transaction validation* for STMs that use an *invisible reader* strategy [Riegel

et al. , 2006], for which the presence of a reading transaction is not visible to concurrent transactions. In this sense, the *conflicts detection* can be implemented in a *pessimistic* approach through the acquisition of locks that protect the memory locations that are accessed by a transaction during its execution, or in an *optimistic* manner allowing access to memory locations without any synchronization and just validating at the end of the transaction whether a concurrent access occurred. In case of conflict, the *contention manager* has the responsibility of deciding whether to wait or to abort a conflicting transaction, while guaranteeing the overall application progress (e.g. Herlihy *et al.* [2006]; Marathe *et al.* [2006]).

To synchronize concurrent accesses to memory locations, some STMs must associate with each transactional memory location additional information—*location's metadata* (e.g., Marathe *et al.* [2006]; Harris *et al.* [2005]; Harris & Fraser [2003]; Herlihy *et al.* [2006]))[1]. Moreover, the *location's metadata* is also used by transactions to keep track of the locations read and written in transaction's private buffers—*read-set* and *write-set*, respectively—that are an essential part of the *transaction validation* and *commit* operations. The validation indicates whether a transaction would be able to commit—that is, whether the *read-set* represents a consistent snapshot. The *write-set* can be used when a transaction succeeds and needs to write data into updated locations, if it follows a *redo log* approach (e.g. Cachopo & Rito-Silva [2006]; Dice *et al.* [2006]; Harris & Fraser [2003]), or to undo changes to updated locations when the transaction aborts, if it follows an *undo log* approach (e.g. Adl-Tabatabai *et al.* [2006]; Harris *et al.* [2006]).

To maintain the *read-set* and *write-set* during the execution of a transaction, the STM must intermediate all accesses to memory locations from inside a transaction. So, instead of just accessing a memory location, it requires a call to a specialized STM function. *STM barriers* are operations that replace normal accesses to memory locations and include the invocation to the specialized STM function.

The main properties of an STM are established by the implementation of the previous features: *read-set* and *write-set*, *location's metadata* and *STM barriers*. The choices made in the design of these features delineate its behavior and induce the overheads incurred by the STM.

[1] However, there are other STMs that do not associate metadata with each memory location, such as NOrec [Dalessandro *et al.* , 2010], RingSTM [Spear *et al.* , 2008], or even simple single-global lock.

1.3 Why STMs do not perform better?

Instead of just accessing a memory location, an *STM barrier* usually needs to consult the *location's metadata* and maintain the *read-set* and *write-set*. These additional tasks can introduce a significant overhead in the STM performance and in some cases may make it perform worse than sequential code.

If we rely on the programmer for introducing *STM barriers* for every memory access, we may get the optimal number of barriers that are necessary to guarantee program correctness. But this methodology is not practical for large programs and significantly impacts one of the STM advantages over lock-based solutions: programmability. So, alternative approaches have been proposed, which delegate to a second party (e.g. STM compiler) the task of automatically translating memory accesses into *STM barriers*—this process is often called *transactification*. This strategy has been seen as a mandatory feature in STM implementations to turn them into a widely adopted solution in concurrent programming. Yet, it is not possible to determine in an automatic form precisely which instructions access shared data. So, the compiler has to instrument the code conservatively, translating every memory access inside a transaction into a read or write barrier. Thus, the compiler-generated code may execute a significantly larger number of unnecessary STM barriers that incur in overhead and reduce the performance—this problem is known as *compiler over-instrumentation* [Yoo *et al.* , 2008].

Moreover, even when *STM barriers* are correctly applied, they can execute redundant and unoptimized tasks, such as consulting the *location's metadata* instead of directly accessing a memory location without indirections overheads. This metadata indirection penalizes accesses to transactional memory locations—resulting in *runtime overheads* [Harris *et al.* , 2006]. Besides the performance penalty, the *location's metadata* also introduces overheads in the amount of memory needed. The additional requirements in memory depend on the STM model and design of the data structures that store the *location's metadata*. But typically, the approaches that reduce the additional amount of memory required for metadata, also induce more false conflicts and unnecessary aborts of transactions, penalizing the overall STM performance. The challenge is to design solutions that require the minimum extra memory while preserving a good performance.

Another major source of overheads in STMs includes *privatization* [Spear *et al.* , 2007], where access to *shared* objects is mediated by transactions, and *private* objects are only accessible to a single thread. The privatization problem arises when objects move

from shared to private access and the STM must avoid correctness violations. Because the benchmarks that I selected to validate my proposals do not require privatization, then I did not explore optimization techniques for privatization.

1.4 Thesis Statement

This dissertation's thesis is that it is possible to substantially reduce the STM-induced overheads for a large-scale program if we assume that the amount of memory under contention—that is, memory being concurrently accessed both for read and for write—is only a small fraction of the total amount of memory accessed by that program.

Namely, that it is possible to achieve that goal by:

- complementing an STM API with additional annotations that let programmers specify which memory locations are not-shared
- enhancing an STM runtime with an optimization technique that is able to identify *non-shared* objects that cannot be identified with the previous approach
- using an adaptive approach to the amount of additional metadata used by any STM to reduce the STM overheads for *non-contended* objects

To validate this thesis, I give specific proposals in the dissertation for adding each of these elements to an STM framework. I describe in detail how these new constructs integrate and demonstrate, by comparison with the existing techniques to avoid over-instrumentation, what are the benefits of my proposals.

To show that it is feasible to eliminate some of the STM overheads and simultaneously keep its API transparent, I demonstrate a concrete implementation of my proposals in Deuce STM—an STM framework with a transparent API. By providing an implementation of my proposals within Deuce STM, I am able to test it with a variety of baseline STM algorithms.

One of the additional benefits of implementing all the techniques is that, having a practical implementation for all of them allowed me to further validate their effectiveness, by performing extensive experimental tests for a variety of benchmarks, including real-world–sized benchmarks that are known for being specially challenging for STMs.

My approach can alleviate some of the major bottlenecks that reduce the performance of STMs in many realistic applications and I report results in different benchmarks to demonstrate that they confirm the assumptions that I used in the design of the proposals of this dissertation.

1.5 Main Contributions

Despite all of the intense research on STMs, which advanced significantly not only the design and implementation of STMs but also the understanding of their shortcomings, the problems originated by the overheads of STMs and the causes behind them are still far from solved.

From my analysis, I observed in different applications (e.g. STMBench7 [Guerraoui *et al.* , 2007], Vacation [Cao Minh *et al.* , 2008] and LeeTM [Ansari *et al.* , 2008]) that the vast majority of the memory locations managed by a program are not under contention. In these cases, the corresponding memory accesses perform useless STM barriers and track further metadata that incur in additional overheads.

The key idea of my work is that for non-contended memory we can avoid the full-blown STM barriers, which should be used only for accessing (the relatively rare) memory under contention. Instead, for the frequent non-contended memory accesses we use lightweight barriers that redirect accesses straight to the target memory, thereby significantly reducing the overheads imposed by the STM. My approach combines different optimization techniques to reduce the overheads of accessing objects that are not under contention in different situations.

But to achieve these solutions, firstly I had to understand and analyse which kind of overheads are incurred by transactional locations and which of them I could suppress through either a manual or automatic mechanism. This research has been a fundamental part of my work, allowing me to design and build the optimization techniques that I propose in this dissertation. Next, I describe each of the main contributions of my work.

The effects on performance of relaxing the transparency of an STM

The first contribution of my work is in the identification of some limitations in concurrent programming environments. For instance, current programming languages do not allow programmers to express immutability for array elements.[2] Thus, instrumenting accesses to immutable array elements incurs in unnecessary over-instrumentation. Part of these scenarios has not been reported yet in previous studies. As a result of

[2] For instance, the Java `final` keyword just avoids the declared variable (array's reference) from being modified after its initialization and it does not mean anything about the characteristics of its elements.

my research in benchmarks for STMs I ported from C# to Java the WormBench benchmark [Zyulkyarov *et al.* , 2008], which I called JWormBench[3] and that helped me to identify these bottlenecks.

The obstacles that I identified in my work are essentially related to the lack of knowledge of the STM compiler about the application semantics. To mitigate those overheads I proposed that an STM should also provide extra annotations that let the programmers directly convey such application-level knowledge to the compiler. Following this approach the programmer is responsible for the proper placement of the annotations and the overall correctness of the application. This is similar to what happens in other solutions [Yoo *et al.* , 2008; Ni *et al.* , 2008; Beckman *et al.* , 2009; White & Spear, 2010] that propose specific annotations to avoid the excessive instrumentation of the STM compiler, alleviating the use of *STM barriers* for every memory access. I developed an extension to Deuce STM that extends its API with extra annotations and my results show that we get an improvement of up to 22-fold in the performance of the JWormBench when we tell the STM framework which data is not transactional. Actually, not only do we get a speedup when we use a less transparent API, my results show that the STM performs as good as a fine-grained lock-based approach, which is particularly easy to implement in JWormBench, but may not be in other applications. So, this work shows that with the appropriate placement of *STM barriers* and avoiding them in useless situations, we can achieve the desirable scalability and performance. These contributions were published in the proceedings of the *11th International Conference on Algorithms and Architectures for Parallel Processing*—ICA3PP11—Volume Part I [Carvalho & Cachopo, 2011].

Runtime elision of transactional barriers for captured memory

But, if the responsibility of placing *STM barriers* is shared between the programmer and the compiler, then an STM cannot guarantee, by itself, the correct synchronization between atomic blocks. Furthermore, the API of an STM must be transparent and the programmer should not be concerned with the transactional definition of the memory locations, but just with the identification of which operations are atomic.

So, my goal is to provide an automatic mechanism that achieves the same results of the previous solution but without relaxing the transparency of the STM API. To that end, I developed a new runtime technique for *lightweight identification of captured memory*—LICM—for managed environments that is independent of the underlying STM

[3] Available at https://github.com/inesc-id-esw/jwormbench

design. *Captured memory* [Dragojevic *et al.* , 2009] corresponds to memory allocated inside a transaction that cannot escape (i.e., is *captured* by) its allocating transaction. My approach is surprisingly simple, yet effective, being up to 5 times faster than its predecessor algorithm [Dragojevic *et al.* , 2009].

A preliminary version of this solution was published as a poster in the proceedings of the *18th ACM SIGPLAN Symposium on Principles and Practice of parallel Programming*—PPoPP13—[Carvalho & Cachopo, 2013b], whereas a further improved version was published as a "Distinguished Paper" in the *13th International Conference on Algorithms and Architectures for Parallel Processing*—ICA3PP13—[Carvalho & Cachopo, 2013a].

Adaptive Object Metadata for multi-versioning STMs

Yet, even when some useless STM barriers are correctly suppressed there are still some STM designs that establish additional metadata and, consequentially further indirections, when accessing transactional objects, including those that are not under contention. This happens, for instance, in the case of *multi-versioning* approaches [Reed, 1978], which establish that a memory location stores a history of committed values, instead of a single value. Although being an optimization technique that aims to reduce the rate of aborted transactions, it introduces additional overhead in all memory accesses including those that should not perform STM barriers. To mitigate this problem, my proposal, which I called *adaptive object metadata* (AOM) [Carvalho & Cachopo, 2012], includes a new object model that swings objects back and forth between two different object layouts: the *compact-layout*, where no memory overhead exists, and the *extended-layout*, used when the object may be under contention.

My proposal was implemented in JVSTM [Cachopo, 2007; Cachopo & Rito-Silva, 2006], a multi-versioning object-based STM and I tested it for a wide variety of benchmarks, including real-world–sized benchmarks that are known for being specially challenging for STMs. The results obtained thus far show that my assumptions are correct and under read dominated scenarios the AOM approach can improve the STM performance and simultaneously reduce the memory overhead.

I first implemented a prototype of my solution in Jikes RVM [Alpern *et al.* , 2005] which confirmed my expectations and allowed me to describe and publish a first proposal of the AOM in the informal proceedings of the *5th Workshop on Programmability Issues for Heterogeneous Multicores*—MULTIPROG'12 [Carvalho & Cachopo, 2012].

After that I developed an improved version of the AOM integrated in Deuce, which allowed me to make a complete analysis in comparison with other STMs already supported by Deuce.

Combining these two techniques, LICM and AOM, we can keep the contention-free execution path as simple as possible (and consequently fast), ideally with the same overhead of accessing non-transactional objects. Therefore, assuming that the number of transactional locations that may be under contention is only a small fraction of the total memory managed by a real-sized application, the overheads introduced by the STM become negligible. The results that I present for large-scale benchmarks, such as the STMBench7 and the Vacation, show that the JVSTM enhanced with LICM and AOM outperforms the throughput of other STMs and can even compete with a fine-grained locking synchronization approach.

My experimental results confirm my expectations that it is feasible to use STMs for real-world-sized applications, provided that the STM is not adding unnecessary barriers and metadata to memory locations that are not under contention. In fact, I believe that integrating the LICM and the AOM together in a managed runtime may further reduce the overhead of my approach and provide a significant boost in the usage of STMs.

1.6 Outline of the Dissertation

This dissertation has a total of eight chapters and it is organized as follows:

- **Chapter 1** *Introduction*. This chapter establishes what is the subject of concern of this dissertation. Specifically, it introduces the core concepts about STMs to clearly understand the following sections. Furthermore, it presents the main reasons of the overheads incurred by STMs and after that presents the thesis statement. Additionally, I also describe the main contributions of my work and how this dissertation is organized.

- **Chapter 2** *Motivation, Problem Statement, and Approach*. This chapter expands on the motivation given in the first chapter and puts this work into concrete contexts of development of programs with shared-memory synchronization. Namely, I present an experimental analysis that shows some of the sources of overhead of an STM. Finally, it identifies more precisely the problem that this dissertation addresses and describes the general approach that was followed to solve it.

- **Chapter 3** *Background & State of the Art*. In this chapter I address the main properties that characterize the implementation of an STM and directly influence its performance. After that I shall describe the state of the art on efficient support for software transactional memory.

- **Chapter 4** *Annotations to avoid over-instrumentation*. This chapter describes the work that I developed to explore the effects on performance of relaxing the transparency of an STM. To that end, it presents a proposal, and an implementation of an extension to the Deuce API that allows to avoid the useless transactification of certain classes. Finally, it presents the benefits of this approach using the JWormBench [Carvalho & Cachopo, 2011] benchmark to evaluate the performance of Deuce with, and without, programmer annotations.

- **Chapter 5** *Lightweight identification of captured memory*. This chapter presents a new runtime technique for managed environments that is independent of the underlying STM design and is able to elide useless STM barriers for transaction local objects. It describes the implementation of this technique within Deuce STM and presents experimental results for a wide variety of benchmarks.

- **Chapter 6** *Adaptive Object Metadata*. This chapter proposes an STM implementation that substantially reduces both the memory and the performance overheads associated with transactional locations that are not under contention. It describes an adaptive object metadata approach for a multi-versioning STM, and provides preliminary results that show an improvement in the performance of workloads where the number of objects written is much lower than the total number of transactional objects.

- **Chapter 7** *Combining LICM and AOM*. This chapter presents experimental results to demonstrate that the combination of two optimization techniques can achieve a better performance than any of those techniques individually. Specifically, I combine the LICM and AOM optimizations techniques to enhance the JVSTM.

- **Chapter 8** *Conclusions*. This chapter summarizes the main contributions of this dissertation and discusses some directions for future research activities.

Finally, I leave some low-level details of my work to appendixes. **Appendix A** *JWormBench*, describes the implementation of JWormBench, which extends the original benchmark in several ways, making it more useful as a testbed for evaluating STMs. **Appendix B** *Extending Jikes RVM's just-in-time compiler*, describes the details of the

modifications made to the Jikes RVM's just-in-time compiler to integrate the JVSTM and the AOM techniques. **Appendix C** describes the new infrastructure of enhancement transformations that I added to Deuce STM engine.

Chapter 2

Motivation, Problem Statement, and Approach

From my observations, the poor performance of STMs in large-scale programs is due to useless STM barriers and to further indirections imposed by additional metadata on objects that are not under contention. Solving this problem is the main motivation of my work, which I expect may contribute to a significant boost in the performance of an STM and, therefore, turn it in an efficient alternative to lock-based synchronization in large-scale programs.

In Chapter 1, I gave a brief introduction to STMs. Now, I start this chapter by establishing the basic terminology that I use to describe the motivation of my work. After that, in Section 2.2, I analyze the overheads incurred by an STM in large-scale applications and then, in Section 2.3, I present the methodology that I used to understand the reasons of those overheads. Based on the previous observations, which are the main motivation for my work, I present the problem statement in Section 2.4. Finally, in Section 2.5, I present the guidelines of the approach that I followed to achieve the solutions proposed in this dissertation.

2.1 Basic Terminology

To guarantee the consistency of concurrent accesses to shared objects, transactions must use STM barriers. An *STM barrier* is a call to the STM runtime that is typically injected before or after, or even replace, a memory access. A *transactional class* is a class whose

code was *instrumented* such that all memory accesses within its methods are guarded by STM barriers. A *transactional object* is an instance of a transactional class.

The main difference between an STM framework with a *transparent API* and one with a *non-transparent API* is that the former implicitly consider all classes as transactional by default, whereas the latter consider all classes *non-transactional*, except for those that are explicitly defined as transactional classes. In other words, this means that, unless the programmer explicitly declares classes as transactional they will be considered non-transactional.

Note that thread-local objects may need to be transactional too, because if a transaction aborts, it needs to revert the objects to their original state, even if they are not shared with other threads.

2.2 What is the overhead of a transparent API?

Some seminal proposals for STM implementations in Java, such as the DSTM2 [Herlihy *et al.* , 2006], the LSA [Riegel *et al.* , 2006] or the JVSTM [Cachopo, 2007], use a non-transparent API where the programmer is responsible for annotating the transactional classes with a specific annotation (e.g. `@Transactional`) or by defining the transactional classes as subtypes of a specific base class provided by the STM framework. This is a very efficient approach, because the STM barriers will be used only for accesses to instances of the transactional classes. All the other accesses to conventional objects (instances of non-transactional classes) from within a transaction will directly access those objects without incurring in the overheads of the STM barriers or the additional metadata. Assuming that the vast majority of the objects in a real-sized program are not shared and thus, their classes avoid a transactional definition, then this approach will incur in a overhead just for a small fraction of the total memory managed by a program, corresponding to shared objects accessed by transactions.

Yet, this approach contrasts with the generally accepted idea that STMs should be *transparent* and that the programmer should be concerned only with the definition of which operations are atomic and not worry about the definition of the memory locations accessed by those operations. Nevertheless, the STMs with non-transparent APIs show performance results that are not achieved by STMs with transparent approaches. This effect was observed by [Fernandes & Cachopo, 2011], who show that using Deuce with either the TL2 STM [Dice *et al.* , 2006] or the LSA STM [Riegel *et al.* , 2006] in the

STMBench7 [Guerraoui *et al.* , 2007] achieves a throughput up to 100 times lower than using a coarse-grained lock. Interestingly, however, they also show that by manually instrumenting the STMBench7 with the JVSTM (rather than with Deuce), it was possible to get better performance than with the medium-grained locks, which suggests that, after all, it is possible to get good performance from STMs in large-scale applications.

I ran this same experiment in a machine with 8 cores and hyperthreading, for each of the synchronization approaches: (1) using Deuce to instrument all of the code and varying the STM used between TL2 and LSA; (2) using the coarse-grained and medium-grained lock-based synchronization of the STMBench7; and (3) using an STMBench7 that was manually instrumented to use the JVSTM. Note that when using Deuce (such as TL2 and LSA), all classes are automatically instrumented, whereas when manually using the JVSTM only the classes of the effectively shared objects are instrumented.

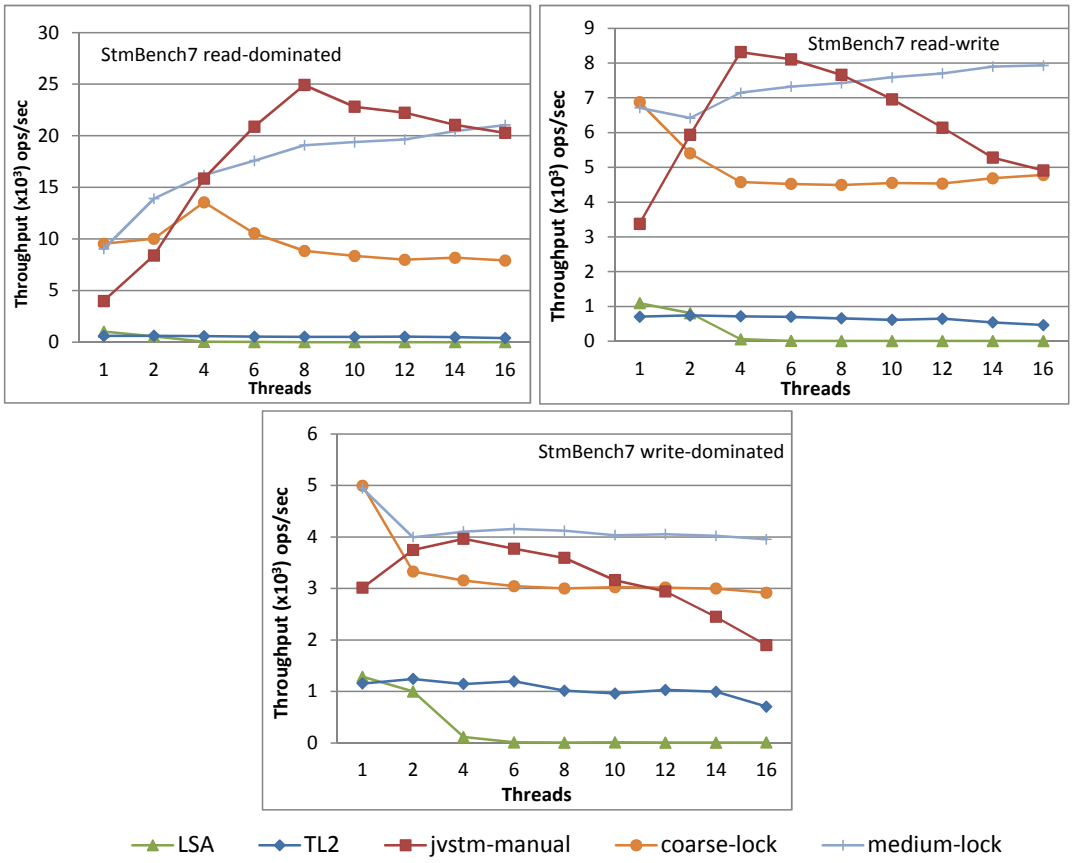


Figure 2.1: The STMBench7 throughput in three different workloads without long traversal operations, for three different STM algorithms: LSA, TL2 and JVSTM, and for two lock-based approaches: a coarse- and a medium-grained lock. In the case of the JVSTM, I manualy instrumented the benchmark, whereas LSA and TL2 were used with the automatic support of the Deuce engine.

In the results of Figure 2.1, I show the throughput of the benchmark, when the number of threads varies from 1 to 16. From these results we can confirm the large gap in performance between the JVSTM and Deuce (with either LSA or TL2), which confirms the observations presented in the work of [Fernandes & Cachopo, 2011]. This gap may be attributed to the difference in the STMs used, to the amount of barriers that are introduced into the benchmark by each approach (Deuce vs manual), or, most probably, to a combination of both.

At the time of the work reported in [Fernandes & Cachopo, 2011], Deuce did not support the integration of an STM that requires additional STM metadata to be stored in-place within the transactional object, such as the JVSTM. Later, [Dias *et al.* , 2012] proposed an API extension to Deuce to support this feature and they provide an integration in Deuce of the lock-based version of the JVSTM [Cachopo & Rito-Silva, 2006]. In my work I propose a new infrastructure of enhancement transformations for Deuce, which preservers its original API and I provide an integration of the lock-free version of the JVSTM (the same that was proposed and used in the results of [Fernandes & Cachopo, 2011]), in the Deuce framework.

Thus, in my research I complete the analysis of [Fernandes & Cachopo, 2011] and I also compare the performance between the two approaches to transactification: *Non-transparent* versus *transparent*, using the same STM algorithm—the JVSTM lock-free. I show in Figure 2.2 the results of running the benchmark manually transactified—label *jvstm-manual*—and automatically transactified by Deuce—label *jvstm*.

The results show that the JVSTM performs better than the other STMs (even when all the instrumentation is made by Deuce), but the more telling aspect is the huge gap in performance between using the JVSTM with Deuce or manually: Whereas when using the JVSTM with Deuce the throughput never gets above the sequential non-instrumented execution, in the manual case we get a speedup of 3 times, outperforming even the medium-grained lock-based approach.

Given that the JVSTM used in both cases is exactly the same, this difference must result from the over-instrumentation made by Deuce. This over-instrumentation has two consequences: (1) it adds more STM barriers to the execution of the benchmark; and (2) it adds to each object extra metadata, which needs to be traversed when accessing those objects. Both affect performance negatively.

My goal is to suppress STM barriers and avoid the metadata indirections in situations where they are not really necessary. By integrating my optimization techniques proposals in Deuce I expect to be able to achieve results similar to those obtained with

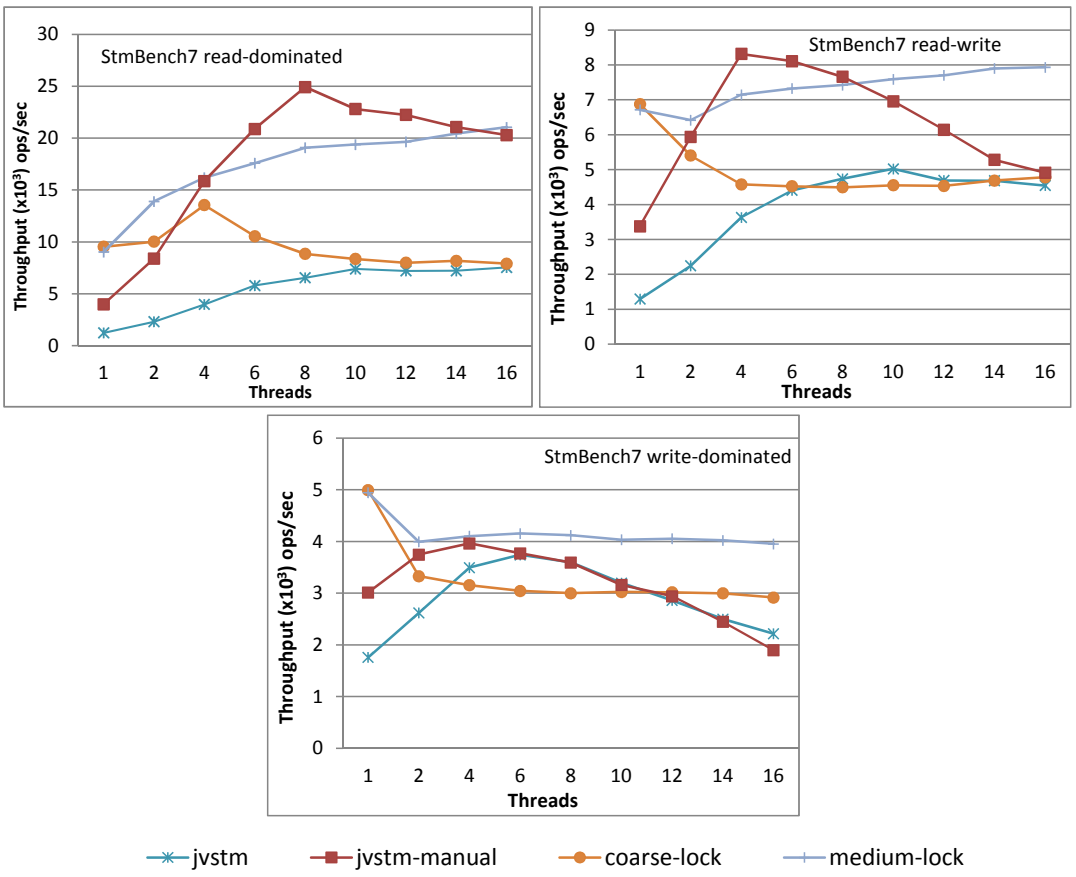


Figure 2.2: The STMBench7 throughput in three different workloads without long traversal operations, for JVSTM and two lock-based approaches: a coarse- and a medium-grained lock. In the case of the JVSTM, I follow two different transactification approaches: Manually—*jvstm-manual*—and instrumented by Deuce engine—*jvstm*.

the manual use of the JVSTM, albeit without the intervention of the programmer (thereby, making the STM easier to use). So, first I would like to understand which STM barriers can we suppress and when and how can we elide them.

2.3 How much overhead can we eliminate?

An STM framework with a transparent API, such as Deuce, cannot predict in advance which memory locations are shared and therefore, it must use a conservative approach and place an STM barrier for every memory access. One of the goals of my work is to identify some scenarios where those barriers are not required and efficiently avoid them with an innovative technique. I hope that such technique can help to improve

the performance of an STM. Namely, I would like to confirm: (1) if there are opportunities to elide unnecessary STM barriers, and (2) if we can improve the performance of a transactified application by as much as we expect, when we avoid the previously identified barriers.

So, I started my research by confirming my expectations for a well-known scenario—eliding STM barriers for *transaction local* memory accesses. Because transaction-local memory is private, accesses to it cannot cause conflicts with other transactions and thus, they do not require STM barriers. Yet, when I suppress STM barriers to access transaction-local memory, I also need to confirm if the result of the optimized synchronization still preserves the overall consistency of the application. Because the STMBench7 provides a verification test to validate if the overall consistency of the program was not broken by an erroneous synchronization, I decided to use this benchmark in my experimental tests. Furthermore, the STMBench7 also fulfils the requirement of modeling a realistic large-scale application.

To analyze the effect of removing all the STM barriers for transaction local memory in the STMBench7, I identified the classes that are instantiated inside a transaction scope, I excluded those classes from being instrumented in the cases where that was possible without compromising the correctness, and then I measured the speedup obtained.

In STMBench7 the operations traverse a complex graph of objects by using iterators over the collections that represent the connections in that graph. Typically, these iterators are transaction local and, thus, accessing them using STM barriers adds unnecessary overhead to the STMBench7's operations. To confirm this intuition, I logged the objects instantiated in the scope of a transaction and I also logged the read-set and the write-set for each operation of the STMBench7. Thus, I could identify which barriers access transaction local objects as shown in the results of Table 2.1. Then, I suppressed those barriers, excluding the whole class definition from being transformed (through the `Exclude` system property of Deuce STM) and I measured the speedup for each operation. For that purpose, I included a new execution mode in STMBench7 that allows me to run the benchmark with just one operation, instead of running a workload that combines several kinds of operations. By, running the STMBench7 with a single operation I could calculate the speedup of that operation when I suppress the identified STM barriers that access transaction local memory.

From Table 2.1, we can observe that there are transaction local objects for almost all of the STMBench7's operations (except for *op1*, *op4*, *op5*, *op9* and *op11*) and the

Operation Type	**read-only**					**read-write**			**ro**	**rw**	**read-only**								**read-write**						
Operation Id	st1	st2	st3	st4	st5	st6	st7	st8	st9	st10	op1	op2	op3	op4	op5	op6	op7	op8	op9	op10	op11	op12	op13	op14	op15
AbstractList$Itr	w	w	w		w	w	w	w	w	w						w	w	w				w	w	w	
AbstractMap$2$1												w	w							w					
HashMap			rw					rw	rw	rw															
HashMap$Entry			rw					w	w	w															
HashMap$Entry []			rw					w	w	w															
HashSet			w					w	w	w															
LargeSetImpl																									w
StringBuilder				rw																					
TreeMap$KeyIterator	w					w						w	w							w					
TreeMap$ValueIterator					w																				
""$AscendingSubMap												w	w							w					
""$EntrySetView												w	w							w					
""$EntryIterator												w	w							w					
Speedup TL2	2.5	1.3	6.1	1.1	3.4	2.4	1.4	3.5	4.4	2.9		3.1	3.1			1.7	1.9	1.9		1.5		1.6	1.5	1.5	1.2

Table 2.1: Barriers suppressed for each STMBench7 operation (r and w denote read and write barrier, respectively) and the corresponding speedup on the operation when we exclude the accessed classes from being instrumented. All classes, except `LargeSetImpl`, belong to the `java.util` package.

majority of their classes are related to the iterators of the `java.util` collections, which confirms my expectations that these iterators are transaction local. In the same table we can also observe a large speedup for each operation, of up to 6-fold in the case of the $st3$, when we avoid the STM barriers that access those transaction local objects. Note that there are operations classified as read-only because they do not change shared objects, but they still need to use write barriers because they change transaction-local objects. When this happens, an STM cannot optimize the execution of read-only transactions.

These observations confirm my two initial expectations that: (1) there are opportunities for further optimizations, and (2) we get a speedup when we perform those optimizations.

Note, however, that the experimental approach that I used to elide STM barriers is not feasible for some realistic scenarios, because I am preventing the use of STM barriers for all instances of the classes identified in table 2.1. Yet, if we need to use one of these classes in the definition of a shared object, then we cannot exclude that class from using STM barriers anymore.

So, another goal of my work is to find an efficient technique that deals with both requirements, meaning that it should avoid unnecessary STM barriers for transaction-local objects, but still preserve STM barriers for shared objects of the same class.

Finally, this optimization technique must be efficient, which means that the overhead of suppressing these barriers must be lower than the overhead of executing those barriers. Otherwise, we cannot expect to improve the performance of an STM.

2.4 Problem Statement

STM frameworks with a transparent API, such as the McRT-STM [Adl-Tabatabai *et al.* , 2006] and the Deuce STM [Korland *et al.* , 2010], provide some built-in optimizations to avoid the unnecessary use of STM barriers in a few common scenarios. For instance, they do not use STM barriers to access *immutable* memory locations, because these locations cannot be updated and thus, they will never be under contention. In Java, immutable memory locations are denoted as `final` fields.

Yet, this approach is of limited use, because the `final` fields can only be initialized by the corresponding constructor and cannot be updated by any other method. So, it cannot be used, e.g., in programs that include a setup phase responsible for initializing the overall environment out of the constructors' scope (such as in the STMBench7).

To suppress this limitation, some STM frameworks with a transparent API, such as Deuce STM, provide an alternative annotation to the `final` keyword that let programmers identify the classes that should not be instrumented by the STM engine. This approach is useful to avoid STM barriers in two different scenarios: (1) for objects that are *unmodified* by transactions, but that are target of updates outside them, and (2) for objects that are *not shared*.[1]

This is a useful approach if we assume that there are just a few cases of objects to which we would like to avoid the instrumentation. At the same time it keeps the main API transparent, which is most often used in the majority of situations. So, only when programmers want to optimize the synchronization process, will they use the annotations to identify the classes that should not be instrumented.

Yet, this approach has also its own limitations. When we exclude a class from being instrumented, we are avoiding STM barriers for all instances of that class. So, if there is only one shared object of that class that is subject of concurrent modifications, then we cannot define that class as non-transactional anymore. Yet, in this case, we are imposing STM barriers to all other *non shared* objects that do not require STM barriers.

Another example of an unsolved problem by this approach is for objects that are subject to concurrent modifications for a small period of time, but which stay unmodified after that period and for the rest of the program execution. These may correspond to shared objects but that are *frequently non-contended*. Again, in this case, we need to

[1] Note, however, that in this case and for thread-local objects you may not preserve the consistency of an object if it is modified by a transaction that aborts.

define the classes of those objects as transactional, because of the possible existence of concurrent updates for a small period of time, but we will not need to use STM barriers for the majority of memory accesses in the rest of their life cycles. Yet, we cannot avoid the definition of transactional classes for those objects.

So, a less transparent API can be a useful approach to avoid over-instrumentation of classes with *non-shared* instances or objects that are *unmodified* by transactions, but, it is not able to solve all problems of the unnecessary use of STM barriers. To that end, we require additional optimization techniques that complement the existing ones and help to eliminate the remaining overheads.

On the other hand, even when a less transparent API is able to eliminate the major sources of overhead, that approach may not be feasible in certain scenarios. For instance, if we do not have access to the classes that we would like to exclude from being instrumented. So, in these scenarios it would be desirable to have an automatic mechanism that does not require the programmer intervention.

2.5 General Approach

The main goal of the work that I describe in this dissertation is to optimize memory transactions for large-scale programs. Because the STMBench7 is a well known benchmark that models the behavior of a real-world–sized application and because the JVSTM presents the best performance in the STMBench7 in comparison to the LSA an TL2, I focused my research in the optimization of the JVSTM. Yet, some of the optimization techniques that I propose are applicable to any STM, as shown in the evaluation of those techniques.

In Section 2.3 I showed that the vast majority of the STMBench7 operations execute useless STM barriers that degrade their performance. Furthermore, I show that when I avoid those barriers I get an improvement on the performance of those operations, of up to 6-fold (in the case of operation *st3*). This speedup is due to the avoidance of the overheads associated with the STM barriers. Namely, when we avoid an STM barrier we are suppressing: (1) *additional tasks* such as maintaining the read-set, lookup for a written location or write buffering; (2) *metadata* that adds further indirections to memory accesses. The example of Figure 2.3 highlights the overheads incurred by an STM: It shows a transactified method—`transactionalOperation`—which instead of directly accessing the memory heap, must perform additional tasks, such as updating

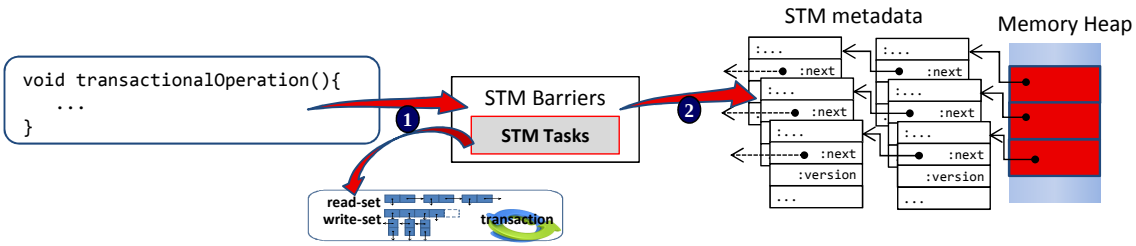


Figure 2.3: Overheads incurred by an STM barrier when accesses a transactional location: additional tasks and further metadata.

the read-set or the write-set(1), and must maintain the metadata associated to every memory location (2).

In this work, I explore some optimization techniques to promote the use of a fast path to access objects that are not under contention and, thus, avoid the overheads illustrated in Figure 2.3. To that end I must consider different approaches for different types of non-contended objects corresponding to transactional classes: (1) whose instances are not shared and therefore, the whole class can be excluded from being instrumented; (2) with both, shared and non-shared instances, and (3) whose instances are shared but are not under contention in the majority of their life cycle, which means that they are frequently non-contended.

For the first category of objects we already have an optimization mechanism provided by Deuce STM, the same that was used in Section 2.3, which allows to define a class as non-transactional and thus, exclude it from being instrumented. Yet, this technique has some limitations regarding the coarse granularity of exclusion at the class definition level. So, if we have a class with both shared and non-shared fields, we have no way to exclude just part of that class from being instrumented. Another limitation of the existing Deuce optimization technique is related with non-shared arrays, which cannot be excluded from being instrumented with the existing Deuce mechanisms.

So, one of the goals of my research is to find a better proposal of a transparent API that mitigates the previous identified limitations and simultaneously provides a finer grained control over what to instrument.

Regarding the second group of non-contended objects, it may include transaction-local objects for instance. In this case, several types of optimization techniques have already been proposed to avoid the use of STM barriers on memory accesses to transaction-local objects. The work of Afek *et al.* [2011] added to Deuce STM a static analysis technique to enable compile-time optimizations that avoid instrumentation of memory accesses to transaction local memory. Yet, static compiler analysis is often imprecise

and conservative, and thus cannot remove all unnecessary barriers, because program modules are dynamically loaded, for example, and it is impossible to perform whole program compiler analysis.

So, in my research I explore alternative techniques to static analysis, such as runtime analysis that differentiates between shared and non-shared objects in an accurate and efficient way by avoiding STM barriers when they are not needed.

Finally, the most challenging optimization is to recognize that a shared object is not under contention for a certain period of its life cycle and thus, avoid the use of STM barriers. Lets revisit the example shown in Figure 2.3 and point that if a memory location is not under contention, then we do not need to keep either additional STM tasks or further STM metadata. So, these are the main overheads that I would like to avoid for non-contended objects, and to that end, in my work I develop an adaptive system that lets me discard the STM metadata for non-contended objects.

Note also, that all the optimization techniques that I explore in my work are complementary to each other and can be combined to avoid different scenarios of unnecessary use of STM barriers. The example shown in Figure 2.4 ilustrates the idea of those optimization techniques, which work by eliminating the overheads of an STM and by providing a direct path for non-contended objects.

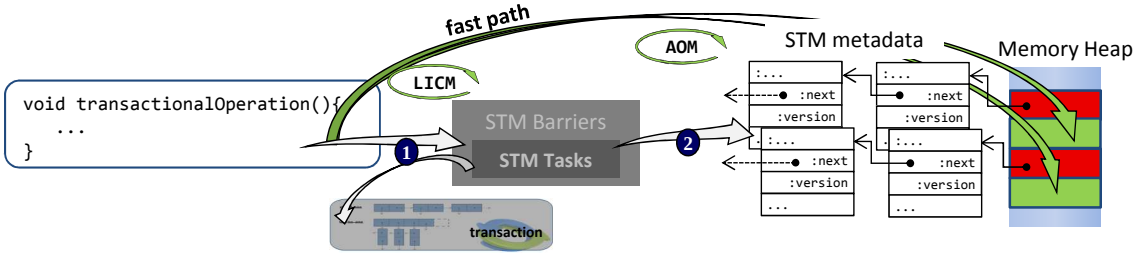


Figure 2.4: A fast path to access objects that are not under contention (represented as green segments on the memory heap). In this case the non-contented objects do not require the STM metadata and we can also avoid the STM tasks if those objects are also transaction local.

So, for non-contended objects we can automatically redirect accesses straight to the targeted object and avoid the full-blown STM barriers, which are imposed by the STM engines for all memory locations. Thus, assuming that in real-sized applications the vast majority of objects are not under contention, the number of objects that require STM barriers should be negligible when compared to the total number of objects in the application. This, in turn, means that if we use lightweight barriers for non-contended objects, we may have a significant improvement in the performance of large-scale programs.

2.6 Summary

STMs are often criticized for introducing unacceptable overheads when compared with either the sequential version or a lock-based version of any realistic benchmark. My experience in testing STMs with several realistic benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among threads.

I started this chapter showing how useless STM barriers can degrade the performance of a large-scale application, such as STMBench7. Then, I presented the speedup achieved by a transactified operation when we suppress those barriers. This analysis gives us an idea of how much speedup we may expect to achieve with a convenient STM optimization.

Based on these observations I describe some of the main causes of the overheads of STM barriers, such as the additional STM tasks and the metadata associated to every transactional location.

Chapter 3

Background & State of the Art

In this chapter I address in Section 3.1 the main properties that characterize the implementation of an STM and directly influence its performance.

Then I shall describe the state of the art on efficient support for software transactional memory according to the following structure. Section 3.2 introduces some of the runtime overheads incurred by many STM implementations and also presents some solutions to mitigate those overheads. Section 3.3 describes some reasons for compiler over-instrumentation and some approaches to reduce it. In the following, Section 3.4 identifies the set of characteristics that I would like to have in a good benchmark for an STM system, and discuss to what extent some of the existing benchmarks satisfy those requirements. Finally, Section 3.5 presents some tools for profiling transactional applications and describes some contributions in this field.

3.1 STM design alternatives

Over the last ten years, the intense research on STMs produced a number of STM implementations that explore a vast design space of choices for several key aspects of an STM runtime. A comprehensive analysis of most of these choices is available in [Harris *et al.* , 2010]. In the following, however, I shall concentrate on the aspects that I believe influence the most the performance of an STM. Namely, I shall discuss design alternatives for *buffering mechanism*, *ownership acquisition time*, *concurrency control*, *validation time*, *transactional memory accesses* and *API decomposition level*.

3.1.1 Buffering mechanism

Regarding the *buffering mechanism* [Dalessandro *et al.* , 2010] STMs may be classified as having either a *redo-log* or *undo-log* strategy. Some STMs [Cachopo & Rito-Silva, 2006; Dice *et al.* , 2006; Harris & Fraser, 2003] just copy the values to the affected memory locations upon successful commit. Such STMs are referred to as *redo-log* STMs, because they redo the writes upon commit. In this approach, the *STM read barriers* need to consult the redo-log before returning a value from a memory location.

Other STMs [Adl-Tabatabai *et al.* , 2006; Harris *et al.* , 2006] directly update memory locations speculatively and maintain an *undo log* that preserves their overwritten values. If a transaction aborts it must restore the written locations with their original values kept in the *undo log*.

3.1.2 Ownership acquisition

Regardless of the *buffering mechanism*, and before writing to a memory location, a transaction must acquire exclusive access to that location—*ownership acquisition*. Acquisition of memory locations can be either *eager* [Adl-Tabatabai *et al.* , 2006; Harris *et al.* , 2006] or *lazy* [Cachopo & Rito-Silva, 2006; Marathe *et al.* , 2004] and it states the *ownership acquisition time* [Marathe *et al.* , 2004] that is also described in some literature as *conflict detection time* [Marathe *et al.* , 2006]. The *eager* and *lazy* approaches are also, respectively, known in [Dice & Shavit, 2006] as *encounter order locking* and *commit time locking*. With *eager* acquisition a transaction tries to acquire ownership of a location when it first writes to that location. In contrast with this, with *lazy* acquisition a transaction acquires ownership of locations only at commit time.

Because *undo-log* STMs store speculatively written data directly in the memory locations during transaction execution, they must acquire ownership *eagerly* to prevent other transactions from either writing to, or reading from those memory locations. *Redo-log* STMs can use either *eager* or *lazy* acquisition.

3.1.3 Concurrency control and validation time

To acquire ownership of a memory location an STM can use different types of *concurrency control* mechanisms, which in turn affects when *validation time* occurs: on *every memory access* or just *before committing*. An STM may use a *pessimistic concurrency*

control approach, such as using a *read-write lock* for every memory location to control the *ownership acquisition*. In this case a conflict can be immediately detected when a transaction accesses a memory location. A *read-write lock* provides two modes for lock acquisition: read and write. In the read mode the *read-write lock* can be held simultaneously by multiple readers—*shared*—, whereas in the write mode it can be held only by a unique writer—*exclusive*. The major disadvantage of this *concurrency control* approach is that it penalizes all read operations with the overhead of the lock acquisition.

On the other hand, a *transactional versioning* approach allows read operations to proceed without the need of acquiring any lock. In this approach, all memory locations have a version number associated with them, which is incremented every time an object is updated. In turn, transactions record the version numbers from objects that they read in a read-set. Then, before a transaction commits it validates the *read-set* by checking if all objects read are in the same version that was read by the transaction. If not, the transaction has to rollback and to undo all changes to updated locations.

When using *transactional versioning*, the transaction's validation can occur on *every memory access* or just *before committing*. Regardless of the approach taken for the *validation time*, the validation entails a traversal of the entire *read-set*. So the overheads of a validation on *every memory access* surpasses in large scale the validation *before committing*. Currently, the majority of the STMs [Adl-Tabatabai *et al.* , 2006; Harris *et al.* , 2006; Cachopo & Rito-Silva, 2006; Saha *et al.* , 2006] use a *transactional versioning* approach to validate the consistency of the *read-set* avoiding the lock acquisition for every read operation. Assuming that read operations significantly outnumber updates in atomic blocks, then mitigating the overhead of *STM read barriers* has a considerable impact in the overall performance.

Some STMs [Dice *et al.* , 2006; Marathe & Moir, 2008] use the *transactional versioning* approach in combination with a *global version clock* (GVC) to mitigate the overhead of iterating over the entire *read-set*. When a transaction commits, it increments the GVC and attaches its value with each modified memory location. The values of the GVC establish a serial order between transactions as follows: considering `i` and `j` versions of GVC, such that $\texttt{i} < \texttt{j}$ and `Ti`, `Tj` the transactions that commit with versions `i` and `j`, then there is a serial order between transactions `Ti` and `Tj` such that `Ti` happens-before `Tj`.

When a transaction begins execution it reads the GVC and stores its current version. To be a valid transaction, it cannot read memory locations with a version bigger than the value observed at the beginning of its execution. So, on every read a transaction

can immediately verify if that memory access makes the transaction invalid and liable to abort.

Another technique complementing the *transactional versioning* approach and GVC is the *multi-versioning*. This idea was first introduced by Reed in 1978 in the context of the distributed execution of atomic actions [Reed, 1983, 1978], which is a well known subject in the field of database management systems.

According to the *multi-versioning* approach a memory location stores a history of committed values, instead of a single value. Each value of the history has attached the version of the transaction that has committed that value. The major advantage of this approach is to enable read-only transactions to always commit successfully, because they can always see a valid snapshot of the memory corresponding to the version captured at the moment of the beginning of their execution. The drawback of this approach is the extra memory space required to store the multiple versions for every transactional location.

3.1.4 Transactional memory accesses

Another key difference among STMs is on *transactional memory accesses*. STMs may follow two distinct solutions: (1) following an *API-based* [Dice *et al.* , 2006; Cachopo & Rito-Silva, 2006; Herlihy *et al.* , 2006; Saha *et al.* , 2006; Marathe *et al.* , 2006], or (2) a *compiler-based* approach [Wang *et al.* , 2007; Korland *et al.* , 2010]. According to the former, an STM delegates on the programmer the responsibility of using specific STM functions to access memory locations inside a transaction. In the latter approach the programmer is unaware of whether the memory locations are accessed inside or outside a transaction and the STM compiler is responsible for automatically translating the transactional memory accesses into STM barriers. This process is called *transactification*.

3.1.5 API decomposition level

An STM API can offer more or less specialized functions to handle specific cases of transactional memory access—*API decomposition level*. Typically an STM may offer a general `STMRead()` method to read transactional memory locations. In this case the STM provides a *homogeneous* API [Dice *et al.* , 2006; Cachopo & Rito-Silva, 2006; Herlihy *et al.* , 2006; Saha *et al.* , 2006; Korland *et al.* , 2010]. On the other hand, an STM

may offer different functions to read different kinds of transactional memory locations. For example the API of an STM can provide a specialized `STMReadTransactionLocal()` that assumes that the value read is thread-local and as a consequence, can optimize away validations of that value. An STM providing such an API has a *heterogeneous* API [Yoo *et al.* , 2008].

3.2 Runtime Overheads

In many STM implementations the *write-set* is used as a private shadow copy of memory that must be updated by write operations and consulted by read operations [Harris & Fraser, 2003; Harris *et al.* , 2005; Cachopo & Rito-Silva, 2006]. Harris *et al.* [2006] have identified this feature as one that incurs in runtime overheads because write operations are duplicated: once to update the write-set and another to update the destination memory location, if the transaction commits successfully. Moreover, it forces all read operations inside atomic blocks to search the write-set, before returning the value of a memory location.

In their work, they tackle this problem by proposing a *direct access* STM that allows objects to be updated directly in the heap—*updates in-place*—avoiding duplicate writes for successfully committed transactions and avoiding lookups to the write-set on read operations. The same approach is proposed by Saha *et al.* [2006] in the implementation of McRT-STM, which allows optimistic reads and *updates in-place*. This solution penalizes only aborted transactions, which must undo changes to updated memory locations with their original value kept in the *write-set*. But assuming that most transactions commit successfully and reads significantly outnumber updates in atomic blocks, we expect to have performance gains by keeping the overhead in transactional reads low.

Although the previous works claim that an *undo log* approach is a better design choice, there are counter-opinions [Dice & Shavit, 2006; Spear *et al.* , 2009] proposing the opposite. In the work of Dice & Shavit [2006] they compare the performance among different STM implementations following different design choices and they observe that an *undo log* solution performs well on uncontended data structures, but degrades on contended ones. This happens due to the *eager ownership acquisition* imposed by the *undo log* approach. So, under high contended scenarios, leaving locks over memory locations during entire execution of a transaction may prevent many other transactions from proceeding successfully and it degrades the overall performance of an application.

Another key improvement of the *direct access* STM of Harris *et al.* [2006] is the integration of *transactional versioning* that is a fundamental part of its implementation and allows a transaction to detect conflicting updates. Harris et al. have presented the *direct access* STM as being the first one to implement the *transactional versioning* feature through an existing object header word, instead of using external tables of versioning records [Adl-Tabatabai *et al.* , 2006; Harris & Fraser, 2003; Saha *et al.* , 2006], additional header words [Harris *et al.* , 2005], or additional levels of indirection between object references and current object contents [Cachopo & Rito-Silva, 2006; Fraser, 2003; Herlihy *et al.* , 2006; Riegel *et al.* , 2006].

Yoo *et al.* [2008] identified another runtime overhead related to bookkeeping costs, which are fixed costs in executing a transaction startup and teardown. Current multi-thread workloads using locks try to keep critical sections as short as possible. When these workloads are transactified, the critical sections give rise to extremely short transactions and bookkeeping costs are poorly amortized over the length of the transactions. In their work with McRT-STM they provide a special execution mode for extremely short transactions that serializes transactions with a scalable global lock, thereby reducing the costs due to bookkeeping.

In the same work, Yoo et al. also point the false conflicts problem that occurs due to the coarse granularity of the conflict detection. This phenomenon may happen for two different reasons. The first one is because it may not distinguish between two different addresses on the same cache line. This is comparable to what happens in object-based conflict detection when it does not distinguish conflicts between fields of the same object. The second reason can be verified even between addresses in different cache lines due to the aliasing used to map addresses to transaction records.

Many STM implementations manage the ownership acquisition over memory locations through the use of an external table of *ownership records* (*orecs*). Mapping the location's addresses to *orecs* via a hash function can reduce the required memory space to store *orecs*, but can induce false conflicts when it maps different addresses to the same *orec*. To reduce the number of cases where this problem occurs, Yoo *et al.* [2008] propose to modify the hash function used in McRT-STM and use 4 additional bits per table entry to store 16 different transaction records into each table entry.

Dalessandro *et al.* [2010] also tackle the problems induced by the use of *orecs* and they propose a new STM implementation — NOrec — that abolishes their usage. The *orecs* have a primordial role in the transaction validation and their abolishment implies a new way to detect conflicting accesses to memory locations. So, instead of

metadata-based validation, which is the established approach for transaction validation using *orecs*, they use a *value-based validation*. Rather than logging the addresses of *orecs*, transactions log the addresses of the locations and the values read. Validation consists of re-reading the addresses and verifying that locations have not been modified, since they were read.

Mannarswamy *et al.* [2010a] also aim to reduce the transactions abort rate due to false conflicts but using a different approach. Their solution applies to STMs that follow a lock based conflict detection scheme and that are dependent on the efficiency of the mapping from accessed memory locations to locks. They have combined the STM lock assignment feature with another technique originated from other trend, which proposes to transform the atomic sections into lock based code—compiler assisted lock allocation (CLA). In their solution they leverage the knowledge of the application's data access patterns collected from the CLA program analysis to selectively assign locks to shared data — *selective compiler assisted lock assignment* (SCLA).

Another approach to reduce the transactions abort rate is through the use of *multi-versioning*, which was firstly proposed in the JVSTM [Cachopo & Rito-Silva, 2006]. One of the main characteristics of the JVSTM is that read-only transactions have very low overheads, and they never conflict with any other transaction. The *multi-versioning* solution was also adopted by the LSA-STM [Riegel *et al.* , 2006], which uses locators (indirection between object reference and object's content) based on the design of DSTM [Herlihy *et al.* , 2006]. More recently, the SMV-STM [Perelman & Keidar, 2010] also implements *multi-versioning* in a design that is most closely related to TL2 [Dice *et al.* , 2006], from which they borrow the ideas of *lazy ownership acquisition* of updated objects and a *global version clock* for consistency checking. Among *multi-versioned* STMs, the closest to SMV is LSA, but instead of a simple solution to garbage collection that keeps a constant number of versions for each object, the SMV keeps versions as long as they might be useful for ongoing transactions as happens in JVSTM.

A different trend aims to reduce the effects of runtime overheads through the redistribution of the STM tasks by auxiliary threads. The FastLane [Wamhoff *et al.* , 2013] introduces the concept of a *master* thread that executes transactions pessimistically without ever aborting, while the *helper* threads can commit speculative transactions only when they do not conflict with the master. This approach has shown good results for programs with low thread counts, typically outperforming a classical STM in the 1-6 threads range.

On the other hand, the STM^2 [Kestor *et al.* , 2011] proposes the division between *application threads* (computation) and auxiliary threads that perform STM management

operations. So, whereas application threads experience minimal overhead, the auxiliary threads, instead, validate read-sets, maintain transaction states and detect conflicts in parallel with the application threads' computation. The STM^2 shows speedups between 1.8x and 5.2x over the tested STM systems, on average. Yet, for the Vacation benchmark, STM^2 performs worse than TL2.

3.3 Compiler Over-instrumentation

One of the main reasons for compiler over-instrumentation is due to missing optimizations opportunities and unnecessary use of STM operations as was described by Harris *et al.* [2006]. In their work they propose to mitigate this problem by enhancing the *direct access* STM with a new decomposed interface that is used in the translation of the atomic blocks and exposed to the compiler, giving new opportunities for optimization.

Adl-Tabatabai *et al.* [2006] also analyze the problems of the compiler over-instrumentation and the effects of the compiler optimizations in the performance of software transactional memory. But instead of optimizing instructions on the compilation process of the atomic blocks, they introduce optimizations at just-in-time compilation level of a Java virtual machine. For that purpose they have extended the compiler intermediate representation—IR—with new STM operations— STIR—in such a way that the existing compiler optimizations can remove redundant STIRs.

Besides this kind of optimizations, it also includes elimination of STM barriers to access immutable memory locations and access them in the same way as any other unsynchronized location, avoiding all overheads and reaching a better performance on its manipulation. For objects allocated inside a transaction—transaction-local objects—the STM barriers may also be eliminated because these objects are not visible and, thus, not shared with other transactions until the transaction commits. So, there is no need to instrument memory accesses for these objects and they can be accessed in the same way as immutable objects.

These scenarios were also analyzed by Dragojevic *et al.* [2009], but for an unmanaged environment. In their work they have introduced new runtime and compiler techniques in the Intel C++ STM compiler. Like the work of Adl-Tabatabai *et al.* [2006], they also propose to elide STM barriers for immutable and transaction-local memory. However in the C/C++ language the support for the declaration of read-only variables constants using C/C++ keyword `const` does not prevent from updates to that

data, because the `const` qualifier could simply be cast away when the data is accessed. Thus, instead of trying to automatically detect which locations are read-only, they expose new API calls—`addPrivateMemoryBlock` and `removePrivateMemoryBlock`—that allow the programmer to annotate memory regions to be (or stop being) safe for accessing without STM barriers. To automatically identify transaction-local data they propose a technique for *capture analysis*. *Captured memory* corresponds to memory allocated inside a transaction, which cannot escape. Thus, accesses to such memory do not require STM barriers. They provide this feature at runtime and also in the compiler using pointer analysis, which determines whether a pointer points to memory allocated inside the current transaction.

They have identified a third case for which some STM barriers can be elided when accessing thread-local memory. But in this case the STM barriers cannot be completely removed because a transaction may have to undo changes to updated locations if it is aborted. Nevertheless, eliding some STM operations, such as open for read, can alleviate part of the overheads and improve the performance of read operations when accessing thread-local memory.

Similar optimizations also appear in [Wang *et al.* , 2007], and [Eddon & Herlihy, 2007], which apply fully interprocedural analysis to discover thread-local data. The work of Riegel *et al.* [2008] propose to tune the behavior of the STM for individual data partitions. Their approach relies on compiler data structure analysis (DSA) to identify the partitions of an application that may be thread-local or transaction-local.

Besides the previous three cases of memory locations: immutable, transaction-local and thread-local, there are other cases corresponding to different behaviors that do not need to be instrumented too, but that cannot be recognized as such via static analysis. For instance, we may have shared memory locations that are immutable with respect to the atomic blocks that access them, but that can be updated outside those blocks, thus preventing them from being declared as immutable. So, these memory locations must be declared in the same way as any other shared locations and there is no additional information that lets the compiler know that it does not need to instrument memory accesses to those locations.

Yoo *et al.* [2008] analyzed the compiler over-instrumentation problem due to its lack of application-level knowledge and suggest a solution that let the programmer directly convey such knowledge to the compiler. Their work is based on McRT-STM [Saha *et al.* , 2006], which was also used by Adl-Tabatabai *et al.* [2006].

The McRT-STM compiler generates two copies of code for each atomic function: a regular version and its transactional twin, which uses read/write barriers for each memory access. A call to an atomic function inside an atomic block is translated into a call to its transactional twin. According to the solution of Yoo *et al.* [2008] they propose a new `tm_waiver` annotation to mark a function or block that would not be instrumented by the compiler for memory access waivered code. Moreover, the `tm_waiver` annotation overrides an atomic function, meaning that when an atomic function is called inside a function annotated with `tm_waiver`, the function will behave as if itself was also annotated with as `tm_waiver`.

So, to prevent the over-instrumentation scenario described above, all functions or blocks that include memory accesses to that kind of locations should be annotated with `tm_waiver` to prevent them from being instrumented.

Likewise, Ni *et al.* [2008] propose that programmers have the responsibility of declaring which functions could avoid the instrumentation through the use of the annotation `tm_pure`. The same approach has been followed in managed runtime environments, such as the work of Beckman *et al.* [2009], which proposes the use of access permissions, via Java annotations, that can be applied on method parameters to describe how references may behave.

White & Spear [2010] have applied the concept of waivered code to hardware transactional memory adding a new `__tm_waiver` construct, which provides a weak form of open nesting. As in the proposal of Yoo *et al.* [2008] the effects of a waivered block cannot be rolled back and the programmer is responsible for preventing races between these blocks and other code.

Afek *et al.* [2011] aim to reduce over-instrumentation by applying some optimization techniques via static analysis. Some of these techniques are common in modern compilers, such as *Partial Redundancy Elimination*. These optimizations also include the automatic elision of STM barriers for transaction local memory. Yet, this approach does not achieve the same performance as solutions based on heterogeneous APIs, such as those previously mentioned, which allow the programmer to annotate functions or blocks of code that are excluded from transactification and thus avoid the use of STM barriers. In fact, static compiler analysis is often conservative, and thus cannot remove all unnecessary barriers, because program modules are dynamically loaded, for example, and it is impossible to perform whole program compiler analysis.

3.4 Benchmarks for STMs

Benchmarks are essential to test and to compare transactional memory (TM) systems. But, as realistic benchmarks are scarce, TM implementers often resort to micro-benchmarks, which are typically too simple to test their systems properly, leading to fair skepticism about the relevance of their results and the applicability of their approaches. Thus, the TM research community is in dire need of good, realistic benchmarks. But, what makes a benchmark good?

Harmanci *et al.* [2009] distinguish two kinds of experimental evaluations for transactional memory (TM) systems: performance evaluation and semantics evaluation (debugging/testing/verification). Regarding performance, they point out that the major challenge is the difficulty of finding benchmarks that are simultaneously precise enough to emphasize the TM features and realistic enough to capture the behavior of most common applications. Regarding semantics, one of the important requirements of a benchmark is the presence of a correctness test that is able to verify if an execution has produced correct results, thereby allowing TM implementers to identify problems with their solutions. Naturally, a good benchmark should allow both types of evaluation.

On the other hand, Ansari *et al.* [2008] argue that a TM benchmark should have as desirable features: large amounts of potential parallelism; several types of transactions; complex contention; and transactions with a wide range of durations (transaction length) and amount of data accesses (transaction size).

To the above requirements, I add that a good benchmark should be flexible enough to allow the integration of new synchronization mechanisms without requiring changes to its source-code. Without this, it is harder to do a fair comparison among different synchronizations strategies, because porting a benchmark to another TM system may change its behavior in ways that may affect the results. Moreover, by separating the core of the benchmark logic from the synchronization code, it allows for an independent and smoother upgrade path to the benchmark logic.

Another desirable feature that should be provided by a benchmark is to provide a synchronization mechanism based on a fine-grained locking approach, which may present a good performance and serves as a reference to achieve by other synchronizations mechanisms.

To address the lack of realistic benchmarks for TM systems, Guerraoui *et al.* [2007] introduced the STMBench7 benchmark: a benchmark for performance evaluation implemented in Java that models a realistic large scale CAD/CAM application. The main

feature of STMBench7 is that it uses long transactions and large data structures, becoming a big challenge to the majority of STMs. In fact, the work of Dragojevic *et al.* [2008] uses STMBench7 to make a performance evaluation of several STMs and concludes that this benchmark stretches STMs too much with regard to memory requirements. In particular, all of the tested STMs in unmanaged (C/C++) runtimes had problems coping with the size of a big data structure and they all crashed. In Java this problem is reduced by the presence of a garbage collector, but there is still a big overhead in memory imposed by the STMBench7 workloads, as happens for DSTM2 (a Java STM) which could actually not run at all due to high memory overheads.

The STMBench7 benchmark provides three kinds of workloads that vary in the ratio of update operations: *read-dominated* (10% update operations), *read/write* (40% update operations), and *write-dominated* (90% update operations). We can also enable or disable long transactions for each workload. However, we can neither easily disable, or enable, specific operations, nor configure the rate between the different kinds of operations.

The data structure of STMBench7 consists of a large graph of different kinds of objects and its operations manipulate large parts of this data structure. These operations vary in the length of the path that is randomly selected from the graph of objects. Typically, STMBench7 operations focus on object manipulation and there are no tasks that apply mathematical functions with different degrees of complexity as exist in other benchmarks, such as STAMP [Cao Minh *et al.* , 2008] and WormBench [Zyulkyarov *et al.* , 2008].

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. Unlike STMBench7, the STAMP applications are configurable at runtime and allow us to vary the level of contention, size of transactions, the percentage of writes, among other parameters. But this benchmark has several drawbacks also. First, not all applications make semantic evaluations of the tested STMs. Second, it is not easy to integrate STAMP applications with some STM algorithms, such as DSTM [Herlihy *et al.* , 2006] and JVSTM [Cachopo & Rito-Silva, 2006], because it requires us to modify its source code and change the type of all memory locations accessed by a transaction. Finally, it provides no lock-based synchronization to evaluate the performance of transactional versions in comparison to alternative synchronization mechanisms. Furthermore, there is no Java implementation for STAMP that is generic enough and able to integrate with any STM framework. The existing Java port is written in a specific dialect (IRC - *Irvine Research Compiler*) and requires manual conversion to be compliant with standard Java.

In the Java world, LeeTM [Ansari *et al.* , 2008] is an alternative to STMBench7 and it has many of the desirable properties of an STM benchmark: it is based on a real-world application and provides a wide range of transaction durations and sizes. The LeeTM also provides a verifier that validates the consistency of the final data structure. This is an advantage over STAMP benchmarks, which do not provide a verifier like this one (except for genome), due to the nature of their computation.

One of the limitations of the LeeTM benchmark, however, is that it does not allow extending it to new kinds of operations and research variations of its contention scenarios. Furthermore, there is no possibility of varying the read/write ratio of the benchmark, because all of the transactions write something, unlike most applications. Finally, the LeeTM benchmark does not provide an extensible API to change the synchronization mechanism, forcing programmers to modify its source-code.

WormBench [Zyulkyarov *et al.* , 2008] is a configurable transactional C# application that was designed to evaluate the performance and correctness of TM systems. The idea behind WormBench is inspired by the Snake game, but in this case the snakes are *worms* moving and performing *worm operations* in a *shared world* of *nodes*. Each *node* has a corresponding pair of coordinates—x, y—stored in a *coordinate* object.

The WormBench shares all the benefits of LeeTM, such as providing a verifier algorithm (correctness test) and a wide range of transaction durations and sizes. But, unlike LeeTM, it applies a broad diversity of mathematical operations and it is able to extend to new ones. Moreover WormBench is totally configurable with regard to: the percentage of update operations; the kind of operations and proportion between them; the maximum execution time or number of iterations; the contention level and synchronization strategy. In addition, the WormBench benchmark has a low complexity domain model (compared to STMBench7), making it easy to understand, and has a simple API. Still, as shown in [Zyulkyarov *et al.* , 2008], despite this simplicity, the WormBench benchmark can still reproduce STAMP workloads with the same characteristics.

Another particularity of WormBench is the ability to previously configure and record a stream of movements and operations that each worm will perform on the execution of the workload. Unlike other benchmarks, such as STMBench7 and STAMP, the stream of operations is randomly generated before the execution of the workload and then it can be executed repeatedly. This feature allows to run exactly the same conditions between different executions of a workload.

3.5 Debug and Profiling Tools for STMs

Most of the work in debug and profiling tools [Harmanci *et al.* , 2009; Herlihy & Lev, 2009; Zyulkyarov *et al.* , 2010a] focus on techniques for identifying correctness errors, rather than investigating performance. Researchers tend to focus in the correctness of applications that use STMs and little has been done to provide tools for profiling and tuning those applications.

Yoo *et al.* [2008] considered that many conclusions taken from studies of STMs on small-scale workloads cannot readily be applied to real-life, large-scale workloads. Another problem pointed to those studies is in how results are evaluated. For instance, blindly measuring the overall transaction abort rate does not suffice in fully characterizing a workload, because simple transactions tend to be executed much more frequently than complex transactions and the high abort rate for complex transactions is overshadowed by the high commit rate of simple ones. To avoid wrong analysis they recommend considering per atomic block statistics instead of overall statistics.

It is also commonly believed that false conflicts and the work wasted executing aborted transactions are sources of performance degradation. In this sense, several researchers have developed tools that help programmers to collect and analyze statistical information about the execution of transactional applications. Lourenço *et al.* [2009] have developed a monitoring framework that collects the transactional events from the execution of an STM into a log file, that is then used by a visualization tool—JTraceView—to display both statistical information and a time-space diagram. The statistical charts include information about abort types, abort reasons, wasted work, as well as other informations.

Chakrabarti [2010a] also refers to the limitations of measuring the STM performance just by looking at the total number of aborts, the total execution time and the scalability trends. Based on the consideration of Yoo *et al.* [2008] that false conflicts are a major source of unnecessary aborts in an STM, Chakrabarti developed a solution that helps the programmer to identify potential conflicts at the transaction level—*coarse*—and at the memory reference level—*fine*. For that purpose he has implemented general extensions to an STM whereby the transaction loads and stores associated with a conflict can be identified. As the program executes, it builds a *dynamic conflict graph* that will be later used by an offline tool—TM_analyzer—to correlate the data and emit meaningful information.

Zyulkyarov *et al.* [2010a] proposed a *conflict point discovery* technique that identifies the first program statements involved in a conflict and that is similar to the *dynamic*

conflict graph approach introduced by Chakrabarti [2010a]. Continuing this work, Zyulkyarov *et al.* [2010b] introduced a series of profiling techniques for transactional applications that provide information about the wasted work caused by aborting transactions, similar to the work of Lourenço *et al.* [2009]. To make analysis simpler, instead of reporting *conflicting points* in machine addresses, they present the results in source language such as variable names. Their profiling tool also provides another view that represents the abort relationship between the atomic blocks, similar to the *coarse dynamic conflict graph* of Chakrabarti [2010a]. In addition, this tool also provides a local summary about the performance of specific parts of the program execution, associates contextual information with the conflicts, and accounts for all conflicting memory accesses within aborted transactions.

3.6 Summary

Several researchers have turned their attention to understand the causes behind the runtime overheads incurred by transactional applications and have pointed some problems related to the following aspects:

- maintenance of the transaction-private log;
- use of external tables for transaction ownership records;
- coarse granularity of the conflict detection;
- fixed costs in executing a transaction startup and teardown — bookkeeping cost.

Among the variety of known solutions to mitigate the runtime overheads there are some contradictory proposals, as happens between the *undo log* approach versus the *redo log* approach. In fact, the better solution is closely dependent of the characteristics of the workload where it is evaluated, leading to different analysis and conclusions, as Dice and Shavit have described in their work [Dice & Shavit, 2006]. So, there are no universal techniques that are globally adopted by all STMs implementations.

Another non consensual solution that complements the use of *transactional versioning* in STMs is the *multi-versioning* approach that was first proposed in JVSTM [Cachopo & Rito-Silva, 2006] and later was also followed by the LSA-STM [Riegel *et al.* , 2006] and the SMV-STM [Perelman & Keidar, 2010]. Although the *multi-versioning*

tends to reduce the number of aborted transactions in read-dominated scenarios, because read-only transactions never conflict with other transactions, it also incurs high memory overheads to store the multiple versions of a transactional location, attenuating the benefits of the performance gains.

Another problem that affects STM performance is compiler over-instrumentation, which can be due to:

- missing optimization opportunities;
- unnecessary use of STM operations;
- lack of application-level knowledge.

These problems were tackled by several researchers proposing different solutions that include some of the following optimizations:

- elimination of redundant STM operations;
- elimination of STM barriers to access immutable memory locations;
- elimination of open for read operations before accessing transaction-local objects;
- `tm_waiver` annotation to mark a function or block that would not be instrumented.

Finally, to understand more clearly the causes behind the STM overheads it is necessary to have benchmarks and profiling tools that evidence the bottlenecks in STM implementations. Yet, to the best of my knowledge there is no benchmark that provides simultaneously the following two very important features: (1) configurability and flexibility in the integration of new synchronization mechanisms, and (2) a broad diversity of operations and extensibility to support new kinds of operations.

Among the various benchmarks for STMs, Wormbench stands out by providing both a simple domain model and the ability to extend it with new kinds of operations among a wide range of functions with different levels of complexity. These two features are very helpful to help in the debugging of an STM, because they allow us to have very fine-grained control over the behavior of the benchmark. Yet, Wormbench has some limitations that makes it hard to evaluate different STM implementations: It uses a macro based approach to integrate different STM implementations and it is available only in C#.

As previously stated, one of the crucial steps of my work was to identify the main overheads of an STM. For this, I need a benchmark with the characteristics of Wormbench. So, I made a port of Wormbench to Java, extending it in several ways and making it more useful as a testbed for evaluating STMs. Moreover, my port, which I called JWormBench,[1] was designed to be easily extensible and to allow easy integration with different STMs. For more information about JWormBench and its design, see Appendix A.

[1] Available at: http://inesc-id-esw.github.io/jwormbench/

Chapter 4

Annotations to Avoid Over-instrumentation

The loss of performance of an STM observed on a real-world–sized benchmark is often attributed to the *over-instrumentation* [Yoo *et al.* , 2008] made by overzealous STM compilers that protect every memory access with an STM barrier.

In this chapter I explore an extension of Deuce that allows programmers to add annotations to their programs specifying that certain memory locations are not under contention, and, therefore, do not need to be instrumented in the same way as are shared data under the control of the STM, which incur in large overheads. This contrasts with the generally accepted idea that STMs should be completely transparent, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations. That is the approach taken by Deuce, which provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

Whereas the transparent API would be ideal for programmers, in practice that leads to unacceptable overheads. So, in this chapter I explore an alternative approach that allow programmers to have some degree of control on what gets transactified. To show the benefits of this, I used the JWormBench [Carvalho & Cachopo, 2011] benchmark to evaluate the performance of Deuce with and without programmer annotations and show a speedup of up to 22-fold, making the optimized version perform similarly to a very fine-grained lock-based scheme.

To identify the overheads caused by over-instrumentation and what may be gained by having finer grained control over what to instrument, I extended the Deuce API

with two Java annotations—`@NoSyncField` and `@NoSyncArray`—that can be applied on fields declarations and type declarations, respectively, to avoid over-instrumentation in certain scenarios.

In this chapter I briefly introduce in Section 4.1 the Deuce STM and why the existing optimizations are not enough to suppress useless STM barriers. After that, in Section 4.2, I describe some scenarios of over-instrumentation in JWormbench that cannot be solved by the optimizations mechanisms provided by Deuce STM. In Section 4.3, I introduce the effects and behavior of the new annotations in the Deuce framework, used to avoid over-instrumentation. Section 4.4 describes the tested configurations of JWormBench and presents a performance evaluation. Finally, in Section 4.5, I conclude with a discussion of my approach and results.

4.1 Deuce STM Optimizations

The Deuce STM compiler is provided as a bytecode instrumentation engine implemented with ASM [Binder *et al.* , 2007] (a Java bytecode manipulation and analysis framework). Its two major goals are: (1) to integrate the implementation of any synchronization technique, and, in particular, different STMs; and (2) to provide a transparent synchronization API, meaning that a programmer using it just needs to be concerned with the identification of the methods that should execute atomically. For this purpose, the programmer should mark those methods with an `@Atomic` annotation (denoting an *atomic* method) and the Deuce engine will automatically synchronize their execution using the synchronization technique specified by the programmer.

To that end, the Deuce compiler instruments the code region of *atomic* methods with calls to the underlying STM runtime. STM runtime exposes an interface to the compiler that includes functions for starting and ending transactions and for performing transactional reads and writes of memory locations (*STM barriers*).

A naive STM compiler translates every memory access inside a transaction into a read or a write barrier and, therefore, the compiler-generated code may include more STM barriers than necessary—*compiler over-instrumentation* [Yoo *et al.* , 2008]. Furthermore, an STM barrier typically requires orders of magnitude more machine cycles than a simple memory access. So, whereas the approach taken by STM compilers ensures the correctness of the whole application, it also degrades its performance significantly.

During instrumentation, Deuce STM performs two optimizations to suppress useless STM barriers. First, Deuce STM does not instrument accesses to `final` fields, as they cannot be modified after creation. This optimization avoids the use of STM barriers when accessing immutable fields, provided that they were correctly identified in the application code. Yet, there is no similar solution to allow the programmers express the same intention about the behavior of arrays' elements. The `final` keyword used in an array type declaration just avoids the declared variable (array's reference) from being modified after its initialization and it does not avoid the array's elements from being modified.

Second, programmers may exclude some classes from being transformed by specifying the names of the classes to be excluded via a runtime parameter (`org.deuce.exclude`). Alternatively, the programmer may use the annotation—`@Exclude`—to exclude the annotated type from being instrumented. Yet, and unlike the runtime parameter `org.deuce.exclude`, this approach does not allow instances of excluded classes to be accessed within a transactional scope. This approach, however, reduces the transparency of the Deuce API. Moreover, it has some limitations: again, it does not work with arrays, nor can it be used when the same class has both instances that are shared and instances that are not shared across the transaction's boundaries.

4.2 Over-instrumented Tasks

To explore the overheads caused by over-instrumentation I used in my analysis the JWormBench. Using profiling analysis I observed that JWormBench, when instrumented with the Deuce framework, spent the majority of the execution time in five different kinds of memory accesses to both objects and arrays:

1. $Coordinate$: fields `x` and `y` of the *coordinate* object;
2. $Worm$: *coordinate* array;
3. $Node$: field `value` of the *node* object;
4. $World$: *node* matrix;
5. $Aux\ array$: auxiliary array to the function that defines a *worm operation*.

Table 4.1 shows the percentage of time spent on each kind of memory access, per operation. From all these operations, only the *Sort* and *Transpose* are read-write, whereas the remaining operations are all read-only. In this analysis I have excluded some read-write operations that are just the combination of two read-only operations in the form of $replaceROwithRO$, such as, *ReplaceMedianWithMin*. So, in these cases the distribution of time consumption for this kind of read-write operations is equal to the sum of the distributions in the corresponding read-only operations.

These results were collected with Deuce configured to use the TL2 and with just one worker thread. Of all the memory locations depicted in Table 1, only the third location—*Node*—should be instrumented, as it is the only one that is shared and updated by concurrent transactions. So, I confirm my expectations that a significant fraction of the time spent by a worm operation is wasted in the execution of useless STM barriers.

#	Operation	(1) $Coordinate$	(2) $Worm$	(3) $Node$	(4) $World$	(5) $Aux\ array$	(6) $Flow$
0	Sum	31%	18%	18%	33%	0%	0%
1	Average	38%	14%	15%	32%	0%	0%
2	Median	2%	4%	1%	2%	90%	1%
3	Minimum	27%	23%	15%	35%	0%	0%
4	Maximum	30%	27%	12%	31%	0%	0%
11	Sort	4%	3%	1%	5%	87%	0%
12	Transpose	26%	20%	6%	27%	18%	2%

Table 4.1: Distribution of the operation execution time accessing each of the five kinds of memory locations, with Deuce configured to use the TL2 and with just one worker thread. There is an extra column—*Flow*—that collects the time spent in the execution of the control flow of the transactions

Yet, we cannot easily avoid these useless STM barriers with the current API of Deuce. Essentially, these memory accesses are limited to objects, or arrays, that are:

- *thread local*, meaning that the most part of the instrumentation incurred by an STM when accessing those fields could be attenuated, except for the *undo log* that is still required to revert the updated data if the transaction aborts (e.g., $Coordinate$).
- *transaction local*, and, therefore, are not shared among different transactions (e.g., $Aux\ array$). So, these objects are private to a single transaction and do not need to be synchronized and thus, do not have to be instrumented.

- *unmodified* inside transactions, but are still subject to changes outside them (e.g., $Worm$ and $World$). In many situations these objects have an initialization phase where they are modified, but after a quiescent period they become immutable. Yet, we cannot define this kind of locations as `final`, because it prohibits changes to the declared fields out of the constructor, which can be too restrictive for some applications. Moreover, for arrays, the `final` keyword just restricts the behaviour of the array's reference and does not forbid any manipulation of its elements.

4.3 New Java Annotations for the Deuce API

To explore what may be gained by having more fine-grained control over what to instrument, I extended the Deuce API to help the Deuce engine avoid STM barriers for both objects and arrays. For that purpose, my solution includes two Java annotations, `@NoSyncField` and `@NoSyncArray`, which should be parametrized with a value of the *enum* type `NoSyncBehavior`. This parameter specifies the behavior of the annotated memory location as *immutable*, *transaction local*, or *thread local*.

The information given by the parameter of the type `NoSyncBehavior` is also relevant in a debug mode in which the Deuce framework verifies if the applied behavior to the annotated memory location is consistent with the way how that location is manipulated during the execution of the program. Under the debug mode the Deuce framework alerts the end user if a location is accessed in such a way that violates the behavior specified by its annotation.

In the following I describe the effects of these annotations in the elimination of over-instrumentation.

4.3.1 `@NoSyncField` annotation

The `@NoSyncField` annotation can be applied to fields' declaration in three different ways: `@NoSyncField(Immutable)`, `@NoSyncField(TransactionLocal)` and `@NoSyncField(ThreadLocal)`. Annotating a field with `@NoSyncField(Immutable)` has an effect similar to the Java `final` keyword on field declarations. Both make the Deuce framework avoid instrumentation when accessing those fields. However, the `final` keyword has another effect at the Java level, prohibiting changes to the declared field

after its initialization. This behavior could be too restrictive for memory locations that are unmodified inside transactions, but are still subject to be modified outside them. Unlike the `final` keyword, annotating with `@NoSyncField(Immutable)` just reports that the annotated field should not be updated inside a transaction, thus avoiding any synchronization when accessing it.

Sometimes, when we are applying this annotation we may also be interested in providing weak atomicity [Martin *et al.* , 2006] (i.e., not guaranteeing that concurrent accesses to a shared memory location from both inside and outside a transaction are consistent).

Another use of the `@NoSyncField` annotation is to annotate fields that are not shared among different threads. So, these fields could be private to a single transaction or to a single thread. In the first case the fields should be annotated with `@NoSyncField(TransactionLocal)` meaning that the annotated fields do not need to be synchronized and therefore do not have to be instrumented. In the second case the fields should be annotated with `@NoSyncField(ThreadLocal)` meaning that most of the instrumentation incurred by an STM when accessing those fields could be omitted, with the exception of the *undo log*, which is still required to revert the updated data if the transaction aborts.

The effects of the `@NoSyncField(TransactionLocal)` on Deuce framework are the same as the `@NoSyncField(Immutable)`: Both avoid instrumentation of the annotated memory locations when they are accessed inside a transaction. They differ only under the debug mode and in the way in which the Deuce framework verifies if the specified behavior is fulfilled by the transactified program: the former is violated if the annotated location is accessed by a different transaction from the one that have initialized the location and the latter is violated if the annotated location is updated inside of any transaction.

The `@NoSyncField(ThreadLocal)` can be applied on the declaration of memory locations that have affinity to only one thread. Therefore, these memory locations do not need to be instrumented in the same way as are the memory locations that are shared among different threads. In this case, for thread local data, the transactions can read and update these memory locations in-place avoiding the overhead of maintaining a *read set* and *write set*. However, the transactions still need to keep an *undo log* where they register the original values of the updated memory locations. Then, when a transaction aborts it uses the *undo log* to revert the data that was updated.

4.3.2 `@NoSyncArray` annotation

Annotating arrays is more challenging than annotating object fields. One difficulty is to find a way to attach the intention—*array elements are immutable, or transaction local, or thread local*—to an array declaration. Let us review the approach followed in the previous subsection to understand the differences. In that case a field will be annotated with an annotation `@NoSyncField`. Then, Java bytecodes for field manipulation have access to the field's metadata and at compile time we can check the field's annotations and decide whether to instrument or not the access to them.

The main difference is that Java bytecodes for arrays manipulation receives an array reference as parameter and not the declared array variable. So, there is no easy way for the compiler to know the array variable at the moment it processes a bytecode for array access.

An alternative approach is to attach the intention to the array object instead of the array variable. However, this strategy postpones the decision from compile-time to runtime and will not eliminate all the unnecessary over-instrumentation on arrays.

As we cannot annotate the array type, I adopted a different solution: to annotate the type of the array's element with the `@NoSyncArray` annotation . This approach has limitations too because instead of annotating the array declaration I propose to do that in the declaration of the type of the array's element.

The `@NoSyncArray(Immutable)` annotation can be applied to the type of the array's element, whose elements are immutable during the array's life cycle. Note that the `final` keyword for arrays declaration has a distinct effect from the one that is specified by the `@NoSyncArray(Immutable)`. In the case of the `final` keyword, it just avoids the declared variable (array's reference) from being modified after its initialization and it does not mean anything about the characteristics of its elements. Whereas the `@NoSyncArray(Immutable)` annotation declares that the elements of the array will not change inside a transaction.

Finally, we can also use `@NoSyncArray` to annotate the declaration of the type of the array's elements as `@NoSyncArray(TransactionLocal)` or `@NoSyncArray(ThreadLocal)` to, respectively, denote that the array's elements are private to a single transaction or to a single thread.

One limitation of this solution is that it is restricted to arrays of non-primitive types, as we have no way to annotate primitive types. One possible alternative, which

is still compliant with my optimization technique, is to use a wrapper class for each primitive type. However, this approach has an additional overhead by the extra indirection incurred by the wrapper object. Depending on each situation, we should evaluate the advantages of avoiding useless STM barriers, over the disadvantages of the extra indirections incurred by the use of wrapper objects.

Outside a transaction, it is more efficient to access a primitive type array than accessing the corresponding array of wrapper types. On the other hand, when accessing the array within a transaction, the overhead of the additional indirection incurred by the wrapper object is lower than the overhead of the useless STM barrier, which we may avoid with my optimization technique for non-shared and immutable arrays. So, we should replace a primitive type array by the corresponding array of wrapper types only if the number of useless STM barriers is bigger than the number of accesses performed outside a transaction. This condition is always true for transactional-local arrays, because there are no accesses to that array outside a transaction. This means that it is always better in terms of performance to use the optimization in this case, which may be accomplished by: (1) replacing primitive type arrays with the corresponding array of wrapper types, and (2) annotating the wrapper type with the `@NoSyncArray` annotation. For thread-local and immutable arrays, however, that condition is not always true and, thus, we should decide on whether to use this optimization technique depending on the expected pattern of accesses to the array.

4.4 Performance Evaluation

Using the new annotations of the Deuce API we can reduce the overhead incurred by the STM barriers in the scenarios 1, 2, 4, and 5 of Table 4.1 on page 46. In the results of Table 4.2 on the next page I show the difference between the performance of each operation whether it performs all useless STM barriers or not, and the corresponding speedup. From these results we can see the speedup achieved when we exclude useless STM barriers, which is between 5 and 38.4.

To evaluate the effect of my approach on the performance and the scalability of the JWormBench benchmark, I compared my optimized version of the Deuce framework against the released version 1.3.0. both using the TL2 STM. Simultaneously, I also made a comparison of the JWormBench with the lock-free implementation of JVSTM [Fernandes & Cachopo, 2011], but manually transactified (without using Deuce). In my

	Sum	Avg	Med	Min	Max	Sort	Trans
All STM barriers	2.220	2.900	49.160	2.820	2.610	42.760	5.660
Excluding useless STM barriers	380	450	1.280	480	520	1.050	420
Speedup	5.8	6.4	38.4	5.9	5.0	40.7	13.5

Table 4.2: The execution time in milliseconds of each *worm operation* with and without useless STM barriers and the corresponding speedup when we suppress those barriers.

analysis I also include an implementation of a fine-grained lock-based *step* for JWorm-Bench, which acquires locks for all the nodes under a worm's head in a pre-specified order to avoid deadlocks (I use an object's monitor per node).

The testing workload has a *world*'s size of 512 nodes and 48 *worms* with a body's length of one node and the head's size varying between 2 and 16 nodes, corresponding to a number of nodes under the worm's head between 4 and 256. The length of the body affects collisions between worms and I excluded that behavior for this analysis. The size of the head affects the size of transactions for each *worm operation* according to the Table A.1 on page 136.

To evaluate the overall performance of the benchmark, I have used three different configurations. In all configurations I keep the ratio between read-write and read-only *worm operations* of 20%-80%, as depicted in Table 4.3 on the next page. The configurations tested are as follows:

1. Combination of *read-only* and *n-reads-1-write worm operations*, excluding operations based on the *median* operation. So, these *worm operations* are not very computationally intensive, having complexity *O(n)*, and the write-set for all read-write transactions has a length of one (n denotes the number of nodes under the worm's head).

2. Equals to the previous configuration, but including operations based on *median*. So, this configuration is more computationally intensive than the previous one, with 4 *worm operations* having complexity $O(n^2)$.

3. Combination of *read-only* and *n-reads-n-writes worm operations*. Two of these operations have complexity $O(n^2)$ and the size of the write-set, for read-write transactions, is equal to the number of nodes under *worm's head*.

	worm operation	Configurations 1	2	3
read-only	Sum	20%	16%	16%
	Average	20%	16%	16%
	Median		16%	16%
	Minimum	20%	16%	16%
	Maximum	20%	16%	16%
n-reads-1-write	ReplaceMaxWithAverage	7%	4%	
	ReplaceMinWithAverage	7%	4%	
	ReplaceMedianWithAverage		3%	
	ReplaceMedianWithMax		3%	
	ReplaceMedianWithMin		3%	
	ReplaceMaxAndMin	6%	3%	
n-reads-n-writes	Sort			10%
	Transpose			10%

Table 4.3: Ratio between *worm operations* for each of the three workloads' configurations.

The tests were performed on a machine with 4 AMD Opteron(tm) 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the `1.6.0_22-b04`, running on Ubuntu with Linux kernel version 2.6.32.

The charts of figure 4.1 on the facing page depict the speedup of each synchronization mechanism over sequential, non-instrumented code. These results show that Deuce 1.3.0 just scales for the first configuration, without the presence of more computationally intensive *worm operations*, such as *median* and *sort*. On the other hand, my optimization proposal for the Deuce framework scales in the three scenarios and performs better than the previous versions: 2 times better in the first configuration and 22 times in both configurations 2 and 3.

In comparison with the fine-grained lock approach, the optimized version of the Deuce framework has a better performance for configurations 1 and 2, but is almost equals in performance for configuration 3. The number of locks acquired by the TL2 with the optimized version of Deuce should be approximately the same of the fine-grained lock approach. Yet, whereas the latter uses an object's monitor per node, the former uses a lock-based hash table. So, although the TL2 approach may have false conflicts, it is simultaneously more lightweight and therefore it gets a better performance in configurations 1 and 2, because these workloads have a very low conflict rate.

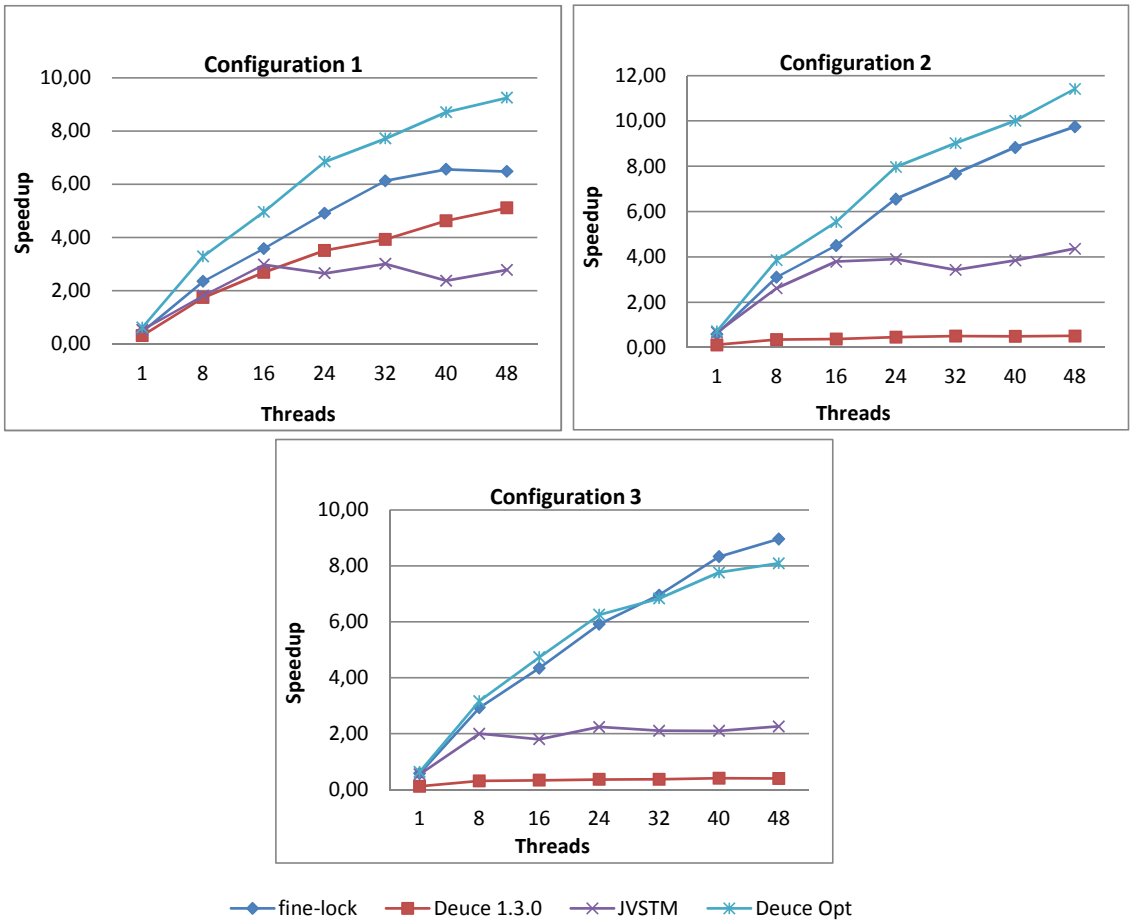


Figure 4.1: Speedup of each synchronization mechanism over sequential, non-instrumented code for each of the configurations shown in Table 4.3

The worst performance verified for the STMs in configuration 3 may happen because the size of the write-set in this case is bigger than in the previous ones, increasing the number of conflicts and causing much aborted transactions.

4.5 Summary

The solution that I explore in this work is to enable programmer to convey to STM that some memory locations are not to be manipulated transactionally. So, Deuce STM may take advantage of the information given by the programmer to eliminate some of the STM barriers that access those memory locations.

This approach reduces the transparency of the STM, which is one of its advantages over lock-based approaches, but it proves to be able to get huge benefits performance-wise. In fact, I have been able to get a 22-fold improvement on the throughput of a

realistic benchmark. This result is consistent with other observations made on benchmarks that use a similar approach (e.g., the results of the JVSTM on the STMBench7 reported in [Fernandes & Cachopo, 2011]).

Actually, not only do we get a huge speedup when we use a less transparent API, my results show that the STM performs better or as good as a fine-grained lock-based approach, which is particularly easy to use in JWormBench, but may not be in other applications. Still, I argue that the lock-based approach is harder to develop and get right than the use of annotations to identify non-transactional memory: To implement a lock-based approach, we need not only to identify the shared resources, as in my approach, but we have to be careful about getting the locks for all of the accessed resources, and doing it in the correct order. So, even if we are losing some of the transparency of the STM approach, I believe that this may be a reasonable tradeoff between easiness of development and performance.

Chapter 5

Lightweight Identification of Captured Memory

In Chapter 4, I described my work to explore a new optimization technique based on a new STM API that allows programmers to specify which types should not be transactified, and thus, avoid useless STM barriers [Carvalho & Cachopo, 2011].

The main idea of this optimization technique is to decompose the STM's API in heterogeneous parts that allow the programmer to convey application-level information about the behavior of the memory locations to the instrumentation engine. The same approach is the basis for other optimization techniques proposed by several researchers (e.g. [Harris *et al.* , 2006], [Yoo *et al.* , 2008] and [Beckman *et al.* , 2009]). Yet, this approach contrasts with one of the main advantages of an STM, which is to provide a transparent synchronization API.

So, although effective, the proposal of an extended STM API for Deuce has the following limitations: (1) it delegates on the programmer the responsibility of identifying the types whose instances are not shared; (2) it may not be applicable for types provided by a third-party library with restricted visibility, and (3) it is not feasible if we have both shared and non-shared objects of the same type.

To address these limitations, other approaches propose automatic mechanisms based on compile-time and runtime techniques that identify and avoid useless STM barriers. For instance, Afek et al. [Afek *et al.* , 2011] added to Deuce STM a static analysis technique to enable compile-time optimizations that avoid instrumentation of memory accesses in several situations, including to transaction local memory. Yet, this approach

does not accomplish the performance improvements shown by solutions based on heterogeneous APIs. However, I argue that automatic approaches that keep the transparency of the STM API are better suited to the overall goal of STMs. So, I propose to tackle this problem and find a technique based on runtime analysis that automatically and efficiently elide STM barriers for transaction local memory.

My work is based on the proposal of Dragojevic et al. [Dragojevic *et al.* , 2009], which introduces the concept of *captured memory* as memory allocated inside a transaction that cannot *escape* (i.e., is *captured* by) its allocating transaction. Captured memory corresponds to newly allocated objects that did not exist before the beginning of their allocating transaction and that, therefore, are held within the transaction until it commits. They use the term *capture analysis* (similar to *escape* analysis) to refer to a compile- or runtime-time algorithm that determines whether a memory location is captured by a transaction or not.

Given the lack of demonstrable effectiveness of the static compiler analysis [Afek *et al.* , 2011], here I am interested in exploring the proposal of Dragojevic et al. [Dragojevic *et al.* , 2009] for *runtime capture analysis*, adapt it to a managed runtime environment and make it more efficient. Note, however, that my proposal cannot be applied to unmanaged languages such as C/C++, which the seminal paper targets.

The main contributions of this chapter are:

- A new runtime technique for *lightweight identification of captured memory*—LICM—for managed environments that is independent of the underlying STM design (Section 5.3). My approach is surprisingly simple, yet effective, being up to 5 times faster than the *filtering* algorithm proposed by [Dragojevic *et al.* , 2009] (which I briefly introduce in Section 5.2).

- I implemented the LICM in Deuce STM. My implementation uses a new infrastructure of enhancement transformations, which is described in Section 5.4. By providing an implementation of my proposal within Deuce STM, I was able to test it with a variety of baseline STM algorithms, namely, LSA [Riegel *et al.* , 2006], TL2 [Dice *et al.* , 2006], and JVSTM [Fernandes & Cachopo, 2011].

- I performed extensive experimental tests for a wide variety of benchmarks (Section 5.5), including real-world–sized benchmarks that are known for being specially challenging for STMs. The goal of these tests was not only to evaluate the performance of my proposal, but, more importantly, to assess the usefulness of the runtime capture analysis, thus completing the analysis of [Dragojevic *et al.* ,

2009] about how many of the memory accesses are to captured locations. Besides the STAMP, I also analyze the STMBench7, and the JWormBench, which were not included in [Dragojevic *et al.* , 2009].

- For the first time, in some of the more challenging benchmarks and without relaxing the transparency of the STM API, the LICM makes STM's performance competitive with the best fine-grained lock-based approaches. Moreover, given its lightweight nature, it has almost no overhead when the benchmark presents no opportunities for optimizations.

The next Section introduces the basics of Deuce STM necessary to understand the adaptation of the runtime capture analysis technique. After that, in Section 5.2 I describe the original proposal of [Dragojevic *et al.* , 2009] in Deuce STM framework. Then, in Section 5.3 I present the design of a new runtime technique for *lightweight identification of captured memory*. Section 5.4 explains the required modifications to Deuce STM to support the LICM technique. Finally, in Section 5.5 I present an experimental evaluation for a variety of benchmarks.

5.1 Deuce STM Overview

Deuce STM is an STM framework for the Java environment, provided as a bytecode instrumentation engine implemented with ASM [Binder *et al.* , 2007]. In the Deuce STM framework the synchronization mechanism is defined by a class that implements the `Context` interface. This interface specifies the event handlers API that each STM implementation must provide and that is used by the code instrumented by Deuce to notify the STM whenever one of the following events occurs during the execution of the instrumented program: the begin of an atomic method (`init` event handler); the end of an atomic method (either `commit` or `rollback` event handlers); or the access to a memory location made inside of a *transactional scope*—i.e., an atomic method or any method invoked in the scope of an atomic method—(`beforeReadAccess`, `onReadAccess`, and `onWriteAccess` event handlers).

On the other hand, the class `ContextDelegator` defines the STM barriers as a set of static methods that delegate the calls from the dynamically instrumented code to the event handlers of the class implementing the `Context` interface. To use memory barriers only when accessing memory from within a transactional scope, Deuce creates a duplicate of every method—a *transactional method*—where every memory access is

replaced by the invocation of the corresponding STM barrier. Depending on whether a method is invoked from inside or from outside a transactional scope, then either the transactional version or the original version will be invoked, respectively.

```
1  class Counter{
2
3    int n;
4    // Synthetic member
5    // generated by Deuce
6    static long n__ADDRESS__ = ...;
7
8    public int next(){
9      int current = n;
10     current++;
11     n = current;
12     return current;
13   }
14   public int next(Context c){
15     ContextDelegator.beforeReadAccess(this, n__ADDRESS__, c);
16     int v = ContextDelegator.onReadAccess(this, n, n__ADDRESS__, c);
17     v++;
18     ContextDelegator.onWriteAccess(this, v, n__ADDRESS__, c);
19     return v;
20   }
21
22 } // End of the class Counter
```

Listing 5.1: Resulting class `Counter` after the instrumentation by the Deuce engine.

In Listing 5.1 we show an example of the resulting class after the instrumentation of an hypothetical class `Counter` by the Deuce engine. For the method `next`, Deuce generated a new transactional version of this method that receives an additional `Context` parameter. Then, every memory access from inside this method, such as reading or writing to the field `n`, is replaced by the invocation of the corresponding STM barrier. Moreover, all calls to the `Counter.next()` method made within other transactional methods will be replaced with calls to this new transactional method `next(Context)`.

The invocation chain of transactional methods begins with a call to an atomic method (marked with `@Atomic`). In this case, the atomic method does not need an uninstrumented version and the body of the original method is replaced by a loop that tries to execute the transactional version of the atomic method within a transaction.

5.2 Runtime Capture Analysis

My proposal is based on the work of Dragojevic et al. [Dragojevic *et al.* , 2009], originally proposed for the Intel C++ STM compiler, which I adapted to the Deuce STM.

In Algorithm 1, I show the pseudo code for a read and a write barrier in Deuce STM when using runtime capture analysis. In both cases, the barrier first checks whether the object being accessed is captured by the current transaction. If so, it accesses data directly from memory; otherwise, it executes the standard full barrier. As in Deuce STM, object fields are updated in place using the `sun.misc.Unsafe` pseudo-standard internal library.

Algorithm 1 Read and write barriers when using runtime capture analysis.

▷ in the following, ref is an object, $addr$ is the address of the field accessed on ref, val is the value read/written, and ctx is the transaction's context

1: **function** $onReadAccess(ref, val, addr, ctx)$
2: **if** $isCaptured(ref, ctx)$ **then**
3: **return** val ▷ returns the field's value if the object ref is captured by ctx
4: **else**
5: **return** $ctx.onReadAccess(ref, val, addr)$ ▷ Uses the full STM barrier
6: **end if**
7: **end function**

8: **function** $onWriteAccess(ref, val, addr, ctx)$
9: **if** $isCaptured(ref, ctx)$ **then**
10: $Unsafe.putInt(ref, addr, val)$ ▷ Updates the field in-place.
11: **else**
12: $ctx.onWriteAccess(ref, val, addr)$ ▷ Uses the full STM barrier
13: **end if**
14: **end function**

The performance of this solution depends on the overhead of the capture analysis, which is made by the `isCaptured` function. So, if the potential savings from barrier elision outweighs the cost of runtime capture analysis, then the average cost of a barrier in an application will be reduced and the overall performance will be improved.

In the Dragojevic et al's original proposal the capture analysis algorithm was intertwined with the memory management process. The key idea of their algorithm was

to compare the address of the accessed object, `ref`, with the ranges of memory locations allocated by the transaction. To perform this analysis, all transactions must keep a *transaction-local allocation log* for all allocated memory.

So, the performance of the `isCaptured` function depends on the performance of the search algorithm that needs to lookup the allocation log for a specific address, which ultimately depends on the efficiency of the data structure used to implement the allocation log. In their work, they implemented and tested three different data structures: a search tree, an array, and a *filter* of memory ranges. The search tree allows insertions and removals of memory ranges and search operations to determine if an address belongs to a memory range stored in the tree. The array implementation of the log simply keeps all memory ranges allocated inside a transaction as an unsorted array. Finally, the *filtering* approach uses a hash table as a filter: When a block of memory gets allocated, all memory locations belonging to the block are hashed and the corresponding hash table entries are marked with the exact addresses of the corresponding memory locations; thus, this filtering scheme allows false negatives.

Dragojevic et al's evaluate the performance improvement of the capture analysis technique by measuring the execution time of all benchmarks synchronized with an unoptimized STM and comparing to the results of the same benchmark and STM with the capture analysis support. Their experimental results show similar performance improvements for the three data structures,[1] peaking at 18% for 16 threads and the Vacation benchmark in a low-contention configuration.

On a managed runtime environment with automatic memory management we do not have readily access to the memory allocation process, so that we can log which memory blocks are allocated by a transaction and, therefore, we cannot implement the capture analysis algorithm based on the search tree or the array data structures. Thus, I adapted the hash table filtering algorithm, replacing it with an `IdentityHashMap` of the JDK and I logged the references of the objects instantiated by a transaction. In my case, and contrary to the original approach, this implementation does not allow false negatives, which increases the reliability of the capture analysis, but incurs in further overhead to maintain the transaction-local allocation log. Nevertheless, using my implementation with the TL2 STM, I get a performance improvement similar to what was shown in [Dragojevic *et al.* , 2009]: For a low-contention configuration of the Vacation benchmark, we achieve a performance improvement of 34% at 16 threads (see Figure 5.2 on page 64).

[1] With the hash table performing slightly worse, 5% in the worst case, than the alternatives.

5.3 Lightweight Identification of Captured Memory

Although the implementation of the Dragojevic et al's filtering technique improves the overall performance of Deuce STM, the `isCaptured` algorithm is still much more expensive than a simple memory access: We have to calculate the `System.identityHashCode()` for the accessed object and then we have to lookup a hash table for that object.

In fact, even with this runtime capture analysis, Deuce STM still does not perform well in some of the most challenging benchmarks, such as the Vacation or the STMBench7, where transactions are more coarse-grained and, therefore, encompass more memory accesses. I claim that is, in part, due to the relative high cost of the `isCaptured` function, and that, if we can lower that cost, we may solve the problem.

In my work, I propose to make the runtime capture analysis algorithm faster by using the following approach: We label objects with unique identifiers of their creating transaction, and then check if the accessing transaction corresponds to that label, in which case we avoid the barriers. For this purpose, every transaction keeps a *fingerprint* that it uses to mark newly allocated objects, representing the objects' *owner* transaction. Thus, the `isCaptured` algorithm just needs to check if the owner of the accessed object corresponds to the transaction's fingerprint of the executing `Context`. In this case, it performs an identity comparison between the fingerprint of the accessing transaction and the owner of the accessed object, as shown in Algorithm 2.

Algorithm 2 The LICM algorithm of the `isCaptured` function.

1: **function** $isCaptured(ref, ctx)$
2: **return** $ref.owner = ctx.fingerprint$
3: **end function**

Every time a new top-level transaction begins, its context gets a new unique fingerprint. So, when a new object is published by the successful commit of its allocating transaction, any previously running or newly created transactions calling the `isCaptured` method for that object will return `false`, because their fingerprint cannot be the same as the fingerprint recorded on that object. At the end of the top-level transaction, we do not need to clear the context's fingerprint because a new fingerprint will be produced on the initialization of the next top-level transaction.

In Figure 5.1 I show an example of the fingerprints creation process and how it directly determines the result of the capture analysis. The STM barriers shown in Algorithm 1 are performed by the `Delegator`, which in turn redirects that invocation to

the `Context` object whenever the accessed object is not captured by the transaction—i.e. is not transaction local. In this example I show three different transactions sharing a `Counter` object that is instantiated by one of those transactions—transaction number 1. The bar bellow each thread has a number representing the `id` of each transaction. In this example, thread `A` performs transactions 1 and 3, while thread `B` performs transaction 2. In this case just transactions 2 and 3 perform full barriers, whereas transaction 1 returns and updates the `Counter` object in place—note that only transactions 2 and 3 reach the `Context` object when invoking the read and write barriers. Moreover, the context object of thread `A` has the same fingerprint of the `Counter` object only during the execution of transaction 1, avoiding in this case a full barrier. After the completion of transaction 1, no other transaction will have the same fingerprint of the `Counter` object and all subsequent transactional accesses to this object must perform a full barrier, as happens for transactions 2 and 3.

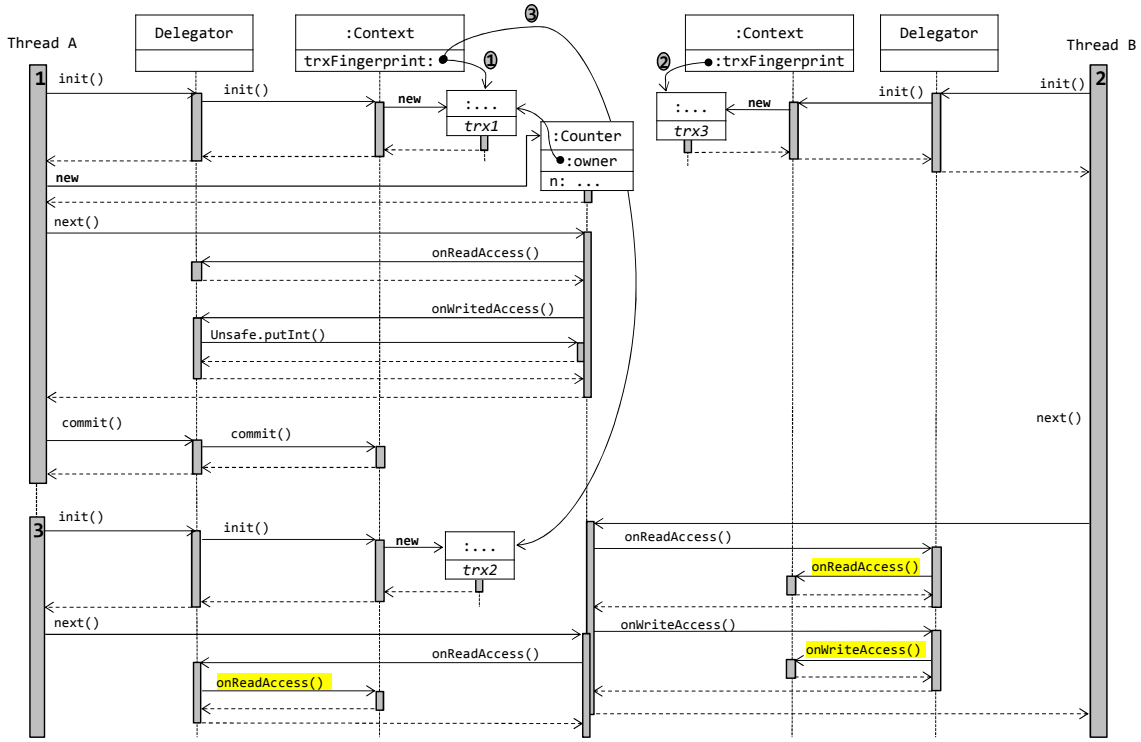


Figure 5.1: Three different transactions accessing a shared object `Counter` that is instantiated by transaction 1, which is the only one that avoids the execution of the full barriers when accessing that object. To clarify the diagrams I put two boxes representing the same class `Delegator` (although there is only one).

The generation of new fingerprints is a delicate process that must be carefully designed to avoid adding unintended overhead to either the Deuce STM engine or the underlying STM. A naive approach to identify each transaction uniquely is to use a

global counter, but this approach adds unwanted synchronization among threads that I would like to avoid. So, the main challenge of LICM implementation is to find an efficient process of generating fingerprints that avoids: (1) further synchronization, (2) the counter rollover and (3) minimize the additional memory overhead per transactional object. Considering these goals, I analysed three different solutions:

1. **global quiescent counter**—A simple scheme increments the counter on every transaction initialization, which would not scale. But there is no need to do that and instead, each thread could access the counter and increment it by a large number (e.g. hundreds of thousands if needed) to acquire a number of transaction identifiers, and then use these identifiers for newly transactions until it runs out of identifiers, at which point it would increment the counter by a large number again. In this case we should use a 64-bit counter because it is unlikely to overflow.

2. **combining a thread identifier with a per-thread sequence number**—This option avoids synchronization, but it requires some mechanism to deal with the wraparound of the numbers. So, again we should use a 64-bit counter.

3. **newly allocated instance of class `Object` as a fingerprint**—This solution avoids rollover and aliasing issues associated with counters. It has the advantage of relying on the garbage collector subsystem to provide uniqueness and the ability of recycling unused fingerprints. However, it imposes an additional memory management burden, because it instantiates an additional object per transaction. Note that in the scope of Deuce STM we cannot use the own transaction object (represented by an instance of the class `Context`) as the fingerprint because these instances are reused by different transactions.

The three solutions require an additional 64-bit metadata field per object to store the fingerprint (regarding the space of a memory address in a 64-bit architecture). So, I chose the third option, because it is the simplest to implement and solves both problems of avoiding synchronization and the counter rollover. Furthermore, I do not expect to see significant differences between the alternatives, given that the fingerprint is created when the transaction starts and corresponds to a very small cost of the entire transaction. According to the results presented in Figure 5.2 on the following page, the TL2 enhanced with the LICM technique (*tl2-licm*) outperforms the filtering approach (*tl2-filter*) and can improve the performance of the baseline STM by 60% at the peak of performance with 32 threads—more than three times of the performance improvement achieved with filtering—for a low-contention configuration of the Vacation benchmark.

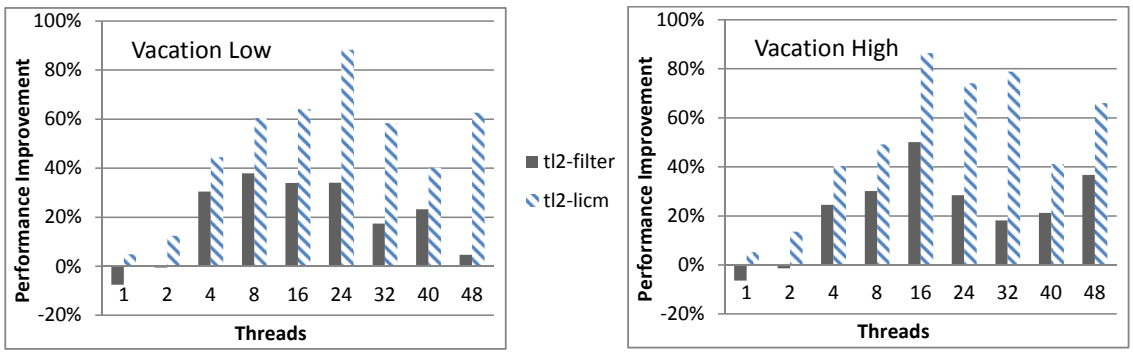


Figure 5.2: Performance improvement from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) in the Vacation benchmark, when using the TL2 STM.

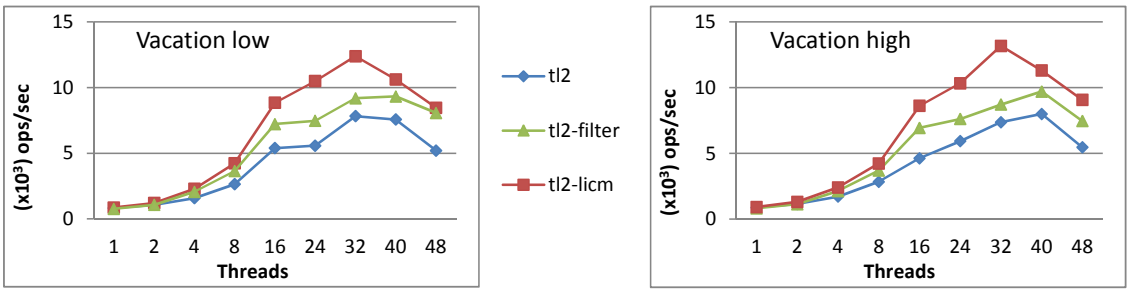


Figure 5.3: The throughput for two workloads (low-contention and high-contention) of the Vacation benchmark, when using the TL2 STM. I show results for the baseline STM (*tl2*), for the STM enhanced with the filtering implementation (*tl2-filter*), and for my LICM approach (*tl2-licm*).

In the results presented in Figure 5.3, we can observe that both optimization techniques: filtering and LICM, present a similar behavior and scalability; however, LICM performs always better than the filtering.

I also compared the performance between the LICM algorithm and the filtering approach in other two benchmarks: the STMBench7 and the JWormBench, for the TL2 STM. In Section 5.5 on page 71, I present a complete analysis of these benchmarks, where I discuss the effects of the LICM over three different STMs: TL2, LSA and JVSTM. For now, I just want to show that LICM is the most effective approach and the TL2 STM enhanced with LICM outperforms the filtering solution in different kinds of applications.

For the STMBench7, we can observe in the results of Figure 5.4 on the next page that we get a speedup between 2 and 14 times for *tl2-licm*, whereas *tl2-filter* get a speedup between 1.5 and 8 times—almost half of the best speedup achieved with LICM. Just for

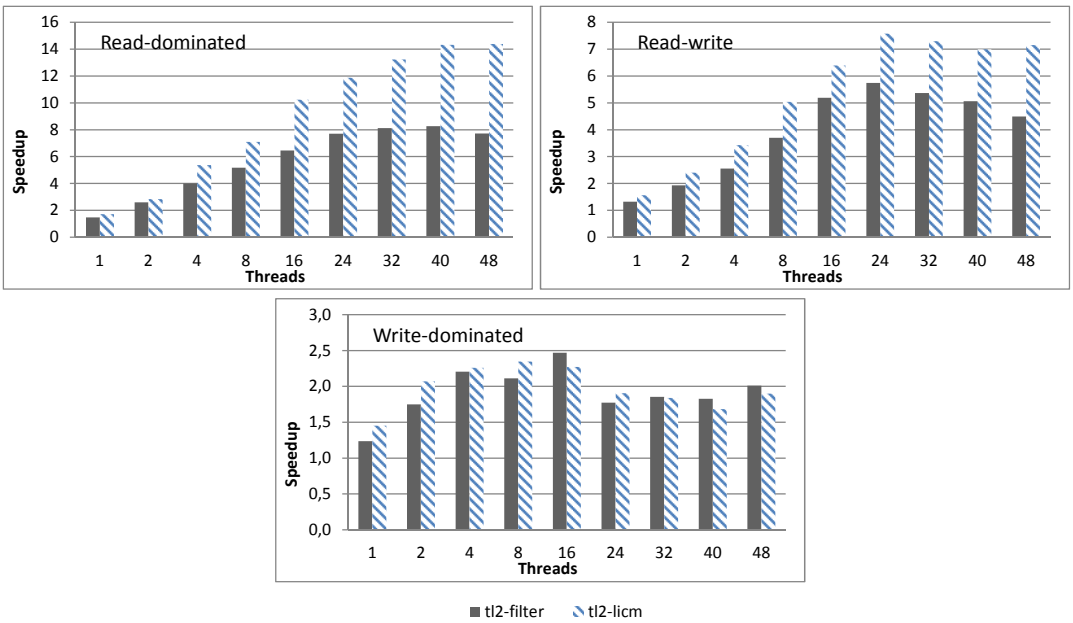


Figure 5.4: Speedup from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) in three workloads (read-dominated, read-write and write-dominated) of the STMBench7 benchmark, when using the TL2 STM.

the write-dominated workload we verify no difference between both approaches. In this case the workload is dominated by read-write transactions and the average length of the transaction local log is lower than in the other two cases—note that most of the transaction local objects are auxiliary objects to the iterators, which are less used by read-write operations. So, the lookup in the hash table is quite efficient because it is too small. Thus, there is no big difference between the two approaches: the LCIM and the filtering, in the write-dominated workload of the STMBench7

For the JWormBench, we observe in the results of Figure 5.6 that *tl2-licm* performs between 2 and 4 times faster than *tl2-filter*. Moreover, in the results of Figure 5.7 we observe that *tl2-licm* scales for an increasing number of threads in the *N-reads-1-write* workload, whereas *tl2-filter* just scales for a maximum of 16 threads. So, for three different benchmarks, we could confirm that LICM outperforms the filtering approach.

5.4 Extending Deuce STM

An object model of a managed runtime environment specifies a set of rules that dictates how to represent objects in memory. The LICM technique requires a specific object model distinct from the one provided by the managed environment, which allows to

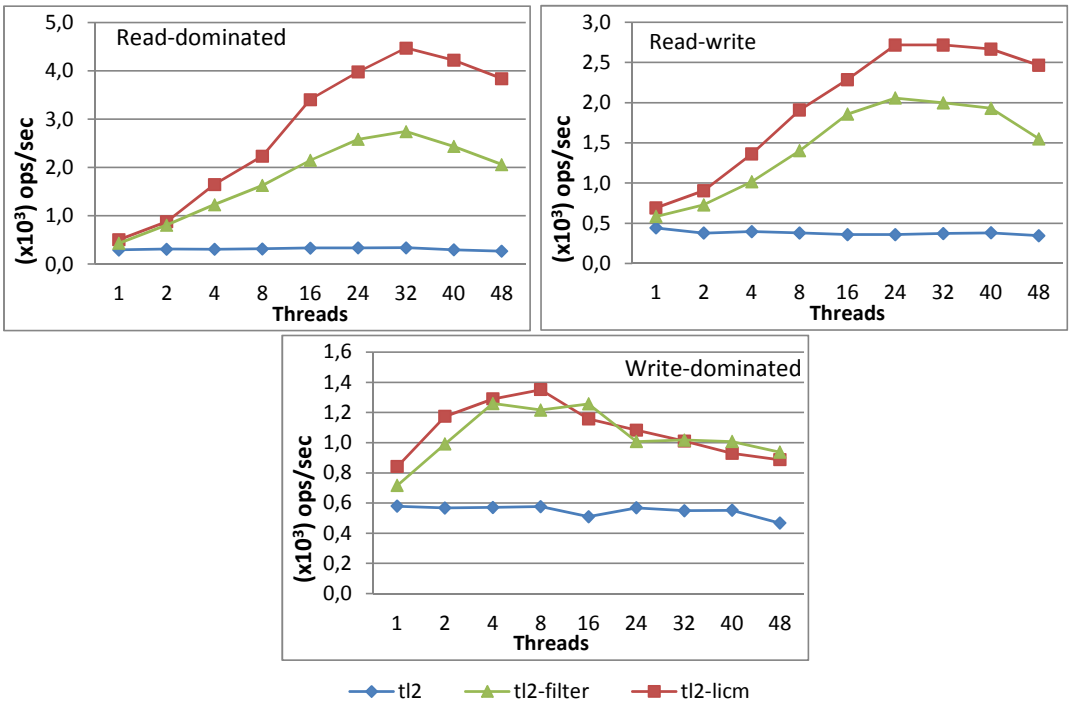


Figure 5.5: The throughput for three workloads (read-dominated, read-write and write-dominated) of the STMBench7 benchmark, when using the TL2 STM, with two different capture analysis techniques: LICM (*tl2-licm*) and filtering (*tl2-filter*)

identify the allocating transaction of an object. Moreover, it also needs to perform additional tasks beyond the standard behaviour provided by the STM barriers. Yet, the original Deuce STM just provides extensibility in terms of the specification of the STM algorithm, but it does not allow either the definition of additional behavior orthogonal to all STMs, or any enhancement to the Java object layout. So, I extended Deuce STM to support the previous requirements and I followed three major guidelines:[2] (1) to avoid changing the current Deuce STM API; (2) to guarantee backwards compatibility with existing applications and STMs for Deuce; and (3) to provide the ability to enhance any existing STM with the capture analysis technique without requiring either its recompilation or any modification to its source-code.

Extending Deuce STM with the capture analysis technique requires two main changes to the Deuce STM core structures: (1) the `Context` implementation of any STM must keep a fingerprint representing the identity of the transaction in execution and must perform the capture analysis shown in Algorithm 1; and (2) a *transactional class* (i.e., a class whose instances are accessed in a transactional scope) must have an additional field, `owner`, to store the fingerprint of the transaction that instantiates it.

[2]This adaptation of Deuce is available at https://github.com/inesc-id-esw/deucestm/

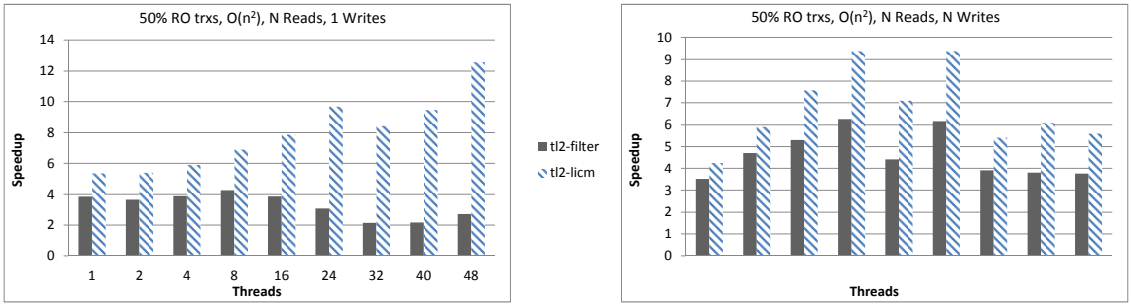


Figure 5.6: Speedup from capture analysis filtering technique (*tl2-filter*) and LICM technique (*tl2-licm*) for two workloads (*N-reads-1-write* with smaller write-sets and *N-reads-N-writes* with larger write-sets) of the JWormBench benchmark.

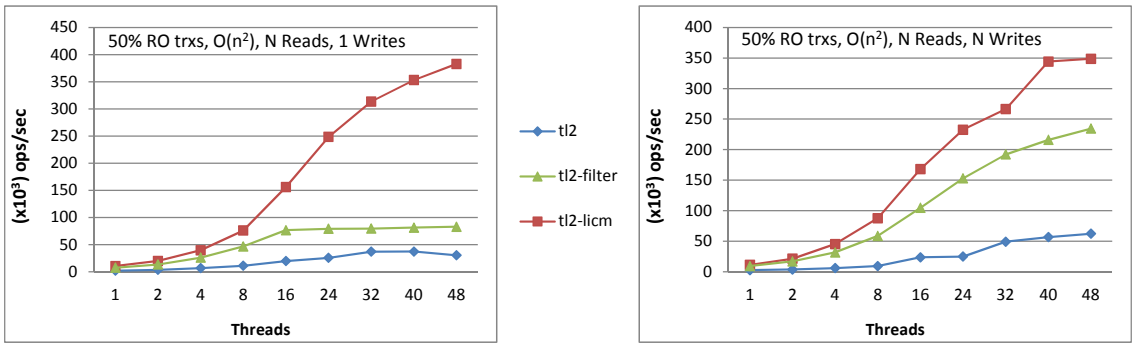


Figure 5.7: The throughput for two workloads (*N-reads-1-write* with smaller write-sets and *N-reads-N-writes* with larger write-sets) of the JWormBench benchmark, when using the TL2 STM, with two different capture analysis techniques: LICM (*tl2-licm*) and filtering (*tl2-filter*)

In this section I describe how I introduced the *filtering* mechanism in Deuce STM to perform additional tasks, such as the capture analysis, without changing the current Deuce API. After that, I address the new infrastructure of enhancement transformations that allows the addition of STM metadata in-place within the transactional objects.

5.4.1 Filtering

Keeping the established guidelines in mind, I added the filtering support through the specification of a *filter context*—that is, a class that implements the `Context` interface and adds some functionality to any existing STM specified by another `Context` (using the decorator design pattern [Gamma *et al.* , 1995]). The new class `ContextFilterCapturedState` uses this approach, so that it can be applied to an existing `Context` of any STM.

The filter is provided to the Deuce STM through the system property `org.deuce.filter`, following the same parametrization approach used to identify an STM.

According to the implementation of the LICM approach, a `Context` object must keep the *fingerprint* of its running transaction. To that end, we need to initialize and store a new fingerprint in the `Context` object of the executing thread, whenever it is notified of the beginning of a new transaction. Besides that, given that Deuce uses flat nesting, meaning that there is no concurrency among nested transactions within a nesting tree, the fingerprint of a top-level transaction can be shared across its nested transactions. In Listing 5.2 I show the code of the class `ContextFilterCapturedState`—a `Context` implementation that controls the initialization of new fingerprints (for simplification I omit the method `rollback` that performs a similar code to the method `commit`).

```
1  public class ContextFilterCapturedState implements Context {
2    protected final Context ctx;
3    protected Object trxFingerprint = null;
4    protected int nestedLevel = 0;
5
6    public ContextFilterCapturedState (Context ctx) {
7      this.ctx = ctx;
8    }
9    @Override
10   public void init(int atomicBlockId, String metainf) {
11     nestedLevel++;
12     if (nestedLevel == 1) trxFingerprint = new Object();
13     ctx.init(atomicBlockId, metainf);
14   }
15   @Override
16   public boolean commit(){
17     nestedLevel--;
18     return ctx.commit();
19   }
20   ...
21 }
```

Listing 5.2: The `ContextFilterCapturedState` class is a context decorator that adds a transaction fingerprint to any existing STM `Context` implementation.

To perform the *runtime capture analysis*, all the STM barriers must check whether the object being accessed is *captured* by the current transaction (i.e. the object is transaction local). This verifications is performed by the function in Listing 5.3 that must be invoked by all STM barriers.

```
1 public boolean isCaptured(Object ref, Context ctx){
2    return (((CapturedState)ref).owner ==
3              ((ContextFilterCapturedState)ctx).trxFingerprint);
4  }
```

Listing 5.3: The LICM algorithm performed by the `isCaptured` function.

According to Deuce STM architecture, for each primitive type there are two STM barriers defined in the class `ContextDelegator`: one to access a field from an object and another to access an element from an array. For example, for the `int` type there is an STM barrier to access an `int` field from an object and another STM barrier to access an element from an `int` array, as shown in the code of Listing 5.4. Then both STM barriers redirect the invocation to the same event handler in the `Context` implementation. Note in the code of Listing 5.4 that both barriers call the same handler of the `Context` interface, which has the following signature: `onReadAccess(Object ref, int val, long addr)`.

```
1  public class ContextDelegator{
2   static int onReadAccess(Object ref, int val, long addr,
3     Context ctx)
4   {
5     return ctx.onReadAccess(ref, val, addr);
6   }
7   static int onArrayReadAccess(int[] arr, int index, Context ctx){
8     int address = INT_ARR_BASE + INT_ARR_SCALE*index;
9     ctx.beforeReadAccess(arr, address);
10    return ctx.onReadAccess(arr, arr[index], address);
11  }
12  ...
13 }
```

Listing 5.4: Default implementation of an STM barrier for the `int` primitive type in regular objects and arrays.

Yet, when we replace the root of the transactional classes hierarchy from `Object` to `CapturedState` then the STM barriers for arrays are no longer compatible with the new arrays' definition. Note, however, that the same problem does not happen for regular objects because `CapturedState` is still compatible with `Object`.

For this reason we need to replace the default implementation of the `ContextDelegator` by a new `delegator` with new STM barriers for the new array types that are wrapped in a `CapturedState` object. So, although I could include the `isCaptured`

check in all event handlers of the `ContextFilterCapturedState`, I put this verification in the STM barriers defined in the new *delegator*.

Deuce instrumentation engine refers to the *delegator* class by its internal name that is stored in a global constant: `CONTEXT_DELEGATOR_INTERNAL`. So, if we replace this constant with the name of the class that implements the new *delegator*, then all instrumented memory accesses are redirected to the STM barriers defined in this new class. To that end, I added a new parameter to Deuce runtime, which is responsible for specifying the name of the class that will replace the default `ContextDelegator`.

I defined the new class `ContextDelegatorCapturedState` whose methods perform the capture analysis check, as shown in the code of Listing 5.5. For simplification I have omitted the code of the write barrier that performs a similar verification of the read barrier.

```
1  public class ContextDelegatorCapturedState extends ContextDelegator
2  {
3   static int onReadAccess(Object ref, int val, long addr,
4     Context ctx)
5   {
6     if (isCaptured(ref, ctx)) return val;
7     else return ctx.onReadAccess(ref, val, addr);
8   }
9   static int onArrayReadAccess(CapturedStateIntArray ref, int idx,
10    Context ctx)
11 {
12    if (isCaptured(ref, ctx))
13      return ref.elements[idx];
14    else
15      return ContextDelegator.
16        onArrayReadAccess(ref.elements, idx, ctx);
17  }
18  ...
19 }
```

Listing 5.5: The code skeleton of two read barriers for both objects and arrays, using runtime capture analysis.

5.4.2 Storing metadata in-place

To store additional metadata in-place with the transactional object we need to provide an additional transformation in the Deuce STM that includes the extra required fields

for all transactional objects. The solution I used to achieve this goal was to replace the root of the transactional classes hierarchy from `Object` to the desired metadata class: i.e. the `CapturedState` class shown in the code of Listing 5.6. This transformation is instrumented by Deuce through a specific *enhancer* processed by the new infrastructure of enhancements transformations that I added to Deuce STM engine (for more information about this new infrastructure and its design, see Appendix C).

```
1 public class CapturedState {
2   private final Object owner;
3   public CapturedState() {
4     this.owner = null;
5   }
6   public CapturedState(Context ctx) {
7     this.owner = ((ContextFilterCapturedState) ctx).trxFingerprint;
8   }
9 }
```

Listing 5.6: `CapturedState` class adds an extra field owner to all transactional classes.

Depending on whether a transactional class is instantiated outside or inside a transactional scope, the constructor invoked will be either the parameterless constructor or the constructor with a `Context` parameter, respectively. If an object is instantiated out of a transactional scope then its owner field will be `null`. Otherwise, it will point to the fingerprint of the executing `Context` (I assume that whenever the class `CapturedState` is used, all contexts will be instances of the class `ContextFilterCapturedState`).

5.5 Validation

Even though the main goal of the LICM technique is to improve the STM performance, it also has a beneficial effect on the size of the application footprint. Although LICM adds an overhead of an allocated fingerprint per transaction and a single word per object, which increases the size of the live objects, it also contributes to the reduction of the size of an application footprint due to the elimination of unnecessary metadata for transaction local objects.

In this section, first I show a performance evaluation of the LICM for a diversity of benchmarks and then, I also show the results of an application's memory consumption.

All the tests were performed on a machine with 4 AMD Opteron(tm) 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the 1.6.0_33-b03, running on Ubuntu with Linux kernel version 2.6.32.

5.5.1 Performance evaluation

To evaluate the performance of my approach, I used the STMBench7, the STAMP, and the JWormBench benchmarks, with the LSA [Riegel *et al.* , 2006], the TL2 [Dice *et al.* , 2006], and the JVSTM [Fernandes & Cachopo, 2011] STMs, all implemented in the Deuce STM framework. In all tests I show the results for both the baseline STM and for the STM with LICM support (identified by the suffix *-licm*).

Moreover, given that the STMBench7 and the JWormBench benchmarks also have a medium/fine-grained locking synchronization strategy, I also compare the performance of the lock-based approach with the STM-based approach, showing that for certain STMs, using LICM makes the performance of the STM-based approach close to (or better than) the performance of the lock-based approach. In particular, for the STMBench7 and a low number of threads, JVSTM outperforms the medium-lock approach.

STAMP benchmarks

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. I tested four STAMP benchmarks: K-Means, Ssca2, Intruder, and Vacation.[3] I ran these benchmarks with the configurations proposed in [Cao Minh *et al.* , 2008] and presented in Table 5.1.

Benchmark	Parameters
Vacation Low-contention	`-n 256 -q 90 -u 98 -r 262144 -t 65536`
Vacation High-contention	`-n 256 -q 90 -u 60 -r 262144 -t 65536`
Intruder	`-a 10 -l 128 -n 65536 -s 1`
KMeans	`-m 15 -n 15 -t 0.00001 -i random-n65536-d32-c16.txt`
Ssca2	`-s 13 -i 1.0 -u 1.0 -l 13 -p 3`

Table 5.1: Configuration parameters used for each STAMP benchmark.

Table 5.2 shows the speedup of each STM with LICM support for 1 thread and for N threads, where N is the number of threads that reach the peak of performance.

[3]The original implementation of STAMP is available as a C library and these four benchmarks are the only ones available for Java in the public repository of Deuce that are running with correct results.

1 thread	Vacation Low-contention	Vacation High-contention	Intruder	KMeans	Ssca2
LSA	**1.2**	**1.2**	**1.4**	0.9	1.0
TL2	**1.1**	**1.1**	**1.2**	0.9	0.9
JVSTM	**1.1**	**1.1**	**1.2**	1.0	1.0
N threads					
LSA	**7.0** (40/8)	**6.0** (40/12)	**1.7** (16/8)	1.0 (32/32)	0.9 (8/8)
TL2	**1.6** (32/32)	**1.6** (32/40)	**1.3** (16/16)	1.0 (12/24)	1.0 (8/8)
JVSTM	**1.1** (8/8)	1.0 (40/40)	**1.1** (8/8)	1.0 (4/4)	1.0 (32/32)

Table 5.2: The speedup of each STM with LICM support for 1 thread and N threads. In the latter case I also show, between parentheses, the number of threads that reach the peak of performance, with and without the LICM support, respectively. I emphasise in bold the speedup values that are higher than 1.0.

Note that a speedup higher than 1 means that the performance improved with LICM, whereas a speedup lower than 1 means that performance decreased with LICM. The results in Table 5.2 show that LICM improves the performance of the baseline STMs for the majority of the evaluated benchmarks and that, when it has no benefits (due to the lack of opportunities for elision of barriers), the imposed overhead is very low.

The speedup we observed in Intruder and Vacation is consistent with the results of [Dragojevic *et al.* , 2009], which provides evidence for some opportunities of elision of transaction-local barriers. From my analysis, Intruder instantiates an auxiliary linked list and a `byte[]`, whose barriers can be elided with my capture analysis technique. On the other hand, Vacation performs three different kinds of operations, each one including an *initialization* phase and an *execution* phase. In the initialization phase it instantiates several arrays with the arguments that should be parametrized in the operations performed by each transaction. These auxiliary arrays are transaction local and their access barriers can be suppressed through capture analysis.

LSA shows better speedup than TL2 and JVSTM, due to scalability problems verified in the LSA when executed without the LICM—in this case we registered a high rate of aborts due to the eager ownership acquisition approach followed by LSA. For the JVSTM we do not observe the same improvement in performance because, although LICM helps to elide useless barriers for transaction local objects, they still incur in additional metadata that penalizes the corresponding memory accesses (in the case of the TL2 and the LSA, there is no in-place metadata associated with the transactional objects).

According to [Dragojevic *et al.* , 2009], neither K-Means nor Ssca2 access transaction local memory and, thus, in these cases there are no opportunities for eliding barriers with capture analysis. My results are consistent with this, but still show that my technique for capture analysis has almost no overhead in performance and that it degrades performance in 10%, in the worst case.

STMBench7 benchmark

Several of the STMBench7's operations traverse a complex graph of objects by using iterators over the collections that represent the connections in that graph. Typically, these iterators are transaction local and, thus, accessing them using STM barriers adds unnecessary overhead to the STMBench7's operations.

LSA and JVSTM perform better than TL2 on STMBench7, because of their versioning approach, which allows read-only transactions to get a valid snapshot of memory and thus, they always commit successfully. This effect is amplified by the fact that only with LICM can we have read-only transactions. Without LICM, most of the read-only transactions are forced to be executed as read-write transactions because they need to use write barriers when using iterators, which perform modifications to their own state. This problem is aggravated by the eager ownership acquisition approach followed by LSA, which acquires a lock for every written location. All this together contributes to a very high rate of aborts that drastically reduces the performance of the LSA for more than 4 threads. Once the useless barriers are elided with capture analysis, the LSA scales for an increasing number of threads, getting an improvement of up to 12-fold in performance as depicted in the results of Figure 5.8.

In the results of Figure 5.8 I omitted TL2, which is the STM with the worst performance. Even though LSA-licm performs better, its results are still far from the results obtained with JVSTM-licm, which is the most performant STM in the STMBench7. In fact, JVSTM-licm gets better results than the medium-lock synchronization approach for a number of threads lower than 24. In this case, JVSTM benefits from its lock-free commit algorithm and from the lazy ownership acquisition approach, in contrast to the eager approach of the LSA.

JWormBench benchmark

In previous chapter I used the JWormBench benchmark to explore the effects on performance of relaxing the transparency of an STM. To that end, I extended the Deuce API

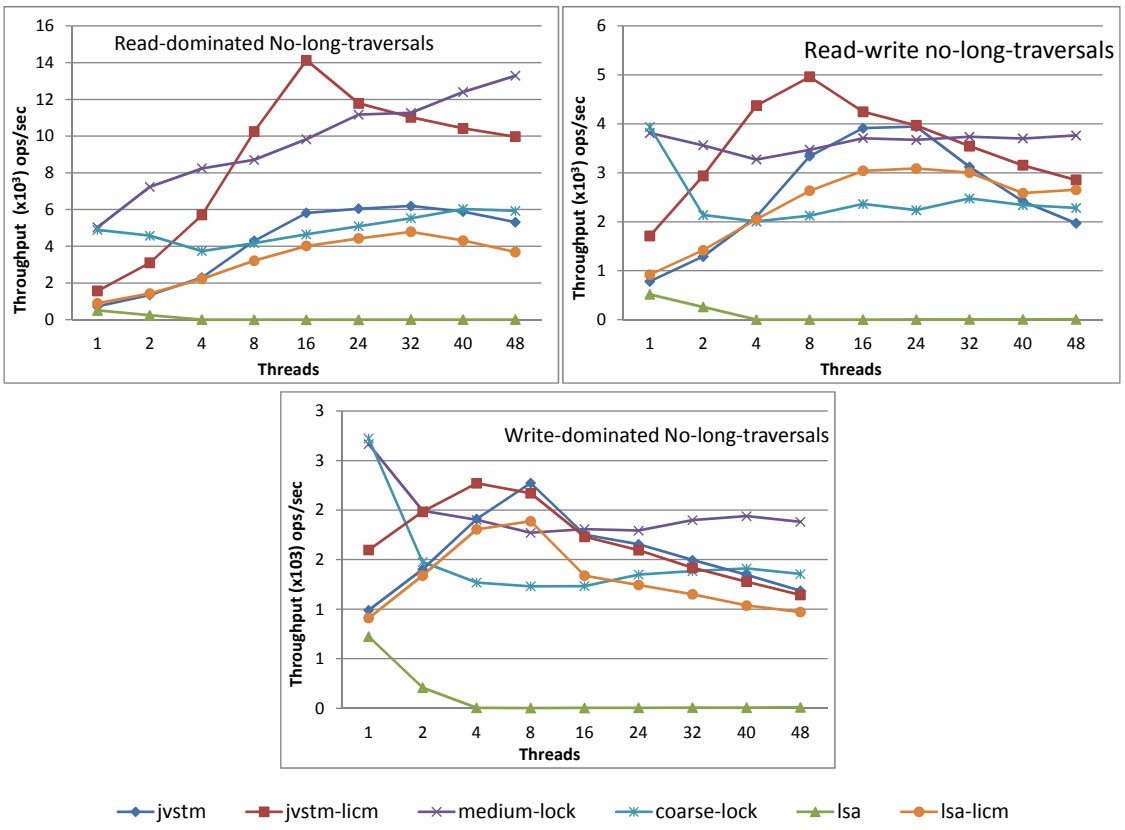


Figure 5.8: The STMBench7 throughput for LSA and JVSTM, in the three available workloads, without long traversal operations. For readability reasons I omitted TL2, which is the worst of the STMs.

with a couple of annotations that allow programmers to specify that certain objects or arrays should not be transactified. Using this approach, we got an improvement of up to 22-fold in the performance. Now, with my new LICM technique, I got similar results but without having to change the original Deuce API.

I ran these benchmarks with the same configuration used in Section 4.4 on page 50: a *world* with 1024 *nodes* and 48 *worms* with a head's size varying between 2 and 16 nodes, corresponding to between 4 and 256 read/write transactional accesses.

There are two major sources of unnecessary STM barriers in the JWormBench: (1) a global immutable matrix containing the world nodes (which cannot be expressed as immutable in Java), and (2) the auxiliary arrays to the worm operations. The first barriers can be suppressed by excluding the class `World` from the instrumentation of Deuce. On the other hand, the second barriers will be automatically elided through my LICM technique.

Figure 5.9 shows the results obtained for the JWormBench benchmark. TL2 and LSA present the same performance in both workloads of the JWormBench and, so,

I show the results for LSA only. Unlike what happened for the STMBench7, LSA with capture analysis is always better than JVSTM in the JWormBench, because these workloads have transactions with a smaller average length and with a lower level of contention. But, most importantly, we can see that both STMs get results close to the results obtained with the fine-grained locking approach, whereas without LICM they were an order of magnitude slower. This is true for the first workload, but when the number of write operations increases too much, as in the case of the $O(n^2)$, $NReads$,-$NWrites$ workload, the performance of JVSTM degrades for a higher number of threads, due to the big overhead of its read-write transactions.

The major overhead of the JWormBench comes from the mathematical operations performed by each worm. When these operations perform useless STM barriers they add a significant overhead to the transactions. In fact, and according to the observations presented in Section 4.2 on page 45, both workloads spend almost 50% of the execution time accessing transactional local arrays through unnecessary STM barriers. Furthermore, this situation increases too much the average length of the transactions and, therefore, increases the rate of aborted transactions. In those circumstances all STMs incur in huge overheads and substantially decrease the overall throughput.

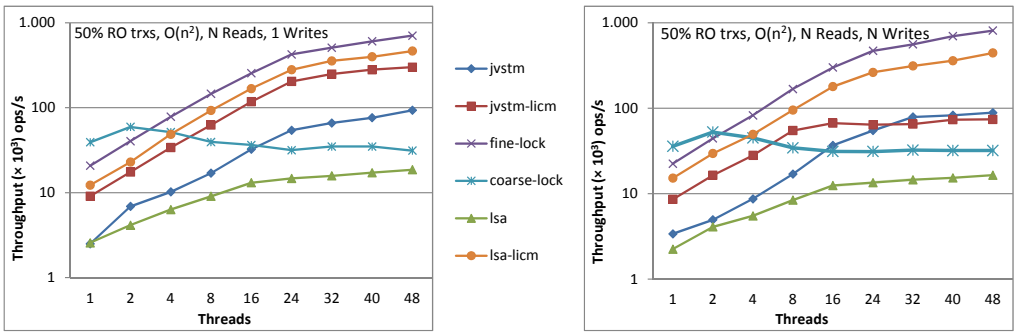


Figure 5.9: The JWormBench throughput for LSA, JVSTM, and locks, for two different wokloads. Note that the vertical axis use a logarithmic scale.

5.5.2 Memory Consumption Evaluation

In terms of memory consumption the LICM has small impact in the behavior of the JVSTM because this STM requires a specific object model, which stores the STM metadata in place within the memory locations. So, even non-shared objects require additional STM metadata to store their values.

So, I chose a non-multi-versioning STM to evaluate the effects of LICM on the application's memory consumption. To that end, I chose the TL2, instead of the LSA, because the latter has shown a performance bottleneck when performed without LICM.

I used the STMBench7 and the Vacation in my experimental analysis, because those benchmarks use large data sets and, therefore, I expect that it better shows impact of the LICM in the memory heap size. In both cases, I just analyze the results of one workload because the behavior of memory consumption resulting from the LICM action is the same for all workloads of the same benchmark.

I ran my tests with just one worker thread for approximately thirty minutes and I collected the results from the verbose garbage collector (GC) output with the default configuration. In this case, and for a server-class machine such as the one used in my tests, the JVM selects the parallel collector by default, which uses an adaptive heap size policy to dynamically adjust the size of the heap. Thus, the behaviour of the GC is based on the values of the following parameters: (1) a maximum garbage collection pause time (by default there is no maximum pause time); (2) a desired throughput for an application, specified as a ratio of the total program execution time spent in garbage collection (the default value is 99, resulting in 1% of the time in garbage collection). So, if the throughput goal is not being met, the size of the heap is increased. Growing and shrinking the heap is done at different rates. By default a generation grows in increments of 20% and shrinks in steps of 5%. So, it is normal that we observe the heap size to grow quickly and to shrink slowly along several collections.

In all results I show the evolution of three parameters during the execution of each benchmark: (1) the combined size of live objects *before GC*, (2) the size of live objects *after GC*, and (3) the total *available* space, corresponding to the maximum heap size.

STMBench7 benchmark

In the results of Figure 5.10, we can observe that there is a small overhead in memory space for the live objects after GC, due to the transaction fingerprint that is kept for all live objects. Yet, I can confirm my expectations that this overhead is very low and there is no significant differences between the size of live objects after GC with, or without LICM.

On the other hand, we can verify a reduction in the total available space of approximately 400 Mb with LICM. Taking advantage of LICM, the transactional local objects avoid STM barriers and also the additional metadata required by the TL2. Hence, the size of the transactions' read-set and write-set are also smaller and contribute to the reduction of the heap size. Thus, we observe in the results of Figure 5.10 that the maximum size of live objects before GC is almost 500 Mb lower than without LICM.

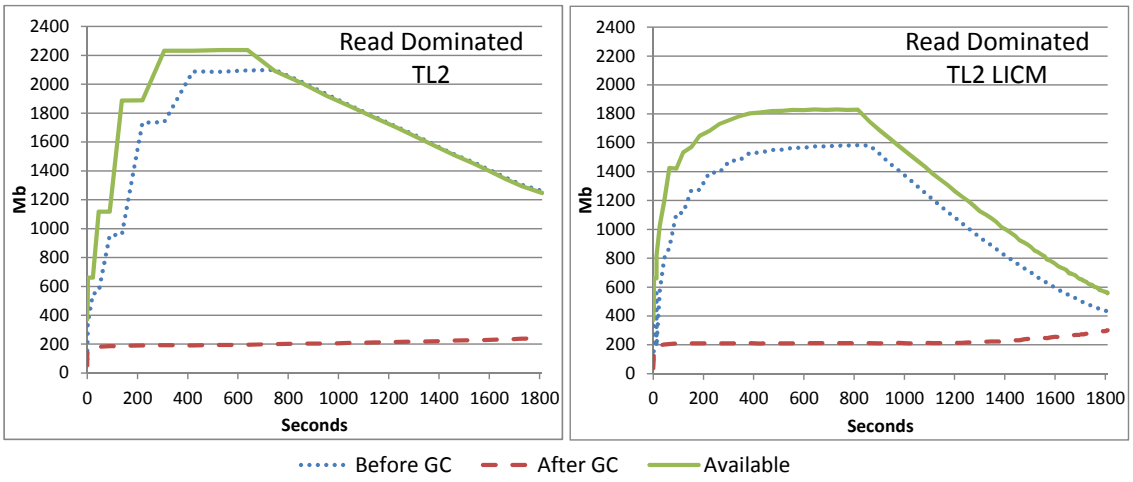


Figure 5.10: The STMBench7 memory consumption for TL2, with and without LICM, in the read dominated workload, without long traversal operations.

Vacation

In the results of Figure 5.11, we can observe the overhead resulting from the transaction fingerprint that is added for all transactional objects by LICM. In this case, the LICM requires almost 100 Mb more for all live objects after GC than without LICM.

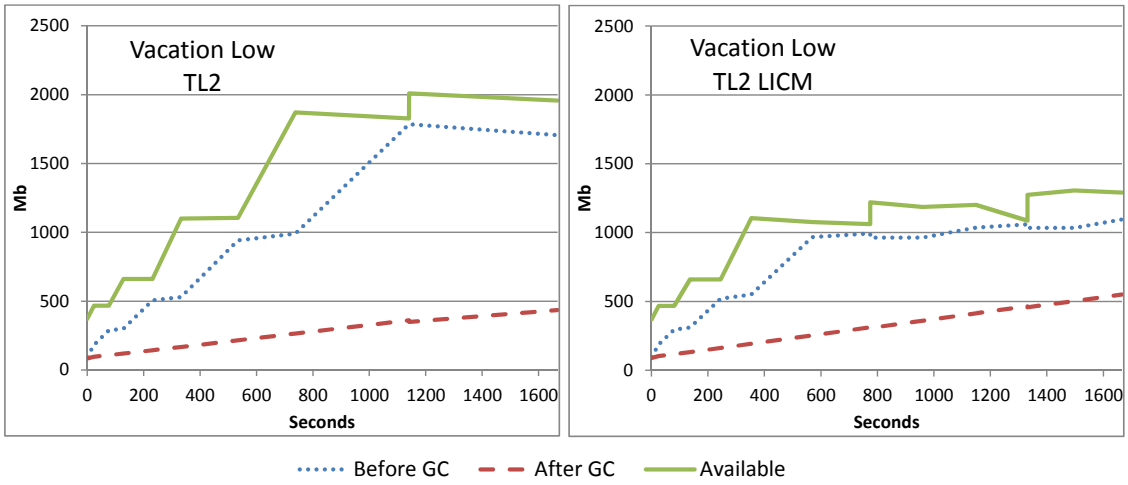


Figure 5.11: The Vacation memory consumption for TL2, with and without LICM, in the low contention workload.

Nevertheless, and as happens for the STMBench7, the LICM is able to reduce the Vacation footprint by as much 700 Mb.

5.6 Summary

STMs are often criticized for introducing unacceptable overhead when compared with either the sequential version or a lock-based version of any realistic benchmark. My experience in testing STMs with several realistic benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among transactions.

Several techniques have been proposed to elide useless STM barriers in programs automatically instrumented by STM compilers. From my analysis, the main contributions in this field follow three distinct approaches: (1) runtime capture analysis; (2) compiler static analysis to elide redundant operations; and (3) decomposition of the STM APIs to allow programmers to convey the knowledge about which blocks of instructions or memory locations should not be instrumented. The latter approach is more efficient and has shown bigger improvements in the performance of the STMs, but has the inconvenient of reducing the transparency of the STMs APIs. Yet, to the extent of my knowledge, none of the previous solutions demonstrated performance improvements with the same magnitude of the results that I present here for the STMBench7 and Vacation benchmarks.

Moreover, even though in this work I did not address the problem of removing unnecessary barriers for accesses to thread local objects, my approach can be easily adapted for this case, also: Rather than having a per-transaction fingerprint, we need a per-thread fingerprint instead.

Chapter 6

Adaptive Object Metadata

In Chapter 5, I described a new runtime optimization technique (LICM) that can identify transaction local objects, corresponding to non-shared memory, and for which STM barriers can be elided when accessing those objects. My experimental results show a significant performance improvement in benchmarks with transaction local data, when we use an STM enhanced with LICM.

Yet, the STM-induced overheads are not only in the unnecessary use of STM barriers for non-shared objects, but also in the overheads of accessing shared objects that are not under contention—*frequently non-contend*. In these cases, we cannot avoid a transactional definition for those classes, because their instances correspond to shared objects that could be under contention, however, we will have an additional overhead every time we access those objects and they are non-contended. For instance, in the JVSTM we incur in the overheads of the additional STM metadata that is added for all transactional locations.

The *multi-versioning* approach, which is used by the JVSTM, has the benefit that read-only transactions never conflict with other transactions, but the drawback of typically requiring both more memory to store the multiple versions and extra indirections to access the value of each transactional location.

Yet, I claim that it is possible to substantially reduce the induced overheads of a *multi-versioning* STM if we assume that the amount of memory under contention—that is, memory being concurrently accessed both for read and for write—is only a small fraction of the total amount of memory accessed by that program.

The key insight that allows eliminating these overheads is that the STM metadata is needed only when several transactions contend for the same transactional object and at

least one of those transactions writes to that object. Thus, assuming that in real-sized applications the vast majority of objects are seldom written, the number of objects that need the metadata should be much smaller than the total number of objects in the application. This, in turn, means that if we use a compact representation for the non-contended objects, we may have a significant reduction both in the memory used and in the performance overhead.

My proposal is based on the idea of transactional objects having *adaptive object metadata* (AOM). AOM is an object-based design that follows the JVSTM general design, but it is adaptive because the metadata used for each transactional object changes over time, depending on how the object is accessed. In its most compact form the metadata adds an overhead of a single word per object, but when an object is changed and several versions of it may be needed, the object is extended with extra metadata to represent the various versions of the object. This extension is not permanent, however, and the object may revert back to its most compact form. With this AOM approach I expect to be able to reduce not only the memory overheads imposed by the STM, but also its performance overheads.

In the next Section, I give an overview of the key aspects of the JVSTM that are relevant to understand the AOM. After that, in Section 6.2, I describe in more detail the new AOM design and its memory model. In Section 6.3, I discuss two different implementation approaches for the AOM. Then, in Section 6.4, I present an experimental evaluation for a variety of benchmarks. Finally, in Section 6.5, I present my concluding remarks.

6.1 JVSTM Overview

The JVSTM [Fernandes & Cachopo, 2011] is a Java library that implements a lock-free, *multi-versioning* STM. It uses the core concept of *versioned boxes (vboxes)*, which can be seen as a replacement for transactional memory locations. Instead of keeping only a single value, a *vbox* (instance of the `VBox` class) maintains a sequence of values—the vbox's *history*—where each value is tagged with a *version* corresponding to the number of the transaction that has committed that value, as shown in Figure 6.1. Each entry in the history of a vbox is a *vbox body* (instance of the `VBoxBody` class), and the entries are sorted by their version in descending order.

During a transaction, writes to a vbox are logged into a per-transaction *write-set* and do not affect the history of the vbox until commit time. Similarly, reads of vboxes are logged into a per-transaction *read-set*, which will be used for validating the transaction.

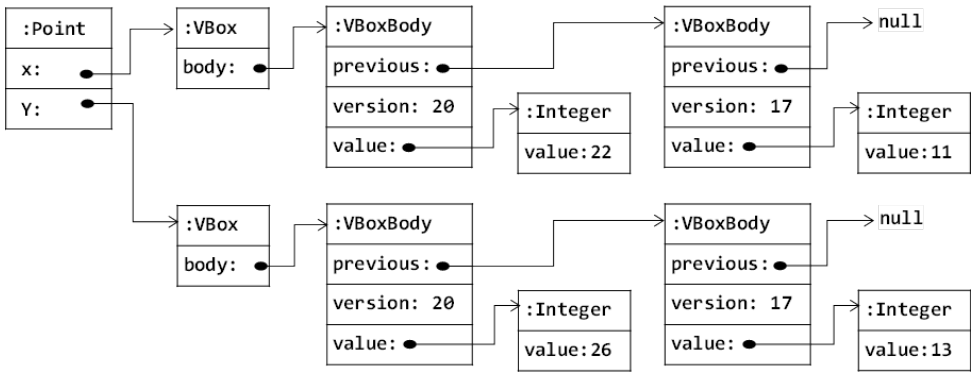


Figure 6.1: Structure that represents a transactional object, instance of a class `Point`, using one *vbox* for each of its fields. In this case, both fields were changed by transactions 17 and 20.

At commit time, read-write transactions are validated[1] by checking that all of the vboxes in their read-set are still up-to-date—that is, that none of the vboxes read by the transaction have been changed in the meanwhile by another concurrent transaction that successfully committed. Otherwise, we say that there was a *conflict* and the transaction cannot commit successfully; instead, it needs to *restart* in the new version of the program's state.

If a write transaction is valid, it will try to enqueue into a global queue of transactions that want to commit; these transactions are represented in the queue by instances of the class `ActiveTransactionRecord` (ATR). By successfully getting into that queue, a transaction obtains a global order for commit and is guaranteed to commit in that order, possibly with the help of other transactions waiting for their turn to commit: This results in a *lock-free* commit algorithm.

During the commit of a (successfully enqueued) transaction, there is the *write-back* phase, which is when the new values of the vboxes changed during the transaction are added to each vbox's history, as shown in Listing 6.1. As a result of the helping during the commit, more than one thread may attempt to write-back to the same vbox. Thus, the `commit` method of class `VBox`, which implements the write-back of a vbox, uses a *compare-and-swap* (CAS) operation to install the new `VBoxBody` at the head of the list of bodies. If the CAS fails, then another thread must have successfully completed the write-back for this vbox. In either case, the write-back of a vbox returns the vbox body that was successfully added to the vbox's history. All these bodies are collected and stored in the ATR corresponding to the committing transaction, and, later, used to allow the garbage collection of old history entries.

[1]In the JVSTM, read-only and write-only transactions do not need to validate as they may always commit successfully.

```
1  VBoxBody commit(Object newValue, int txNumber) {
2    VBoxBody currHead = this.body;
3    if (currHead == null || currHead.version < txNumber) {
4      VBoxBody newBody = new VBoxBody(newValue, txNumber, currHead);
5      return CASbody(currHead, newBody);
6    } else {
7      return currHead.getBody(txNumber);
8    }
9  }

11 VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
12   if (compareAndSwapObject(this, bodyOffset, expected, newBody)) {
13     return newBody;
14   } else { // if the CAS failed the new body must already be there!
15     return this.body.getBody(newBody.version);
16   }
17 }
```

Listing 6.1: Algorithm used by the JVSTM to write-back to a vbox. The `commit` method receives the new value and the transaction number corresponding to the version of the new value.

The history of a vbox is needed to ensure that reads will always be able to access a consistent view of the shared-memory state. At transaction begin, transactions obtain the number of the latest committed transaction and store that number as their *reading version*. This version is used in all reads to ensure that they are consistent: Reading a vbox traverses the vbox's history to obtain the value corresponding to the transaction's reading version (the value with the largest version smaller or equal to the transaction's version). This is one of the key elements that guarantees that the JVSTM satisfies the *opacity* correctness property [Guerraoui & Kapalka, 2008].

Thus, old entries in a vbox's history—that is, all entries except for the most recent one—may be discarded as soon as there are no active transactions that may need to access those entries. To discard no-longer accessible versions, the JVSTM implements its own garbage collection (GC) algorithm.

The JVSTM's GC runs in its own thread and whenever it finds out that the versions overridden by a given commit are no longer accessible, it calls the `clean` method shown in Listing 6.2.

The `clean` method of an ATR iterates over all of the bodies that were written back for the commit of that ATR's transaction and trims them—that is, it sets the `previous`

```
1 // in class ActiveTransactionRecord:
2 void clean() {
3   for (Pair<VBox, VBoxBody> pair : this.allWrittenVBoxes) {
4     pair.body.previous = null;
5   }
6 }
```

Listing 6.2: Algorithm to clean the `VBoxBody` objects committed by that record's transaction.

field to `null`—because the previous entries in the history are no-longer needed. As we shall see next, my AOM approach piggybacks into this `clean` method.

6.2 The Adaptive Object Metadata approach

I showed what a transactional object looks like in Figure 6.1. Compare that with a plain object, which would consist only of the instance of the `Point` class. It becomes clear that using the JVSTM adds significant overhead, not only in terms of the extra memory needed to store all of the metadata, but also in terms of the cost needed to traverse all of that metadata during accesses to the object. Even when the application reaches a quiescent state where the GC is able to trim all of the vboxes' histories, there is still significant overhead due to the vboxes and their single entry.

The novelty of the AOM approach is that it uses two different layouts for transactional objects—the *compact layout* and the *extended layout*—and changes objects back and forth between these two layouts, so that, most often, objects are in the compact layout, which does not need metadata.

The extended layout is similar to the original layout used by the JVSTM, except that AOM uses only one vbox for the entire object, rather than one vbox per field, as shown before. Moreover, given that there is always one and only one vbox for each object, AOM coalesces the vbox with the object. So, in my approach, every transactional class must inherit from `VBox`, thereby guaranteeing that each transactional object has a field `body` that points to its history of values.

The structure shown in Figure 6.2 corresponds to the extended layout of my AOM approach. In this layout, the values contained in the fields of the `Point` class are ignored by the STM, because the values are contained in the history stored in `body`. Each value

of the history corresponds to a *snapshot* of an instance of `Point`, created whenever a transaction commits some changes to that instance. Because in these snapshots the field `body` is not used, I omit it in Figure 6.2. I say that an object with the extended layout is an *extended object*.

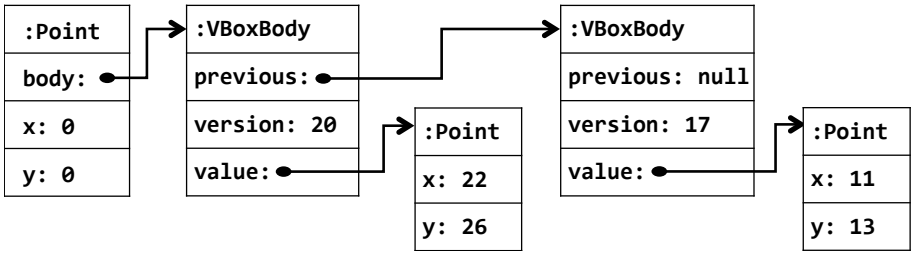


Figure 6.2: An instance of the class `Point` in the AOM's *extended layout*. Whereas in Figure 6.1 there was a vbox for each field, here there is a single vbox for the entire object.

The goal, however, is that most objects should use (for most of the time) the compact layout, in which case the field `body` is `null` (so, there is no history) and the fields declared in the class `Point` contain the object's current values, as shown in Figure 6.3. An object with the compact layout is a *compact object*.

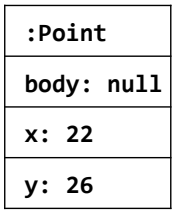


Figure 6.3: `Point` object in the *compact layout*.

In my approach, transactional objects may be in any of these two layouts and they may swing back and forth between the two: A compact object is *extended* whenever it is changed by a transaction and, therefore, it needs to maintain more than one version of its fields; an extended object may be *reverted* (to a compact layout) whenever it has only a single version in its history.

To support the extension and reversion operations, I added two auxiliary methods to the class `VBox`—`snapshot()` and `toCompactLayout(Object)`—which are overridden by every class that inherits from `VBox`. The execution of `snapshot()` returns a clone of the object containing the current values of the object's fields. The method `toCompactLayout(Object)` is responsible for copying the fields of a given snapshot into the transactional object. The implementation for both methods is injected into

every transactional class instrumented by Deuce through a specific *enhancer* that is responsible for this transformation (for more information about the new infrastructure of enhancement transformations for Deuce and its design, see Appendix C).

With this approach of swinging back and forth between the two layouts, I intend to reduce both the memory and the performance overheads caused by the STM's metadata, but obviously there is a tradeoff here. Not only because extending and reverting objects has costs, but also because it may interfere with the rest of the STM operations.

I designed my AOM operations—the extension and the reversion of an object—such that all of the JVSTM's operations preserve their progress guarantees. Namely, that reading a vbox and writing to a vbox are wait-free, and that committing a transaction is lock-free. Even though I use locks in the reversion of an object, the reversion is performed during the GC, which runs in separate threads, and, therefore, does not affect the rest of the transaction's operations.

In the following three subsections, I describe each of the AOM operations in detail: reverting, extending and reading. After that, in Subsection 6.2.4, I discuss informally the correctness of my approach by looking into the various scenarios of concurrent interleavings of the STM operations over a transactional object.

6.2.1 Reverting Objects

The process of reverting an object is piggybacked into the JVSTM's GC: When the GC trims an object's history and only the most recent version remains, it may revert the object to the compact layout.

In Listing 6.3 I show both the `tryRevert` method that implements the reversion of an object and its call during the GC's `clean` method. Figure 6.4 illustrates the process of reverting an object with a single version.

When the GC algorithm trims the history of an object (line 4), it calls the `tryRevert` method (line 5). Given that the JVSTM's GC may run with multiple threads, the `tryRevert` method may be called concurrently for the same object. Thus, it acquires the object's monitor (line 10) to prevent that more than one thread tries to revert the same object concurrently. Once the monitor is acquired, the method checks whether the head of the object's history is the `VBoxBody` that was just trimmed (line 11). If it is, which means that there is only one version in the history, it copies the values contained in that body to the corresponding fields in the object (line 12) and tries to CAS

```
1  // in class ActiveTransactionRecord:
2  void clean() {
3    for (Pair<VBox, VBoxBody> pair : this.allWrittenVBoxes) {
4      pair.body.previous = null;
5      tryRevert(pair.vbox, pair.body); // new call for AOM
6    }
7  }

9  boolean tryRevert(VBox vbox, VBoxBody body) {
10   synchronized (vbox) {
11     if (vbox.body == body) {                              // step 1
12       vbox.toCompactLayout(body.value);                   // step 2
13       compareAndSwapObject(vbox, bodyOffset, body, null);// step 3
14     }
15   }
16 }
```

Listing 6.3: Algorithm to revert an object as part of the GC's clean task. I show in bold the line that was added to the `clean` method.

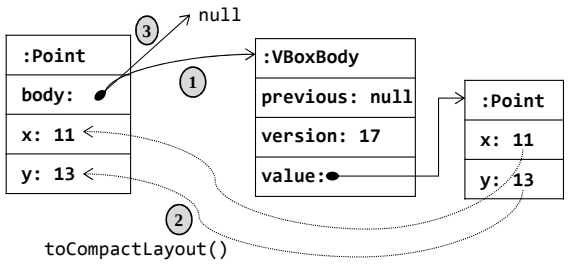


Figure 6.4: Reverting an object that is in the extended layout and that stores the values 11 and 13 as the most recent, and only, committed values.

the `body` of the object to `null` (line 13). The CAS fails if the current value of the field `body` is no longer the body that was trimmed, which may happen only if one or more transactions commit new values for this object. So, when the CAS fails nothing else needs to be done, as the object cannot be reverted.

6.2.2 Extending Objects

When an object is created, it is naturally in the compact layout. Moreover, when using LICM, the object will stay in the compact layout even when its owning transaction

commits and publishes it. This is safe because the object is not shared until that transaction successfully commits, and, therefore, no other transaction may need another version of that object.

So, when does a transactional object need to be extended? As mentioned before, we need to extend an object only when we need to have more than one version of the object, which happens only when a transaction writes a new value to any of the object's fields and successfully commits those changes.

Thus, similarly to what was done for the reversion of an object, the extension operation is piggybacked into the write-back phase of the commit: The object is extended during the write-back if it is in the compact layout. In this case, however, we cannot use locks in the extension operation if we want the entire commit operation to be lock-free.

In Listing 6.4 I show the new code for the write-back phase, which includes the code to extend objects if needed. When trying to add a new version to an object, the write-back operation must now check if the object is in the compact layout (line 6). If it is, then the previous version of the object's state is in the object's own fields, rather than in its (nonexistent) history. Thus, to add the new version to the previous version, it must extend the object such that both versions are in the object's history (because after the extension no transaction will look into the object's fields). This is accomplished by: (1) creating a snapshot of the object (line 7), (2) creating an entry for the snapshot that is marked with version `0` (line 8), and (3) creating the entry for the new value and version that points to the entry with version `0` (line 9). If the object is not in the compact layout, then the write-back works as before, by creating a new entry for the new value and version (line 11). In either case, the operation proceeds by calling the `CASbody` method (line 13) to update the history, which in the case of an extension corresponds to installing a new history. An example of a successful extension that goes through all these steps is shown in Figure 6.5.

The method `CASbody` was also changed. Previously, it attempted to CAS to the new body only once and returned even if the CAS failed, because the only reason why the CAS operation could fail was if some other thread performed the write-back of an identical body, in which case there was nothing else to do. With the AOM, however, there may be another reason for a failure of the CAS: A concurrent GC thread may revert the object back to its compact layout between the read of the object's `body` (line 3) and the attempt to CAS it to the new value (line 20); such a reversion changes the value of `body` to `null` and the CAS fails. But in this case the new value was not added to the object's history and, therefore, we need to try to add it again. This

```
1  // in class VBox:
2  VBoxBody commit(Object newValue, int txNumber) {
3    VBoxBody currHead = this.body;
4    if (currHead == null || currHead.version < txNumber) {
5      VBoxBody newBody = null;
6      if (currHead == null) {
7        Object v0 = this.snapshot();                          // step 1
8        VBoxBody body0 = new VBoxBody(v0, 0, null);           // step 2
9        newBody = new VBoxBody(newValue, txNumber, body0);    // step 3
10     } else {
11       newBody = new VBoxBody(newValue, txNumber, currHead);
12     }
13     return CASbody(currHead, newBody);                      // step 4
14   } else {
15     return currHead.getBody(txNumber);
16   }
17 }

19 VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
20   if (compareAndSwapObject(this, bodyOffset, expected, newBody)) {
21     return newBody;
22   } else {
23     // If the CAS failed then either the new value must already be there,
24     // or the object may have been reverted by a concurrent GC Task,
25     // in which case we need to retry the CAS to commit the new body.
26     // If the second CAS fails, then some other thread did the write-back.
27     VBoxBody currHead = this.body;
28     if (currHead == null) {
29       if (compareAndSwapObject(this, bodyOffset, null, newBody)) {
30         return newBody;
31       } else {
32         return this.body.getBody(newBody.version);
33       }
34     } else {
35       return currHead.getBody(newBody.version);
36     }
37   }
38 }
```

Listing 6.4: New write-back algorithm of the `VBox` class including the extension. I show in bold the code that was added to the original code of the JVSTM.

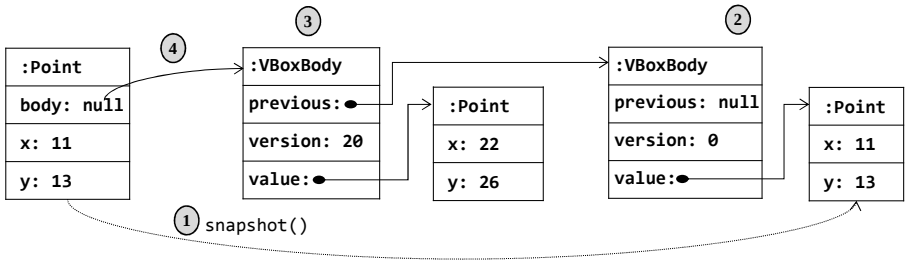


Figure 6.5: An example of a transaction that commits the values 22 and 26 to the fields `x` and `y`, respectively, of a `Point`, which was in the compact layout and was storing the values 11 and 13 before.

scenario is identified by reading again the object's `body` (line 27) and checking whether it is `null` (line 28), in which case the CAS is attempted once again (line 29) to swing the field `body` from `null` to the new body. If this second attempt of the CAS fails, then it is because this body was written back by some helper. I shall return to why this is so in Section 6.2.4, where I discuss the correctness of my AOM algorithms.

6.2.3 Reading Objects

Given that with AOM objects may be in one of two possible layouts, the read operation must take the layout into account when accessing an object. So, the first thing that a read operation does is to check whether the field `body` of the object is `null`. If it is, the read operation uses a fast path that directly accesses the other fields of the object—I refer to these as *proper fields*.

Consider the case of a thread Th_1 that is executing transaction Tx_1 with a reading version v_1. To read the proper field of an object Ob, Th_1 must find `null` at the field `body`. Otherwise, it goes through the history found in `body` until it finds the correct version and then reads the proper field of the corresponding snapshot. The latter case is the original algorithm of the JVSTM, which traverses only immutable data structures.

For read-write transactions we must still log the read operation in the read-set for objects in both layouts. But for read-only transactions we do not need to keep any log and, therefore, reading an object in the compact layout has almost no overhead over accessing directly an object's field without an STM barrier: We just have to check whether the `body` field is pointing to `null` or not.

On the other hand, if the object is extended, then the read operation will always have to search for the correct version in the history. So, even if the required version is

found at the head of the history, the read operation still has the overhead of traversing two pointers: One from the field `body` to the entry at the head of the history and another from that entry to the snapshot containing the value to be accessed. This pointer chasing typically results in poor cache locality, further contributing to the degradation of the performance.

6.2.4 Correctness of the AOM Operations

After presenting the algorithms for reverting, extending, and reading an object, I discuss now the correctness of those algorithms while giving the rationale for my design.

As in the original JVSTM, in AOM the objects representing an history entry—instances of the class `VBoxBody`—are immutable. They are created during the write-back phase of a transaction's commit and once created cannot be changed. Likewise for the snapshots, which are the instances of the transactional classes that are created to represent the value of an object at a particular version. Snapshots are created also during the write-back (which may include the extension of an object) and stored in the history entries before they are made available to other threads.

So, the only interesting cases that may be both read and written by multiple threads are the fields of a transactional object. Whereas without AOM only the field `body` is used, with AOM the proper fields of a transactional object may also be read and written.

To help us reason about the correctness of AOM it is useful to make a couple of observations about the JVSTM first.

In the JVSTM, a thread Th_1 will help a transaction Tx to commit (e.g., by doing its write-back) only if Th_1 is itself executing a transaction Tx_1 (possibly Tx) that wants to make progress. This means that Tx_1 must have started before the commit of Tx. So, if Tx commits with the version v, the version with which Tx_1 started (i.e., its *reading version*), v_1, is necessarily smaller than v (i.e., $v_1 < v$). Moreover, all commits are done in order—that is, only after the write-back for version v is completed may any thread start doing the write-back for version $v + 1$. Given these observations, I may now establish some results regarding AOM.

I start with a theorem establishing the impossibility of a late write-back.

Theorem 1. *A thread that arrives at the write-back of an object Ob for some transaction Tx after at least another thread has successfully finished that write-back will not change the object in any way.*

Proof. This result is trivially guaranteed in the original JVSTM because each helping thread reads the value of the field `body` before deciding whether it needs to do the write-back. If it finds an entry corresponding to the write-back that it is trying to do, it returns without changing the object; otherwise it tries to install a new entry with a CAS. Yet, only the first attempted CAS will succeed and all others will fail because late threads will find the value of `body` changed since they first saw it. With AOM, however, the field `body` may swing back and forth between `null` and some other values (due to reversions). So, we need to check whether a late thread may find a `null` value in the field `body` and decide to add a new version, even though that version has been written before.

Assume that Th_1 is a late thread in the write-back of Ob for Tx. That means that some other thread Th_2 must have completed the write-back of Ob for the same transaction Tx, necessarily leaving in the field `body` of Ob an history containing at least two versions: the version v and its previous version. So, for Th_1 to find a `null` value in the field `body` of Ob after the write-back done by Th_2, Ob must have been reverted in the meanwhile. Yet, if Tx_1 is executing, Ob cannot be reverted, because it will need to keep at least two versions until Tx_1 finishes (the version v, which cannot be accessed by Tx_1, and the previous version). On the other hand, if the reversion occurs before Tx_1 starts, then it is because the commit of Tx had already finished and, therefore, Tx_1 would not have a version $v_1 < v$. □

Theorem 2. *If a thread Th_1 is executing the write-back of a transactional object Ob, then Ob is reverted at most once until Th_1 finishes the write-back, regardless of how long it takes.*

Proof. Let us assume that, while Th_1 is executing the write-back of Ob, there are two reversions for Ob. If that is the case, then Ob must have been extended in between the two reversions, while Th_1 is still running. Given that by Theorem 1 there are no late write-backs, that extension must have occurred as part of the write-back of Ob for some version v such that $v > v_1$ (where v_1 is the reading version of the transaction that Th_1 is still executing). So, at least until Th_1 finishes the execution of the write-back of Ob, Ob cannot be reverted again, because Tx_1 cannot access the version v of Ob. □

Theorem 2 justifies why the method `CASbody` in Listing 6.4 tries to attempt the CAS only once after a failure of the CAS at line 20: The failure of the first CAS may

be due to a concurrent reversion, but the failure of the second CAS cannot, because it is not possible to have two reversions for the same object while a write-back is in progress. A similar, simpler result is the following.

Theorem 3. *If a thread Th_1 executing a transaction Tx_1 reads a transactional object Ob and sees Ob in the compact layout, then Ob cannot be both extended and reverted until Th_1 finishes executing Tx_1.*

Proof. The reasoning for this proof is similar to the previous one. Assume that the reading version for Tx_1 is v_1. If Ob is extended after Th_1 sees it in the compact layout, then it must be because of a write-back to Ob that created a version v such that $v_1 < v$. After that extension, Ob cannot be reverted while Tx_1 is still running, because it cannot access v. □

Given these three theorems, I may now discuss the correctness of each of the AOM operations.

Correctness of the read operation

Consider the case of a thread Th_1 that is executing transaction Tx_1 with a reading version v_1. If Th_1 finds the object Ob in the compact layout, then it must be because the most recent version of Ob, let us call it v_{ob}, is such that $v_{ob} < v_1$ and the proper fields of Ob contain the values corresponding to that version v_{ob}. Otherwise, either the reversion could not have occurred, or Tx_1 would have another, higher version. Moreover, by Theorem 3, we know that while Th_1 is executing Tx_1, Ob may be extended but it cannot be reverted again. Given that only the reversion writes into the proper fields of an object, we know that the values in the proper fields of Ob are the values that Th_1 needs to read and, thus, that it is safe for Th_1 to read them.

If, on the other hand, Th_1 finds Ob in the extended layout, it will be able to find the version that it needs in its history. To see why, consider the two possible scenarios: (1) the last change to Ob occurred before Tx_1 started, and (2) Ob was changed after Tx_1 started, but before Th_1 attempted to read it. In the first scenario, if Ob is still in the extended layout, it is because no reversion for Ob occurred in the meanwhile and the head of the history contains the entry that Th_1 needs to read. In the second scenario, either Ob was already extended and the new commits simply added new versions to the head of the history, or Ob was extended by the first of the commits, creating not only a newer snapshot but also a snapshot for version `0` with the values of Ob's proper

fields. In both cases, Th_1 will jump over the entries at the head of the history until it finds an entry with a version that it can read (possibly, version `0` that results from the extension). We shall see below why it is correct to create this version `0`.

Correctness of the reversion operation

Only the reversion operation writes to proper fields, through the execution of the method `toCompactLayout`, which copies the values of a snapshot into the proper fields. Given that doing this copy requires acquiring the object's monitor, no two concurrent threads may be copying values back to the proper fields at the same time and, thus, after a reversion the values at the proper fields of an object will necessarily reflect a consistent set of values for some version (obtained from a given immutable snapshot).

Yet, while the copy is executing, the values in the proper fields may correspond to an inconsistent set of values, as some fields have already been copied while others have not. So, we need to ensure that while such a copy is in progress no reads of the proper fields are made.

With AOM, proper fields may be read only in two cases: (1) when reading an object during a transaction, and (2) when extending an object during the write-back (by the execution of the method `snapshot`). In both cases, the proper fields are read only if the field `body` was found to be `null`.

On the other hand, the reversion operation tries to change the field `body` back to `null` only after copying all values from a snapshot into the proper fields (see lines 12 and 13 in Listing 6.3). This is done with a tentative CAS of `body` from the entry containing the snapshot to `null`. This reset of the field `body` may fail if another thread concurrently commits a new value to this object, in which case it installs a new entry in `body`. In this case, the copy of the snapshot back into the proper fields was in vain because the object may no longer be reverted. Still, the values copied are consistent after the copy. But, more importantly, given that `body` was never `null`, no read of the proper fields may have occurred during the copy. So, reversions are guaranteed to occur only when they are safe with regard to potential concurrent reads.

Correctness of the extension operation

The behavior of the write-back of an object when the object is in the extended layout was not changed with the AOM. So, we just need to consider the case of doing a write-back for an object that is in the compact layout.

We have two scenarios to consider: (1) when the write-back operation first sees the object extended but the object is reverted during the write-back, and (2) when the write-back operation sees the object already in the compact layout.

In the first scenario, the write-back obtains the object's current history and creates a new entry to add at the beginning, as in the normal case. Yet, when it tries to install the new history the CAS fails because the object was reverted in the meanwhile. Still, the newly computed history is still correct and may be installed in the `body`, effectively extending the object again. This is accomplished by the second CAS in the method `CASbody`. As discussed before, after this second CAS it is guaranteed that the new version will be installed in the object.

In the second scenario, the write-back operation needs to create a snapshot of the object to capture the current values of its proper fields. This snapshot is tagged with version `0` and added to the history, just after the newly created version. As we saw before, this snapshot is guaranteed to obtain a consistent view of the object's state. The version `0` is used because an object in the compact layout has no information about which version its values correspond to. Still, we know that it must be the case that no previous version would be needed by any of the running transactions (or else the object would not be in the compact layout), and, thus, we may use any version that is lower than the oldest running transaction; version `0` satisfies trivially this constraint.

6.3 Implementation Approaches

As happens for other optimization techniques the AOM requires a specific object model distinct from the one provided by the managed runtime environment, which allows to store additional STM metadata in-place within the transactional objects. Thus, to build the AOM solution, I developed two different implementations that follow two implementation approaches: (1) at the virtual machine runtime level, and (2) through Java bytecode instrumentation. Both solutions are implemented in the Java platform.

In the following subsection I introduce the basics of the Java Object Model, which is the basis for both implementations. After that, in Subsections 6.3.2 and 6.3.3, I give an overview about the requirements of both implementations: the former implementation uses the Jikes Research Virtual Machine (Jikes RVM) [Alpern *et al.* , 2005] and the latter was integrated in the Deuce STM engine [Korland *et al.* , 2010].

In Appendix B, I give further details about my extension to Jikes RVM's just-in-time compiler to support the integration of the JVSTM with AOM.

6.3.1 Basics of the Java Object Model

Managed runtime environments for object-oriented programming languages such as Java represent objects in memory according to a set of rules on how to layout, not only the fields of an object, but also all the remaining metadata associated to each object—for instance, the type of the object, its lock, or its hashcode. The set of rules describing how to layout an object is called the *object model* of the runtime environment. Naturally, different runtime environments, even if for the same language, have different object models, which may depend on various factors, including the computer architecture on which the runtime environment executes.

The Java type system is organized in two distinct groups of data types: primitives and references. Instances of primitive types are stored in-place. This means that they are allocated in the stack frame of the method that has instantiated them, or in the storage of an object or a class. Primitive types in Java are used and passed by value. In other words, when a primitive value is assigned or passed as an argument to a method, it is simply copied.

Instances of reference types (also known as objects), on the other hand, have their storage allocated in the heap memory space, which keeps the values of object's instance fields. The value of each field is stored in a minimum of one word slot and a maximum of two slots.[2]

Unlike primitives, objects are accessed by reference. A reference is simply a pointer for the storage of an object. In turn, a variable of a reference type holds a reference to the corresponding storage. For example, in Jikes RVM a reference to an object always

[2]Most virtual machines make some exceptions to improve alignment and pack the fields in an object. For instance, two fields of a 16-bits type could be stored in the same slot. Yet, this optimization is beyond the scope of this discussion. My implementation does not interfere with the way how the virtual machine lays out the object's fields.

points to the second slot of the instance field's region, as shown in Figure 6.6. Instance fields are recorded in heap space in the same order that they are declared in the Java source file[3], as shown in Figure 6.6.

```
A a = new A((short)32767, 34.7, 2147483647);
```

```
public class A {
  short x;
  double y;
  int z;
}
```

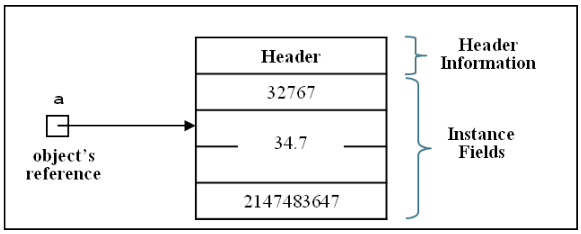


Figure 6.6: Object's storage layout. For each instance field declared by class `A` there is a specific storage (one or two slots) that keeps its value. In the case of the `y` instance field, it takes two slots to keep a double word value.

Besides the slots for instance fields, all objects have a fixed number of specific slots for metadata information. This portion of an object's heap storage is called the header of the object, and is depicted also in Figure 6.6. According to my AOM design, transactional objects may have the layout that is natively specified by the managed runtime environment, avoiding any extra STM structure and, consequently, extra indirections.

6.3.2 Integrating JVSTM with AOM in Jikes RVM

The Jikes RVM (Jikes Research Virtual Machine) [Alpern *et al.* , 2005] is a research project to create an open-source Java virtual machine. One of the key differences from many other virtual machines is that it is implemented in Java and that it only uses a small amount of C code for accessing the operating system.

In this work, I describe my approach on top of the object model used by the Jikes RVM runtime environment for the Intel's x86 architecture. Yet, my proposal is applicable to other architectures.

In the following subsection I describe the STM interface of the AOM implementation. After that, I explain the modifications required in the Jikes RVM to support the integration of an STM.

[3]Except if the virtual machine packs the fields to improve alignment.

STM Interface

To support program-level transactions I provide a method annotation, `@Atomic`, and a type annotation, `@Transactional`. Every method that is supposed to be executed as a transaction is marked with `@Atomic` and is called an *atomic method*. The objects accessed by atomic methods should be instances of classes annotated with `@Transactional`, except for immutable objects, such as strings, which are naturally thread safe.

If a thread executes a method `m` atomically, all changes to transactional objects in the context of that method `m` are serializable with respect to other transactions. When `m` completes successfully, its effects over transactional objects become globally visible. For transactional objects, even when they are accessed outside of a transaction the system guarantees *strong atomicity* [Martin *et al.* , 2006] because non-transactional accesses to those objects are implemented as single-instruction transactions. Yet, the system just satisfies this property for instances of classes that are correctly annotated. For non-transactional objects the system does not guarantees even weak atomicity [Martin *et al.* , 2006], because it does not guarantee any isolation between transactional and non-transactional accesses to those objects.

When a transaction starts, it becomes the thread's *current transaction* until it finishes or another nested transaction starts. When a transaction finishes, its parent, if any, becomes the thread's current transaction. Because nested transactions are executed by the same thread of their parents, when a nested transaction is executing, the parent transaction is not. Once a transaction starts, it is stored in the corresponding `RVMThread` object that represents a Java thread's execution context in Jikes RVM. At runtime the `RVMThread` object is referred by `RVMProcessor` object, which is accessible via the processor register—`esi` for IA32 architecture—as presented in Figure 6.7. So, from any point of execution we can know if we are, or not, in a transactional context by consulting the `RVMThread` object.

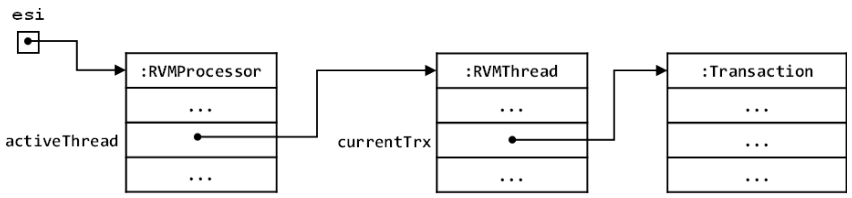


Figure 6.7: Processor register

Calling an atomic method `m` corresponds to wrap that invocation as shown in listing 6.5. This code is generated by the just-in-time compiler (see Appendix B for further details). For now, to better express the algorithm I choose the Java language.

```
1 do{
2   Transaction.begin(getCurrentThreadFromProcessorRegister());
3   m();
4 }while(Transaction.tryCommit() != Status.SUCCEED)
```

Listing 6.5: Wrapping `m` call in a transaction control flow

According to the Java specification a method caller is responsible for pushing arguments on the stack and the callee for cleaning those arguments and push the method result, if it exists. This means that we must preserve a backup of the method's arguments, in case we have to repeat the invocation of the atomic method. I also leave the details of that solution to Appendix B.

So, to turn a method `m` into an atomic method we must call `Transaction.begin` before calling `m` and `Transaction.tryCommit` at the end of `m` execution. Finally, we have to evaluate `tryCommit` returned value and if it has not succeeded we must restart a new transaction. The static methods `begin` and `tryCommit` of the class `Transaction` encapsulate all logic details about transactions management.

The `begin` method creates an instance of a `Transaction` subclass and initializes the required data structures (*read-set* and *write-set*, in case of a read-write transaction) to record all read and write operations made in the scope of a transaction. The `tryCommit` operation checks if the current transaction is valid and if so, it performs the commit operation storing all mappings from the *write-set* into box bodies of the corresponding versioned boxes.

Making the Jikes RVM Runtime Transactional

To execute atomically, a method has to perform the following actions:

1. Create a new transaction object, which for read-write transactions includes initializing its internal data structures: *read-set* and *write-set*;
2. Register all method's read and write operations in the transaction's *read-set* and *write-set*, instead of accessing the standard object fields locations.
3. At the end, try to commit the transaction and in case of failure it must repeat the execution restarting on the first point.

To accomplish the first and the third points, we must wrap the method invocation into a transaction control flow, as shown in Listing 6.5. This integration is made at the just-in-time compiler level by evaluating the method's annotations: if `Atomic` is present we must insert the appropriate STM calls. For a method to behave as explained in the second point, we can follow two different approaches. One possible solution is to create a second version of each atomic method—a *transactional twin* (see, e.g., [Yoo *et al.* , 2008] and [Korland *et al.* , 2010] for a similar approach)—that replaces every field access with an *STM barrier*. This method should be called from the transaction control flow (Listing 6.5), replacing the invocation of the original method and every invocation to an atomic method from inside this method should also be replaced with a call to the corresponding *transactional twin*.

I chose a different approach that changes the default behavior of the `getfield` and `putfield` *bytecodes*. If the target of these operations is an object not marked with `Transactional`, we have nothing else to do and these *bytecodes* perform just their default behavior. This choice is made at just-in-time compile time and compromise my solution to the assumption that if a class is transactional, then all inherited fields must be from transactional classes too. On the other hand, given that I have modified the Jikes RVM object model, we must change the way that the `putfield` and `getfield` *bytecodes* access values from fields of transactional objects.

Every time a `putfield` operation is performed to a transactional object it just has to register that operation in the current transaction's *write-set*. If there is no transaction, a new one must be created and committed at the end. These tasks are encapsulated in the corresponding *STM write barrier*, that is defined by a static method `write<field's type>` of the class `VBox`. For each primitive type there is a corresponding static method in the class `VBox` with a parameter according to that primitive type. There is also a `writeObject` static method with a parameter of type `Object` that handles all write operations to reference type fields.

The `getfield` operation has to access in the first place the object's header to recognize the layout of the object and find out where it should read the field's value: from the object's field location or from the versioned box. If that object is in the *extended layout* it must read its fields values via an *STM read barrier*. This is done through the `read<field's type>` static methods of `VBox` class. As happens in *STM write barriers*, for each primitive-type there is a corresponding static method in the class `VBox`, which has the return type corresponding to the field's type. The *just-in-time* compiler is responsible for determining to which method should emit the call. The value returned from the *STM read barrier* stays on the evaluation stack as the result of the `getfield`

```
1  ; Object's reference is on the top of stack and is copied to ecx.
2  I0: mov ecx, [esp]
3  ; Compare object's header with Null.
4  I1: cmp [ecx - HEADER_OFFSET], NULL
5  I2: jeq stdRead
6  I3: ; The 1st argument is the object's reference (already on stack)
7      ; The 2nd argument is the field's offset
8      ; The 3rd argument is the VBox's reference
9  I4: push fieldOffset
10 I5: push [ecx - HEADER_OFFSET]
11 I6: call VBox.read<field's type>
12 I7: jmp done
13 stdRead:
14 I8: ; Default behavior of getfield.
15 I9: call VBox.registerRead
16 done:
```

Listing 6.6: Behavior of `getfield` operation for atomic objects expressed in IA32.

operation. If the read object is in the *compact layout*, then the `getfield` can read the object's fields directly. Yet, at the end, it must still call the `registerRead` static method of `VBox` class, to record that operation in the transaction's *read-set*.

So, at runtime and for atomic objects, the `putfield` bytecode is always translated into a simple call to the corresponding *STM write barrier* and the `getfield` bytecode is translated to the algorithm presented in Listing 6.6.

6.3.3 Extending Deuce STM

Despite the promising results obtained with the prototype of the JVSTM with AOM in Jikes RVM, this prototype falls short of the efficiency gains that I was aiming for. Essentially, the lack of efficiency of this implementation is due to the undesired effects of the modifications made to the just-in-time compiler, which interfere with the code optimization process and degrades the overall performance.

So, I chose a different technological environment and I redesigned my solution to the model of the Deuce STM framework. Yet, the original Deuce STM provides extensibility only in terms of the specification of the STM algorithm and it does not allow either the definition of additional behavior orthogonal to all STMs, or any modification to the standard type system. To support the JVSTM and the AOM it is mandatory

to store metatada in-place within the transactional object. So, I had to extend Deuce STM to support this feature.[4]

Yet, I could not follow the same approach of the Jikes RVM implementation, which takes advantage of the object's header to store the handle of the object's history. So, I have to add to all transactional objects an additional field `body` to point to its history of values. To that end, I change the hierarchy of the transactional classes to inherit from the `VBox` class.

In the following, I explain how transactional classes are enhanced to inherit from `VBox`. After that, I describe the solution for transactional arrays.

Enhancing Regular Objects

To support this feature, I added to the Deuce framework a new infrastructure that allows the specification and execution of *enhancers*, which are additional transformations to the standard Deuce instrumentation (for more information about this new infrastructure and its design, see Appendix C). For instance, in the case of JVSTM, we need to combine a *post* enhancer that transforms the definition of transactional arrays—`EnhanceVBoxArrays`—and we also need to include a *pre* enhancer that makes a required transformation to support static fields—`EnhanceStaticFields` (all JVSTM enhancers belong to the package `org.deuce.transform.jvstm`). For static fields, and due to restrictions of the Java object model, I chose to transform static fields into instance fields of a well-known singleton class. After the previous transformations, all memory locations, including arrays and static fields, are instrumented by the same enhancer—`EnhanceTransactional`.

Enhancing Arrays

The enhancer `EnhanceTransactional` defines the transformation responsible for replacing the root of the transactional classes hierarchy from `Object` to `VBox`. However, I cannot apply the same approach for arrays, because we cannot change their base class. So, we need to wrap arrays with instances of the class `AbstractVBoxArray`, which in turn inherits from `VBox`. But, this class does not hold an `elements` field array, because we cannot define a generic array comprising arrays of all primitive types. So, we have

[4]This adaptation of Deuce is available at https://github.com/inesc-id-esw/deucestm/

one class for each primitive type and another class for arrays of reference types. The name of these classes follow the convention `VBox<T>Array`.

The enhancer `EnhanceVBoxArrays` is responsible for this transformation and for replacing all operations that deal with arrays with new operations manipulating instances of the `AbstractVBoxArray` class. This approach allows us to store metadata in-place for regular objects only, or for both objects and arrays, depending on the enhancers that we pass to Deuce.

Furthermore, wrapping arrays in `VBox` objects adds an extra indirection when they are accessed from non-transactional code, because these objects are no longer arrays themselves. Note that for regular objects, I do not change their behavior: I preserve their original structure and interface, which can be accessed either from transactional or non-transactional code. So, unlike with regular objects, when I access arrays from non-transactional code I have to get the encapsulated array from the `VBox` object. But, unlike unidimensional arrays, where the unwrap operation consists only in getting its `elements` field, for multidimensional arrays I need to instantiate a new array and copy the elements from each unidimensional array that is encapsulated in its own `VBox` object array. For this reason, in my current approach, the unwrap of a multidimensional array has a huge overhead.

Note that we cannot let the inner dimensions of a multidimensional array be ordinary arrays. In Java, the elements of a multidimensional array are accessed as a series of unidimensional accesses (e.g. in the case of a bidimensional array stored in a field we need to perform an `aaload`[5] to get the reference to the inner array, followed by another array access). All of these operations (which may not appear collocated in the code, because we may pass an inner array as an argument to a method) are instrumented by the Deuce STM and replaced by STM barriers. So, if we let the inner array be an ordinary array then we cannot access the object's history in the second barrier.

Thus, the reverse process of unwrapping a multidimensional array from the `VBox` object forces us to rebuild the entire multidimensional array from the elements encapsulated on each array of every inner `VBox` object, instead of just getting the reference to the internal array, as happens for unidimensional arrays. Yet, I disallowed the unwrap of multidimensional arrays (throwing an `UnsupportedOperationException`), due to the overhead of this operation and in this case, I recommend one of the following alternative options, if possible: (1) to exclude the owner class from Deuce instrumentation; or (2) to access the array always from atomic methods, such as atomic indexers.

[5]Load onto the stack a reference from an array

6.4 Validation

In Chapter 5, I evaluated the improvement in performance and memory consumption of an STM enhanced with the LICM optimization technique. In this Section, I evaluate the behavior of the JVSTM enhanced only with the AOM, and without the LICM support. Later, in Chapter 7, I will show the effects of both optimization techniques over the JVSTM.

In the following subsection I show a performance evaluation of the AOM for a diversity of benchmarks and, then, in subsection 6.4.2, I also show the results of an application's memory consumption with and without the AOM.

6.4.1 Performance Evaluation

To evaluate the performance of the AOM approach, I compare the original lock-free implementation of the JVSTM (*jvstm*) with the same algorithm enhanced with the AOM (*jvstm-aom*). In both cases I use the Deuce STM to automatically transactify the evaluated benchmark.

I analyze the performance of the STMs in the STMBench7 benchmark, in the Vacation application of the STAMP benchmark, and the JWormBench. Given that the STMBench7 and the JWormBench benchmarks also have a medium/fine-grained locking synchronization strategy, I also compare the performance of the lock-based approach with the STM-based approach.

Vacation

The Vacation application emulates a travel reservation system where *customers* concurrently make requests that affect some of the application's *items* (such as flights and cars): Each request from a customer is composed of operations—such as reserving a car, or cancelling a reservation—that must be performed atomically. Because all the operations that take place in a request include at least some transactional writes, this application does not take advantage of read-only transactions. The application holds the global items in red-black trees, one for each type of item, including the customers themselves. On the other hand, each customer contains a list that points to each resource that it reserved.

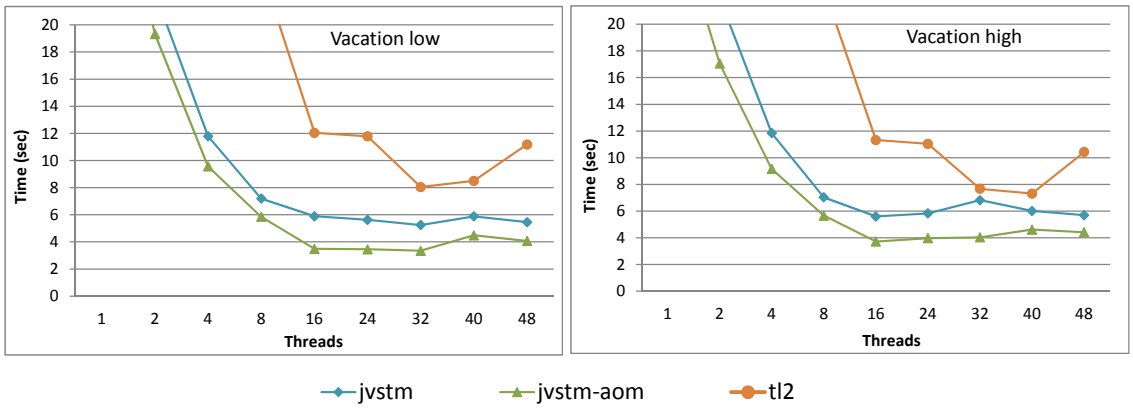


Figure 6.8: The results for Vacation with TL2 and JVSTM, in the two workloads (low and high contention).

I ran this benchmark with the configurations proposed by [Cao Minh *et al.* , 2008], but because I want to emulate the behavior of large-scale programs, I used larger data sets. So, instead of the proposed value for the parameter *n* (between 2 and 4), which specifies the number of *items* operated by session and that is directly related to the transaction's length, I used a higher number of 256 items. Thus, for the low contention scenario I used the parameters "-n 256 -q 90 -u 98 -r 262144 -t 65536", whereas for the high contention scenario I used the parameters "-n 256 -q 60 -u 90 -r 262144 -t 65536".

These tests were performed on a machine with 4 AMD Opteron^{TM} 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the 1.6.0_33-b03, running on Ubuntu with Linux kernel version 2.6.32.

I present in Figure 6.8 the results obtained for the two workloads of the Vacation benchmark with three different STMs. Because Vacation does not provide a lock-based synchronization alternative, I include also the results of the TL2 for comparison with the JVSTM and the JVSTM-AOM (LSA presents similar performance to the TL2).

In Vacation, the *items* operated by each transaction are randomly selected by each *customer*. Thus, because we use large data sets, there is a low probability that two consecutive transactions performed by the same *customer* (or even two concurrent transactions) update the same *items*. Furthermore, all transactions need to traverse the global structures containing the application items to find the selected resources. These lookups perform many memory reads over transactional objects that are seldom changed. So, I expect to be able to improve the performance of the benchmark if I keep transactional objects in a compact layout, thereby promoting the use of lightweight STM barriers. In fact, there is an advantage in reverting the transactional objects to the compact layout whenever possible to accelerate the lookup phase of each transaction

and therefore improve the overall performance of the benchmark. This is the reason why the AOM improves the performance of the JVSTM in both workloads of the Vacation: As shown in Figure 6.8 the *jvstm-aom* reduces the time taken to execute the benchmark to almost half of the time taken by *jvstm* in the best case.

STMBench7 benchmark

These tests were performed on a machine with two quad-core Intel Xeon CPUs E5520 with hyper-threading, resulting in 8 cores and 16 hardware threads.

As in the case of the Vacation benchmark, in the STMBench7 the AOM can also improve the performance of the JVSTM when the traverse operations are dominated by read accesses. In a read dominated workload the AOM can duplicate the performance of the baseline JVSTM as depicted in the results of Figure 6.9. On the other hand, when we increase the update rate, the benefits of the AOM are reduced because of the overhead incurred by the extension of transactional objects. If we have too many objects being extended and reverted back consecutively between the two layouts, then the benefits of the lighter read barriers cannot overtake the overheads of the layout transitions. Yet, the results of Figure 6.9 show that even in the write-dominated workload the AOM adds only a residual overhead that does not degrade the performance of the JVSTM.

JWormBench benchmark

In my experimental tests with the JWormBench, I used the following configuration for both workloads: a *world* consisting of 1024 *nodes* and 48 *worms* with a head's size varying between 2 and 16 nodes, which dictates transactions with an average length between 4 and 256 read/write accesses.

As we can observe in the results shown in Figure 6.10, for the read dominated workload the AOM improves the performance of the JVSTM by 2.5-fold. In the case of the workload dominated by write operations we can observe the same effect verified in the STMBench7 with the write-dominated workload. When we increase the number of worker threads, then we also increase the contention and the number of objects that stay in the compact layout reduces. Therefore, the overhead of the increasing number of layout transitions degrades the performance of the AOM.

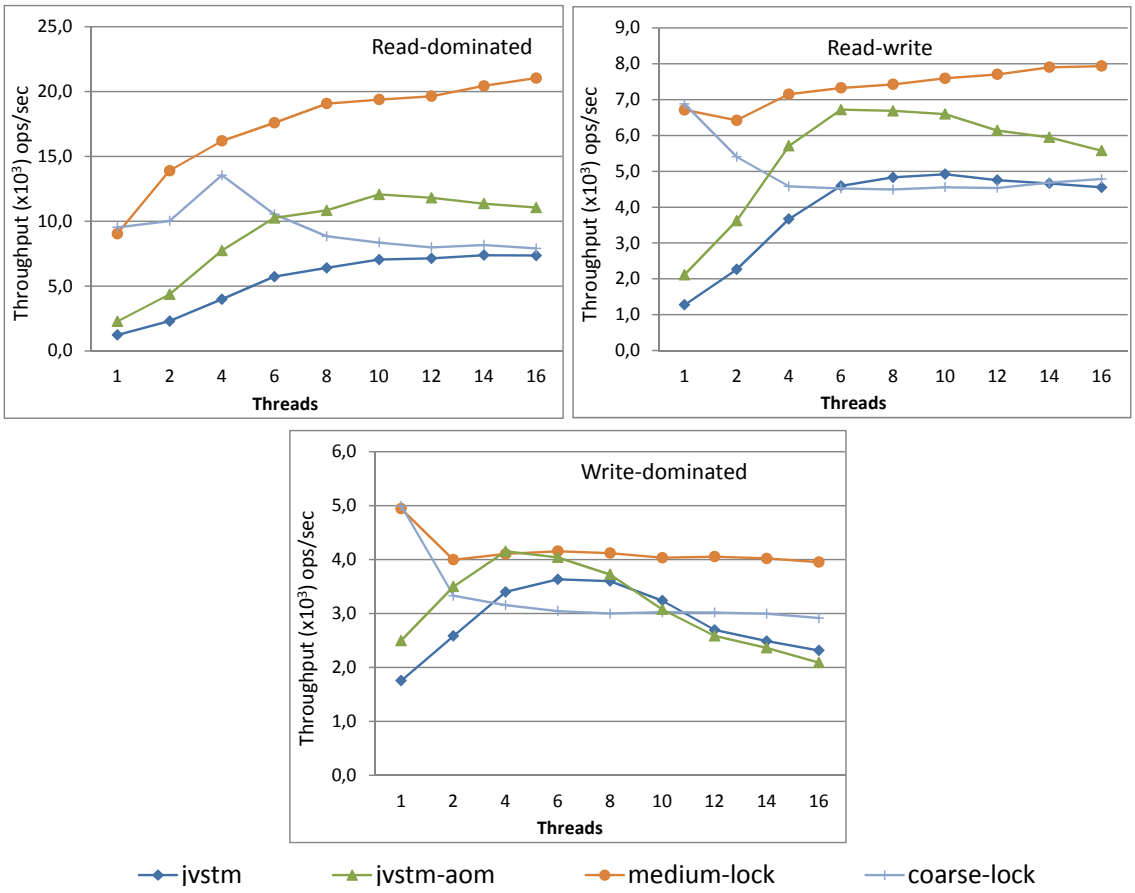


Figure 6.9: The results for STMBench7 with JVSTM, in the three available workloads, without long traversal operations.

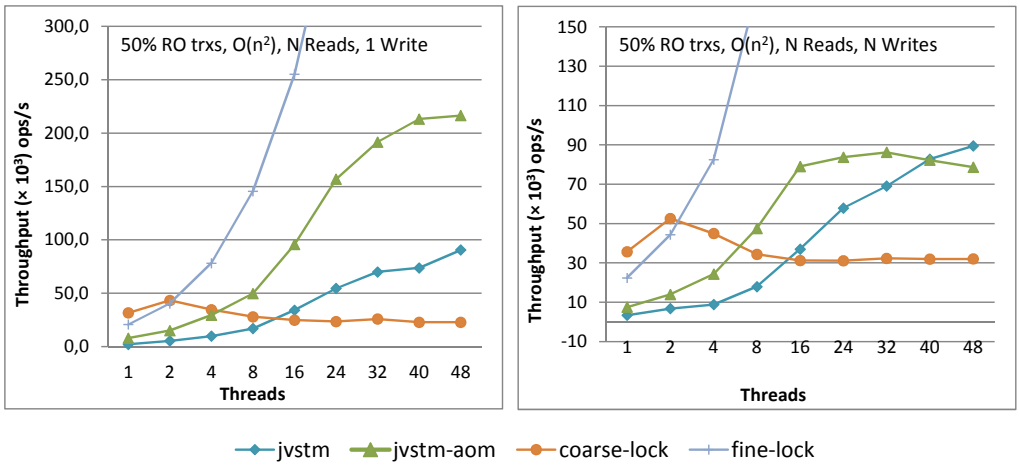


Figure 6.10: The JWormBench throughput for JVSTM and locks, for two different workloads. The latter workload performs the same number of write accesses as read accesses.

6.4.2 Memory Consumption Evaluation

I do not include the JWormBench in these experimental analysis, because this benchmark does not use large data sets and, therefore, are not visible in this case the effects of the AOM over the memory heap size.

I ran my tests with just one worker thread for approximately thirty minutes and I collected the results from the verbose garbage collector (GC) output. In all results I show the values of three parameters during the execution of each benchmark: (1) the combined size of live objects *before GC*, (2) the size of live objects *after GC*, and (3) the total *available* space, corresponding to the maximum heap size.

STMBench7 benchmark

In the results of Figure 6.11, we can observe that there is a reduction of around 20% in memory consumption of live objects after GC, when we execute the JVSTM with the AOM, except for the write-dominated scenario where we expect to have just a few number of objects in the compact layout due to the high rate of write operations.

Most visible is the reduction of the application footprint, which saves almost 50% of memory space in the case of the read write workload.

Vacation benchmark

In the results of Figure 6.12 on page 111, we can observe a significant reduction in the three parameters of the GC, around 1 Gb of memory space, when we execute the JVSTM with the AOM. In this case we do not register any differences between the results collected for both workloads of the Vacation benchmark, because the workloads use the same update rate and they just differ in the contention level.

6.5 Summary

In this Chapter I introduce an *adaptive object metadata* approach into an object-based design for a multi-versioning STM, and I provide experimental results that show an improvement in the performance of workloads where the number of objects written is much lower than the total number of transactional objects.

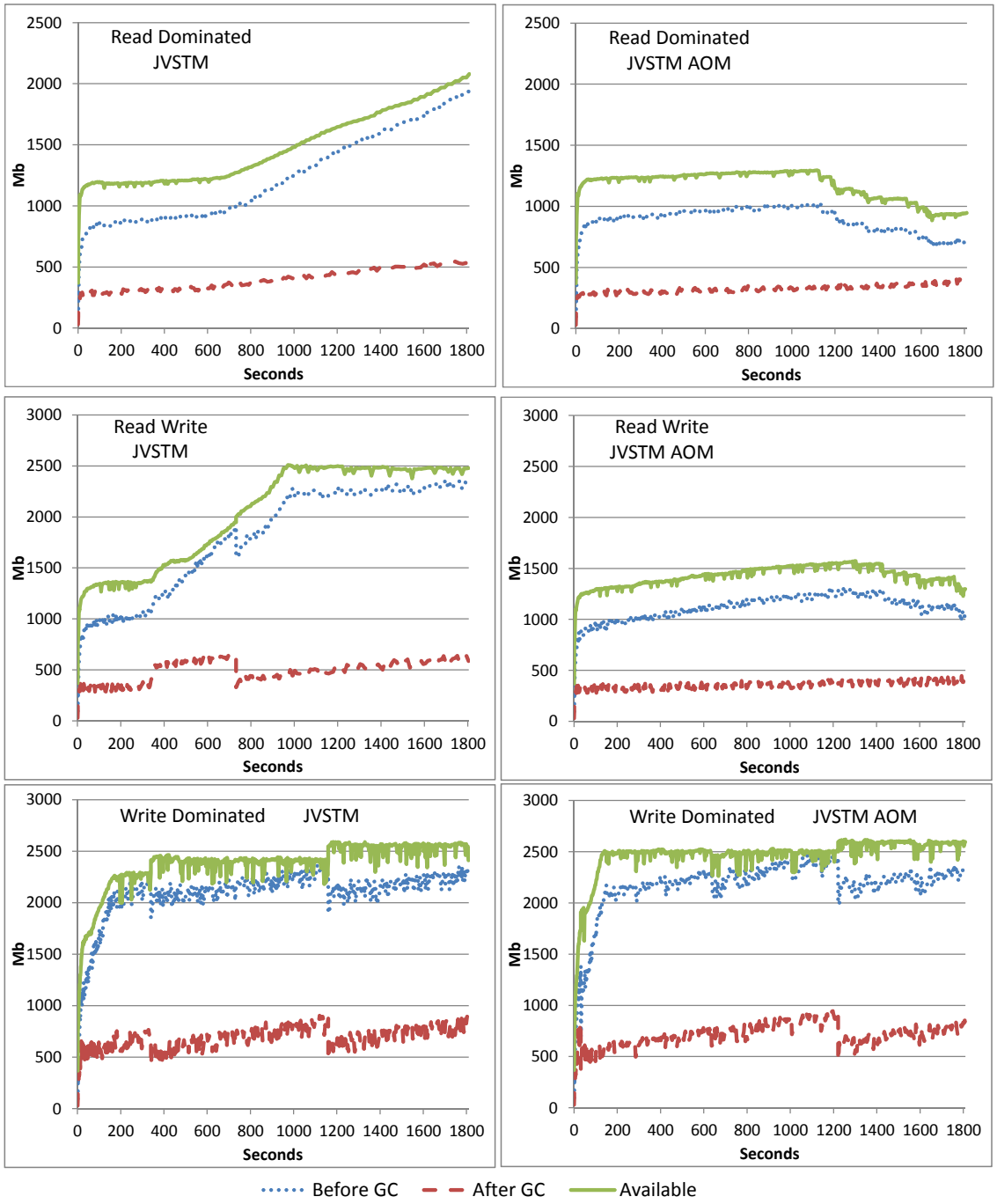


Figure 6.11: The STMBench7 memory consumption for JVSTM with, and without AOM, in three different workloads, without long traversal operations.

Furthermore, the use of a lock-free commit algorithm requires an in-depth analysis of the possible inter-leavings of concurrent layout transitions, which I also discuss to prove the correctness of the new AOM algorithm while giving the rationale for my design along that discussion.

Finally I analysed two possible implementations of my proposal that follow two dif-

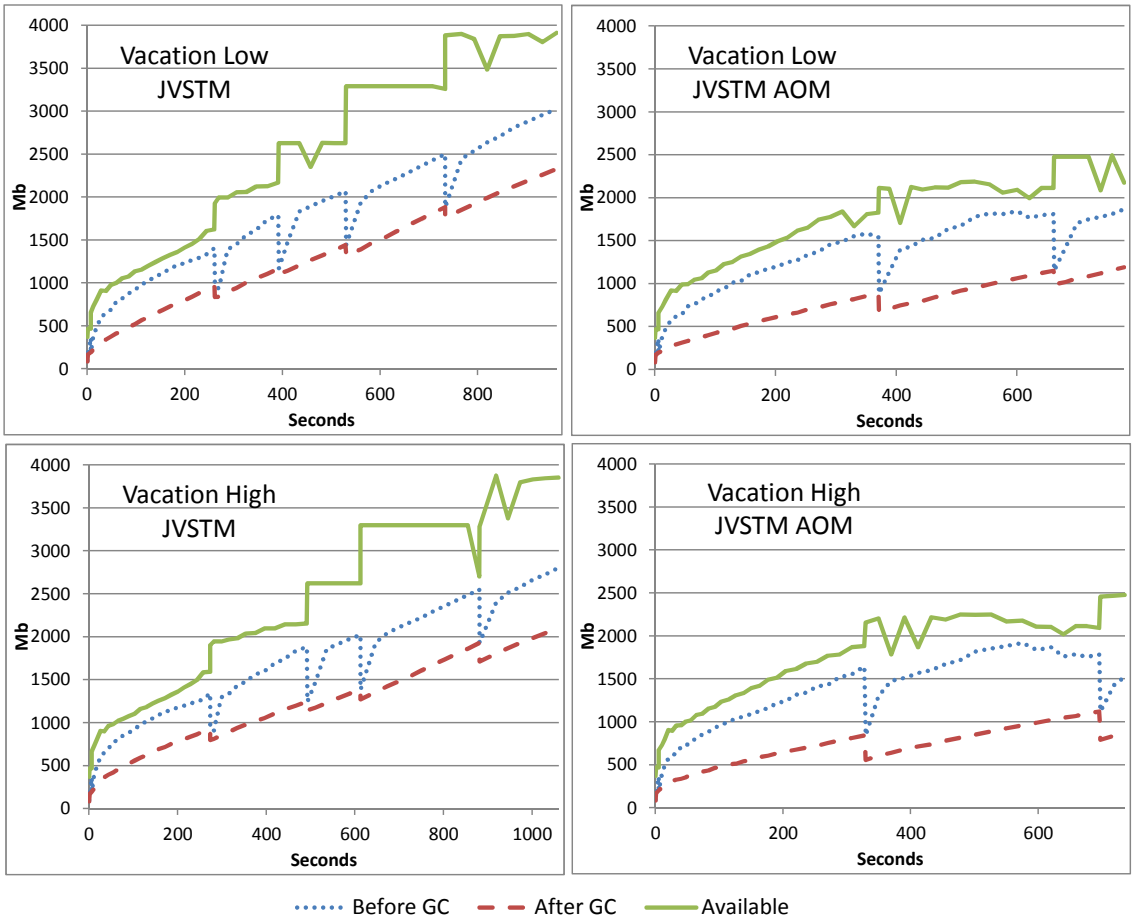


Figure 6.12: The Vacation memory consumption for JVSTM with, and without AOM, in the low and high contention workloads.

ferent approaches: (i) at the just-in-time compiler level, and (ii) through Java bytecode instrumentation. Although the latter approach was more effective, I believe that using a different integration solution at the virtual machine level could be even more efficient and suppress some handicaps of the bytecode instrumentation.

Chapter 7

Combining LICM and AOM

In the previous chapters I presented two runtime optimizations techniques (LICM and AOM) to reduce the STM-induced overheads when accessing objects that are not under contention. The LICM optimizes an STM to avoid useless STM barriers when accessing transaction local objects (corresponding to non-shared objects), whereas the AOM optimizes a multi-versioning STM, such as JVSTM, to avoid further STM metadata for non-contended objects. So, when an STM is enhanced with LICM it may avoid useless STM barriers, but it still incurs in the overhead of the STM metadata added by some STMs, such as the JVSTM. On the other hand, when an STM is enhanced with the AOM it may avoid the overheads of additional STM metadata for non-contended objects, but it still incurs in the overhead of useless STM barriers for non-shared objects, such as transaction local objects.

So, through the combination of both techniques: LICM and AOM, I expect to avoid simultaneously the overheads that are mitigated by each technique individually. Note that only with the combination of both techniques we can initialize the life cycle of a transactional object in the compact layout of the AOM. Otherwise, even for transaction local objects, when the instance constructor initializes the object's proper fields, later, when the transaction commits it would leave that object in the extended layout.

The work that I describe in this Chapter explores how the combination of two optimization techniques can achieve a better performance than any of those techniques individually. Since my proposals were developed and integrated in Deuce STM, I use those optimizations techniques separately or together.

So, the LICM and the AOM techniques complement each other in a synergistic way, substantially reducing the overheads of an STM in large-scale programs. As we shall see,

my approach can solve one of the major bottlenecks that reduces the performance in many realistic applications and simultaneously preserve the transparency of an STM API, as shown with its implementation in Deuce.

In the following Section I show how the JVSTM enhanced with both LICM and AOM (*jvstm-aom-licm*) achieves the best performance for all evaluated benchmarks in comparison with any other solution of the JVSTM. After that, in Section 7.2, I compare the performance of *jvstm-aom-licm* with other synchronizations approaches, namely, lock-based synchronization, other STM algorithms (LSA and TL2) and manual instrumentation with the JVSTM (*jvstm-manual*).

7.1 Enhancing the JVSTM with both LICM and AOM

In this section I evaluate the performance of the JVSTM in four different scenarios: (1) the original unmodified implementation of the JVSTM (*jvstm*), (2) the JVSTM enhanced with AOM (*jvstm-aom*), (3) the JVSTM enhanced with LICM (*jvstm-licm*), and (4) the JVSTM enhanced with both techniques (*jvstm-aom-licm*).

In my experimental tests, I used the STMBench7 benchmark, the Vacation application of the STAMP benchmark and the JWormBench.

7.1.1 Vacation

I present in Figure 7.1 the results obtained for the two workloads of the Vacation benchmark. I include the results of the TL2 for comparison with the JVSTM but I omit the LSA, which presents similar performance to the TL2. These tests were performed on a machine with 4 AMD Opteron^{TM} 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the 1.6.0_33-b03, running on Ubuntu with Linux kernel version 2.6.32.

As we can see, in this benchmark LICM shows no benefits for the JVSTM, because the transaction local objects still incur in additional metadata that penalizes the corresponding memory accesses (in the case of the TL2 and the LSA, there is no in-place metadata associated with the transactional objects). Yet, these results show the lightweight nature of LICM: Its penalty on the performance of the JVSTM (either with or without AOM) is just below 10% in the worst case and is negligible in most cases.

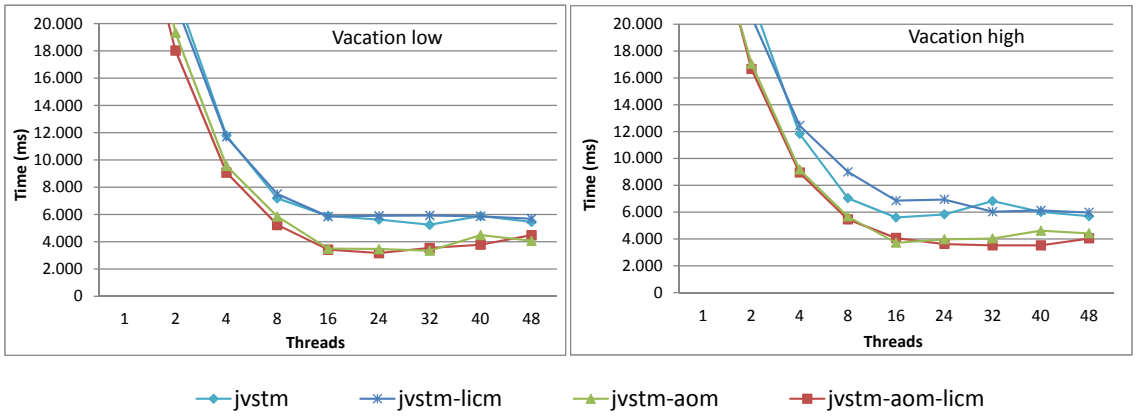


Figure 7.1: The results for Vacation with JVSTM, in the two workloads (low and high contention).

On the other hand, the AOM improves the performance of the *jvstm* and the *jvstm-licm* in both workloads of the Vacation: As shown in Figure 7.1, both *jvstm-aom* and *jvstm-aom-licm* reduce the time taken to execute the benchmark to half of the time taken by *jvstm*.

7.1.2 STMBench7

As shown in Chapter 5, the majority of the barriers elided by LICM in the STMBench7 access instances of classes related to the iterators of the `java.util` collections. So, it is no surprise that using capture analysis has a great impact on the performance of the STMBench7 with Deuce, as shown in Figure 7.2, where we can see that LICM improves the performance of both enhanced STMs.[1]

Finally, when we combine the LICM with the AOM, we can observe that LICM can take one step further and still improve the performance, getting an improvement of up to 2.5-fold in performance of the *jvstm-aom* and of up to 4-fold in the performance of the *jvstm*. In fact, the *jvstm-aom-licm* is the best synchronization approach in the STMBench7, except for the write-dominated workload for which both optimization techniques cannot improve the performance of the baseline JVSTM for more than 8 threads.

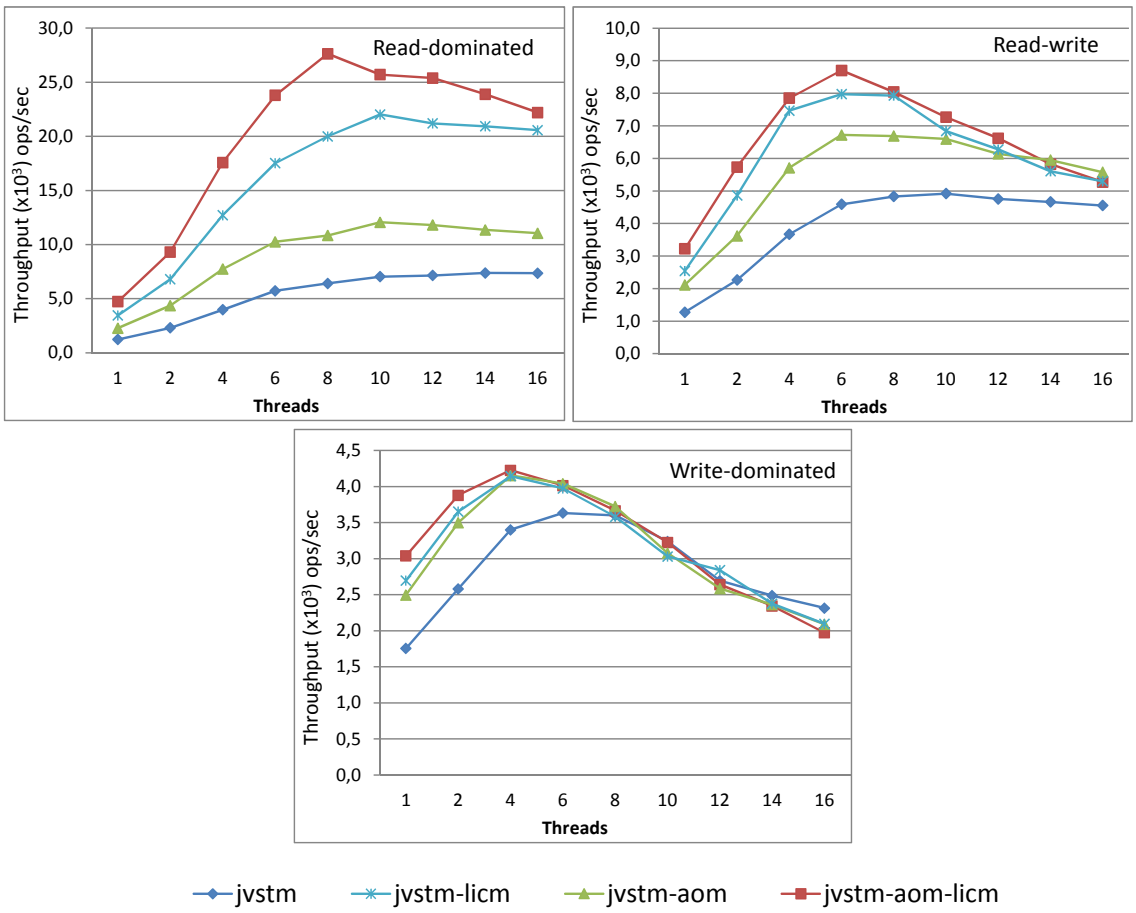


Figure 7.2: The results for STMBench7 with JVSTM, in the three available workloads, without long traversal operations.

7.1.3 JWormBench

I ran this benchmark with the same configuration presented in Chapters 6 and 7: a *world* with 1024 *nodes* and 48 *worms* with a head's size varying between 2 and 16 nodes, corresponding to a total number of nodes between 4 and 256 operated per transaction. In comparison with the Vacation workloads, this configuration of the JWormBench uses shorter transactions. In Vacation a transaction operates on up to 256 items, each one performed on 90% or 60% (depending on the workload) of the total number of records (configured with a value of 262144 records).

The results of Figure 7.3 confirm that either LICM or AOM can improve the performance of the JVSTM and both together can achieve a better performance than any of the previous single techniques. More precisely and for the read dominated workload

[1] These tests were performed on a machine with two quad-core Intel Xeon CPUs E5520 with hyper-threading, resulting in 8 cores and 16 hardware threads.

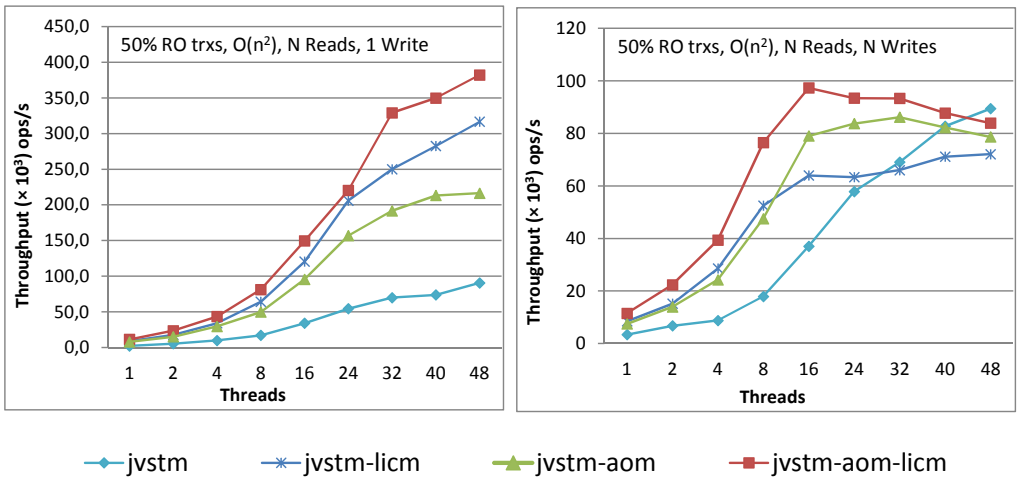


Figure 7.3: The JWormBench throughput for JVSTM for two different workloads. The workload on the right performs the same number of write accesses as read accesses.

the AOM improves the performance of the JVSTM by 2.5-fold and the LICM improves the performance by 3.5-fold. When we enhance the JVSTM with both techniques we get an improvement of up to 4.2-fold in performance.

Yet, when the number of write operations increases too much, as in the case of the $O(n^2), NReads, NWrites$ workload, the performance of the enhanced versions of JVSTM degrades for a higher number of threads, due to the big overhead of its read-write transactions with a high number of write accesses, which prevents any of the JVSTM enhanced versions to scale for more than 16 threads.

7.2 Comparing *jvstm-aom-licm* with other approaches

In this section I compare the best JVSTM algorithm enhanced with both techniques (*jvstm-aom-licm*), with LSA and TL2 enhanced with LICM. Both of these latter STMs were available with the original Deuce framework distribution, whereas the JVSTM was integrated in the enhanced version of Deuce STM (for more information about this new version of Deuce STM, see Appendix C).

In these tests I used the same experimental environment including machines, benchmarks and their configurations, as described in Section 7.1.

Given that the STMBench7 and the JWormBench also provide a medium/fine-grained locking synchronization approach, I also evaluated these strategies in my experimental tests for comparison with the STM-based approaches.

In Chapter 2, I presented the results obtained with a manually instrumented version of the STMBench7 benchmark to show that it was possible to achieve better performance with an STM, provided that the programmer collaborated in identifying which objects needed to be instrumented by the STM. In fact, I established as a goal for my work to be able to obtain comparable performance without requiring the intervention of the programmer.

The results depicted in Figures 7.4, 7.5, and 7.6 compare the results of the enhanced version of the JVSTM (*jvstm-aom-licm*) with the results of manually instrumenting the benchmarks with the JVSTM (*jvstm-manual*). These results show that the goal was achieved: Instrumenting automatically the entire program while using the LICM and AOM allows us to obtain performance results comparable and even better than with the manual instrumentation.

The better results of the automatic approach may be surprising at first, but they may be explained by a series of factors: (1) the manual approach eliminates useless barriers, much in the same way as LICM, but it does not benefit from the performance benefits of the AOM approach; (2) the *jvstm-aom-licm* uses an object-level conflict detection approach, whereas the *jvstm-manual* uses a word-level conflict detection; and (3) the manual approach is applied statically at the class level, whereas the automatic approach decides on a per-object basis whether to do the optimizations or not.

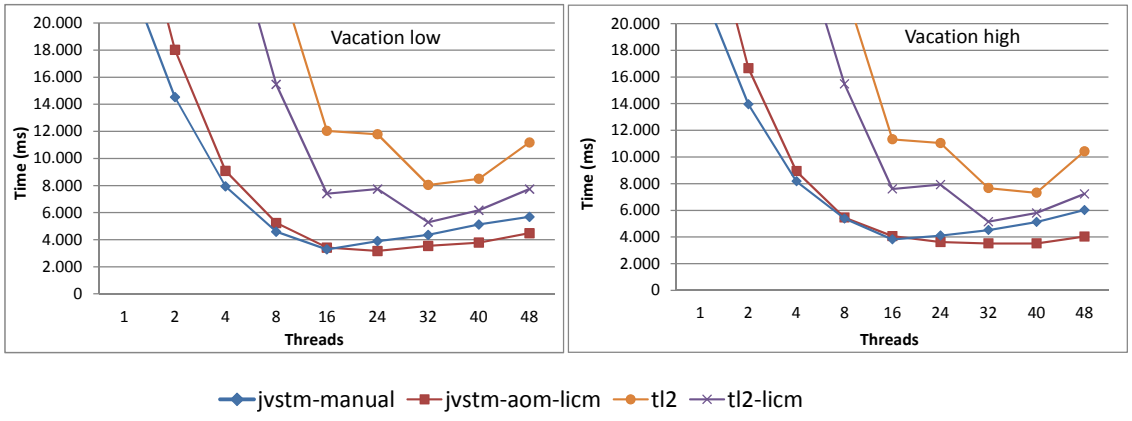


Figure 7.4: The results for Vacation with TL2 and JVSTM, in two workloads (low and high contention).

For readability reasons I omitted the LSA in the results of Figure 7.4, which presents similar performance to the TL2 with and without LICM.

In the case of the STMBench7, and for comparison with my optimized version of the JVSTM, I include the *lsa-licm* in my analysis. Even though LSA performs better

than TL2 with capture analysis, its results are still far from the results obtained with JVSTM, which is the most performant STM in the STMBench7, as shown in the results of Figure 7.5. In fact, *jvstm-aom-licm* gets better results than the medium-lock synchronization approach for the read and read-write dominated workloads. In this case, JVSTM benefits from its lock-free commit algorithm and from the lazy ownership acquisition approach.

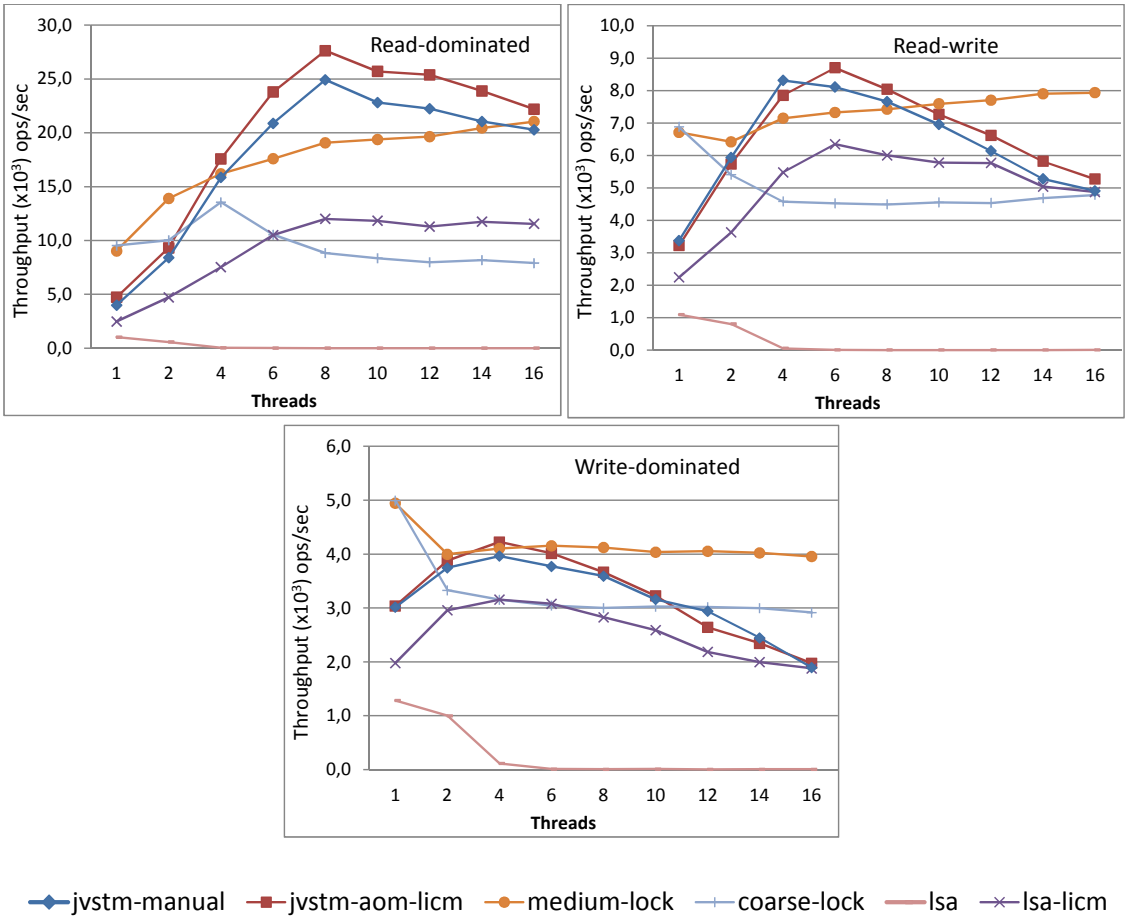


Figure 7.5: The results for STMBench7 with LSA, JVSTM, and locks, in the three available workloads, without long traversal operations.

For the JWormBench and as we observed in the results of Figure 7.3, the LICM and AOM help to improve the performance of the JVSTM. Yet, this enhancement is not enough to get the *jvstm-aom-licm* close to the performance of the *lsa-licm*, as shown in the results of Figure 7.6. Unlike what happened for the STMBench7 and the Vacation, the LSA with capture analysis performs better than JVSTM in the JWormBench, because these workloads have smaller transactions.

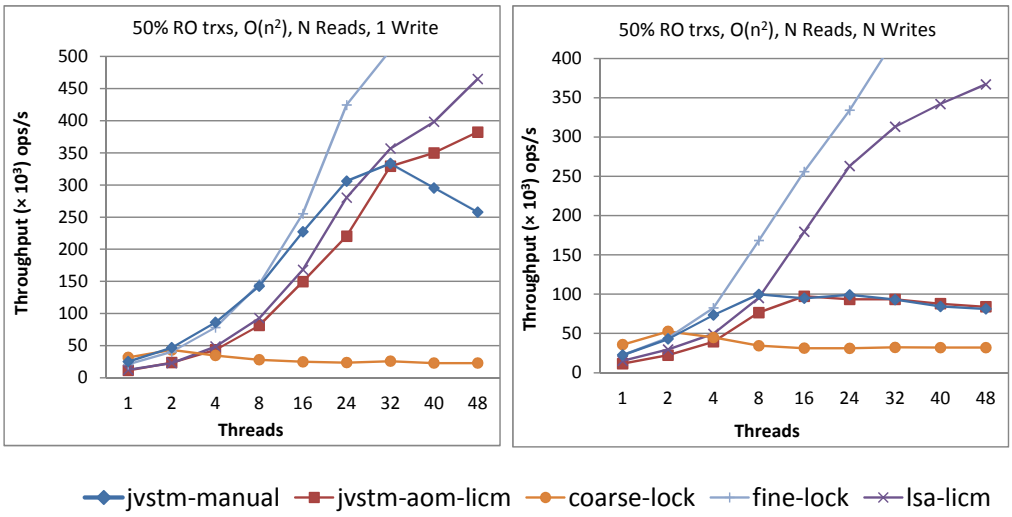


Figure 7.6: The JWormBench throughput for LSA, JVSTM, and locks, for two different workloads. The workload on the right performs the same number of write accesses as read accesses.

7.3 Summary

In my tests I observed that both LICM and AOM are able to improve the performance of an STM that is enhanced with either one of these techniques. Simultaneously, I observed that individually none of these techniques can achieve the performance that is achieved by an STM enhanced with the combination of both techniques. So, I claim that each of the techniques can still optimize the remaining opportunities that were left by the other technique in some benchmarks, because they aim to solve distinct kinds of overheads. In the worst case, the combination of the techniques does not degrade the benefits of the best one.

My approach can solve one of the major bottlenecks that reduces the performance in many realistic applications and simultaneously preserve the transparency of an STM API, as shown with its implementation in the Deuce STM framework. Although LICM adds a minor overhead in memory space to all transactional locations and an extra indirection for arrays, we still get a huge speedup in the Vacation and the STMBench7 benchmarks. In fact, for the first time in the case of STMBench7, I am able to get better performance with an STM than with the medium-grain lock strategy.

Finally, my experimental results confirm my expectations that it is feasible to use STMs for real-world–sized applications, provided that the STM is not adding barriers to unnecessary memory locations. In fact, I believe that integrating the LICM and the

AOM together in a managed runtime may further reduce the overhead of my approach and provide a significant boost in the usage of STMs.

Chapter 8

Conclusions

Programming languages evolved to simplify the development of parallel applications. The most popular environments, such as Java and .Net still provide the same abstractions to solve a common problem in concurrent programming—*shared memory synchronization*. Typically, these abstractions are intrinsic to the programming languages—e.g. `synchronized` keyword in Java and `lock` in C#—and are implemented internally through lock-based solutions, which may reduce parallelism when applied naively. To bridge the gap of alternative abstractions for shared memory synchronization, several frameworks have emerged to complement the APIs provided by managed runtime environments. In this context, software transactional memory is one of the most promising techniques used in the implementation of these frameworks.

Despite all the benefits of an STM, it still incurs in unacceptable overheads and, in many cases, an STM is unable to efficiently solve the shared memory synchronization problem, in comparison to a lock-based solution. In fact, an STM adds further indirections and extra metadata to all memory locations managed by an application. And these are some reasons why accessing a transactional location requires orders of magnitude more machine cycles than a simple access to a conventional memory location. So, as the memory heap managed by an application becomes larger, also the overheads induced by the use of an STM become higher, because the space required by the STM metadata is proportional to the number of transactional memory locations.

In my work I am particularly interested in the optimization of memory transactions for large-scale programs where the side-effects resulting from the STM-induced overheads are more harmful in the overall performance and memory space of these applications. The idea behind my research was to find a solution that removes all the

STM overheads for the majority of transactional locations and the cost of accessing these locations should become similar to accessing non-transactional memory. So, if the overheads of an STM become circumscribed to a small fraction of the overall memory managed by an application then these overheads may become negligible.

To tackle this problem first I studied what kind of overheads I could suppress in transactional applications and then I explored different optimization techniques to avoid these overheads. Finally, I confirmed the usefulness of my proposal for a diversity of benchmarks, some of them well-known for being particularly challenging to STMs, which reinforces the advantages of my techniques to make STMs as a valid alternative to lock-based synchronization in large-scale programs.

Next, I present a detailed description of the contributions of this dissertation and directions for future work.

8.1 Main Contributions

In this section I summarize the main contributions that result from the research work that I describe in this dissertation: (1) JWormBench—A flexible benchmark for transactional synchronization; (2) An heterogeneous API to optimize memory transactions; (3) LICM—Lightweight Identification of Captured Memory; (4) AOM—Adaptive Object Metadata; (5) Use of a fast access path for non-contended objects, and (6) Support for in-place metadata.

JWormBench—A flexible benchmark for transactional synchronization

To clearly understand the overheads incurred by an STM and what kind of optimizations we could perform, we need applications that gather a couple of characteristics that we just meet individually in different benchmarks. Namely, I was looking for a benchmark that provides simultaneously: (1) a correctness test (i.e. sanity check) that verifies if the STM made a valid synchronization (e.g. provided by STMBench7 and LeeTM); (2) different kinds of mathematical functions with different degrees of complexity to stretch STMs along different axis (e.g. provided by WormBench); (3) long transactions that stress the STM under intensive workloads (e.g. provided by STMBench7 and Vacation); (4) fine-grained lock synchronization for comparison with an STM; (5) a flexible API that allows the integration of different STM implementations (e.g. provided by STMBench7). Given the nice features of the WormBench, I decided to port

it to Java and further extend it with the missing mechanisms. JWormBench extends the original benchmark in several ways and has some key differences: (1) it replaces the STM integration approach based on macros, with a new extensible API that allows easy integration with different STMs; (2) it implements the correctness test based on the results accumulated on each thread's private buffer; (3) it decouples the number of threads from the environment specification allowing to keep the same contention along different numbers of worker threads; (4) it allows to specify the proportion between each kind of operation to produce workloads with different ratios of update operations. These new parameters allowed me to evaluate STMs under new test conditions, which were not available yet, and in this way I could find the most notorious bottlenecks that contribute to the performance degradation of an STM.

A heterogeneous API to optimize memory transactions

Several other researchers proposed different kinds of heterogeneous APIs to help reduce the overheads of an STM. In common, all these solutions allow the programmer to specify which blocks of code should not be instrumented by the STM compiler. My proposal follows a different approach that allows programmers to specify their intentions over memory locations, instead of on code blocks. So, my solution of a heterogeneous API provides two distinct aspects: (1) it allows a fine-grained control of instrumentation at the fields level; (2) for the first time in a managed environment, it also includes arrays as locations that can avoid the STM instrumentation. Thanks to these two features I could identify the most harmful overheads of an STM compiler in some challenging benchmarks. I also observed the speedup achieved by an application synchronized with an STM when we suppress these overheads, which came close to the performance of the best fine-grained lock synchronization approach. These results opened good perspectives about the use of STMs as an alternative to fine-grained lock synchronization in large-scale programs.

LICM—Lightweight Identification of Captured Memory

The previous contributions allowed me to confirm that it was feasible to improve the STM performance. Yet, there are some situations where we cannot predict in advance if the STM barriers are required or not, and, thus, we cannot suppress those barriers beforehand. So, my next step was to develop a runtime technique that was able to accurately identify whether an object is not shared, such as for transaction local objects,

and therefore avoid useless STM barriers. In this case, one ot the requirements of my new approach was to keep the STM API transparent.

Although over-instrumentation was a well-known consequence of the transparent synchronization, there were no automatic mechanisms that efficiently avoid these overheads. My proposal of a technique for *lightweight identification of captured memory*—LICM—was the first automatic mechanism that made STM's performance competitive with the best fine-grained lock-based approaches. A key aspect of this approach is that LICM keeps the STM API transparent and does not require any further intervention from the programmer.

AOM—Adaptive Object Metadata

In my work I was particularly concerned with the optimization of an STM in read-dominated scenarios, which I believe are most commonly found in real-world–sized applications. In this context, even when an object is shared it may be frequently non-contended and, thus, in these cases we may eliminate some of the STM overheads of accessing non-contended objects. Regarding this problem, my idea was to eliminate the extra STM metadata for non-contended objects. To that end, I explore the *object model* provided by the managed runtime environments and instead of a unique layout, which includes the STM metadata, I researched the effects of providing an *adaptive layout* that swings objects back and forth between two different object layouts: a compact layout, where no memory overheads exist, and an extended layout, used when the object may be under contention. My approach not only improves the STM performance, but also reduces the size of the memory heap managed by an application.

Fast access path for non-contended objects

Although LICM improved the performance of all baseline STMs, I noted that: (1) in the STMBench7 it was still lagging behind of the best results achieved with the JVSTM manually transactified; (2) in certain benchmarks, such as the Vacation, the LICM does not achieve the same speedup for the JVSTM as we observe in other benchmarks, such as the STMBench7. From these two observations I analysed the behaviour of the JVSTM and I concluded that LICM cannot completely suppress the overheads of useless STM barriers in JVSTM due to the additional metadata associated with all transactional locations. So, by combining both LICM and AOM, I can simultaneously avoid the additional tasks performed by STM barriers and also the further indirections induced

by the extra metadata, and thus provide a *fast access path* for non-contended objects. This new approach enhances the JVSTM and for the first time makes the performance of Deuce with JVSTM even better than a fine-grained lock approach in the STMBench7. At the same time, this new solution achieves the same performance of the JVSTM manually transactified, but within an automatic instrumentation engine such as Deuce STM.

Support for in-place metadata

All mentioned techniques (multi-versioning, LICM, and AOM) require additional support from the execution environment to store metadata in-place, which was not available yet in any STM compiler. In my work I also bridge this gap and I researched two different approaches to make a managed runtime environment include additional metadata within memory locations. In my first implementation I extended the object model of a Java runtime (in this case the Jikes RVM [Alpern *et al.* , 2005]) to natively integrate the transactional metadata of a multi-versioning STM—JVSTM. Yet, this implementation also required modifications to the just-in-time compiler, which affected the optimization process and compromised the overall performance of the virtual machine. So, in my second implementation I followed a different approach and I integrated a similar mechanism through a bytecode instrumentation engine instead of the just-in-time compiler. For that purpose I modified a well-known STM engine for Java—Deuce STM—and I integrated full support for in-place metadata. My implementation is innovative in the following aspects: (1) it maintains the Deuce API; (2) it guarantees backwards compatibility with existing applications and STMs for Deuce; and (3) it provides the ability to enhance any existing STM with additional filtering features without requiring either its recompilation or any modification to its source-code.

8.2 Future Research

Even though the main goal of the work described in this dissertation is quite easy to summarize—to optimize memory transactions—it is simultaneously quite ambitious, especially when I compare my proposals with the best fine-grained locking solutions in some of the more challenging benchmarks. Nevertheless, I believe that some of the techniques that I proposed and developed could be further optimized and therefore produce even better results.

Next, I present some of the areas which I believe would benefit from further work in the future.

Optimizing transactional read operations

In my work I assume that real-world–sized applications are dominated by read operations and the number of write operations is just a small fraction of the total number of memory accesses performed during the application's life cycle. So, the expectation that I confirm in my work is that if we optimize read operations for the majority of transactional locations then we will have a substantially speedup in the overall performance of an application. Both techniques—LICM and AOM—allowed me to optimize read operations. In fact, for transactional local objects these techniques can completely remove the overheads of an STM and reading a transaction local object has the same overhead of reading a non-transactional one, plus the overhead of an additional identity comparison. Yet, for non-local objects that are seldom written I could not completely suppress the overheads of the STM. In this case, we are able to avoid the additional metadata, but we must still keep track of the read-set. This means that for seldom written objects, which take advantage of the compact layout provided by the AOM, we can directly read the object fields but we must log this operation in the transaction's read-set. So, the remainder overhead of a read operation corresponds to the cost of maintaining the read-set.

Again, assuming that the majority of the objects are seldom written, then we may improve the application's performance if we avoid the read-set log for compact objects. Thus, we may achieve the same reading performance for seldom written objects, as for transaction local objects. Yet, this kind of optimization requires additional validation at commit phase to guarantee that a transaction that has read a compact object still got a valid snapshot of memory. So, there is a tradeoff between the speedup obtain from suppressing the read-set log and the overhead of extra validation at commit phase. The implementation and possible adoption of this optimization could be a enhancement for the AOM design and still requires future research.

Layout transitions policies

One of the problems regarding the application of the AOM to read-write workloads is related to the overhead of the layout transitions. If an object is being extended and reverted back repeatedly, then the benefits of accessing a compact object are overshadowed by the accumulated cost of the layout transitions. In fact, we should not revert an object to the compact layout if it will be written soon. One possible approach is to keep an object in the extended layout by a quiescent period before reverting it to

the compact layout. In my work I was specially focused on read-dominated workloads and I did not observe any advantage of delaying the layout reversion. However, this analysis was biased by the circumstances of the workloads and I believe that different analysis can be made for other kinds of workload configurations. In that case, more complex reversion policies may reduce the overheads of layout transitions and enhance the benefits of the AOM.

Influence of the object's size in the AOM overheads

Object layout transitions are the core primitive of the AOM. These operations encompass the copy between the object fields and its versions. Obviously, the overhead of this copy is directly dependent on the size of the object. However, all tested benchmarks use domain classes with few fields and, thus, I did not expect a big overhead from the layout transitions. Nevertheless, it would be interesting to evaluate the cost of layout transitions according to each type of transactional object of a benchmark and, maybe for certain classes there would be advantages in delaying the reversion, which probably could prevent the object from being reverted. Concluding, the decision of immediately reverting an object or delaying its reversion should be also assessed taking the size of the object into account. Therefore, it is to be expected that in the future a more deep study analyzes the influence of the object's size in the AOM overheads and develops a more efficient policy for the layout transitions.

Adaptive Memory Accessors

The AOM establishes further validation to all memory accesses. In this case it includes an identity comparison with the `null` value, to check whether an object is in the compact or extended layout. Although my goal was to avoid the STM overheads for non-contended objects, we still incur in the overhead of the branch resulting from the AOM validation. The inclusion of adaptive memory accessors together with the adaptive layout could suppress the overhead of the AOM validation. As we swing between layouts, we could also swing between the implementation of memory accessors. For that purpose, and assuming that all object fields have private access, then all methods that access instance fields should provide two implementations: one that accesses the proper fields of the object and another that accesses the versioned history without previously checking if it exists or not. So, when an object changes its layout it must also update its methods table to point to the correct implementations of the methods corresponding to the layout of the object. This approach raises other problems regarding the

built-in optimizations of the just-in-time compiler such as methods inlining and, thus, it requires further research to its correct adoption in the AOM.

These topics are, by no means, an exhaustive list of all the research work that may follow from what is described in this dissertation.

8.3 Concluding Remarks and Future Directions

In my work I start to show how a naive implementation of an STM with a transparent API is harmful for a large-scale program and after that, I also show how appropriate optimization techniques may effectively avoid the STM induced overheads and enhance the overall performance of an application synchronized with an STM. In fact, and for the first time, my approach allows me to make STM's performance competitive with the best fine-grained lock-based approaches in some of the more challenging benchmarks. These results show that it is feasible to use STMs for real-world-sized applications.

I developed my solution for the JVM with the support of a well-known STM instrumentation engine for Java—Deuce STM. However, I believe that the native support of an STM at the managed runtime level, rather than at the bytecode instrumentation level, could suppress some limitations and avoid the workarounds of my solution, such as for supporting arrays, and give further gains to this integration. For instance, we could take advantage of the object layout and optimize the STM metadata storage directly using the object's header portion. Another facility available at the virtual machine level is the direct access to the processor instructions set. In this case, we could take advantage of Hardware Transactional Memory—HTM—that is available in some architectures such as the Haswell from Intel, which provides the RTM (Restricted Transactional Memory).

The RTM API includes specific instructions to mark the start and the end of a transactional code region. All memory accesses within the transaction region—transactional accesses—are maintained in the transaction's private buffers: read-set and write-set, instead of directly accessing memory locations. This approach complies with the idea of a transparent API where all transactional accesses are implicit, instead of requiring an explicit kind of instruction for accessing memory locations.

So, the RTM API is a low-level programming alternative to lock-based synchronization mechanisms and thus, it may be used to synchronize critical sections with a

limited number of memory accesses that does not exceed the maximum size of the internal buffers. This approach is not intended for higher-level domain languages that should encompass many memory accesses that easily lead to the overflow of the HTM buffers. For instance, in the scope of a managed runtime environment, such as the JVM, most of the code executed within a transaction region might not be part of the domain logic, but part of the runtime services, such as the garbage collector. So, the number of STM barriers performed within a transaction region can easily exceed the maximum capacity of the transaction's private buffers, which in turn aborts the transaction.

Some of the optimization techniques proposed in this dissertation avoid useless STM barriers. To that end, we take advantage of the domain application knowledge to suppress unnecessary STM barriers, such as for transaction local objects. To apply this kind of optimizations to an HTM we need to have fine-grained control over which memory accesses are transactified, or not. A transparent API such as the one provided by Haswell does not allow us to use these techniques. For the previous reasons I do not consider the current RTM API exploitable from the JVM point of view. Yet, I believe that the hardware support for TM may contribute to further improve the performance of an integrated TM solution in a managed runtime environment, such as the JVM. To that end, HTM implementations should provide explicit instructions for transactional memory accesses that let the compiler decide whether it should use, or not, an STM barrier instead of a conventional memory access.

The development of large-scale applications is still requiring alternatives to lock-based synchronization and an HTM by itself does not provide a valid option yet. However, managed runtime environments may provide an abstraction over an HTM and directly integrate the support of HTM to provide a more effective software transactional mechanism, such as Deuce STM, but using hardware transactional barriers instead of software-based barriers. The combination of this approach together with the optimization techniques proposed in this dissertation may lead to an efficient use of an STM in the JVM.

To finish my dissertation, I would like to reinforce my claim that managed runtime environments need to provide an efficient alternative to lock-based synchronization, which does not exist yet; I believe that my work, which defined new runtime optimization techniques to STMs, is a significant step toward achieving this goal.

Appendix A

JWormBench

The lack of realistic benchmarks is one of the factors that has been hampering the development, testing, and acceptance of Software Transactional Memory (STM) systems. Many of the developments made on STMs are evaluated on micro-benchmarks [Herlihy *et al.* , 2006; Dice *et al.* , 2006; Riegel *et al.* , 2006; Felber *et al.* , 2008], which are fairly criticized by some researchers (e.g. [Cascaval *et al.* , 2008]) that question the usefulness of STMs, given their lack of demonstrable applicability to real-world problems.

Within the set of constraints presented by the majority of well-known benchmarks for STMs there are two main aspects that make the evaluation and comparison of different synchronizations approaches difficult: (1) the lack of configurability and flexibility on the integration of new synchronization mechanisms, and (2) the limited variety of operations adopted by a workload and shortage extensibility to other kinds of operations.

In this Appendix I describe a port that I made of the WormBench benchmark [Zyulkyarov *et al.* , 2008], from C# to Java. From the existing characteristics of WormBench I was particularly interested in the simplicity of the domain model and the ability to extend it with new kinds of operations among a wide range of functions with different levels of complexity. This port extends the original benchmark in several ways, making it more useful as a testbed for evaluating STMs [Carvalho & Cachopo, 2011]. Moreover, my port, which I called JWormBench[1], has some key differences from the original WormBench:

[1] Available at https://github.com/inesc-id-esw/jwormbench

1. Unlike the WormBench, which follows an STM integration approach based on macros, the JWormBench was designed to be easily extensible and to allow easy integration with different STMs. For that purpose, the JWormBench has a new solution architecture based on *inversion of control*, *abstract factory*, and *factory method* design patterns [Gamma *et al.* , 1995];

2. The core engine of the JWormBench benchmark is deployed in a separate and independent library, whose features can be extended with other libraries;

3. Unlike JWormBench, the WormBench distribution does not implement the correctness test (i.e. sanity check for the STM system) based on the results accumulated on each thread's private buffer;

4. In WormBench it is not easy to maintain the same contention scenario when varying the number of threads, whereas in JWormBench the number of threads is totally decoupled from the environment specification and we can maintain the same conditions along different numbers of worker threads;

5. The operations generator tool in JWormBench allows us to specify the proportion between each kind of operation, which is an essential feature to produce workloads with different ratios of update operations.

In the following section I describe in more detail the WomBench benchmark, and in Section A.2 I explain the main differences introduced in my port of that benchmark to Java—JWormBench.

A.1 The WormBench benchmark

Given the nice characteristics of the WormBench, I decided to port it to Java and further extend it to improve its usefulness. The WormBench benchmark was built to research new workloads (creating, testing, and running) for TM systems' evaluation. The main data structures in the WormBench include *worms* formed by a *body* and a *head*, moving in a shared *world* — matrix of *nodes*. Each *node* has an integer *value* and the worms' *group id* that is over that node (worms belonging to different groups should not cross through each other).

The total sum of the values of all world's nodes is the *world's state*. For read-only workloads, the world's state should remain unchanged by the execution of the benchmark.

A *worm* object has the following properties: unique *identifier*, *group id*, *speed*, *head's size*, *coordinates* of the *head*, and *coordinates* of the *body*. A *worm* provides a *move* method that receives a *direction* as parameter and moves the *worm*. Moving the worm includes two tasks: (1) updating the *coordinates* of the *body* according to the new direction, and (2) updating the *nodes* below the *body* with the *worm*'s *group id*.

The *worms* perform *worm operations* on the *nodes* under the *worms*'s *head*. The number of *nodes* under the *head* of the worm is equal to the square of its *head's size*. In Figure A.1, I show an example of a worm with a head's size of 10 nodes, corresponding to a total number of 100 nodes under the worm's head. Given that a *worm* reads all the *nodes* under its head for each *worm operation* and a transaction performs just one worm operation, this means that the length of the transaction read-set grows quadratically with the *head's size* of the *worm*.

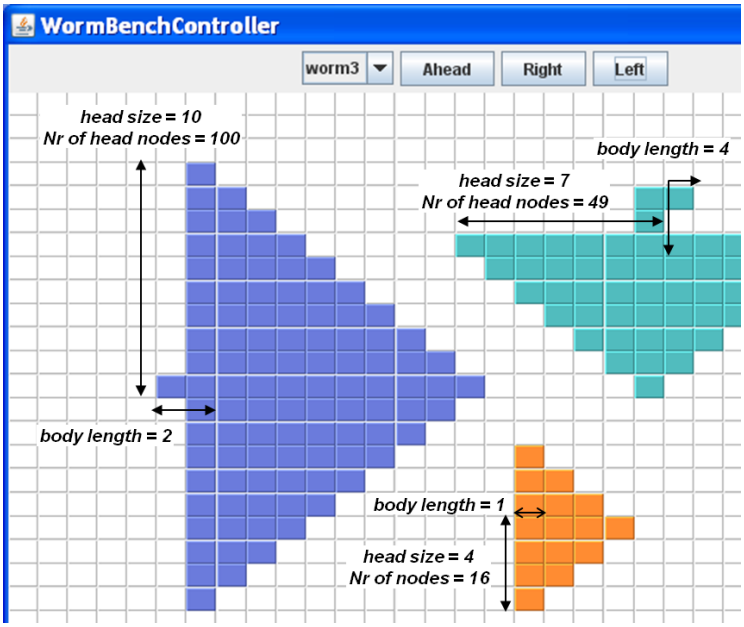


Figure A.1: Example of Worms layout in JWormBenchGui application.

In the WormBench application a *worm* object is associated with one thread and is initialized with a stream of *worm operations* and *movements* that will be performed by that thread during the execution of a workload. On each iteration, a *worm* performs the following tasks: (1) updates the *coordinates* of the worm's head; (2) reads the values of the *nodes* under the worm's head to a private buffer; (3) performs a *worm operation* on the previously read private buffer; (4) copies the values of the private buffer to the *nodes* under the worm's head; (5) moves the *worm* (updates the *coordinates* of the *body* and updates the *nodes* below the *body*). Each one of the previous tasks should be performed atomically and their source-code is annotated with macros that delimit the beginning

and the end of the atomic block. Then, each synchronization mechanism should translate those macros to invocations to the corresponding synchronization API.

#	Operation	read-set	write-set	complexity	description
0	Sum	*head's size*^{2}	0	*O(n)*	returns the sum of the nodes' values under the worm's head
1	Average	*head's size*^{2}	0	*O(n)*	returns the average of the nodes' values under the worm's head
2	Median	*head's size*^{2}	0	*O(n^2)*	returns the median value of the nodes under the worm's head
3	Minimum	*head's size*^{2}	0	*O(n)*	returns the minimum value of the nodes under the worm's head
4	Maximum	*head's size*^{2}	0	*O(n)*	returns the maximum value of the nodes under the worm's head
5	ReplaceMaxWithAverage	*head's size*^{2}	1	*O(n)*	replaces the maximum by the average of the nodes' values under the worm's head
6	ReplaceMinWithAverage	*head's size*^{2}	1	*O(n)*	replaces the minimum by the average of the nodes' values under the worm's head
7	ReplaceMedianWithAverage	*head's size*^{2}	1	*O(n^2)*	replaces the median by the average of the nodes' values under the worm's head
8	ReplaceMedianWithMax	*head's size*^{2}	1	*O(n^2)*	replaces the median by the maximum value of the nodes under the worm's head
9	ReplaceMedianWithMin	*head's size*^{2}	1	*O(n^2)*	replaces the median by the minimum value of the nodes under the worm's head
10	ReplaceMaxWithMin	*head's size*^{2}	1	*O(n)*	replaces the maximum by the minimum value of the nodes under the worm's head
11	Sort	*head's size*^{2}	*head's size*^{2}	*O(n^2)*	sort the values of the nodes under worm's head and write them back to those nodes
12	Transpose	*head's size*^{2}	*head's size*^{2}	*O(n)*	transpose the values of the nodes under worm's head and write them back to those nodes

Table A.1: 13 *worm operations* provided in the WormBench implementation. All *worm operations* with complexity *O(n^2)* could be implemented with lower complexity, but it was my intention to provide operations computationally intensive.

The WormBench implementation provides 13 types of *worm operations*, which are presented in Table A.1. These operations may be grouped in three categories:

- *read-only* — *Sum*, *Average*, *Median*, *Minimum*, and *Maximum* (from #0 to #4). Each of these operations reads all the nodes under the worm's head, corresponding to *head's size*^{2} nodes;
- *n-reads-1-write* — *Replace<read-only>With<read-only>* (from #5 to #10). Each of these operations combines two of the read-only operations described in the previous item: They use the value returned by the first operation to update the node returned by the second. Each operation makes *2*head's size*^{2} reads and one write, updating the *world's state*.

- *n-reads-n-writes* — *Sort* (#11) and *Transpose* (#12). When these operations are properly synchronized with other concurrent worm operations, they preserve the *world's state*, i.e. the total value of all nodes in the world remains the same before and after the execution of these operations.

The *worm operation Sort* and those based on the Median have complexity $O(n^2)$. These algorithms could be implemented with lower complexity, but it was the author's intention to provide operations computationally intensive.

A.2 JWormBench: A port of WormBench to Java

One of the advantages of WormBench is the ability to create new configurations of the *world*, *worms*, and *worm operations*, producing new workloads with complex contention characteristics and different transaction durations and sizes, without modifying its source-code. The JWormBench keeps this approach and adds two new features important for the research of new workloads and evaluation of STM scalability: (1) the ability to specify the proportion between different kinds of operations, and (2) the ability to set the number of worms independently of the number of threads.

Furthermore, the JWormBench provides a simple API, easy to integrate with any STM implementation in Java. So, anyone may add a new synchronization mechanism (based on STM or other), implementing the appropriate abstract types and providing those implementations to JWormBench via a configuration *module*. In the same way you can also extend JWomBench with new kinds of *worm operations* without modifying the core JWormBench library.

In this section I will describe the main features of JWormBench that differ from the WormBench implementation.

A.2.1 JWormBench applications

To increase the extensibility of JWormBench, I have designed it according to the *inversion of control* (IoC) design pattern. Then, to run a workload on JWormBench we must create an instance of the `WormBench` class and invoke the `RunBenchmark` method.

But the `WormBench` class has *dependencies* to several abstract types, whose implementations in turn depend on other abstract types and so on. So, I used Guice as the *dependency injection* framework[2] to automatically resolve and inject *dependencies* based on a configuration Guice *module* (a Java class that contributes configuration information — *bindings*).

This new architecture promotes the implementation and easier integration of new synchronization mechanisms without the need to interfere and modify the source code of JWormBench. Also note that this modular design does not add any additional overhead to the synchronization mechanism during the execution of the workload and while it is collecting measurements. The additional levels of indirection imposed by IoC and Guice will just delay the setup and will not affect the performance analysis.

We also provide the JWormBenchApp application that gives an easier way to parameterize and run a JWormBench's workload using the default implementations of the mentioned abstract types. By default JWormBenchApp uses a Guice *module* that specifies the default implementations for all abstract types. Then, each synchronization strategy must define its own Guice *module* overriding *bindings* that should provide implementations with a distinct behavior. For instance, a Guice *module* to configure Deuce STM just have to define the binding for `IStepFactory` type. On the other hand, a Guice *module* for JVSTM must define `IStepFactory` and `INodeFactory` *bindings* as shown in Listing A.1.

```
1  public class JvstmSyncModule extends AbstractModule{
2    @Override
3    protected void configure() {
4      bind(IStepFactory.class)
5          .to(JvstmStepFactory.class)
6          .in(Singleton.class);
7      bind(INodeFactory.class)
8          .to(JvstmBenchNodeFactory.class)
9          .in(Singleton.class);
10   }
11 }
```

Listing A.1: Guice *module* for JVSTM configuration

The JWormBenchApp is a Java console application that extends the JWormBench framework with some built-in Guice *modules* for several synchronization strategies. The running strategy can be specified by the command line argument `-sync`, which

[2]Available at: http://code.google.com/p/google-guice/

receives one of the following values: `none` (default *module* that provides no synchronization); `deuce`; `jvstm`; `dstm`; `boost` (a highly-concurrent transactional version of a linearizable implementation of *node* [Herlihy & Koskinen, 2008]); `fine-lock` (a fine-grained lock-based scheme).

If we run the JWormBenchApp application with a different `-sync` argument, it will be interpreted as the name of a new *module* defining a new synchronization strategy. So, to integrate a Java STM implementation with JWormBenchApp we just need to: (1) implement the necessary factories, compile and include them in the Java classpath; (2) define a new Guice module specifying *bindings* to those factories; (3) run JWormBenchApp and give to the `-sync` argument the name of the Guice *module*.

The JWormBench is available on GitHub repository in 5 projects:

- JWormBench - the core framework class library.
- JWormBenchApp - a console application that parses command line arguments, setup and runs a workload, based on a *configuration module*.
- JWormBench-unit.tests - unit tests that cover 90% of the source code of JWormBench class library;
- JWormBenchGui - A GUI application for layout evaluation and to move worms on the world.
- WormBenchTools - Tools from the original WormBench application, written in C#, able to generate configurations' files of *worms*, *world*, and *worm operations*.

A.2.2 STM integration

One of the key differences from WormBench is the STM integration model of JWormBench. For that purpose, the JWormBench introduces the concept of *step* that is an abstraction of an iteration performed by the worm's thread. A step includes the following tasks: (1) performs a *worm operation*, (2) moves the *worm*, updating the *coordinates* of the body and the head of the worm, and (3) updates the *nodes* under the *body* with the *worm*'s reference.

Each synchronization strategy must provide a concrete implementation of the `IStepFactory` interface, whose instances are responsible for creating a collection of *step* objects, based on the information gathered from an iterable `IStepSetup` instance (the

default implementation of `IStepSetup` loads *step* data from a configuration file). The easiest way of providing an implementation of the `IStepFactory` interface is by extending the `AbstractStepFactory` class, according to the *factory method* design pattern [Gamma *et al.* , 1995]. Listing A.2 presents `DeuceStepFactory` class as an implementation of the `IStepFactory`.

```
1  public class DeuceStepFactory extends AbstractStepFactory{
2    public DeuceStepFactory(...) {
3      super(stepSetup, opFac);
4    }
5    protected IStep factoryMethod(IOperation<?> op, ...) {
6      return new AbstractStep(direction, op) {
7        @Override
8        public Object performStep(IWorm worm) {
9          Object res = performAtomicOperation(worm);
10         performmAtomicMove(worm, direction);
11         return res;
12       }
13       @org.deuce.Atomic
14       Object performAtomicOperation(IWorm worm){
15         return op.performOperation(worm);
16       }
17       @org.deuce.Atomic
18       void performmAtomicMove(IWorm worm, ...) {
19         worm.move(direction);
20         worm.updateWorldUnderWorm();
21       }
22     };
23   }
24 }
```

Listing A.2: Implementation of *step* factory for Deuce STM.

Depending on the STM implementation, it may be required to provide other implementations of the JWormBench core entities, such as *world*, *node*, or *coordinates*, among others. For instance, in JVSTM the fields accessed in the context of a transaction must be of the `VBox` type. In the case of JWormBench this means that the *value* field of *node* must be of the `VBox` type. So, the configuration of JVSTM for JWormBench must provide also an implementation of a *node*'s factory — an implementation of `INodeFactory` interface.

A.2.3 Correctness test

The JWormBench's *world* is accessed by *worms* that can update the nodes' values. If we use a read-only workload and the nodes' values are not modified by worms, then the *world's state* must be the same before and after workload execution.

On the other hand, for read-write workloads the total value can change, or not, depending on the kind of update operations. For instance, if we just use operations of *n-reads-n-writes* category as *sort* and *transpose* (see section A.1), the *world's state* will be preserved.

But for operations of *n-reads-1-writes* such as *Replace<read-only>With<read-only>* the total value of the world nodes will change. To verify if the total value is correct at the end of a workload execution, we must store in the thread's private buffer the difference between the new and the old value updated by a worm operation. At the end, if we subtract the accumulated differences on each thread's private buffer to the total value of all nodes, the result must be equal to the initial sum of nodes' values.

A.2.4 Contention level

Another of the limitations of WormBench is the difficulty of maintaining the same contention level among different number of worker threads. This happens because in WormBench the number of *worms* is equal to the number of threads. So, if we increase the number of threads we will automatically have more *worms* in the *world* and more collisions between them, affecting the contention level. In some scenarios you may require a behavior like this, but there are other cases where you may want to keep the same contention level among a different number of worker threads. The JWormBench allows both scenarios.

In JWormBench the number of threads and the number of worms are independent. The former is specified by an integer argument and the latter corresponds to the number of worms defined in the configuration file. So, by varying these parameters we can obtain three situations: (1) if the number of worms is exactly the same as the number of threads, then each worm will be assigned to one thread as in WormBench application; (2) if the number of worms is greater than the number of threads, then worms are distributed among threads; (3) if there are less worms than threads, then we have an unsupported scenario and an exception is raised.

Appendix B

Extending Jikes RVM's just-in-time compiler

I have implemented a prototype of JVSTM with AOM in the Jikes Research Virtual Machine (Jikes RVM) [Alpern *et al.* , 2005]. In my work I have focused in the details of the managed runtime object model to achieve the AOM approach and I show an STM integration at the virtual machine level.

This Appendix describes the details of the modifications made to the Jikes RVM's just-in-time compiler.

In Section B.1 I start by giving an overview about the architecture of Jikes RVM's just-in-time compiler. Then, in Section B.2 I describe the required modifications to wrap an atomic method call into a transaction control flow. Finally, in Section B.3 I explain the changes to the implementation of `getfield` and `putfield` operations.

B.1 Architecture of Jikes RVM's just-in-time compiler

The just-in-time compiler of Jikes RVM is organized in three main classes: the `TemplateCompilerFramework`, the `BaselineCompilerImpl` and the `Assembler`. The `TemplateCompilerFramework` is responsible for parsing a method's bytecodes and for generating the corresponding machine code. An instance of the `TemplateCompilerFramework` delegates the conversion of each Java bytecode on `emit_<opcode>` protected methods. These methods behave like hook methods overridden by the class

`BaselineCompilerImpl`, which is responsible for the translation into a specific machine code. So, for each CPU architecture there is a concrete `BaselineCompilerImpl` implementation. Finally, the `BaselineCompilerImpl` uses the `Assembler` class, which implements the low-level assembler for IA32 processor architecture and contains functionality for encoding specific instructions into an array of bytes.

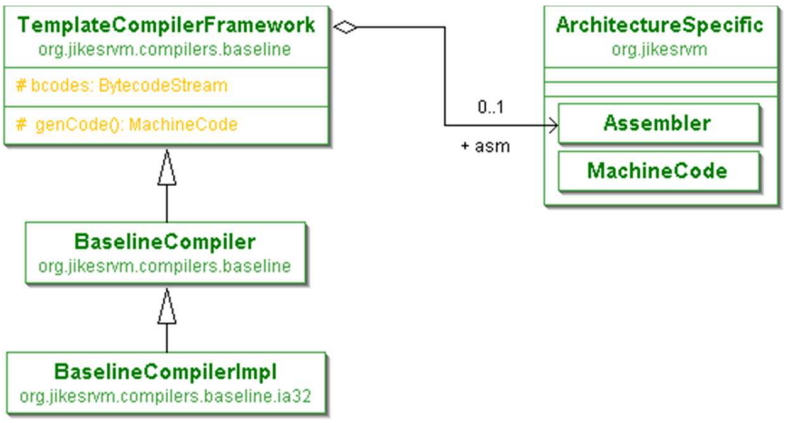


Figure B.1: Architecture of the Jikes RVM's just-in-time compiler

For the compilation of each method there is a corresponding instance of the `BaselineCompilerImpl` responsible for that job. This instance has a `bcodes` reference to the bytecodes stream of the method being compiled and a `genCode` method responsible for translating those bytecodes into machine code. In Listing B.1, I show the skeleton of the `genCode` method of class `TemplateCompilerFramework` and how it delegates on `emit_<opcode>` hook methods the responsibility of converting bytecodes into machine code.

B.2 Wrapping a method call in a transaction control flow

I have extended the just-in-time compiler respecting the existing design for translating bytecodes into machine code, as shown in Listing B.1. For each bytecode there is a corresponding case statement in `genCode` method, which delegates on `emit_<opcode>` hook method the responsibility for translating that bytecode. Given that for each kind of method invocation there is a specific bytecode `invokeinterface`, `invokespecial`, `invokestatic`, or `invokevirtual`, then there is a corresponding hook method `emit-`

```
1  protected final MachineCode genCode() {
2    emit_prologue();
3    while (bcodes.hasMoreBytecodes()) { //Main code generation loop
4      starting_bytecode();
5      int code = bcodes.nextInstruction();
6      switch (code) {
7        case JBC_nop: break;
8        case JBC_iaload: emit_iaload(); break;
9        case JBC_laload: emit_laload(); break;
10       ...
11       case JBC_invokestatic: ... emit_invokestatic(); ... break;
12       ...
13     }
14     ending_bytecode();
15   }
16   return asm.finalizeMachineCode(bytecodeMap);
17 }
```

Listing B.1: `genCode()` method from `TemplateCompilerFramework`

`_invokeinterface`, `emit_invokespecial`, `emit_invokestatic`, or `emit_invokevirtual`.[1]

I followed a similar approach to wrap a method call in a transaction control flow and I added in `TemplateCompilerFramework` two hook methods: `emitSTM_begin` and `emitSTM_tryCommit`. Before emitting code for the method's invocation, I emit code to start a transaction—`emitSTM_begin`—and at the end I emit code that tries to commit the transaction and in case of failure restarts the transaction—`emitSTM_tryCommit`. To restart the execution at the point where the transaction was created, then the `emitSTM_begin` method must return the address of the first instruction it emits. This address is used by `emitSTM_tryCommit` to emit a `jmp` to the beginning of the wrapping, in case of transaction's failure. If the `emitSTM_begin` returns a zero value, it means that the calling method is not transactional and no transaction has been created, so I should not call the `emitSTM_tryCommit`.

The Listing B.2 shows how the just-in-time compiler uses the `emitSTM_begin` and `emitSTM_tryCommit` to instrument a method call. This example just depicts a static invocation but I follow the same approach for other kinds of method calls.

The `emitSTM_begin` and `emitSTM_tryCommit` methods are abstract and should be overridden for each `BaselineCompilerImpl` class implementation, according to each

[1] For each kind of invocation there are two hook methods in format `emit_<resolved|unresolved><opcode>`. Yet, the way a method is wrapped into a transaction is independent of its resolution.

```
case JBC_invokestatic: {
  MethodReference mRef = bcodes.getMethodReference();
  RVMMethod callingMethod = mRef.resolve();
  // emit a call to Transaction.begin
  int methodInitImm = emitSTM_begin(callingMethod);
  // Default translation of invokestatic
  emit_invokestatic(mRef);
  if(methodInitImm > 0)
  // emit a call to Transaction.tryCommit
  emitSTM_tryCommit(callingMethod, methodInitImm);
  break;
}
```

Listing B.2: `genCode()`: emitting code to wrap a method call into a transaction.

CPU architecture. The implementation of these methods is architecture dependent, on how `RVMThread` context is accessed and how is created a backup for the method's arguments. In my solution I have just implemented these methods for the IA32 architecture. These methods emit code to invoke the corresponding `begin` and `tryCommit` methods of the class `Transaction`, which receive a `RVMThread` object by argument in both cases. So, the `emitSTM_begin` method should perform the following tasks:

1. Checks if the calling method is annotated with `Atomic` and if not, there is nothing more to do and it returns;
2. If the calling method is atomic then it emits code for invoking `begin` static method from `Transaction` class. This is the point to where execution should jump in case of failure on `tryCommit`;
3. Creates a backup of the method's arguments (explained later);
4. At the end it will return the offset for the instruction corresponding to the invocation of the method `begin`—variable `methodInitImm` of the Listing B.3.

According to the Jikes RVM just-in-time compiler the method caller is responsible for pushing arguments on the stack and the callee for cleaning those arguments and push the method result, if exists. This means that we must preserve a backup of the method's arguments in the case we have to repeat the invocation of the atomic method. Listing B.4 shows the code responsible for emitting the creation of a backup in stack for the callee method's arguments and Listing B.5 presents the corresponding IA32 code emitted by the previous code. Figure B.2 depicts three snapshots of the stack

```
protected int emitSTM_begin(RVMMethod cM){
 int methodInitImm = 0;
 // ---------------------- 1st task -----------------
 if(fullyBootedVM && cM.isAnnotationPresent(Atomic.class)){
  // ---------------------- 2nd task -----------------
  methodInitImm = asm.getMachineCodeIndex();
  asm.emitPUSH_RegDisp(ESI, activeThread_field_offset);
  emit_resolved_invokestatic_without_stm(stmTransactionBegin);
  // ---------------------- 3rd task -----------------
  // Creates a backup for method's arguments
  ...
 }
 // ---------------------- 4th action -----------------
 return methodInitImm;
}
```

Listing B.3: `emitSTM_begin()`

frame corresponding to its state before the execution of the code shown in Listing B.5, after the execution of the instructions I1.1 to I1.3 and at the end of the execution of the arguments duplication.

The `emitSTM_tryCommit` method emits code to call the `tryCommit` method of the class `Transaction` and evaluate its returned value. In case of transaction's failure then it will jump to the `methodInitImm` offset instruction. Otherwise the last thing it has to do is to emit code to clean argument's backup from stack. Due to the complexity of emitting code for the different branches of the control flow, I have chosen to just present a sketch of the resulting IA32 code from the execution of the `emitSTM_tryCommit` in Listing B.6.

B.3 Changing the `getfield` and `putfield` default behaviors

When the `genCode` method of the class `TemplateCompilerFramekork` is processing the `putfield` and `getfield` bytecodes, we must check if the declaring class of the targeting field is transactional and in that case we have to emit adequate machine code. For that purpose we add two new hook methods: `emit_transactional_getfield` and `emit_transactional_putfield` and change the case statements for corresponding

```
// ---------- 3rd task ----------
int nrParams =
  callingMethod.getParameterWords()
  + (callingMethod.isStatic() ? 0 : 1);
Offset nrArgsWords = NO_SLOT.plus(
  WORDSIZE * nrParams);
asm.emitSUB_Reg_Imm(SP, WORDSIZE);
for(int i = nrParams; i > 0; i--){
  asm.emitMOV_Reg_RegDisp(
    ECX, SP, nrArgsWords);
  asm.emitMOV_RegDisp_Reg(
    SP, NO_SLOT, ECX);
  asm.emitSUB_Reg_Imm(SP, WORDSIZE);
}
asm.emitADD_Reg_Imm(SP, WORDSIZE);
```

Listing B.4: Emit code for duplicating method's arguments.

```
;Adjust stack pointer down one word
I0:     sub esp, WORDSIZE
;Save argument at ECX
I1.1:   mov ecx, [esp
        +( nrArgsWords * WORDSIZE)]
;Copy argument to nrArgsWords
;slots forward
I1.2:   mov [esp], ecx
;Adjust stack pointer down one word
I1.3:   sub esp, WORDSIZE
;Repeats three last operations
;for the number of method's arguments
I...:
;Adjust stack pointer up one word
I.nrArgs: add esp, WORDSIZE
```

Listing B.5: Machine code for duplicating method's arguments.

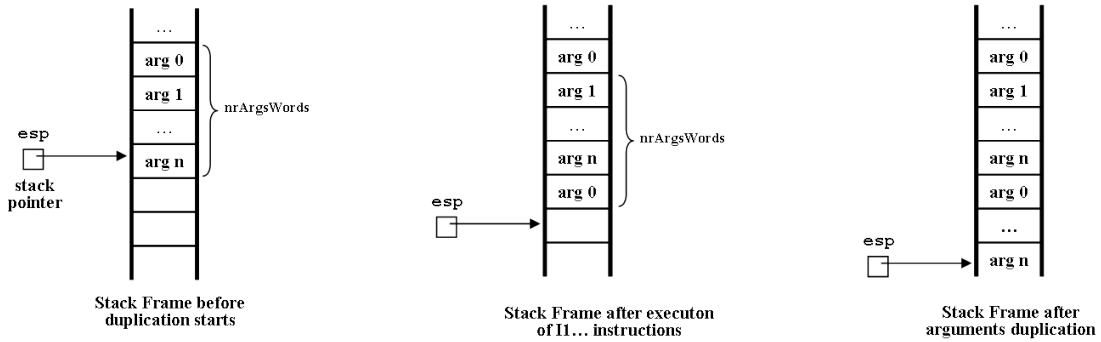


Figure B.2: Stack frame before, during and after arguments duplication.

```
;Invokes the tryCommit method of the class Transaction passing
;the current RVMThread object as argument
  push [esi + fieldActiveThreadOfRvmProcessor.peekResolvedField().getOffset()]
  call stmTransactionTryCommit
;Checks if the tryCommit has succeed comparing the method's result
;with Transaction.Status.SUCCEED.
  cmp [esp], Transaction.Status.SUCCEED.ordinal()
  jeq done
repeat:
  add esp, WORDSIZE                 ;Pop stmTransactionCommit result
  add esp, nrReturnWords            ;Pop the result of the atomic method call
  jmp methodInitImm
done:
  add esp, WORDSIZE                 ;Pop stmTransactionCommit result
  add esp, nrArgsWords * WORDSIZE   ;clean argument's backup
```

Listing B.6: Machine code for trying to commit a transaction.

bytecodes, as shown in Listing B.7. The `emit_getfield` and `emit_putfield`[2] pre-

[2]There are two and not one emit method for each `getfield` and `putfield` operations, depending on the resolution, or not, of the field's declaring class. But this fact does not interfere in the conditions

serve the default behavior of corresponding `putfield` and `getfield` bytecodes, whereas the `emit_transactional_getfield` and `emit_transactional_putfield` call the STM to get, or store, values from, or into, the transaction's context.

```
...
case JBC_getfield:{
 FieldReference fieldRef = bcodes.getFieldReference();
 RVMField field = fieldRef.resolve();
 if(fullyBootedVM){
  RVMClass declaringClass = field.getDeclaringClass();
  if(declaringClass.isAnnotationPresent(Transactional.class))
   emit_transactional_getfield(fieldRef);
 }
 else{...; emit_getfield(fieldRef); // default behavior
 }
 break;
}
case JBC_putfield: {
 FieldReference fieldRef = bcodes.getFieldReference();
 RVMField field = fieldRef.resolve();
 if(fullyBootedVM && !method.isObjectInitializer()){
  RVMClass declaringClass = field.getDeclaringClass();
  if(declaringClass.isAnnotationPresent(Transactional.class))
    emit_transactional_putfield(fieldRef);
 }
 else{{...; emit_putfield(fieldRef); // default behavior
 }
 break;
}
...
```

Listing B.7: `genCode()` method from `TemplateCompilerFramework`

Yet, there is one scenario for which the `getfield` operation does not need STM for targeting a transactional object: when that object is on its compact layout and the `getfield` is performed out of the scope of any transaction. Both conditions must be met to read a transactional object in the standard way. In that case the `getfield` operation just have to read a slot from the object's storage.

On the other hand, if an object is accessed in the scope of a transaction and it is already in its *extended layout* we must read its fields values through the STM. If an object is in the *compact layout*, we also must call STM to record that operation in the

for generating a transactional `getfield`/`putfield` and for simplification I keep it out of this discussion.

transaction's context. So, independently of the object layout, every time we read an object from inside a transaction we must do it via STM.

Appendix C

Extending Deuce STM

Some STMs and optimization techniques require a specific object model distinct from the one provided by a managed runtime environment. Yet, the original Deuce STM just provides extensibility in terms of the specification of the STM algorithm, but it does not allow any enhancement to the Java object layout.

In Deuce, the entry point of the instrumentation engine is defined by the class `Agent`, which implements the interface `java.lang.instrument.ClassFileTransformer`. This class is responsible for instrumenting all the classes from a jar archive, or from the class loader, depending on whether the instrumentation engine is performed in offline mode (running the `main` method), or by a Java agent (running the `premain` method), as depicted in the code of the Listing C.1. In both cases, it is invoked the core method `transform`, which receives a `byte[]` with the bytecodes of the original class definition and returns a new `List<ClassByteCode>` with the bytecodes of the resulting class from the Deuce transactification and other auxiliary classes added by the transformation. So, the standard Deuce transformation is defined by the class `org.deuce.transform.asm.ClassTransformer`, which is invoked by the `Agent` class.

In my work, I added to the Deuce framework a new infrastructure that allows the specification and execution of *enhancers*, which are additional transformations to the standard Deuce instrumentation.[1] These enhancers are instances of classes implementing the interface `ClassEnhancer` presented in Listing C.2. This interface specifies a unique method with the same signature of the core `transform` method of the class `Agent`, which is responsible to perform the standard Deuce transactification. But now,

```
1  public class Agent implements ClassFileTransformer {
2
3    public static void main(String[] args) {
4      ...
5      for (JarEntry nextJarEntry : . . . ) {
6        List<ClassByteCode> transformed = transform(...);
7        ...
8      }
9    }
10
11   public static void premain(..., Instrumentation inst) {
12     inst.addTransformer(new Agent());
13   }
14
15   @Overrides
16   public byte[] transform(..., byte[] bytecodes){
17     List<ClassByteCode> transformed = transform(...);
18     for(ClassByteCode cb : transformed){
19       loadClass(cb.getClassName(), cb.getBytecode());
20     }
21     ...
22   }
23
24   List<ClassByteCode> transform(..., byte[] bytecodes){
25     ClassTransformer cv = new ClassTransformer(...);
26     bytecodes = cv.visit(bytecodes);
27     ...
28   }
29 }
```

Listing C.1: Skeleton of the class `Agent`, with two different entry points: `main` and `premain`, depending on whether the instrumentation is performed in offline mode or by a Java agent.

I will allow the end user programmer to specify further transformations beyond the standard Deuce transformation.

The *enhancers* may be added to the Deuce engine through the system properties `org.deuce.transform.pre` and `org.deuce.transform.post`, depending on whether they should be executed before or after the standard Deuce instrumentation. Moreover, the enhancers may be combined in a chain of transformations, when more than one en-

[1]This adaptation of Deuce is available at https://github.com/inesc-id-esw/deucestm/

```
1 public interface ClassEnhancer{
2
3   public List<ClassByteCode> transform(
4       boolean offline,
5       String className,
6       byte[] classfileBuffer);
7
8 }
```

Listing C.2: `ClassEnhancer` interface.

hancer is specified in the same *pre* or *post* property. For instance, in the case of JVSTM, we need to combine two *post* enhancers that transform the definition of transactional classes and arrays, and we also need to include a *pre* enhancer, which makes a required transformation to support static fields. So, for the JVSTM we need to run Deuce with the configuration presented in Listing C.3.

```
1 org.deuce.transform.pre=
2   org.deuce.transform.jvstm.EnhanceStaticFields
3 org.deuce.transform.post=
4   org.deuce.transform.jvstm.EnhanceTransactional,
5   org.deuce.transform.jvstm.EnhanceVBoxArrays
```

Listing C.3: Required enhancers to run Deuce with the JVSTM.

Bibliography

Adl-Tabatabai, Ali-Reza, Lewis, Brian T., Menon, Vijay, Murphy, Brian R., Saha, Bratin, & Shpeisman, Tatiana. 2006. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, **41**(June), 26–37.

Afek, Yehuda, Korland, Guy, & Zilberstein, Arie. 2011. Lowering STM overhead with static analysis. *Pages 31–45 of: Proceedings of the 23rd international conference on Languages and compilers for parallel computing*. LCPC'10. Houston, TX: Springer-Verlag.

Agrawal, Kunal, Leiserson, Charles E., & Sukha, Jim. 2006. Memory models for open-nested transactions. *Pages 70–81 of: Proceedings of the 2006 workshop on Memory system performance and correctness*. MSPC '06. San Jose, California: ACM.

Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K. S., Mergen, M., Moss, J. E. B., Ngo, T., & Sarkar, V. 2005. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, **44**(January), 399–417.

Ansari, Mohammad, Kotselidis, Christos, Jarvis, Kim, Luján, Mikel, Kirkham, Chris, & Watson, Ian. 2008. Lee-TM: A Non-trivial Benchmark for Transactional Memory. *In: ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS, Springer.

Beckman, Nels E., Kim, Yoon Phil, Stork, Sven, & Aldrich, Jonathan. 2009. Reducing STM overhead with access permissions. *Pages 2:1–2:10 of: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*. Genova, Italy: ACM.

Binder, Walter, Hulaas, Jarle, & Moret, Philippe. 2007. Advanced Java bytecode instrumentation. *Pages 135–144 of: Proceedings of the 5th international symposium on Principles and practice of programming in Java*. PPPJ '07. Lisboa, Portugal: ACM.

Cachopo, João, & Rito-Silva, António. 2006. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, **63**(December), 172–185.

Cachopo, João. 2007. *Development of Rich Domain Models with Atomic Actions*. Ph.D. thesis, Instituto Superior Técnico, Technical University of Lisbon.

Cao Minh, Chi, Chung, JaeWoong, Kozyrakis, Christos, & Olukotun, Kunle. 2008 (September). STAMP: Stanford Transactional Applications for Multi-Processing. *In: IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*.

Carvalho, Fernando Miguel, & Cachopo, João. 2012 (January). Adaptive object metadata to reduce the overheads of a multi-versioning STM. *In: Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers*. MULTIPROG.

Carvalho, Fernando Miguel, & Cachopo, João. 2013a. Lightweight identification of captured memory for Software Transactional Memory. *In: Proceedings of the 13th international conference on Algorithms and architectures for parallel processing*. ICA3PP'13. Sorrento, Italy: Springer-Verlag.

Carvalho, Fernando Miguel, & Cachopo, João. 2013b. Runtime elision of transactional barriers for captured memory. *Pages 303–304 of: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '13. Shenzhen, China: ACM.

Carvalho, Fernando Miguel, & Cachopo, João. 2011. STM with transparent API considered harmful. *Pages 326–337 of: Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*. ICA3PP'11. Melbourne, Australia: Springer-Verlag.

Cascaval, Calin, Blundell, Colin, Michael, Maged, Cain, Harold W., Wu, Peng, Chiras, Stefanie, & Chatterjee, Siddhartha. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, **6**(5), 46–58.

Chakrabarti, Dhruva R. 2010a. New abstractions for effective performance analysis of STM programs. *SIGPLAN Not.*, **45**(January), 333–334.

Chakrabarti, Dhruva R. 2010b. New abstractions for effective performance analysis of STM programs. *Pages 333–334 of: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '10. Bangalore, India: ACM.

Cormen, Thomas H., Stein, Clifford, Rivest, Ronald L., & Leiserson, Charles E. 2001. *Introduction to Algorithms*. 2nd edn. McGraw-Hill Higher Education.

Dalessandro, Luke, Spear, Michael F., & Scott, Michael L. 2010. NOrec: streamlining STM by abolishing ownership records. *Pages 67–78 of: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '10. Bangalore, India: ACM.

Dias, Ricardo J., Vale, Tiago M., & Lourenço, João M. 2012. Efficient support for in-place metadata in transactional memory. *Pages 589–600 of: Proceedings of the 18th international conference on Parallel Processing*. Euro-Par'12. Rhodes Island, Greece: Springer-Verlag.

Dice, Dave, & Shavit, Nir. 2006. What really makes transactions faster? *In: Proc. of the 1st TRANSACT 2006 workshop*. Transact, Ottawa, Canada, 06.11.2006.

Dice, Dave, Shalev, Ori, & Shavit, Nir. 2006. Transactional locking II. *Pages 194–208 of: Proceedings of the 20th international conference on Distributed Computing*. DISC'06. Stockholm, Sweden: Springer-Verlag.

Dragojevic, Aleksandar, Guerraoui, Rachid, & Kapalka, Michal. 2008. Dividing Transactional Memories by Zero. *In: TRANSACT '08: 3rd Workshop on Transactional Computing*. Transact, Salt Lake City, Utah, USA, 23.02.2008.

Dragojevic, Aleksandar, Ni, Yang, & Adl-Tabatabai, Ali-Reza. 2009. Optimizing transactions for captured memory. *Pages 214–222 of: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. SPAA '09. Calgary, AB, Canada: ACM.

Dragojević, Aleksandar, Guerraoui, Rachid, & Kapalka, Michal. 2009. Stretching transactional memory. *Pages 155–165 of: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '09. Dublin, Ireland: ACM.

Eddon, Guy, & Herlihy, Maurice. 2007. Language support and compiler optimizations for STM and transactional boosting. *Pages 209–224 of: Proceedings of the 4th international conference on Distributed computing and internet technology*. ICDCIT'07. Bangalore, India: Springer-Verlag.

Felber, Pascal, Fetzer, Christof, & Riegel, Torvald. 2008. Dynamic performance tuning of word-based software transactional memory. *Pages 237–246 of: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. Salt Lake City, UT, USA: ACM.

Fernandes, Sérgio Miguel, & Cachopo, João. 2011. Lock-free and scalable multi-version software transactional memory. *Pages 179–188 of: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP '11. San Antonio, TX, USA: ACM.

Fowler, M. 2003. *Patterns of Enterprise Application Architecture*. A Martin Fowler signature book. Addison-Wesley.

Fraser, Keir. 2003. *Practical lock freedom*. Ph.D. thesis, University of Cambridge, Computer Laboratory.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Guerraoui, Rachid, & Kapalka, Michal. 2008. On the correctness of transactional memory. *Pages 175–184 of: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. Salt Lake City, UT, USA: ACM.

Guerraoui, Rachid, Kapalka, Michal, & Vitek, Jan. 2007. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, **41**(March), 315–324.

Harmanci, Derin, Felber, Pascal, Gramoli, Vincent, & Fetzer, Christof. 2009. TMunit: Testing Transactional Memories. *In: 4th ACM SIGPLAN Workshop on Transactional Computing*. Transact.

Harris, Tim, & Fraser, Keir. 2003. Language support for lightweight transactions. *Pages 388–402 of: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. OOPSLA '03. Anaheim, California, USA: ACM.

Harris, Tim, Marlow, Simon, Peyton-Jones, Simon, & Herlihy, Maurice. 2005. Composable memory transactions. *Pages 48–60 of: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '05. Chicago, IL, USA: ACM.

Harris, Tim, Plesko, Mark, Shinnar, Avraham, & Tarditi, David. 2006. Optimizing memory transactions. *Pages 14–25 of: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM.

Harris, Tim, Larus, James, & Rajwar, Ravi. 2010. *Transactional Memory, 2nd Edition*. 2nd edn. Morgan and Claypool Publishers.

Herlihy, Maurice, & Koskinen, Eric. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. *Pages 207–216 of: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. Salt Lake City, UT, USA: ACM.

Herlihy, Maurice, & Lev, Yossi. 2009. tm_db: A Generic Debugging Library for Transactional Programs. *Pages 136–145 of: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society.

Herlihy, Maurice, & Moss, J. Eliot B. 1993. Transactional memory: architectural support for lock-free data structures. *Pages 289–300 of: Proceedings of the 20th annual international symposium on computer architecture*. ISCA '93. San Diego, California, United States: ACM.

Herlihy, Maurice, & Shavit, Nir. 2008. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Herlihy, Maurice, Luchangco, Victor, & Moir, Mark. 2006. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, **41**(October), 253–262.

Herlihy, Maurice P., & Wing, Jeannette M. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, **12**(3), 463–492.

Kestor, Gokcen, Gioiosa, Roberto, Harris, Tim, Unsal, Osman S., Cristal, Adrian, Hur, Ibrahim, & Valero, Mateo. 2011. STM2: A Parallel STM for High Performance Simultaneous Multithreading Systems. *Pages 221–231 of: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. Washington, DC, USA: IEEE Computer Society.

Knight, Tom. 1986. An architecture for mostly functional languages. *Pages 105–112 of: Proceedings of the 1986 ACM conference on LISP and functional programming*. LFP '86. Cambridge, Massachusetts, United States: ACM.

Korland, Guy. 2010. *Deuce STM*. https://sites.google.com/site/deucestm/.

Korland, Guy, Shavit, Nir, & Felber, Pascal. 2010 (March). Noninvasive concurrency with Java STM. *In: Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers*. MULTIPROG.

Lindholm, Tim, & Yellin, Frank. 1999. *Java Virtual Machine Specification*. 2nd edn. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Lourenço, João, Dias, Ricardo, Luís, João, Rebelo, Miguel, & Pessanha, Vasco. 2009. Understanding the behavior of transactional memory applications. *Pages 3:1–3:9 of: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. PADTAD '09. Chicago, Illinois: ACM.

Mannarswamy, Sandya, Chakrabarti, Dhruva R., Rajan, Kaushik, & Saraswati, Sujoy. 2010a. Compiler aided selective lock assignment for improving the performance of software transactional memory. *SIGPLAN Not.*, **45**(January), 37–46.

Mannarswamy, Sandya, Chakrabarti, Dhruva R., Rajan, Kaushik, & Saraswati, Sujoy. 2010b. Compiler aided selective lock assignment for improving the performance of software transactional memory. *Pages 37–46 of: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '10. Bangalore, India: ACM.

Manson, Jeremy, Pugh, William, & Adve, Sarita V. 2005. The Java memory model. *Pages 378–391 of: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '05. Long Beach, California, USA: ACM.

Marathe, Virendra J., Scherer, William N., & Scott, Michael L. 2004. Design tradeoffs in modern software transactional memory systems. *Pages 1–7 of: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*. LCR '04. Houston, Texas: ACM.

Marathe, Virendra J., Spear, Michael F., Heriot, Christopher, Acharya, Athul, Eisenstat, David, Scherer III, William N., & Scott, Michael L. 2006 (Jun). Lowering the Overhead of Software Transactional Memory. *In: ACM SIGPLAN Workshop on Transactional Computing*. Transact, Ottawa, Canada, 06.11.2006.

Marathe, Virendra Jayant, & Moir, Mark. 2008. Toward high performance nonblocking software transactional memory. *Pages 227–236 of: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. Salt Lake City, UT, USA: ACM.

Martin, Milo, Blundell, Colin, & Lewis, E. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.*, **5**(July).

McKenney, Paul, Michael, Maged, Triplett, Josh, & Walpole, Jonathan. 2010. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. *SIGOPS Oper. Syst. Rev.*, **44**(August), 93–101.

Mellor-Crummey, John M., & Scott, Michael L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, **9**(1), 21–65.

Moss, J. Eliot B. 2006. Open Nested Transactions: Semantics and Support. *In: Workshop on Memory Performance Issues*.

Ni, Yang, Welc, Adam, Adl-Tabatabai, Ali-Reza, Bach, Moshe, Berkowits, Sion, Cownie, James, Geva, Robert, Kozhukow, Sergey, Narayanaswamy, Ravi, Olivier, Jeffrey, Preis, Serguei, Saha, Bratin, Tal, Ady, & Tian, Xinmin. 2008. Design and implementation of transactional constructs for C/C++. *Pages 195–212 of: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA '08. Nashville, TN, USA: ACM.

Papadimitriou, Christos H. 1979. The Serializability of Concurrent Database Updates. *J. ACM*, **26**(4), 631–653.

Perelman, Dmitri, & Keidar, Idit. 2010. SMV: Selective Multi-Versioning STM. *In: TRANSACT '10: 5th Workshop on Transactional Computing*.

Reed, David P. 1978. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. thesis, Massachusetts Institute of Technology Cambridge, MA, USA.

Reed, David P. 1983. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, **1**(February), 3–23.

Riegel, Torvald, Felber, Pascal, & Fetzer, Christof. 2006. A lazy snapshot algorithm with eager validation. *Pages 284–298 of: Proceedings of the 20th international conference on Distributed Computing*. DISC'06. Stockholm, Sweden: Springer-Verlag.

Riegel, Torvald, Fetzer, Christof, & Felber, Pascal. 2008. Automatic data partitioning in software transactional memories. *Pages 152–159 of: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. SPAA '08. Munich, Germany: ACM.

Saha, Bratin, Adl-Tabatabai, Ali-Reza, Hudson, Richard L., Minh, Chi Cao, & Hertzberg, Benjamin. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. *Pages 187–197 of: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '06. New York, New York, USA: ACM.

Shavit, Nir, & Touitou, Dan. 1995. Software transactional memory. *Pages 204–213 of: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95. Ottowa, Ontario, Canada: ACM.

Smith, Jim, & Nair, Ravi. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Spear, Michael F., Marathe, Virendra J., Dalessandro, Luke, & Scott, Michael L. 2007. Privatization Techniques for Software Transactional Memory. *Pages 338–339 of: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '07.

Spear, Michael F., Michael, Maged M., & von Praun, Christoph. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. *Pages 275–284 of: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08.

Spear, Michael F., Dalessandro, Luke, Marathe, Virendra J., & Scott, Michael L. 2009. A comprehensive strategy for contention management in software transactional memory. *Pages 141–150 of: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '09. Raleigh, NC, USA: ACM.

Sutter, Herb, & Larus, James. 2005. Software and the Concurrency Revolution. *Queue*, **3**(7), 54–62.

Wamhoff, Jons-Tobias, Fetzer, Christof, Felber, Pascal, Rivière, Etienne, & Muller, Gilles. 2013. FastLane: improving performance of software transactional memory for low thread counts. *Pages 113–122 of: Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM.

Wang, Cheng, Chen, Wei-Yu, Wu, Youfeng, Saha, Bratin, & Adl-Tabatabai, Ali-Reza. 2007. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. *Pages 34–48 of: Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07. Washington, DC, USA: IEEE Computer Society.

White, Sean, & Spear, Michael. 2010 (September). On reconciling hardware atomicity, memory models, and `__tm_waiver`. *In: 2nd Workshop on the Theory of Transactional Memory (WTTM)*.

Yoo, Richard M., Ni, Yang, Welc, Adam, Saha, Bratin, Adl-Tabatabai, Ali-Reza, & Lee, Hsien-Hsin S. 2008. Kicking the tires of software transactional memory: why the going gets tough. *Pages 265–274 of: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. SPAA '08. Munich, Germany: ACM.

Zyulkyarov, Ferad, Cristal, Adrian, Cvijic, Sanja, Ayguade, Eduard, Valero, Mateo, Unsal, Osman, & Harris, Tim. 2008. WormBench: a configurable workload for evaluating transactional memory systems. *Pages 61–68 of: Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*. MEDEA '08. Toronto, Canada: ACM.

Zyulkyarov, Ferad, Harris, Tim, Unsal, Osman S., Cristal, Adrían, & Valero, Mateo. 2010a. Debugging programs that use atomic blocks and transactional memory. *Pages 57–66 of: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '10. Bangalore, India: ACM.

Zyulkyarov, Ferad, Stipic, Srdjan, Harris, Tim, Unsal, Osman S., Cristal, Adrián, Hur, Ibrahim, & Valero, Mateo. 2010b. Discovering and understanding performance bottlenecks in transactional applications. *Pages 285–294 of: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT '10. Vienna, Austria: ACM.