UNIVERSITÉ DE NEUCHÂTEL

Institut d'informatique
Université de Neuchâtel

Semantics, Performance and Language Support for Transactional Memory

par

Mehmet Derin Harmanci

Thèse

Soutenue le 19 mars 2012 à la Faculté des Sciences
pour l'obtention du grade de
Docteur ès sciences

Acceptée sur proposition du jury:

Prof.	**Pascal Felber**	Université de Neuchâtel	Directeur de thèse
Dr.	**Vincent Gramoli**	Université de Neuchâtel et EPFL	Co-directeur de thèse
Prof.	**Peter Kropf**	Université de Neuchâtel	Rapporteur
Prof.	**Oscar Nierstrasz**	Université de Berne	Rapporteur
Prof.	**Ian Watson**	Université de Manchester	Rapporteur

UNIVERSITÉ DE
NEUCHÂTEL

FACULTE DES SCIENCES
Secrétariat-Décanat de la faculté
Rue Emile-Argand 11
CH-2000 Neuchâtel

IMPRIMATUR POUR LA THESE

Semantics, Performance and Language Support for Transactional Memory

Mehmet Derin HARMANCI

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel,
sur le rapport des membres du jury

MM. Pascal Felber, (directeur de thèse),UniNe
Vincent Gramoli, (co-directeur de thèse), UniNe et EPF Lausanne
Peter Kropf, UniNe,
Ian Watson, Uni. Manchester,
Oscar Nierstrasz, Uni. Berne

autorise l'impression de la présente thèse.

Neuchâtel, le 14 juin 2012

Le doyen :
P. Kropf

Téléphone : +41 32 718 21 00 E-mail : secretariat.sciences@unine.ch www.unine.ch/sciences

To my brother

Acknowledgements

This thesis document and all studies it represents are fulfilled thanks to the support of numerous people.

First of all, I would like to thank my thesis advisor, Pascal Felber, for his guidance and support throughout this thesis. His availability, pragmatic approach to problems and timely advices allowed me to easily keep up on the right track. His broad knowledge has always been at the rescue of even the minor of the difficulties. I'm also grateful for his patience to kindly support me from the very beginning to the delayed end.

Special thanks goes to Vincent Gramoli, who has supervised my thesis alongside my advisor. With his constant presence and guidance, his input to this work is invaluable. His remarks, feedback and advices have always been succinct, clear and to the point and his constant support and stimulation always helped me to keep up with my work.

I would like to thank Christof Fetzer, for his valuable inputs and remarks on the TMunit work. I am also thankful to the members of the thesis jury for showing interest in my studies and for helping to raise the quality of my work.

I would like to thank Peter Kropf, for his warm support to me and my small family, for his kindness and attention.

The research work conducted in this thesis is supported in part by the Swiss National Foundation under grant 200021-118043 and the European Union's Seventh Framework Programme (FP7/2007-2013) under grants 216852 and 248465. I'm grateful to both foundations for their support has allowed me to perform my studies.

I would like to thank the colleagues of Computer Science Department of University of Neuchatel for the warm and pleasant working environment. A special thanks to my office mate, Patrick, for his kindness, attention and positive attitude and especially for his constant readiness to help no matter what the subject is. I would also like to thank Nada for her warm company both in and outside the institute, always helping to be in good mood.

I can not be more thankful and grateful to my parents and my brother for bringing me to the point I'm now and for supporting me all along my entire studies. Although very young, I would like to thank my small daughter Ipek from whom I repeatedly learn how joyful life can be. Last but definitely not least, I would like to thank my wife, Dilek, for her unconditional support, love, care and patience. Thank you for giving the strength to carry on and for bringing color and joy to my life.

Abstract

Keywords: Concurrent programming, transactional memory, semantics, performance, language support, exception handling.

The emergence of multi-core architectures suggests that programmers should write concurrent programs for exploiting the continuously increasing computing capacity of the hardware. Currently, the predominant approach to write concurrent programs is to use locks, but using locks for writing correct concurrent programs is difficult. A recent approach to simplify concurrent programming is to introduce transactional blocks into programming languages. Such code blocks would be demarcated by a specific language construct so that the block executes in a transactional manner (i.e., the block can be aborted and restarted whenever required). Providing these language-level blocks is expected to simplify concurrent program development by hiding data synchronization between concurrent executions from the programmer and, hence, is an appealing alternative to locks.

Although transactional blocks have advantages over locks, convincing a programmer to use transactional blocks for data synchronization requires the following: *(i)* the semantics of the transactional behavior should be clear and simple, *(ii)* the performance of a program using transactional blocks should be scalable with the number of concurrent hardware threads (as long as the workload represented by the program allows it) as well as being better than the sequential execution performance when run on multiple threads, and *(iii)* the language support should be such that it is easy to use transactional blocks both in terms of syntax and interoperability with other language features. The first two requirements should mostly be met by the run-time support that implements the actual transactional behavior, called Transactional Memory (TM). Although many different TM implementations exist there are no approaches to verify whether these implementations meet both requirements at the same time. Moreover, few studies target rich language support to meet the third requirement. This thesis aims at providing a solution to each of these shortcomings by proposing two tools: TMunit and TMJava.

TMunit is the tool we propose for programmers to verify whether a TM implementation meets the semantic and performance requirements. The ability of TMunit to experiment with both aspects (semantic and performance aspects) makes it a unique aid for TM designers. For the semantics side, TMunit is capable of describing scenarios at the level

of individual transactional operations and their interleavings, allowing to design detailed tests to verify semantics of TMs. For the performance side, TMunit, allows us to assess performance of TM implementations by describing complex scenarios (such as stressing a concurrent data structure) where access patterns to shared data can be specified. TMunit has an abstract interface that can bridge any TM and it comes with a test-suite so that common semantic and performance issues of TMs can readily be tested on any TM implementation.

TMJava aims at providing a rich language support for transactional blocks. To that end, it proposes a language extension to Java in terms of new transactional constructs that support *(i)* automated data synchronization among concurrent threads, (*ii)* transactional control flow, and *(iii)* enhanced exception handling. A notable merit of TMJava is that it proposes to use transactional blocks not only for data synchronization but also for exception handling while most existing language support approaches address only the data synchronization aspect. In particular, TMJava introduces an original extension, Atomic Boxes, that allows programmers to handle exceptions among multiple threads in a coordinated manner. This is particularly useful to propagate exceptions to all concerned threads and to perform coordinated recovery actions at a safe point in the execution of a concurrent program.

Résumé

Mots Clés: Programmation concurrent, mémoire transactionelle, sémantique, performance, support de langage, traitement d'exception.

Avec l'apparition des architectures multi-cœurs, les programmeurs doivent écrire des programmes concurrents pour exploiter pleinement la capacité de calcul de ces architectures. Actuellement, l'approche prédominante pour l'écriture des programmes concurrents est l'utilisation des verrous, mais il est, en général, difficile d'écrire un programme correct en utilisant les verrous. Une approche récente pour simplifier la programmation concurrente est d'introduire des blocs transactionnels dans les langages de programmation. De tels blocs de code sont delimités par des éléments spécifiques du langage et s'exécutent de manière transactionnelle (c.à.d. le bloc peut être avorté et reexécuté au besoin). Inclure ces blocs de langage doit permettre de simplifier le développement des programmes concurrents en masquant au programmeur la synchronisation des données entre fils d'exécution concurrents et, par conséquant, offre une alternative attractive à l'utilisation des verrous.

Malgré ses avantages pour la programmation concurrente, cette approche doit satisfaire les exigences suivantes pour qu'un programmeur soit convaincu d'utiliser des blocs transactionnels pour la synchronisation des données: *(i)* la sémantique du comportement transactionnel doit être simple et claire *(ii)* la performance d'un programme utilisant des blocs transactionnels doit passer à l'echelle par rapport au nombre de fils d'exécutions concurrents (à condition que la charge de travail représentée par le programme le permette) et doit en même temps offrir une performance supérieure à celle de l'exécution séquentielle lorsque le programme s'exécute sur plusieurs fils d'exécution, et *(iii)* le support de langage associé aux blocs transactionnels doit être simple à utiliser, tant en termes de syntaxe que d'intéropérabilité avec les autres éléments de langage. Les premières deux conditions doivent être satisfaites par le support d'exécution, nommé Mémoire transactionnelle (MT), qui met en œuvre le comportement transactionnel. Malgré l'existence de nombreuse MTs, il n'y a pas d'approche permettant de vérifier si elles satisfont ces deux conditions en même temps. De plus, il n'existe que peu de travaux visant à avoir un support de langage riche pour satisfaire la troisième condition. La présente thèse a pour but de trouver des solutions a chacun de ces problèmes à travers deux nouveaux outils: TMunit et TMJava.

TMunit est l'outil que nous proposons aux programmeurs pour vérifier si une MT satisfait certaines conditions en termes de sémantique et de performance. La possibilité d'expérimenter avec ces deux aspects (sémantiques et performance) des MTs rend TMunit unique pour la conception des MTs. Côté sémantique, TMunit est capable de décrire des scénarios au niveau des operations transactionnelles, et permet donc de définir des tests détaillés pour vérifier la sémantique des MTs. Côté performance, TMunit permet d'évaluer la performance des MTs en décrivant des scénarios plus complexes où les motifs d'accés aux données partagées peuvent être spécifié. TMunit propose une interface abstraite qui lui permet de stimuler n'importe quelle MT et il est accompagné d'une batterie de test pour que les aspects communs de sémantique et performance des MTs puissent être vérifiés aisément.

TMJava vise un support de langage riche pour les blocs transactionnel. Il propose une extension pour Java sous forme d'élément de langage transactionnels qui fournissent (i) la synchronisation des données automatisées entre les fils d'exécutions, (ii) le flot de contrôle transactionnel, et *(iii)* un traitement des exceptions augmenté. L'atout distinctif de TMJava est le fait qu'il propose l'utilisation des blocs transactionnels non seulement pour la synchronisation des données, mais aussi pour le traitement des exceptions, tandis que les supports de langage existants abordent uniquement l'aspect de synchronisation des données. En particulier, TMJava introduit une extension de langage novatrice, Atomic Boxes, qui permet aux programmeurs de traiter les exceptions de manière coordonnée. Cette extension est particulièrement intéressante pour propager les exceptions parmi tous les fils d'exécution concernés et pour exécuter des actions coordonnées de récupération d'exception à partir d'un état sûr du programme concurrent.

List of Related Publications

The contributions of the present thesis have resulted in the following workshop, conference and journal publications (in chronological order):

- Vincent Gramoli, Derin Harmanci, Pascal Felber. **Toward a theory of input acceptance for transactional memories**, *Proceedings of the 12th International Conference On Principles Of Distributed Systems (OPODIS'08)*, LNCS 5401 p.527-533, December 2008.
- Derin Harmanci, Pascal Felber, Vincent Gramoli and Christof Fetzer. **TMunit: Testing transactional memories**, *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, February 2009, Raleigh, North Carolina, USA.
- Vincent Gramoli, Derin Harmanci, Pascal Felber. **On the input acceptance of transactional memory**, *Parallel Processing Letters (PPL)*, 20 (1) p.31-50, March 2010.
- Yehuda Afek, Ulrich Drepper, Pascal Felber, Christof Fetzer, Vincent Gramoli, Michael Hohmuth, Etienne Riviere, Per Stenstrom, Osman S. Unsal, Walther Maldonado Moreira, Derin Harmanci, Patrick Marlier, Stephan Diestelhorst, Martin Pohlack, Adrian Cristal, Ibrahim Hur, Aleksandar Dragojevic, Rachid Guerraoui, Michal Kapalka, Sasa Tomic, Guy Korland, Nir Shavit, Martin Nowack, Torvald Riegel. **The Velox transactional memory stack**, IEEE Micro 30 (5) p.76-87, September-October 2010.
- Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. **Extensible transactional memory testbed**, *Journal of Parallel and Distributed Computing (JPDC)* - Special Issue on Transactional Memory, 70 (10) p.1053-1067, October 2010.
- Derin Harmanci, Vincent Gramoli, Pascal Felber. **Atomic boxes: Coordinated exception handling with transactional memory**, *European Conference on Object-Oriented Programming (ECOOP)*, vol. 6813 of LNCS, pp. 634-657, July 2011.

Table of Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

ACI	Atomicity, Consistency and Isolation properties of transactions
ALA	Asymmetric Lock Atomicity
AOP	Aspect-oriented Programming
ASM	An all purpose Java bytecode manipulation and analysis framework
BCEL	Byte Code Engineering Library
CAS	Compare and Swap
CCR	Conditional Critical Region
CM	Contention Manager
CTL	Commit-time Locking
CTR	Commit-time Relaxation
DLA	Disjoint Lock Atomicity
DSTM	Dynamic Software Transactional Memory
DSTM2	Dynamic Software Transactional Memory (2nd release)
ELA	Encounter-time Lock Atomicity
ETL	Encounter Time Locking
HTM	Hardware Transactional Memory
IWIR	Invisible Write Invisible Read
JVM	Java Virtual Machine

ABBREVIATIONS

LSA-STM	Lazy Snapshot Algorithm based Software Transactional Memory
McRT-STM	Multi-core Run Time Software Transactional Memory
Norec	No-orec Software Transactional Memory
NZTM	Nonblocking Zero-indirection Transactional Memory
OSTM	Object-based Software Transactional Memory (by Fraser&Harris)
RSTM	Rochester Software Transactional Memory
RTR	Real-time relaxation
SC	Sequential Consistency
SLA	Single Lock Atomicity
SNZI	Scalable Non Zero Indicators
SSTM	Serializable Software Transactional Memory
STM	Software Transactional Memory
SXM	C# Software Transactional Memory
TDRF	Transactional Data Race Freedom
TL2	Transactional Locking II
TM	Transactional Memory
TSC	Transactional Sequential Consistency
TSTM	Toronto Software Transactional Memory
VWC	Virtual World Consistency
VWIR	Visible Write Invisible Read
VWVR	Visible Write Visible Read
WRC	Weakest Reasonable Condition
WSTM	(The first) Word-based Software Transactional Memory

Chapter 1

Introduction

1.1 Concurrent programming and transactional memory

Until recently, the mainstream programming model was sequential. This programming practice was widespread because the previous generation of processor architecture, the uni-processor, could only perform instructions one after the other. For this generation of processors the increasing demand in processing power was satisfied by building faster and more powerful uni-processors. However, this trend has recently been broken, since hardware manufacturers could not practically continue increasing processing power of a single processor. Instead they offered a new generation of processors that incorporate multiple cores with average computing power integrated into a single chip [145, 179]. This change in processor architecture has a direct impact on the software development, because exploiting the computing power of the new processing architectures now absolutely necessitates parallelization. Hence, mainstream programming practice should now change its course towards concurrent programming [179, 180, 90, 113, 48].

Unfortunately, concurrent programming is more challenging than its sequential counterpart because it introduces additional difficulties for the programmer [180, 113, 48, 90]. A concurrent program design requires coordinating concurrently executing sequential tasks. In a way, a programmer needs to deal with multiple sequentially executing entities at the same time. Moreover, those sequential execution entities[1] are meant to communicate with each other to achieve a final common objective. A common challenge in such systems is to mask the non-determinism of the program state observed due to concurrency. In sequential programs the state of the program is well-defined by the set of instructions previously performed by the single execution entity. In a concurrent program, this is not anymore true

[1] In the context of a multi-threaded program an execution entity corresponds to a thread.

since between the execution of one instruction of a given execution entity to the next, other execution entities could change the program state. Despite this non-determinism of the program state, a concurrent program should still deliver a deterministic and correct result. This is one of the major reasons why parallel programming has been considered to be a difficult task.

Since the architectural trends force mainstream programmers to become concurrent programmers, recently researchers started looking for solutions that simplify the job of coding concurrent programs. As done for solving most of complex engineering problems, the current quest is to find new abstraction(s) that hide(s) most of the complexity of concurrent programming such that the programmer could focus on the task rather than struggling with difficulties merely due to the concurrency in the design.

The current widespread concurrent programming approach is **lock-based programming** [9, 113, 90]. This approach is appealing for two reasons:

- It applies to the shared memory programming model. This model is considered to be convenient to program with because the concurrent execution entities share the same memory space and hence there is no need to explicitly transfer data from one execution entity to the other. Instead, each entity can obtain/modify the data it is interested in, simply by accessing it at the corresponding memory address, as it would do in a sequential program.
- The programmer can mark the code sections, named critical sections, where s/he knows there will be accesses to shared data. By enclosing code into critical sections the programmer actually requests exclusive access to the shared data as long as the execution stays in the critical section. The exclusive access in critical sections is provided by acquisition and release of locks at the beginning and at the end of critical sections respectively. This programming discipline allows the programmer to code the same way as a sequential program all through the execution entity code.

Although having appealing properties, the main problem with lock-based programming is that the programmer needs to specify locks manually to enter and to quit critical sections. This is an error-prone process where the mapping of the shared data to the lock that protects it should be done manually by the programmer. Apart from that, the concurrency of operations in execution entities makes the use of locks even more difficult and results in classical problems such as deadlocks, livelocks, priority inversion etc. (see **Section 2.2** for more details). Since handling these problems is not trivial, lock-based programming is not well suited as a concurrent programming paradigm [180, 9, 113, 90].

In the last decade one approach that attracted a lot of attention, especially in multi-threaded application domain, is to write concurrent programs using the runtime support called Transactional Memory (TM). TM can be considered as a system running under the multi-threaded application to provide the consistency of the accesses to shared data without explicit intervention of the programmer (such as managing the locks). TM is capable of doing this by introducing the transaction abstraction to the programmer (this type of transaction is generally called **memory transaction** in the literature to distinguish it from database transaction [55]). With this abstraction the programmer can group a set of operations such that they appear to be executed in a single step to other threads. Another convenient property of the abstraction is isolation where the transaction ensures that it runs as if it were the only thread running in the application. These two properties allow the programmer to ignore the operations concurrent to the transaction, as if they are not being executed. This way, writing a multi-threaded program using the transaction abstraction gets closer to writing a single-threaded sequential program.

Writing programs using TM, more specifically using the transaction abstraction provided by TM, necessitates a different way of coding and design compared to lock-based programming and, hence, the corresponding programming paradigm is generally called **TM-based programming**. At first sight, TM-based programming may be considered the same as programming with databases, since both are based on using the transaction abstraction. However, programming that makes use of databases isolates the data under the control of a database from the rest of the application data. Furthermore, accessing data in a database is performed through a set of functions. TM-based programming opts for simplicity of programming and hence eliminates both the isolation of data[1] and the data access through a function interface [55, 90]. The programmer using TM only needs to spot the code sections that should execute in a transaction and to mark them in the code. Note that, when compared to lock-based programming, marking such code sections is even simpler in TM-based programming since the programmer does not need to manage locks.

[1]The elimination of isolation simplifies programming for the application programmer since s/he does not need to think whether data is stored in a database or not, instead it shifts the burden to the TM designer. The data isolated in a database can only be accessed transactionally through the database. A notable difference of TM-based programming (introduced through the elimination of isolation) is that data can be accessed both in a transactional manner and directly, and if care is not taken there can be transactional and direct concurrent accesses to the same data, introducing inconsistencies in application execution [55, 26, 90].

1.2 Challenges in TM-based programming

Although TM-based programming simplifies concurrent programming with its elegant language interface and by resolving many of the issues that make concurrent programming hard with lock-based programming, there are still open issues remaining to be solved.

Using TM-based programming requires a significant amount of support to be introduced in the software stack. We can group the support required in two levels: run-time and compile-time. Different challenges exist at these two levels in order to satisfy the required support.

1.2.1 Run-time challenges

The run-time level support corresponds to TM implementations that provide transactional behavior. At this level we see a variety of TM implementations that provide different transactional guarantees but, unfortunately, there is still not a consensus on which of the possible transactional guarantees is the best to support application requirements. Even when the transactional guarantee is determined, the inherent complexity of providing a transaction abstraction leads to sophisticated TM implementations and makes it hard to specify clearly the semantic properties of a given TM implementation. The same complexity also introduces challenges in testing TM implementations.

Another aspect of a TM implementation that requires attention is the performance it delivers. This aspect is even the mostly studied aspect of TMs because without acceptable performance, it is very hard for TM-based programming to be considered as a widespread programming paradigm. In that respect, the large set of proposed TM implementations are classified broadly on the basis of whether they are software-based (Software TM, STM for short), hardware-based (Hardware TM, HTM for short) or a hybrid solution which has both software and hardware components. STMs introduce a significant overhead since each data access of a code section running under transactional semantics corresponds to a large number of instructions instead of a simple memory access instruction. While this hampers the performance of an STM, its flexibility allows *(i)* transactions of any size and of any type to be supported (hence it is attractive to be used at least as a fallback alternative) *(ii)* exploring different design alternatives. Conversely, HTM solutions deliver good performance but are limited by transaction size or type.

When designing a TM runtime neither the semantic nor the performance aspects can be overlooked and they need to be considered together for an acceptable solution. This makes the TM design even more challenging.

1.2.2 Compile-time challenges

The compile-time support required for TM-based programming mainly consists of the language support required to provide the transaction abstraction. This language support has two main components: the specification of the language constructs required for transaction abstraction and the transformation of the code enclosed in transactions to code that uses TM implementations through TM specific function calls (this transformation is also called **transactification**).

For the specification of language constructs, the ideal language construct for a transaction is a transaction block that encloses arbitrary program statements and executes them under transactional semantics. However, it is not straightforward to use such ideal transaction blocks in programming languages. There are two reasons for this:

- The transaction abstraction is not a solution for all concurrent programming issues [73, 167]. It offers simplified data synchronization, while it cannot be used with the same ease for synchronizing tasks for coordination. In this sense, the language constructs that provide transaction abstraction need to co-exist with task synchronization mechanisms without harming the correctness of the application and keeping the simplicity they offer for data synchronization.
- The ability to use some language features in a transaction block prevents us to use a simple transactional semantics [190, 167, 59]. For example, the use of irrevocable language constructs (such as I/O accesses, systems calls, external library calls) disallows the rollback-and-restart behavior of transaction constructs. Another example is the necessity to support different types of exceptions in a transaction: some exceptions can be handled only if the transaction commits its changes before raising the exception (the recovery of such exceptions can be performed only if the program state at the moment of exception raise is known), while for some others it is better to roll the transaction back in order to preserve application consistency.

In short, the required language constructs to support TM should be flexible enough to cover all the necessary semantic properties discussed above while keeping the simplicity of the ideal transaction block as much as possible.

The transactification process, compared to specification, is a mechanical process. The main challenge in the transactification is to generate code in all possible allowed situations and keeping the semantics of the generated code as required by the specification.

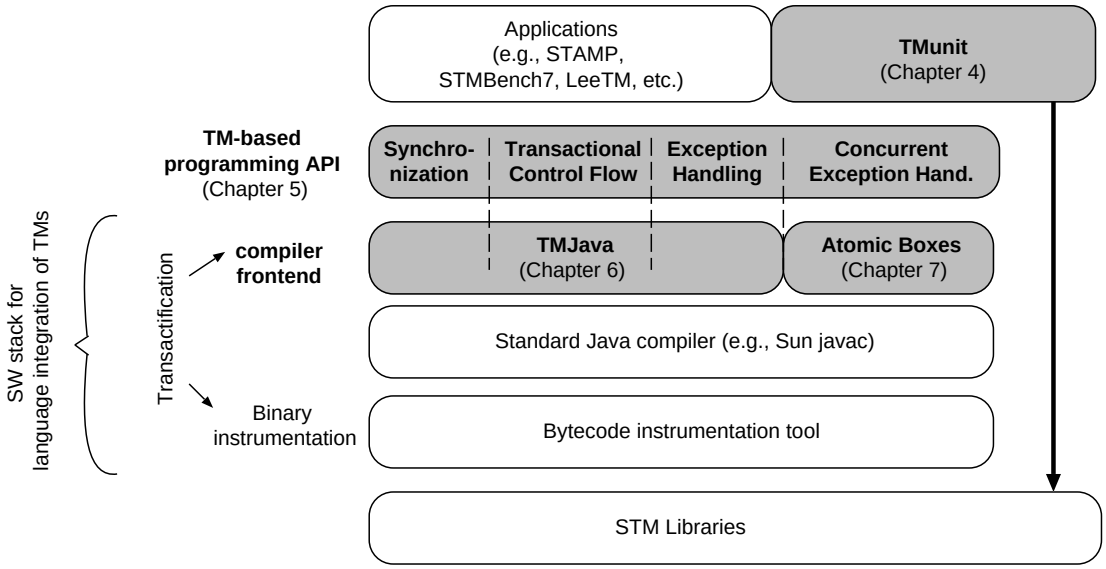


Figure 1.1: An overview of the novel tools constituting part of the contributions of the thesis. The figure illustrates the locations of each tool (the shaded components) on the software stack. Each shaded element is labeled with the name of the tool to which it corresponds and its location in the thesis text.

1.3 Thesis contributions

This thesis contributes to the development of the TM support for programming languages at both the run-time and the compile-time levels. The major part of the contributions have been embodied in tools which are depicted by shaded components in **Figure 1.1**. Although the contributions of the thesis apply most of the time also for HTMs, we mainly focused on STMs for their availability and their ability to offer diverse designs.

For the run-time level of TM support, our first contribution is a qualitative classification of STM designs (**Chapter 3**). In this work, we first point that many STM designs abort more often than necessary (i.e., STMs abort even in some cases where application correctness is preserved). We find different classes of aborts STM designs perform unnecessarily and classify designs accordingly. The classification sheds light on the tradeoff between the simplicity and the effectiveness of an STM. This work has been published in a conference publication *"Toward a Theory of Input Acceptance for Transactional Memories"* (OPODIS'08) [69] and a follow-up journal publication *"On the Input Acceptance of Transactional Memory"* (PPL'10) [70].

Our second contribution for the run-time level (appearing in **Chapter 4**) is a domain-specific language and an associated framework, TMunit, which allows testing both semantic and performance aspects of TM implementations (work on TMunit has been published in the workshop publication *"TMUNIT: Testing Transactional Memories"* (TRANSACT'09) [85], and in two journal publications *"Extensible Transactional Memory Testbed"* (JPDC'10) [87] and *"The VELOX Transactional Memory Stack"* (MICRO'10) [14]). The language is unique in that it allows specification of deterministic scenarios for TMs, which we call semantic tests, with which it is possible to test exactly the desired semantic property of a TM. The language is simple and concise, leading to a rapid specification of such deterministic scenarios. The language also offers the specification of synthetic workloads for TMs (we call these performance tests), to test TMs under desired workload conditions. This way, it helps TM designers to evaluate the performance of TM implementations. The fact that the language permits the specification of both semantic and performance tests is an important asset since a TM designer needs to consider both aspects together to provide the best possible design. The TMunit framework (appearing at the top level of the software stack in **Figure 1.1** and directly driving the STM libraries) interprets this domain-specific language and offers an abstract interface for TMs so that theoretically any TM can be tested with the specified semantic and performance tests. Last but not least, the deterministic scenarios designed for TMs can be used to test and debug TM implementations by generating deterministic interleavings of TM library function executions.

For the compile-time support, our first contribution (appearing in **Chapter 5**) is the specification of a language extension (denoted as the TM-based programming API on **Figure 1.1**) for supporting a transactional construct in Java. Although part of the specification reuses the existing specification for C++ [12], our specification goes beyond the features proposed by the C++ specification. The focus of the C++ specification is only the data synchronization aspect that can be offered by TMs. However, we argue that transaction abstraction can also be useful in other aspects of programming and we describe novel transactional control flow and exception handling constructs (both for single-threaded and multi-threaded exception handling) that are not present in the C++ specification.

An implementation of the language specification is also made available to the programmers via a compiler framework, TMJava (**Chapter 6**). As can be noticed from **Figure 1.1** the transactification that we have performed is split into two layers in the software stack that we have used for integrating TM to the Java language. In this setting, TMJava appears only on the layer above the traditional Java compiler and operates in coordination with a bytecode instrumentation tool that performs (only) the transactification of class methods annotated specifically to be executed in transactional context. The role of TMJava in this

design is to transform the source code expressed according to our language specification to an equivalent Java code where transactional regions are mapped to annotated class methods (that are to be transactified by bytecode instrumentation tool).

As part of our Java language specification, we also propose an original extension, Atomic Boxes (presented in **Chapter 7**), that allows programmers to handle exceptions among multiple threads in a coordinated manner (see also the API and compiler frontend layer of **Figure 1.1** where this extension is made explicit). The extension makes use of the transaction abstraction to keep a multi-threaded application always in a consistent state even when exceptions are raised. On the basis of this feature, the extension is capable of reverting the complete application to a consistent state upon an exception and provides recovery actions that can be coordinated among multiple threads. This is particularly useful to handle exceptions having implications on threads other than the one raising the exception. Our work on Atomic Boxes has been presented in the conference ECOOP'11 under the title *"Atomic Boxes: Coordinated Exception Handling with Transactional Memory"* [86].

1.4 Thesis organization

The content of the thesis is organized in two major parts. The first part comprises an analysis of TM from different aspects. Background material on TM as well as semantic issues on TM and associated transactions are presented in **Chapter 2**. This is followed by **Chapter 3** that qualitatively classifies STM designs with respect to their abort performance. **Chapter 4**, the last chapter of the first part, presents our testing framework, TMunit and an associated testbed, that allows experimenting both semantic and performance properties of TM implementations.

The second part of the thesis explains our Java language support for TM. **Chapter 5** introduces the language extension syntax and its associated semantics. This language extension includes the language constructs we propose to provide automated data synchronization as well as transactional control flow and coordinated exception handling. While **Chapter 6** explains the implementation details of data synchronization and transactional control flow constructs specified in **Chapter 5**, **Chapter 7** discusses the semantics of the proposed construct for coordinated exception handling, presents its implementation details and evaluates the performance of the construct.

The thesis is concluded by **Chapter 8** where we summarize the main findings of our work and present possible future research directions.

Chapter 2

Background

2.1 Introduction

A software system can be described using two complementary approaches; one data-centric and the other activity-centric [114]. The data-centric approach describes the system based on data structures that get involved in the execution. In this approach *i)* data is grouped into data structures that represent components or subsystems *ii)* data structures interact with each other (as components or subsystems) to perform the required functionality. The activity-centric approach describes the system in terms of possible activities. An activity can involve sets of function call-chains, event sequences, tasks, sessions, transactions and threads. Some activities can even span multiple threads and may even correspond to system-wide use cases.

Neither the data-centric nor the activity-centric approach provide a complete understanding of the whole software system, since a data structure can be involved in multiple activities and an activity can span multiple data structures. However, these two approaches lead to two complementary group of properties for software systems [114]:

- **Safety:** *Nothing ever bad happens*. This group of properties are data-centric and deal with correct manipulation of data structures. Failure of a safety property leads to unintended and unexpected execution behavior.
- **Liveness:** *Something eventually happens within an activity*. This group of properties are activity-centric and deal with progress of activities. Failure in ensuring a liveness property can lead to lack of progress of some activities towards completion, which can even bring the system to a halt.

Some of the safety and liveness properties of systems are related more to the quality of the software and this further leads to two groups of quality concerns [114]:

- **Reusability:** The utility of objects and classes across multiple contexts.
- **Performance:** The extent to which activities execute soon and quickly.

Even without considering quality concerns, ensuring safety and liveness properties can sometimes be contradictory in designing programs, i.e., improving or introducing a safety property may destroy some liveness property and vice versa. Obviously, all safety properties can be guaranteed such that no activity can ever terminate, but of course this does not produce a useful program. Hence, designing a software system requires a programmer to care both the safety and liveness properties expected from a program.

Taking both types of properties into account is already not easy for sequential programs. In concurrent programs, however, an additional difficulty in ensuring safety and liveness properties is the need for correct synchronization among threads. A particular type of synchronization that introduces difficulties in ensuring safety and liveness properties of multi-threaded programs is data synchronization.

The work presented in this thesis mainly focuses on a recent data synchronization mechanism, transactional memory (TM) and in particular the semantics and performance associated to it. Hence, in this chapter, we present fundamental terms and issues related to semantics, performance and design of both TMs and TM-based programs. We start by describing the difficulties of programming with locks in **Section 2.2** to motivate the use of TMs for concurrent programming. Then, we present TM and its fundamental semantics in **Section 2.3**. **Section 2.4** discusses semantic issues specific to the transaction abstraction that should be supported by TM implementations and presents a spectrum of possible transaction guarantees that can be provided by TMs. Semantic issues specific to a language construct that provides the transaction abstraction and existing solutions to solve the related problems are presented in **Section 2.5**. **Section 2.6** discusses fundamental notions in evaluating TM performance and **Section 2.7** presents different STM implementations focusing on their distinguishing design properties. The chapter is concluded by **Section 2.8** explaining how the content of the chapter is related to the work presented in other chapters of the thesis.

2.2 Data synchronization and locks

The shared memory paradigm allows the programmer to communicate data between different threads through the sharing of the memory address space. This is very convenient for programming, however, for an application to run correctly, the accesses to shared data should be synchronized properly.

Until recently, the mainstream solution for providing data synchronization in shared memory programs was synchronizing threads at the entry and exit of critical sections (sections of code where shared data is accessed). The aim of this synchronization is to protect shared data from concurrent threads that would try to access the same data and/or modify it. This protection is achieved through mutual exclusion, i.e., by allowing only one thread to access the shared data during the execution of a critical section. Mutual exclusion is possible thanks to the use of locks such that critical sections are surrounded with acquisition and release of a set of locks (for this reason the programming discipline of using locks to protect critical sections is also called **lock-based programming**). While a thread is in a critical section, no other thread accessing data used by the critical section is meant to perform execution. Such exclusion of threads is only possible if all threads protect their critical sections using locks.

Semantically, the use of locks is appealing since, when correctly used, they provide the isolation of a critical section from the rest of the system. This isolation allows the programmer to access shared data as if s/he were writing a sequential program inside a critical section. Practically, however, using locks to isolate correctly a critical section has numerous difficulties [98, 90]:

- **Error-prone:** Writing code using locks is error-prone. There are several factors that make using locks error-prone:
  - **Manual handling:** Each protected data item or object requires a separate lock and the programmer needs to keep track of the mapping between data and the corresponding locks. Moreover, the programmer should be careful not to forget acquiring/releasing locks and performing the acquisition/release operations later/earlier than required. Failing to handle acquisition and release of locks correctly causes **data race**s, i.e., situations where multiple threads try to access the same data at the same time and the resulting behavior of the code depends on the sequence of accesses performed by the threads (hence the behavior is non-deterministic). So, simple mistakes in handling locks can cause serious bugs.
  - **Deadlock:** When acquiring a lock a thread accepts that it can be blocked if some other thread owns the lock. In other words, by using a lock, a thread accepts that its progress can depend on the behavior of another thread. It is possible that during execution such progress dependencies form a circle of dependence relations and, as a result, each of the threads that have one of those dependencies waits for one another and none of them are able to progress, bringing the application to a complete halt. Such situations violating liveness

are called **deadlock**. Deadlock situations occur very frequently because locks are acquired in different orders by concurrent threads (e.g., two threads acquire two locks in reverse order). Even in cases where deadlock is caused merely due to the lock acquisition orders, it is not easy to detect, neither during application design nor during testing.

  - **Waiting and convoying:** The progress dependencies mentioned for deadlocks may not always violate liveness, but may cause waits. For example a thread T that acquired a lock can be waiting for some other reason (e.g., communication over the network etc.) and other threads that try to acquire the same lock can wait for T until T stops waiting. If the lock protects a data used by many threads that would generate a long queue of threads waiting for T. This phenomenon is called **convoying**. For some applications (e.g., real-time applications) such long waiting times result in erroneous behavior and situations like convoying should be avoided.
  - **Priority inversion:** The blocking nature of locks can result in cases where thread priorities are not respected. Assume that a low priority thread owns a lock that is required by a high priority thread. When the high priority thread tries to acquire the lock it will be blocked until the low priority thread frees the lock. This way the priority of the high priority thread becomes ineffective at least for a temporary duration. If in the above situation a medium priority thread delays the low priority thread, it will also be delaying the high priority thread indirectly and will act as if it had a higher priority than the high priority thread. The fact that priorities are not respected poses a problem in real-time systems and can even violate safety since such delaying prevents high priority threads to meet their deadlines.

- **Lacking composition:** Lock-based code is not composable. In other words, the reusability of lock-based code is limited in that it is often very difficult, if not impossible, to take two correct lock-based pieces of code, combine them without modification and expect that safety and/or liveness is still ensured. Composability requires hiding the internals of a code that is used as a component, while lock-based code breaks the hiding requirement, because the user of the lock-based code needs to know the locks and the order in which they are acquired to ensure that his/her manipulations will not cause a data race or a deadlock. Consider the example by Harris *et al.* [91] where we would like to compose atomic `insert` and `remove` operations of a hash-table to move an element from one table to the other atomically. The intermediate state where the element is removed from the source table and not yet added to the desti-

nation table should not be visible to other threads to avoid data races (i.e., to ensure safety). However to eliminate data races, the lock acquisition and release contained in the insert and delete operations should be removed and combined differently to wrap around the insert and delete. This breaks the abstraction that is offered by the `insert` and `remove` operations of the hash-table, because the details on locking needs to be exposed for composition. An example where liveness is not ensured (instead of safety) due to the composition of lock-based code is the *nested monitor problem* [114]: a lock-based object A is contained in another lock-based object B and the contained object A performs waiting in its lock-based method (which is also called in a lock-based method of B). While the waiting performed in the method of object A results in the release of the lock for object A, the lock held for object B will still be held, blocking the other threads to use object B. If the only way to notify the waiting thread in the method of object A is to call a method of the object B (which is also guarded by a lock), this will not be possible since its lock is held. The solution to this problem again needs the careful redesign of the objects and hence, does not allow direct inclusion of object A inside object B.

Similar to data/object composition, composing actions that are within lock-based code and that require waiting can lead to problems. Suppose we would like to obtain two consecutive elements of a buffer atomically using its `get` method which return only one element and which blocks until there is at least one element in the buffer [91]. Calling the `get` method twice is not an option since it is not guaranteed to return consecutive elements. Implementing the correct behavior requires the waiting condition of the `get` method to be modified, which again breaks the hiding principle required by composition.

- **Lacking fault tolerance:** A thread that acquired a lock can always crash and cease to execute. In such a case other threads that require the lock can never acquire it since the crashing thread can never free the lock. With systems using locks such situations violationg liveness are hard to avoid.
- **Hard to achieve scalability:** Using locks can be simplified by **coarse-grained locking** where the programmer uses a lock to protect more than one shared data item. This results in few locks and large critical sections. Although it is easier to manage code in this way, such a choice usually hampers the performance since larger critical sections cause serialization of thread executions to a big extent, leading to unscalable code. It is possible to obtain scalable code but this requires associating each lock to (typically) each shared data item (**fine-grained locking**). This solution is hard to

program correctly since all the problems described above get more difficult to handle with an increasing number of locks.

All the above difficulties make it complex to develop correct concurrent code using locks. Dealing with this complexity seriously hampers the software development time and there is an increasing need to develop concurrent programs due to emerging multi-core processors. Hence, software designers and researchers alike look for data synchronization solutions other than using locks.

2.3 Transactional memory

One recently proposed alternative to using locks for the protection of critical sections is transactional memory (TM). With the use of locks, consistent shared data access in critical sections is provided by mutual exclusion, however, many of the problems described in **Section 2.2** are due to the waiting mutual exclusion induces. TM instead adopts speculative execution of concurrent critical sections and evaluates the results of their speculative executions to check whether consistency is violated or not.

Syntactically, critical sections are enclosed in a code block with TM, much as is done with locks protecting them. In programs using TM, the resulting code blocks are called transactions and they are delimited, and distinguished from regular code, by a starting operation (generally called *start*) and a terminating operation (generally called *commit*). A higher level syntax to delimit such code blocks is

```
atomic{
    // transaction code block
    ...
}
```

where the starting operation is expressed with the `atomic` keyword combined with opening brace and the terminating operation corresponds to the closing brace. The code that appears between start and commit operations is defined as **transactional code**. Any code that is not transactional is called **non-transactional code**. Any memory access that takes place in transactional code is called **transactional access**, while a memory access performed in non-transactional code is called **non-transactional access**.

Semantically, a transaction is defined as a set of operations that performs a transformation on the system state and that has the atomicity, consistency, isolation and durability properties [71, 72, 90]. These properties can be described as follows:

- **Atomicity:** The execution of a transaction maintains an *all-or-nothing* semantics: a transaction either executes fully (i.e., it *commit*s) or acts as if it did not execute at all (i.e., it *abort*s). An implication of atomicity is that a transaction is seen as a single *indivisible* operation.
- **Consistency:** A transaction always acts on a consistent state. In other words, a transaction is capable of capturing a consistent snapshot for the state of the data items it accesses and performs modifications based on this consistent snapshot. Starting from a consistent snapshot and applying modifications atomically, as defined by the atomicity property, the transformation of a transaction execution results also in a consistent state. As long as the application does not require consistency requirements other than the consistency of concurrent accesses to the shared data (an example for such requirements can be complex invariants defined on several data items), consistency is satisfied by the atomicity and isolation properties of a transaction. For this reason, in the rest of the document we do not consider consistency property of transactions; atomicity and isolation will be enough to satisfy transaction semantics.
- **Isolation:** Each transaction executes as if it were the only one executing in the system. This implies that a transaction cannot see tentative modifications performed by other transactions. A transaction can only be aware of modifications made effective by committed transactions.
- **Durability:** Once a transaction commits, its result should persist.

These properties express the fundamental behavior expected from transactions as defined by the database community and are generally expressed altogether as ACID properties of transactions. TM adapts the transaction abstraction defined by the database community but slightly modifies the properties such that the required promises of transaction abstraction is kept, while any computational overhead that is not important in program execution is ruled out. The resulting transaction is generally called **memory transaction**. One notable modification in memory transaction is the strictness of durability. It is generally accepted that memory transactions exclude the durability property [113, 55, 90]. We can restate this claim as memory transactions supporting a lightweight version of durability: committed state persists as long as required by the program (as is the case for any variable for example) [55]. Since the durability requirement of memory transactions is quite weak they are said to be ACI instead of being ACID.

Although semantically database and memory transactions may seem similar, there is an important distinction between them: in database transactions the data stored in the database is separated from the data in the program. Database transactions perform transactional access only for data on the database, while memory transactions can perform

transactional access on any data structure of the program [55, 90]. Removing the separation of data in the program, a new problem arises for memory transactions. If care is not taken a multi-threaded program can perform transactional and non-transactional accesses to the same data item concurrently. This fact requires an adaptation of the atomicity and isolation properties for TMs (this issue is treated in **Section 2.4**).

If we compare the use of locks to transactions in terms of the ACI properties they provide to a critical section, we can say that locks, when used correctly, also achieve atomicity and isolation. Atomicity is achieved because only one thread at a time can modify data items of common interest and isolation is achieved because, by the nature of mutual exclusion, threads other than the one in the critical section will be blocked, allowing the thread in the critical section to execute in isolation. Meanwhile, contrary to using locks, a transaction has the ability to rollback and restart. This rollback ability allows speculative executions of transactions and frees transactions from blocking behavior.

Thanks to their speculative nature, transactions also allow concurrent executions of critical sections even if these executions include accesses to common data items. So if consistency would not be violated, transaction execution allow parallel executions, increasing concurrency. This is not possible when locks are used since mutual exclusion impose the execution of only one thread in its critical section.

Last but not least, transactions are clearly simpler to use for the programmer since the programmer is exempted from the onus of acquiring and releasing locks. Instead, TM manages the protection of any data accessed within its code block. The programmer only needs to delimit the code blocks where shared accesses occur.

When we consider TM semantics, we distinguish two different semantics: *transaction semantics* and *transaction block semantics*. Transaction semantics describe the semantics associated to TM implementations, while transaction block semantics are concerned with the semantics of the language constructs providing the transaction abstraction to the programmer. We analyze each of them in separate sections below.

2.4 Transaction semantics

Although we have presented the fundamental transaction semantics in terms of transaction properties in section **Section 2.3**, a transaction in TM-based programming has additional semantic requirements [79]. Hereafter, the semantics of transactions with these additional requirements will be called TM semantics (we also use term *transaction semantics* interchangeably with TM semantics). In this section, we discuss TM semantics in detail and describe different propositions for it.

The first study that defines TM semantics in a comprehensive way is due to Scott [162]. In his work, Scott describes formal definitions for the guarantees that TMs should provide. The formalism uses histories, i.e., interleavings of transactional operations, that represent program executions and a TM guarantee is described by sets of histories that it accepts. Note that program executions could have been defined using interleavings of instructions constituting the transaction operations instead of interleavings of transactional operations. But to reduce complexity, generally transactional operations are assumed to execute atomically and this style of semantic description is called the **sequential semantics** of TMs [90]. In general, sequential semantics are accepted to be representative of TM semantics (e.g., a more recent formal study by Doherty *et al.* [44] on TM semantics also uses sequential semantics), hence, in the sequel, we consider only sequential semantics.

Scott [162] identifies two types of guarantees that constitute sequential semantics of TMs: correctness and progress. We take the same approach and group the studies on sequential semantics according to these two guarantees in **Sections 2.4.4** and **2.4.5**. But before explaining different guarantees, we need to analyze an important aspect of TM-based programs that serves to group different correctness guarantees: *transactional data races*. We describe what transactional data races are and how correctness guarantees are classified according to transactional data races in the following section.

2.4.1 Transactional data races and levels of isolation

An important question that has been first raised by Blundell *et al.* [26] is "*What should be the semantics of transactions when there are concurrent accesses to same data both inside and outside transactions?*". This questions reveals the fact that if a transaction does not take the necessary measures to protect itself against concurrent non-transactional accesses, as it does against concurrent transactional accesses, unexpected data races may occur such as lost updates, inconsistent or dirty reads [74, 166, 7, 130, 120, 90]. Such data races that occur between transactional and non-transactional accesses are called **transactional data race**s [39] or **violation** [7].

The basic transaction criteria, which have been defined for databases, ignores the existence of transactional data races simply because such races do not exist in database systems; a database system handles nothing but concurrent transactions. Conversely, in TM-based programming shared data can be accessed both in transactional or non-transactional code, so transactional data races threatens TM-based code correctness as any ordinary data race. This lead Blundell *et al.* [26] to define two level of guarantees according to whether the guarantee allows transactional data races or not. The guarantee that ignores transactional

data races is **weak atomicity** and the guarantee that does not allow transactional data races is **strong atomicity**.

In the literature, strong and weak atomicity are also called, respectively, strong and weak isolation by different groups. We prefer to adopt this terminology instead of the original name(s) weak/strong atomicity for the following reason. Actually both atomicity and isolation properties of transactions define the interference among transactions in opposite directions with respect to a transaction T. By enforcing other threads to perceive the changes of transaction T in an indivisible step, atomicity prohibits the execution of T to interfere in the execution of other concurrent transactions/threads until its commit is effective. Similarly, by enforcing T to execute as if it were the only one executing in the system, isolation property prohibits the execution of other transactions/threads to interfere in the execution of T. A transactional data race can break the interference in both directions. We think that the term isolation better reflects this interference effect of transactional data races, so we also refer to the guarantee(s) that allow/disallow transactional data races as **weak/strong isolation**.

Strong isolation effectively removes all the transactional data races that could occur under weak isolation and hence is more suitable for multi-threaded applications compared to weak isolation. Without strong isolation, the simplicity of using transaction abstraction is seriously degraded, since the programmer should still consider interactions between transactional and non-transactional code. Strong isolation assures the programmer that enclosing data accesses in a transaction frees him/her from considering any data races that can occur associated with those accesses.

Although strong isolation is the ideal semantics for TM, especially for the acceptance of TM as a mainstream data synchronization mechanism, many STMs support only weak isolation for performance reasons (however, HTMs guarantee strong isolation by design). It should be noted that data races similar to transactional data races in nature exists also for lock-based programming. They arise when two concurrent accesses target the same location, one protected using a lock and the other not[1] . Hence, it may be claimed that STMs that follow weak isolation at least provide the same guarantee as lock-based programs without going into the hassle of managing locks explicitly. Unfortunately, this is not true and weak isolation provides a weaker guarantee then lock-based semantics. This has been shown in the literature for two canonical programming patterns that are handled correctly under lock-based semantics while causing data races under weak isolation: *publication* and *privatization* [130, 7, 90, 128, 169].

[1]Such data races are called asymmetric races [152] and are generally regarded as an incorrect omission of locking for the non-protected access.

Publication and privatization are important and frequently used programming patterns in multi-threaded programming since they allow programmers to determine when a given data is private or shared at a given point in the program code. If data is known to be private it can be accessed directly by a thread, if not the data needs to be accessed using a synchronization mechanism. In the case of STMs, the first practical need for a simple form of privatization has been mentioned by Dice and Shavit [43] for freeing deleted nodes in a shared list. In general, for multi-threaded programs adopting STMs the possibility to switch between private and shared usage of data is advantageous in two ways [171, 116]: *(i)* Data synchronization using STMs is costly in terms of performance. Hence, wherever possible, a programmer may like to privatize shared data to speed-up execution. A good example on how privatization can speed-up transactional programs has been shown by Scott *et al.* on a delaunay triangulation benchmark [163]. *(ii)* Use of library calls and execution of I/O actions that are not under the control of STM are either not supported by STMs or results in significant performance degradations. Switching to private usage of shared data for library calls and I/O actions can be used to avoid these problems.

These appealing properties of publication and privatization urged researchers to augment the isolation of weakly-isolated STMs such that publication and privatization patterns that are correct under lock-based semantics are also correct under the augmented TM guarantee. Although the mechanisms proposed in these studies provides an isolation guarantee weaker than that of strong isolation, the practical use of publication and privatization makes them valuable for ensuring isolation. In the next sections, we discuss in detail publication and privatization patterns and the transactional data races associated to each of them.

2.4.2 Publication

Publication occurs when a thread makes a(n) object/variable visible to other threads using a synchronization mechanism [113, 130]. Put differently, publication is a phase of a multi-threaded program where a piece of data switches from private usage (where it can be accessed directly) to shared usage (where it should be accessed using a synchronization mechanism) in a controlled manner. The control is ensured by the use of a synchronization mechanism. In the case of STM, the synchronization construct to be used is a transaction and the publication is to be performed inside the transaction.

	Thread 1	Thread 2
1	SG1;	
2	{*publishing action*}	
3		atomic{
4		SG2;
5		}

	Thread 1	Thread 2
1	atomic{	
2	SG2;	
3	}	
4		{*privatizing action*}
5		SG1;

Figure 2.1: The patterns for publication (on the left) and privatization (on the right). The pattern illustrations are derived from the Figure 4 (for publication) and Figure 3 (for privatization) of [130].

The publication pattern is depicted on the left side of **Figure 2.1**. In the figure, `SG1` is a group of statements preparing the content of data to be published (e.g., this can be private initialization of an object). The *publishing action* is an action that makes the data visible to the other threads using a synchronization mechanism (e.g., a flag can be set to true to indicate other threads that the data is now published, or a private object reference can be added into a shared list). In the case of STM the synchronization mechanism used for the publishing action is a transaction, which we call a **publishing transaction**. `SG2` is a group of statements in which some statements access the published data.

Publication safety is the ordering guarantee enforcing that all the non-transactional access in `SG1` occurs *before* any of accesses in `SG2` if the transaction enclosing `SG2` is ordered *after* the publishing transaction (hence left side of **Figure 2.1** depicts the desired ordering). The definition of publication safety implies that if the transaction enclosing `SG2` is ordered *before* the publishing transaction no ordering guarantee is required. This is normal since without the result of publishing transaction being visible to other threads, the publication is not complete and statements in `SG2` should not be able to use the shared data (by design of publication pattern).

An STM implementation based on merely weakly-isolated STM can violate publication safety with an inconsistent read as shown on line 3 of **Figure 2.2** (the example is inspired from Figure 1 of [130] and Listing 7 of [40]). The problem in the execution presented in **Figure 2.2** is that Thread 2 can access the state of the shared object `ShObject` prior to publication without its transaction detecting any conflicts. Although Thread 2's transaction is ordered *after* the publishing transaction, it reads the state of `ShObject.x` *before* initialization (i.e., reading an object state *before* the publishing transaction)[1] .

[1]One can argue that what leads to the problem is the way the code is written, i.e., the programmer did not perform the publication check (i.e., whether the shared object `ShObject` is published or not) before accessing it. While this is true, when we consider that a transaction should be atomic, it should not matter where the publication check is performed (even a compiler could have reordered the check in the position

	Thread 1	Thread 2
1	ShObject = **new** Object();	
2		**atomic**{
3		tmp_x = ShObject.x;
4	ShObject.x = val1;	
5	ShObject.y = val2;	
6		
7	// publishing transaction	
8	**atomic**{	
9	published = true;	
10	}	
11		tmp_y = ShObject.y;
12		if(published)
13		z = tmp_x + tmp_y;
14		}

Figure 2.2: Execution schedule for the publication pattern depicting an inconsistent execution under weak isolation. The inconsistency is the result of the inconsistent read observed by Thread 2 (line 3) while `ShObject` is being initialized in Thread 1 (lines 1-5).

2.4.3 Privatization

Privatization occurs when a thread makes a(n) variable/object inaccessible to other threads by using a synchronization mechanism [113, 130]. Hence, privatization is a phase in the multi-threaded program (conceptually the reverse of publication) where a piece of data switches from shared usage to private usage using a synchronization mechanism.

The privatization pattern, which is symmetric to the publication pattern, is illustrated on the right side of **Figure 2.1**. In this pattern, `SG1` is a group of statements where the privatized data is accessed directly, the privatizing action is the transaction that privatizes the data (named generally **privatizing transaction**), and `SG2` is a group of statements that access the privatized data assuming that data is still shared.

Privatization safety is the ordering guarantee enforcing that all the non-transactional access in `SG1` occurs *after* any of accesses in `SG2` if the transaction enclosing `SG2` should logically be ordered *before* the privatizing transaction (hence **Figure 2.1** depicts the desired ordering). Privatization safety is hence violated when some actions (or their effects) in `SG2` are delayed and actually occur after the privatizing transaction, causing data races between the transactional accesses in `SG2` and non-transactional accesses in `SG1`. The situations

shown in the example for optimization) and, hence, the code of the programmer is semantically correct and a programming language providing such semantics for transactions should behave according to this semantics.

where privatization safety is violated can be classified in two groups: *delayed cleanup* and *delayed conflict detection*.

Delayed cleanup problem [128, 169, 116, 90] occurs when private accesses of `SG1` in **Figure 2.1** are interleaved with a commit or an abort operation of `SG2` where modifications to be performed by the commit or abort operation are delayed to take effect and are ordered after the privatizing transaction. Figure 2.3 illustrates the inconsistent read and lost update problems that can occur due to delayed cleanup both in direct and deferred update STMs[1] (a detailed explanation of the example is deferred to the end of the section). As it can be noticed, the violation of privatization due to delayed cleanup causes incorrect behavior of the non-transactional accesses only.

Delayed conflict detection problem occurs when the transaction enclosing `SG2` continues to execute concurrent to the statements of `SG1` until it becomes aware that it is invalidated (thus made doomed to abort) by a committed privatizing transaction[2] . The concurrent accesses in `SG1` and in the invalidated transaction can result in infinite loops, null reference exceptions, divide-by-zero errors since the invalidated transaction can perform reads and deliver inconsistent values. Figure 2.4 depicts an inconsistent read example (line 33) causing the program to crash due to a null reference. The figure also illustrates inconsistent reads that can be observed by the non-transactional accesses for direct update STMs.

Details on the example used in Figures 2.3 and 2.4: In our example, we see two threads, Thread 1 and Thread 2, executing according to specific schedules and acting on a shared list `L`. The elements of the list have two fields: `val` and `flag`. Thread 1 privatizes the shared list `L` using transaction `T1`. The rest of Thread 1's actions is performed under the assumption that the shared list is only accessed by Thread 1. In this part of its code Thread 1 traverses the list, updates the `val` fields of the elements, counts the number of elements where the `flag` field is set and stops the traversal (deallocating the rest of the list with the `free_list` function) if the updated value of the visited element is higher than a threshold `y`. Thread 2, inside its transaction `T2`, traverses the list up to an element of the list which represents a `val` field above the threshold `x` and updates all the fields of the elements it traverses. Due to the initial values of the list `L` and the thresholds `x` and `y` both

[1] Direct update STMs make their updates on shared data items visible to other threads right away (without waiting the commit), whereas deferred update STMs log the updates during transaction execution and make their updates effective during transaction commit. See details on direct and deferred update STMs in **Section A.3.2**.

[2] This class of privatization violation is also known as *doomed transaction problem* since it is always caused by the transaction enclosing `SG2` which is invalidated by the privatizing transaction. However, we prefer to distinguish this class of violation as delayed conflict detection since doomed transactions also cause delayed cleanup problems for direct-update systems as seen in Figure 2.3.

Initially the shared list L has two elements. The list, variables x,y and thread-local variable counter have the following content: List elements: {val=8, flag=false, next=...}, {val=13, flag=false, next=NULL}, x=10; y=15; counter=0;

Direct update STM

```
1  // Thread1                        // Thread2
2                                    // Transaction T2
3                                    atomic{
4                                       if(L.head != NULL){
5  // Transaction T1 (privatizer)
6  atomic{
7    privL.head = L.head;
8                                           n = L.head;
9                                           while( n.val < x){
10                                             n.flag = true;
11                                             n.val = n.val+2;
12                                             if (n.next == NULL)
13                                                 break;
14                                             else
15                                                 n = n.next;
16                                          } // end of while
17                                       } // end of if
18   L.head = NULL;
19   }
20                                    // abort preempted before
21                                    // rollback
22 // private actions
23 n = privL.head;
24 while(n != NULL){
25   n.val = 2*n.val;
26   if ( n.flag == true ) {
27     counter++;
28   }
29   if( n.val > y ) {
30     free_list(n.next);
31     n.next = NULL;
32   }
33   n = n.next;
34 }
35                                    } // abort terminated
```

Deferred update STM

```
1  // Thread1                        // Thread2
2                                    // Transaction T2
3                                    atomic{
4                                       if(L.head != NULL){
5  // Transaction T1 (privatizer)
6  atomic{
7    privL.head = L.head;
8                                           n = L.head;
9                                           while( n.val < x){
10                                             n.flag = true;
11                                             n.val = n.val+2;
12                                             if (n.next == NULL)
13                                                 break;
14                                             else
15                                                 n = n.next;
16                                          } // end of while
17                                       } // end of if
18                                    // commit preempted before
19                                    // write back
20   L.head = NULL;
21   }
22 // private actions
23 n = privL.head;
24 while(n != NULL){
25   n.val = 2*n.val;
26   if ( n.flag == true ) {
27     counter++;
28   }
29   if( n.val > y ) {
30     free_list(n.next);
31     n.next = NULL;
32   }
33   n = n.next;
34 }
35                                    } // commit terminated
```

Expected outcome: T2 is serialized after T1:
First list element: {val=16, flag=false, next=NULL}
Second list element: *deleted*
x=10; y=15; counter=0;

Observed outcome:
First list element: {val=8, flag=false, next=NULL}
Second list element: *deleted*
x=10; y=15; counter=1;

Problematic accesses:

(line 25) **inconsistent read:** tentative value of n.val set by Thread 2 (i.e., 10) is read.

(line 26) **inconsistent read:** tentative value of n.flag set by Thread 2 (i.e., true) is read. Reason for inconsistent counter value.

(line 35) **lost update:** abort overrides n.val written on line 25 (i.e., n.val is restored as 8).

Expected outcome: *T1 is serialized after T2:*
First list element: {val=20, flag=true, next=NULL}
Second list element: *deleted*
x=10; y=15; counter=1;

Observed outcome:
First list element: {val=10, flag=true, next=NULL}
Second list element: *deleted*
x=10; y=15; counter=0;

Problematic accesses:

(line 25) **inconsistent read:** uncommitted value of n.val (i.e., 8) is read.

(line 26) **inconsistent read:** uncommitted value of n.flag (i.e., false) is read. Reason for inconsistent counter value.

(line 35) **lost update:** commit overrides n.val written on line 25 (i.e., n.val becomes 10).

Figure 2.3: Illustration of the data races caused by delayed cleanup.

Initially the shared list L has two elements. The list, variables x,y and thread-local variable counter have the following content: List elements: {val=8, flag=false, next=...}, {val=13, flag=false, next=NULL}, x=10; y=15; counter=0;

Schedule	Outcomes

```
1  // Thread1                          // Thread2
2                                      // Transaction T2
3                                      atomic{
4  // Privatizer transaction T1         if(L.head != NULL){
5  atomic{
6    privL.head = L.head;
7                                        n = L.head;
8    L.head = NULL;
9  }
10                                       while( n.val < x){
11                                         n.flag = true;
12                                         n.val = n.val+2;
13                                         if(n.next ==NULL)
14                                            break;
15                                         else
16 // private actions
17 n = privL.head;
18 while(n != NULL){
19   n.val = 2*n.val;
20   if ( n.flag == true ) {
21     counter ++;
22   }
23   if( n.val > y  ) {
24     free_list(n.next);
25     n.next = NULL;
26   }
27   n = n.next;
28 }
29                                            n = n.next;
30                                       }// end of while
31                                       // Beginning of
32                                       // second iteration
33                                       while( n.val < x){
34                                         ...
```

Expected outcome: T2 is serialized after T1:
Since L.head is NULL, T2 does nothing.
First list element: {val=16, flag=false, next=NULL}
Second list element: *deleted*
x=10; y=15; counter=0;

Observed outcome:
Program crashes at line 33 (access to NULL reference)

Problematic accesses for direct update STMs:
(line 19) **inconsistent read:** tentative value of n.val set by Thread 2 (i.e., 10) is read.
(line 20) **inconsistent read:** tentative value of n.flag set by Thread 2 (i.e., true) is read. Reason for inconsistent counter value.

Problematic access for direct&deferred update STMs:
(line 29) **inconsistent read:** T2 reads non-transactionally modified (on line 25) next pointer (which is NULL).

Figure 2.4: Illustration of the data races caused by delayed conflict detection. Note that the figure illustrates a schedule where Thread 1's code is interleaved with the first iteration of Thread 2's while loop (Thread 2's loop is unrolled). Beginning of the second iteration causes a null reference on line 33.

threads can only update the first element of L. More specifically, if `T1` is serialized after `T2`, the private actions of Thread 1 observe L such that only its first element is modified with the `val` field set to 10 and the `flag` field set to `true`. If, however, `T1` is serialized before `T2`, `T2` should see L empty and should commit without doing anything.

2.4.4 Correctness guarantees

The basic criteria that must be guaranteed for TM correctness are defined by the ACI properties of transactions (see **Section 2.3**). However, with the integration of TM to programming languages, the basic transaction criteria are not always enough and stronger guarantees are of interest [79]. There is no agreement yet on the guarantee a TM should provide to the programmer and, hence, there are many guarantees defined for TM in the literature which we elaborate in this section.

2.4.4.1 Guarantees ensuring weak isolation

As explained in **Section 2.4.1** weak isolation allows transactional data races. This is equivalent to saying that the ACI properties of transactions are valid only among transactions and not between transactional and non-transactional code. There are a number of different guarantees that ensure this requirement. In general, the basic rationale of these guarantees is equivalence of the current transactional execution to a **sequential history**, i.e., an execution where transactions execute one after the other without interruption. A sequential history trivially ensures the ACI properties and so does a history equivalent to a sequential history.

Below, we present a series of weak isolation guarantees that are based on equivalence to a sequential history. For the sake of simplicity, we consider in the remainder only read/write objects and we assume that all executions respect the sequential specification of these objects meaning that a read to an object must return the last written value or the default one in case no previous write exists.

Ensuring correctness of committed transactions: We start with guarantees that consider correctness *only* for committed transactions. Although these guarantees are not suitable for TM systems, we start by introducing them since they form the foundations of correctness guarantees specific to TMs.

- **Serializability** [146] is a correctness guarantee that has been extensively used to characterize transactional database systems. Serializability accepts an execution as being correct if it is equivalent to a sequential history.
- **Strict serializability** can be considered as linearizability [100] defined for transactions. As such, it is the same as serializability with an additional constraint: the occurrence order of non-concurrent transactions must be preserved in the equivalent sequential history (we say that two transactions are non-concurrent if there is no common point in time where the two transactions have started and none of them have terminated yet). In other words, strict serializability takes the occurrence times of start and commit operations of a transaction into account and provides the illusion that a transaction takes effect instantaneously at some point in time between its start and commit operations (and does not allow any transaction ordering that would violate this illusion). As strict serializability restricts the set of possible equivalent sequential histories, it is strictly stronger than serializability.

Ensuring correctness of all transactions: Ensuring correctness of only committed transactions is a valid approach in database systems since the user of the system is not effected

by the execution of the transaction, but rather sees the committed result. So a database system can allow aborted transactions to execute based on inconsistent states. However, in TM-based programs a transaction is a part of the program logic, so any fault that could occur in the execution of a transaction (even if the transaction would abort) can lead to anomalies such as infinite loops, access violation exceptions, divide-by-zero errors [79]. For a multi-threaded program such anomalies should be avoided even for aborted transactions to ensure the correctness of the program. Transactions that have observed inconsistent values but that have not yet aborted (and hence can cause anomalies) have been named **doomed transaction** [171] (or **zombie transaction** [42]). In the following, we analyze correctness guarantees that forbid the execution of doomed transactions (while still ensuring correctness of committed transactions):

- **Opacity** [79] requires that strict serializability is ensured for *all transactions (both committed or aborted)* and that all transactions including aborted ones access only consistent system states at any time.
- **Virtual world consistency (VWC)** [104] is the same as opacity except that strict serializability is ensured for *all committed transactions* rather than all transactions (both committed and aborted). It should be noted that VWC still ensures the execution of aborted transactions on consistent state, as in opacity. This is made possible by enforcing aborted transactions to execute based on *some* possible set and/or order of committed transactions. The set and/or order of committed transactions observed by aborted transactions may disagree with the one finally observed by all threads. The rationale behind this guarantee is that since the aborted transaction's execution is consistent with a possible set and/or order of committed transactions (does not matter if it is the finally observed one), this execution should not result in any anomalies; the transaction should be executing correctly as if the set and/or order accepted by the aborted transaction were correct. With its restriction on aborted transactions, VWC is stronger than strict serializability but weaker than opacity.
- **Weakest Reasonable Condition (WRC)** [44] shares the same rationale as VWC for the consistency of aborted transactions. However, it also allows any transaction to *pretend* as a commit-pending transaction that has committed, even though it may ultimately abort (a transaction is said to be **commit-pending** if it has invoked commit, but has not yet committed or aborted). This allows WRC to include some uncommitted transactions to be included in the order of committed transactions. Another difference between WRC and VWC is that VWC allows an aborted transaction T to ignore committed transactions that precede T in the order of real-time occurrence while WRC takes this real-time occurrence into account. With such properties, WRC

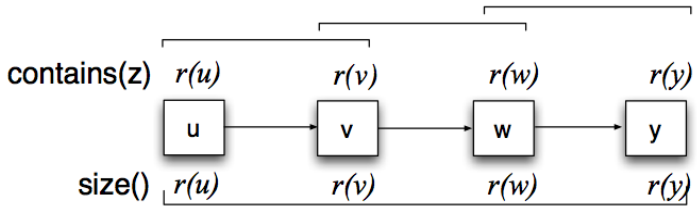


Figure 2.5: Two operations with same accesses and differing semantics.

is stronger than neither opacity nor VWC [44]. But it is obviously stronger than strict serializability.

Relaxing correctness requirements when not needed: In a concurrent environment, two operations may look very similar even though they do not share the same semantics. For example, this is the case for a `contains(z)` operation traversing an ordered linked list data structure and failing in finding element `z` and another operation `size()` capturing an atomic snapshot of the number of elements of this data structure. Both operations have the same sequence of read/write accesses, yet they have distinct semantics. **Figure 2.5** depicts the reads `r(*)` and writes `w(*)` of these operations.

The `contains(z)` is consistent even though `y` is concurrently inserted after `r(u)` occurs. Conversely, the `size()` requires for example that `u` and `y`, which are both counted, were both present in the linked list at the same time. Hence, `contains(z)` enables theoretically more concurrency than `size()` as it allows a weaker guarantee than `size()` requires. Below, we describe the guarantees that exploit the high-level application level semantics of operations such as the `contains(z)` and that improve the performance of the TM for executions including these operations:

- **Elastic-opacity** [58] assumes a model with two types of transactions, elastic and regular, and requires that if we cut elastic transactions into smaller sub-transactions, then the resulting execution is an opaque execution composed of regular transactions and sub-transactions. A programmer can choose to associate elastic-opacity to a transaction if the transaction as a whole does not need to appear as atomic, whereas all couples of consecutive operations or the operations delimited by write operations in this transaction need to appear as atomic. As long as these requirements are satisfied by the transaction code, elastic opacity provides fast and safe execution of the transaction among other elastic and regular transactions.

- **View transactions** [15] uses multi-versions [22] which allow accessing older versions of an accessed memory location. Having the possibility to access older versions (rather than only the current version) a read operation can return a value based on a consistent snapshot, which does not need to be the same as the snapshot used at commit-time. It is sufficient for program correctness that only a subset of the actual snapshot encountered at the commit-time is the same as the subset of the snapshot constructed during execution (the snapshot constructed during execution can point to older versions of several objects compared to the commit-time snapshot). The subset that needs to be the same is determined by the program level correctness criteria, e.g., for the `contains(z)` example above, it would be last consecutive two reads. This subset that needs to be valid should be identified by the programmer as a **critical view**. As with VWC and WRC, aborted view transactions execute on some consistent state of the system, since a transaction executes based on a possible snapshot that does not need to be the same as the actual one. This way, inconsistency anomalies that could occur are avoided. Note that this is not the case with elastic opacity.

2.4.4.2 Guarantees ensuring lock-based semantics

A programmer using transactions based on weak isolation should take care of transactional data races himself/herself. Some of these transactional data races, such as the ones explained in **Section 2.4.2** and **Section 2.4.3** for publication and privatization, are even subtler than data races occurring in lock-based programming, since the races are the result of TM implementation decisions rather than incorrect omissions of synchronization by the programmer. This fact has urged researchers to define *lock-based* transactional guarantees where the semantics of a transaction are equivalent to the behavior of the transaction content executed under lock-based synchronization. The objective of these guarantees is to free the programmer from reasoning about TM implementation details and offer the programmer transaction semantics that are clean and simple. Note that such lock-based semantics avoid only some of the non-transactional data races, especially the ones presented for publication and privatization in **Section 2.4.2** and **Section 2.4.3**, and do not ensure strong isolation.

The most widely known lock-based guarantee is **single-lock-atomicity** (SLA) where a program executes *as if* all the transactions acquired a single, program-wide lock [131, 90]. Such guarantee naturally ensures that transactions execute according to a sequential history: when one transaction executes, it is the owner of the global lock so no other transaction can execute during its execution. The semantic simplicity of SLA is quite appealing while

it limits the concurrency of the program more than necessary. Especially, SLA enforces transactions to be serialized (due to the use of global lock) even if they access disjoint sets of data. Menon *et al.* [131] describe several other lock-based guarantees that raise this limitation; e.g., disjoint lock atomicity (DLA) enforces that the transaction content acquires a set of locks that corresponds to the data it accesses (rather than a single global lock) so that two transactions that access disjoint sets of data can execute concurrently. Other lock-based guarantees described by Menon *et al.* (asymmetric lock atomicity (ALA) and encounter-time lock atomicity (ELA)) share the same rationale as DLA but allow more concurrency by delaying the lock acquisitions up to the point where they are first needed.

2.4.4.3 Guarantees ensuring strong isolation

Although lock-based semantics eliminate some transactional data races, it does not guarantee to remove all. This means a programmer should still care about data races. However, the objective of introducing the transaction abstraction to programming languages is to free the programmer from solving problems due to concurrency, letting him/her focus on the implementation of tasks. This can be achieved only if a TM implementation guarantees strong isolation. While an HTM inherently provides strong isolation, STMs rarely ensure stronger guarantees than lock-based guarantees. The reason is usually the performance cost of ensuring strong isolation.

The first efforts to formally define the semantics related to strong isolation were concurrently done by Abadi *et al.* [7] and Moore *et al.* [135]. Later, Dalessandro *et al.* [39] have identified that transactional code should ideally have semantics that correspond to sequential consistency[1] (SC) [112] which is the most natural semantics expected by the programmer using shared memory. They introduced this corresponding semantics under the name **transactional sequential consistency** (TSC) and define it as SC with the following additional condition: *"memory accesses from a given transaction should appear to be contiguous in the total execution order"*. With such a definition, TSC enforces operations of transactions to execute as if they were not interleaved with any operations (transactional or non-transactional) by other threads. It should be noted that TSC is stronger than strong isolation in that TSC imposes ordering restrictions on operations, especially enforcing the execution order of program statements to result into semantics equivalent to the program code, which is not necessarily required by strong isolation [39]. Program statements having

[1] Sequential consistency is defined as the consistency guarantee for shared memory accesses where "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [112].

data dependencies across threads due to concurrent accesses may restrict some re-orderings a compiler would perform because locally there is no data dependency on the program statements. With strong isolation the dependencies across threads are not necessarily checked and thus the re-orderings a compiler performs may result in inconsistent executions.

Historically, the tradeoff between the semantic simplicity of sequential consistency and the cost of its implementation has resulted into a midway solution, proposed by Adve and Hill [13], imposing a programming discipline to the programmer. This discipline proposes to write programs free of data races (called **data-race-free** programs). Eliminating data races allows sequential consistency to be ensured only on the borders of critical sections (through the use of explicit operations for synchronization) and elsewhere the execution is performed under a weaker semantics, resulting into an efficient execution. As this approach has been widely accepted by the parallel programming community, a similar approach has also been taken by various researchers in the TM community for defining corresponding semantics. For example, Dalessandro *et al.* [39] named the analogous programming discipline **transactional data-race-freedom** (TDRF) where a code should be written free of both transactional data races and ordinary data races (races between concurrent non-transactional code) for the resulting code to ensure TSC. Similarly, but more restrictively, Abadi *et al.* [7] defines **violation-freedom** where an execution should not have any transactional data race. TDRF and violation-freedom still require programmers to adhere to a race-free programming discipline. However, thanks to the elimination of transactional data races, they allow efficient TM implementations guaranteeing weak isolation to be used for ensuring the isolation required by programs instead of costly TM implementations guaranteeing strong isolation. Such semantics are interesting since the resulting code is both semantically clean and practically efficient.

2.4.5 Progress guarantees

An important aspect that constitutes transaction semantics is their progress. By progress we refer to **forward progress** which can be informally defined as "*performing useful computation towards termination*" [102]. As with any synchronization mechanism, STMs are expected to have progress properties and synchronization mechanisms are classified into two major groups with respect to forward progress: *blocking synchronization* and *non-blocking synchronization*.

Blocking synchronization is a synchronization mechanism where there is no guarantee that the execution will always perform forward progress, i.e., the execution can block for an unbounded number of steps. Such execution uses mutual exclusion to protect shared

data and, hence, gives exclusive permission to only one thread for access to shared data. Mutual exclusion is ensured by acquiring and releasing ownerships through locking and unlocking respectively. Other threads trying to access a locked data wait until the thread that locked the data unlocks it. Since this waiting can be controlled only by the thread that has acquired the lock, there is no guarantee that the application does not block (i.e., it is possible to find an execution where all other threads wait for a thread that locked a data and, hence, cannot perform forward progress). This blocking behavior is the reason for some of the problems described in **Section 2.2** (namely deadlock, convoying and lacking fault tolerance).

Non-blocking synchronization is, contrary to blocking synchronization, a synchronization mechanism where there is a specified guarantee of forward progress. Since there will be at least some forward progress, the solutions that are non-blocking exclude the use of locks. Instead, they minimize the number of data locations that threads can contend and avoid race conditions on those data locations by using some costly atomic operations (usually hardware instructions) such as *compare-and-swap* or *load-linked/store-conditional*. There are mainly three classes of forward progress guarantees proposed for non-blocking synchronization: *wait-freedom*, *lock-freedom* and *obstruction-freedom* (a synchronization mechanism that ensures one of these guarantees is said to be *wait-free*, *lock-free* and *obstruction-free* respectively).

Wait-freedom is the progress guarantee where all threads that take part in a concurrent execution make forward progress in finite number of steps, regardless of the delay or failure experienced by other threads [93, 95]. Wait-freedom is the strongest progress guarantee that can be offered by a synchronization mechanism. However, wait-free implementations are generally known to be inefficient and slow, which often makes weaker progress guarantees more appealing for implementations [95, 99, 90].

Lock-freedom requires that at least one thread that takes part in a concurrent execution makes forward progress in finite number of steps [64, 95, 99]. One particularity of lock-freedom is that if a thread T is continuing to execute, it is not guaranteed that it is thread T that makes forward progress [90]. Thus, lock-freedom is weaker than wait-freedom and does not avoid starvation (i.e., the inability of a thread to perform forward progress) [95, 64, 99]. Lock-free implementations usually use **helping**, where if one thread finds that it cannot make forward progress with its own work, then it will help a thread (one that prevents itself from progressing) to make forward progress [90]. This helping mechanism opens the door for starvation because when a thread is helping other threads on its way, it may end up with not performing forward progress itself. Being weaker than wait-freedom, lock-freedom allows more efficient implementations than wait-freedom does.

Obstruction-freedom requires that a thread that takes part in a concurrent execution makes forward progress in finite number of steps as long as other threads do not introduce synchronization conflicts [95, 127]. As in wait-freedom and lock-freedom, obstruction-freedom still ensures that a thread makes forward progress even if other threads experience delays or faults. The only guarantee not ensured by obstruction-freedom is the forward progress in the presence of synchronization conflicts. Since synchronization conflicts can be monitored (e.g., by the synchronization mechanism), it is possible to detect conflicts and remove the cause of conflict in the system. If it can be ensured that all encountered conflicts are removed from the concurrent execution in a finite number of steps by some mechanism, obstruction-freedom can ensure forward progress for all the threads that take part in the concurrent execution. Herlihy *et al.* name the mechanism that removes the cause of conflict as **contention management** [97] and show that obstruction-freedom coupled with a contention manager provides a high level of progress guarantee [95]. Obstruction-freedom, requiring an even weaker guarantee than lock-freedom, leads to the most efficient implementations among the class of non-blocking synchronization mechanisms. Since the contention manager is an out-of-band mechanism, it can further enhance obstruction-free implementation performance by dynamically changing the way it removes the conflicts [95].

2.5 Transaction block semantics

We define a **transaction block** as a code block of a multi-threaded program that should execute in a transaction. A transaction block is thus a language construct that has a special semantics. We indicate that code enclosed in a transaction block is subject to these special semantics by saying that such code executes in **transactional context**.

Semantics of a transaction and a transaction block may be considered to be same but, in this work, we differentiate them based on the following reasoning: the semantics of a transaction describe the semantics of a TM implementation as provided to the programmer, whereas the semantics of a transaction block describe the semantics of the language construct that use a transaction as its basis. As such, the semantics of a transaction block are concerned by not only the semantics of the underlying transaction, but also the interaction of the transaction with other language features and constructs. In other words, a transaction block can have complex behavior required by the language (or added to the language as part of its evolution) but not concerning the underlying TM implementation. Moreover, being a language construct, a transaction block is subject to compiler optimizations that effect memory access orderings, and, hence, language memory models also affect the semantics of a transaction block.

2.5.1 Challenges

The major challenges in defining the semantics of a transaction block with respect to its interaction to other language constructs (or features) can be listed as follows:

1. **ACI violation:** Some language constructs do not fit into the atomicity or isolation semantics implied by a transaction block. For example, task synchronization constructs, such as waiting, require that two threads communicate in the middle of the execution of the construct and that the execution of the transaction block continue after the communication is terminated (making it impossible to defer the communication to the commit operation). As a result of this communication, either the thread executing the transaction block needs to reveal its partial effects to some external thread (which breaks atomicity), or some information from an external thread needs to be received during the execution of the transaction block (which breaks isolation). We call such semantic incompatibility of language constructs with a transaction block **ACI violation**.

2. **Irrevocability:** Some language constructs or statements prohibit the rollback semantics of a transaction block. Once the construct/statement is executed, its effect cannot be rolled back. Such constructs/statements are called **irrevocable**. One striking example to such constructs/statements is missile firing. If during the execution of a transaction block we decide to fire a missile and we execute it, there is no turning back. In general, the irrevocability of a construct/statement is due to the fact that the action implied by the construct/statement is not under the control of TM. Note that although the task synchronization constructs mentioned in the previous item are also irrevocable, overcoming their irrevocability does not solve the ACI violation implied by these constructs.

These challenges complicate the use of transaction block in a language since simply forbidding the use of constructs that cause ACI violation or irrevocability restricts the flexibility of a programmer to a great extent. Hence, it is preferable for a language designer to propose solutions allowing the use of as many language constructs as possible within a transaction block. We analyze existing solutions to these challenges in the next section.

2.5.2 Existing approaches to challenges

Since the language constructs and features that can be used in a transaction block are numerous and vary in their interaction with the transaction block, there are several known techniques to address the challenges explained in **Section 2.5.1** [90]:

- **Prohibition:** This technique simply forbids the use of a construct in a transaction block. Although simple, if there are solutions that allow the use of a language construct within a transaction block this technique restricts the flexibility of the transaction block. Hence, its use is appropriate *(i)* when applied temporarily until another solution is provided (since it allows the application of other solutions seamlessly [37]), or *(ii)* only for language constructs that result in ACI violation (and if there are no known solutions to overcome this issue).
- **Transactification:** Some constructs or statements that are enclosed within a transaction block modify only memory state and result in irrevocability just because the memory accesses resulting from its execution are not under the control of TM. If this is the case, a compiler or a binary instrumentation tool can automatically generate a version of the construct/statement that is under TM control (as explained in **Chapter 6** this process of the compiler or the instrumentation tool is called *transactification*) and use this version for execution in the transaction block. This technique allows the use of such constructs/statements in the desired manner without any unwanted side effects. For example, many standard library functions can be used inside a transaction block using this technique.
- **Integration:** For some constructs/statements it is possible to have a transactional version of a construct/statement that work correctly in a transaction block, but automatic transactification is not enough; the transactional version should be provided manually. For example, transactional versions of some system calls can be provided for use in transaction block, but such versions require the kernel programmer to manually write the code.
- **Extension:** While some constructs/statements require integration (previous item) for correctly executing within a transaction block, some of these constructs introduce behaviors that does not exist in the language. A complete integration of such constructs/statements also requires the extension of the language with appropriate new constructs (e.g., blocks or keywords). Constructs that represent a good example of such extention requirement is exceptions, where the propagation of an exception may require either the commit or the abort of the transaction according to the exception type and this may be controlled by the programmer only if new constructs are provided by the language.
- **Irrevocable execution:** This technique can be used for all statements that result in the irrevocability challenge (see **Section 2.5.1**). It disables the ability of a transaction to rollback, hence it forbids the execution of a transaction in a speculative manner. However, despite the rollback restriction, irrevocable execution still ensures

the ACI guarantees of a transaction. Being a constrained transactional execution, it degrades the overall performance of a multi-threaded program compared to normal transactional execution. So, it is generally preferred to use it as a fall-back technique when there is no other solutions to apply. A statement that requires the irrevocable execution of transaction block is one that includes system calls. If a system call has no available transactional version, this technique allows the execution of the transaction without giving up the ACI guarantees.

2.5.3 Solutions specific to language constructs

Having seen the classification of solutions to address the challenges in **Section 2.5.1**, in this section we overview the techniques applied for frequently used language constructs and features.

2.5.3.1 Memory allocation

Memory allocation[1], used all over applications, is subject to problems that weak isolation presents. Problems can arise in situations where *(i)* both transactional and non-transactional code demand allocation-related actions concurrently, or *(ii)* some transactional code is attempting to deallocate a memory location while some non-transactional code is concurrently accessing the same location. Hence, a solution to memory allocation requires support for strong isolation [90]. However, a solution that covers only allocation for the user code is not enough since allocation is inevitable for any code used by the program (regardless of whether it is user, library code or a system call). Additionally, allocation requires system calls to be executed in cases such as expanding the heap or allocating new memory pages for an application. All these issues suggest that problems related to memory allocation should be solved using the integration technique [90]. Fortunately, an attempt to allocate memory locations in transactional code can be performed as a thread-local operation and can be rolled-back easily. However, the deallocation of memory locations requires more care in the integration effort. It is usually not enough to defer deallocation to the transaction commit since this may retain more memory than actually required. Moreover, it is not trivial to decide when a deallocated memory is eligible for re-use, especially due to concurrent accesses. Similar problems exist for lock-free synchronization, hence solutions can be borrowed from this field [62, 90].

[1]Here we use the term allocation to refer to both allocation and deallocation of memory. We use the term *deallocation* explicitly when we would like to express the action of deallocating a memory location.

2.5.3.2 Exceptions

In considering the interaction of exceptions with transaction blocks, only propagation of an exception out of a transaction block (defined as *escaping a transaction block* by Adl-Tabatabai *et al.* [11]) raises semantic issues. Such issues have been brought to the attention of the TM community by Harris [88] and a more thorough discussion of these issues is presented by Crowl *et al.* [37] and Adl-Tabatabai *et al.* [11].

There are mainly two alternative behaviors among which a transaction block should choose upon escaping a transaction; commit or abort (Adl-Tabatabai *et al.* [11] express these behaviors as *commit-on-escape* and *abort-on-escape* respectively). Both behaviors raise different semantic issues and, hence, proposed solutions differ according to the chosen behavior:

- **Commit-on-escape:** With these semantics a transaction block commits the modifications it has performed up to the point where the exception is raised and propagates the exception after commit. These semantics allows an exception handler to resolve the exceptional situation by analyzing the state of the system at the time of exception raise and, hence, is compatible with the exception handling mechanism. Ringenburg and Grossman [156] advocate these semantics in their work since they consider the raising of an exception to be a non-local control transfer (which should not change the program state) and the transaction block merely a synchronization construct (so it needs to ensure its guarantees only on error-free situations). Yang *et al.* [140] adopt these semantics since their implementation of transaction block is based on lock-based semantics and only commit-on-escape semantics correspond to the exception propagation semantics of a critical section protected by locks. They also indicate that the use of abort-on-escape semantics for a transaction block enclosing irrevocable code is practically infeasible. It should also be noted that commit-on-escape semantics of a transaction block do not prevent the programmer to mimic the abort-on-escape behavior by explicitly aborting the transaction block upon catching exceptions (which are to be caught inside the transaction block) [37, 11].

  While commit-on-escape has the assets explained above, it also introduces semantic issues. It allows cases where partial modifications of a transaction block can be visible by other concurrent threads, breaking the atomicity guarantee of the transaction block. This is especially problematic for situations where the program continues to execute after the exception, propagated according to commit-on-escape behavior, has been handled. Such situations imply that a programmer cannot always assume that a transaction block has executed atomically (the block may not have been executed

atomically due to a commit-on-escape behavior) and should take necessary measures to check whether its execution was atomic. In addition, committing a transaction upon an exception, makes an inconsistent system state due to the exceptional situation visible to concurrent threads, threatening the safety of the code (thus also breaking the *exception safety* suggested by the C++ language [178]).

- **Abort-on-escape:** These semantics suggests that a transaction block rolls all its modifications back before propagating the exception out of the block. As opposed to the commit-on-escape semantics, the abort-on-escape semantics satisfy the requirements of atomicity guarantee of a transaction block while becoming incompatible with the exception handling mechanism. The incompatibility stems from the fact that the rollback behavior prevents the transfer of information out of transaction block, whereas an exception object is meant to carry information about the exceptional situation so that the handler can perform necessary actions to correct the situation. This problem can be circumvented by keeping an exception object out of the control of TM during transactification. However, a subtle problem still persists if the exception object has fields referencing to variables allocated during the execution of transaction block: these fields become invalid after the rollback and, hence, cannot be used for handling the exception. Except the exception object referencing allocated variables, transactification techniques are enough to provide the necessary support. Supporting a complete solution require integration techniques to be applied.

  Fetzer and Felber [59] suggest that the use of abort-on-escape semantics helps writing more robust exception handling code since, thanks to the rollback behavior, it automatically removes the inconsistencies created by the exceptional situations from the system. Based on the same reasoning, two other studies suggest to augment a `try` block with transactional semantics (i.e., a `try` block becomes also a transaction block [30] or an additional `try` block which has transactional semantics is added to the language [165]). With the abort-on-escape semantics proposed by all these studies, the other semantics, commit-on-escape, can be explicitly coded by the programmer by catching the exception in the transaction block and exiting it without aborting [91, 37].

  All the above work advocates abort-on-escape but ignores the problem with exception objects enclosing references to variables allocated in the transaction block. Harris *et al.* [91] propose a partial abort-on-escape semantics such that only write effects of a transaction block are rolled back while allocation effects are retained, avoiding invalid references. However, this solution is quite confusing since part of the exception

object could be rolled back while the rest is not. Another proposition for the problem is making deep copies of the exception objects but this is generally accepted to be infeasible since it is difficult to know the levels of indirections that should be considered to fulfill a deep copy [88, 37]. A final proposition is to restrict the types of exceptions (or the types of information) propagating out of a transaction block such that exception objects cannot include references to other variables [37]. This final solution requires the use of the extension technique in order to forbid exceptions not following the restriction.

Since both commit-on-escape and abort-on-escape are useful in different situations, some researchers provided solutions where both semantics can be supported. Adl-Tabatabai *et al.* [11] use the extension technique and propose a different syntax to the programmer to express both semantics. Harris [88] suggests to distinguish between exceptions that cause abort from the ones that cause commit. He proposes a new base exception type for which abort-on-escape semantics will be applied (obviously descendants of this types will behave the same) while other exceptions will be propagated according to commit-on-escape (this approach requires the integration technique to be applied). Both studies minimize the semantic ambiguity for exception propagation by requiring the programmer to explicitly indicate that s/he desires to use abort-on-escape semantics. This way the programmer knows exactly which semantics s/he should expect for each exceptional situation.

2.5.3.3 Synchronization actions

TM has been proposed as an alternative to lock-based programming for data synchronization where the aim is to ensure the atomicity and isolation of a series of accesses to shared data within a transaction block. Since lock-based synchronization has similar purposes, in general TM is supposed to replace lock-based synchronization and the use of locks in transaction blocks is generally not an expected programming pattern. However, use of libraries and legacy code can easily introduce the use of lock-based code inside transaction blocks. As this is more related to the use of external code, we defer such issues to **Section 2.5.3.4**, where interaction of transaction blocks with external code is treated.

Apart from data synchronization, lock-based programming proposes also task synchronization constructs (such as condition variables, barriers etc.) allowing coordination of threads to perform a task together. Hence, to be a competing alternative to lock-based programming, task synchronization should also be supported by TM-based programming, i.e., task synchronization constructs should be usable in transaction blocks.

Task synchronization involves two groups of threads: waiting and notifying threads. Waiting threads stop their execution at a certain point in their execution and wait for notifying threads to signal them when to resume. Due to the dependency of waiting threads on the notifying threads, the execution of waiting threads cannot be isolated from that of notifying threads, at least for the portion of the execution where a wait is performed. Additionally, once a notification is signaled it is generally not possible to reverse this action. These issues represent both ACI violation and irrevocability challenges and, hence, the use of task synchronization in a transaction block conflicts with transactional semantics.

In the literature, we observe two approaches for allowing task synchronization actions within transaction blocks. The first is a *strict* approach where the ACI guarantees of the transaction blocks are fully respected. The second is a *relaxed* approach where the ACI guarantees are broken (only) to allow communication between transactions that need to synchronize. As it can be expected, the first approach cannot support all types of task synchronization, while still covering a good deal of required synchronization actions[1] . Both approaches propose the use of the integration technique so that the use of task synchronization can be performed without much effort by the programmer.

The first study for the **strict approach** is by Harris and Fraser [89] where *Conditional Critial Regions* (CCRs) are integrated into the programming language as a synchronization construct. This construct suspends the execution of a thread until the condition specified by the CCR is satisfied, and a transaction block specified as a CCR can only execute when this condition is satisfied. With this construct, synchronization can only be performed at the beginning of a transaction. A more elaborate synchronization mechanism is later proposed by Harris *et al.* [91] where the condition to suspend a thread can also be specified inside a transaction block. The approach proposes to use the `retry` keyword within transaction block which can be used when a required condition is not satisfied. Invoking this keyword rolls the transaction block back and suspends the thread until any of the addresses read by the transaction block (before the execution of `retry`) are modified by another transaction block (the set of such monitored addresses is generally called a **watch-set**). With this approach, a thread effectively waits until the conditions under which the transaction block executes change without any explicit notification required by notifying threads (as opposed to existing task synchronization approaches where notifying threads need to explicitly notify waiting threads). This watch-set based approach has been very influential and both hardware [156, 129, 198] and software [10, 33] implementation variants have been realized by

[1] An important limitation of strict task synchronization approaches is that they do not correctly execute under nested transaction blocks. The interested reader can refer to the **Section 3.2.1.4** of Transactional Memory book [90] for the issues raised due to nesting when trying to implement a barrier.

other researchers. In some these variants [156, 129, 33] it is possible to define the watch-set explicitly, increasing the performance of the approach[1] .

Both CCR and watch-set based approaches broadcast the notification of shared variable change in a transaction to all other transactions through TM. Dudnik and Swift [49] argue that notifying specific threads instead of all threads is valuable for certain applications and, hence, provide an implementation of condition variables using TM. This approach proposes two implementations for condition variables. The first is a simple implementation that defers the notification after the transaction commit. However, due to its deferring nature, it is not suitable for nested transaction blocks[2] . The second implementation speculatively notifies other threads. However, this implementation requires execution of non-transactional actions during the execution of a transaction block. A second issue is that this implementation requires a conflict resolution policy that is aware of conditional variables and that can prevent livelocks in the case where a waiting thread causes the notifying thread to abort.

A final work on the strict approach is proposed by Welc *et al.* [189]. This work presents a practical approach to use synchronization constructs in transaction blocks. Any lock-based synchronization construct is used as before under the condition that they are also protected by mutual exclusion locks as in non-transactional code. This provides a kind of double protection on the shared data (one through locks, the other through TM). In addition to that, they define the semantics of `wait` and `notify` Java primitives as follows. The effect of `wait` is immediate while the effect of all the code before `wait` is deferred until commit. For `notify`, the notification is deferred until commit. This approach tries to allow the task synchronization while not breaking the ACI guarantees, however, since the effects of statements before a `wait` are not effective while a `wait` is effective, the programmer needs to communicate information between synchronizing threads through non-transactional means. So, devising a working task synchronization with this approach may eventually break ACI guarantees.

The studies in the **relaxed approach** either break atomicity [32, 167] or isolation [122] guarantee of transactions. **Atomicity-breaking** solutions require a transaction performing a wait to commit at the point where the waiting occurs, i.e., they force a transaction commit earlier then the normal commit point. A first study on atomicity-breaking solutions is by Carlstrom *et al.* [32] where they propose to replace a Java `wait` statement with a

[1] Among these studies, [156] is proposed only for uni-processors where threads do not execute concurrently and only yield the processor to other threads to give the illusion of concurrency.

[2] This limitation is also mentioned by the authors, while they also add that even the use of lock-based conditional variables cannot be nested. However, it should be noted that the deferred nature of the implementation rules out any nesting of transaction blocks rather than the nesting of *only* transaction blocks using conditional variables.

transactional commit and transactional start pair[1] . A more elaborate atomicity-breaking solution is proposed by the TIC programming model [167]. This model wraps task synchronization code in functions and imperatively executes them in an `expose` block followed by an `establish` statement. This new combined construct actually splits a transaction into two: *top* and *bottom* transactions. The transaction that has started before an `expose` block is the top transaction, while the part of the transaction code after the `establish` statement is the bottom transaction. When an `expose` block is encountered (for execution) three actions are performed: *(i)* the top transaction commits (breaking the atomicity), *(ii)* the contents of the `expose` block are executed in non-transactional mode, and *(iii)* the `establish` statement executes (again in non-transactional mode) to ensure that the consistency of data of top transaction is still valid. If `establish` statement successfully terminates such that consistency is preserved, the bottom transaction starts executing. In this model, since the task synchronization is performed in non-transactional code, any existing lock-based synchronization primitive can be used for task synchronization. The difficulty of using this model is to provide a correct `establish` statement.

An **isolation-breaking relaxed approach** is proposed by Luchangco and Marathe [122]. The rationale of the approach is to provide partial isolation, i.e., transactions that need to synchronize among each other cannot be isolated, however, such transactions can still be isolated from the rest of the transactions. In their approach, Luchangco and Marathe provide such isolation by encapsulating data that allow synchronization into a special data object called **transaction synchronizer**. Transactions that use the same transaction synchronizers either all commit or all abort. This way all these transactions can be kept isolated from the rest of the transactions. However, it should be noted that transaction synchronizers can directly be accessed by all threads running transactions on these synchronizers, thus programmers should take charge of the data synchronization on transaction synchronizers.

2.5.3.4 External code

Claiming to simplify the practice of concurrent programming by its virtue of composability, a transaction block should also allow software to be reused within its body to satisfy its promises completely. One of the most prevalent and essential software reuse techniques adopted in software development is the use of external code, such as library code (either in the form of binary or source code). Hence, programmers naturally expect being able to use such external code in transaction blocks (such as printing some string to the console from within a transaction block). However, this is a challenging expectation since it is difficult to

[1] It should be noted that this study is designed to work only for a specific transactional model named continuous transaction model where there is no execution out of transactions.

reverse the effects of external code, resulting into incompatibility with the rollback semantics of transaction blocks.

While studies revealed that there is no one-size-fits-all solution to include external code in a transaction block, we observe that external code can be categorized according to its semantic implications with respect to transaction blocks and there are different solutions for each category. Researchers analyzed existing multi-threaded code (e.g. using locks) and library code to discover the usage and semantics of external code and understand the issues that made external code incompatible with transaction blocks. Such a study performed by Baugh and Zilles [19] re-introduces a classification, by Gray and Reuters [72], borrowed from the database field[1]:

1. **Side-effect-free code:** In this category, we see code that is either read-only, or that modifies only CPU or memory state. Examples of such code are functions such as `getpid` or `gettimeoftheday` [184, 19, 136]. In theory, side-effect-free code does not represent any semantic incompatibility with transaction blocks. The problem represented by this category is that such code needs to be transactified so that TM can take necessary measures to roll back modifications.

2. **System code:** Such code include system calls, usually performed by the operating system, and performs actions that are not under the control of TM (e.g., actions manipulating kernel data structures or data out of reach of the TM). A good example of such code is the majority of file handling functions such as `lseek`, `open`, `close` [184, 19]. Hence, solutions other than transactification are required to reverse the effects of such code. Additionally, since TM cannot isolate such data from concurrent accesses, solutions addressing code this category should also provide such isolation. The issues represented by this category are generally cumbersome to deal with since they require programmers to introduce additional code to the application manually.

3. **Irreversible code:** Code in this category (such as launching a missile) is irreversible in nature, i.e., once executed it is not possible to roll its effects back. Part of the system calls, such as process maintenance functions (e.g., `tgkill`) or I/O operations (e.g., `ioctl`), are of this nature. Being irreversible, the use of such code is in direct conflict with roll-back semantics of transactions.

Although it can be further detailed (as Volos *et al.* [184] or Moravan *et al.* [136] did in their work), the above classification corresponds well to the major issues that make

[1] Gray and Reuters name the following categories as *protected*, *unprotected* and *real* code while we rename them respectively as *side-effect-free*, *system* and *irreversible* code.

external code incompatible with transaction blocks. So, we use the same classification to present existing solutions for transaction blocks enclosing external code. Below, we present proposed methods separately for external code available in binary form or as source code.

If the external code is in binary form, the major approach applied to execute such code in a transaction block without breaking its semantics is using the irrevocable execution technique since it is hard to modify the binary code such that it is executed according to transactional behavior[1] . Of course, a long term solution is to apply the integration technique and to have the built-in support for TM execution in the binary library codes. An example of such approach is the integration of `dietlibc` by Miletić *et al.* [132] where the original library interface is kept while adding support code in the library for transactional execution.

If the external code used by the application is provided as source code, there are more possibilities to provide a solution. Below we explain different solutions that apply for different categories of external code:

- **Solutions for side-effect-free code:** The solution that is to be applied for side-effect-free code is using binary instrumentation tools such as LSA-STM [153], Deuce [111] and Multiverse [4], or just-in-time compilation tools such as Judo [144] that can automatically transactify such code. These binary instrumentation tools may require the insertion of some annotations to indicate the parts of the code that are transactional, but no more other changes are required for supporting execution of such code in a transaction block.
- **Solutions for system code:** Depending on the nature of the system code there are two solutions that work in different cases: *deferral* or *compensation*. Common to all the solutions in this category is that there is always a need for manual code restructuring and addition, and, as such, they are all part of the integration technique applied to support external code.

  Deferral consists of deferring the execution of the external code only after the transaction is sure to commit, hence as part of commit. While this solution is simple and useful, it is appropriate only if data produced by the external code is not later required within the transaction block. Since it is a restricted solution, researchers applying this solution use it as a complementary solution for executing system code [88, 156, 33, 129, 136, 184]. Meanwhile, the same restriction breaks the composability of transaction semantics. For example, if a library function has been adapted

[1] One rare approach addressing transactification of binary code is the TARIFA tool proposed by Felber *et al.* [56]. This tool can be used to automatically transactify external binary code, especially if the code is side-effect-free.

for use with transaction blocks using deferral but the result of a call to this function is used by another statement in the transaction block following the call (which has been shown to be quite likely by Baugh and Zilles [19] in their analysis of external code usage), the deferral solution does not provide the intended transaction semantics. Furthermore, Miletić *et al.* [132] explain that deferral becomes impossible if transactions are allowed to be nested: *(i)* Under flat nesting deferral requires that we delay the commit of an external code in an inner transaction until the commit of outermost transaction. Hence, flat nesting semantics disallows an outer transaction to use data produced by a deferred external code in an inner transaction, *(ii)* Under both open and closed nesting the external code is executed as part of the inner transaction commit, but if the outer transaction aborts, the inner transaction repeats, leading to a repetition of the external code, which is generally undesirable. From the above discussion, we deduce that a complete and composable solution for external code would exclude deferral even as a complementary solution.

Compensation consists of letting an external code run right where it is intended to execute in a transaction block and executing a related compensation code (taking the effects the external code back) as part of transaction abort if a rollback is required. This technique is applicable only if the effects of the external code can be totally taken back by software, thus can be applied for all external code that is not irreversible. We observe two distinct groups of techniques that support compensation in the literature: *weakly-isolated* and *non-isolated* techniques. The difference between these two groups of techniques is that weakly-isolated ones isolate the data manipulated by the external code against concurrent transactions (note that the isolation is only between the transactions and not between transactional and non-transactional code), while non-isolated ones leave the handling of isolation to the programmer (which breaks the isolation promises of a transaction block).

As a **weakly-isolated compensation** technique McDonald *et al.* [129] propose to execute external code in a dedicated open-nested transaction using the HTM support they provide. Since the transaction is open-nested, its effects are visible by all threads when the transaction commits (i.e., even before the parent transaction commits). The open-nested transaction also registers some compensation code before committing so that if its parent transaction aborts, its committed effects can be rolled back. The isolation of the external code is naturally provided by the fact that it is executed in a transaction. Touching the same data as the external code, the compensation code is also executed in open-nested transactions to ensure isolation. The TIC program-

ming model [167] by Smaragdakis *et al.* also adopts the open-nesting solution, but requires a small amount of code restructuring: it wraps external code into functions marked with `suspending` annotation. Each such function is additionally marked to have compensation code which is expressed in a parameterized `undo` annotation where the parameter indicates the function enclosing the associated compensation code. Open-nesting solutions are generally attacked because they can only be used in restricted situations, i.e., an open-nested transaction should not write to data written by ancestor transactions in order to maintain data consistency [136]. Although seemingly strict, practically this restriction may not be a problem since external code data structures are independent of the application data structures (in a sense, that is why they are external!). However, another downside of open nesting is that it adds considerable complexity to TM design. Hence, other approaches [136, 184] proposed to use sentinels, lightweight locks to be used for shared data structures used in external code (e.g., kernel data structures). In these approaches isolation is ensured by acquiring the sentinels at the first access to data and release it during commit. A third approach proposed by Miletić *et al.* [132] is to run external code using user space copies of data structures rather than the data structures (e.g., kernel data structures) themselves. In this case, the external code portion is transactified as any other code, and since the user space copies are controlled directly by the TM, TM provides the required isolation. However, in this technique it is the programmer's responsibility *(i)* to make copies of external code data structures before external code execution, *(ii)* to update them after external code execution, and *(iii)* to provide the required isolation during updates.

Non-isolated compensation techniques execute external code out of transaction context, i.e., without the data accesses being controlled by TM. Hence, the effect of external code is immediate and can be reversed by compensation actions which are registered during external code execution. Harris [88] requires the programmer to define an `ExternalAction` object for each external code and a `ContextListener` object for each compensation code associated to external code. This proposal requires a significant amount of code insertion to the source to allow required semantics. A similar approach based on providing callback functions for compensation is taken by AtomCaml [156]. Zilles and Baugh [198] propose an HTM solution where two hardware instructions respectively allow *pausing* and *resuming* transactions on a thread while not stopping the execution of the thread running the transaction. Any code running in between these two instructions run as non-transactional code. This way, external code can be executed in the middle of a transaction and the transac-

tion is resumed later on. While the solution is simple, all the burden of providing the transactional semantics to a transaction block using these instructions is left to the programmer, including the registration of compensation code. One semantical support, however, is that a transaction is not allowed to abort in between the two instructions (pause and resume) so that external code can fully run and register its compensation code permitting the abort of the transaction to be performed correctly.

- **Solutions for lock-based system code:** A special case of system code is the lock acquisition and release code. Although deliberate use of lock-based code inside transaction blocks is unlikely, use of library or legacy code that includes lock-based code is quite likely. Thus, some researchers have studied the problems and solutions resulting from the co-existence of lock-based and transactional code in the same execution [183, 90]. Volos *et al.* [183] analyzed different pathological cases due to the interaction of locks and transaction blocks and observed the following: *(i)* the use of locks reintroduces the blocking problem into transaction blocks, *(ii)* for weakly-isolated TM systems, the lack of isolation between transactional and non-transactional code results in dirty-read and lost update races on the lock variables. Such races break the mutual exclusion guarantee of lock-based code by resulting into cases where a shared variable is owned by multiple threads or a shared variable's lock being released earlier than the programmer's intention, *(iii)* for strong-isolated TM systems, the fact that the conflicting accesses to lock variables from transactional and non-transactional code are treated as an ordinary conflict on any shared variable (rather than a special conflict on a lock variable) can cause deadlock and livelock situations.

  Volos *et al.* [183] propose *transaction safe locks* (TxLocks) to overcome these issues. In their approach, they introduce the following modifications for lock related code: *(i)* they execute operations modifying lock variables out of TM control, and they use compensation to reverse their effects when needed, *(ii)* using deferral, they postpone the lock releases inside transaction blocks to transaction commit, *(iii)* they extend conflict resolution of TM systems such that the TM becomes lock-aware. To achieve lock-awareness, timestamps are associated not only to transaction commits but also to lock acquisition and releases: owners of the locks are made visible to the TM system and transactions notify the locks they are waiting for to the TM system before blocking and yielding the processor. With these extensions blocking, deadlock and livelock situations are resolved during conflict resolution of TM system. Gottschlich *et al.* [68] takes a static approach where the programmer is required to annotate conflicting lock use (whereas the system of Volos *et al.* tries to identify the lock use and their dependencies at runtime). The programmer can either tell whether a lock can

conflict with any transaction or not, or s/he can be more specific and describe all the locks a transaction would use before a transaction starts executing. This allows the resulting code to perform better in general because the TM system does not need to be pessimistic about false conflicts that actually do not exist, and can only focus on true conflicts indicated by the programmer. However, this approach brings a significant burden to the programmer and mistakes done by the programmer can result in inconsistent executions. A final study in incorporating locks and transactions is by Welc *et al.* [197] where they propose to convert all critical regions of lock-based code to transaction blocks that have SLA semantics (see **Section 2.4.4.2** for SLA semantics). This way, both transactions and locks are tied to the lock semantics and they can easily co-exist. However, compared to the approach of Gottschlich *et al.*, this approach is overly pessimistic since all critical regions or transaction blocks are serialized, while the ones that act on different data could have been executed concurrently.

- **Solutions for irreversible code:** A transaction block that encloses irreversible code has different semantics before and after the execution of the first irreversible code in the block. Before the irreversible code execution the transaction behaves as a transaction that does not include irreversible code, while after the irreversible code executes, the transaction should ensure that it does not abort (in general a transaction that has executed an irreversible code is called an **irrevocable transaction**). Ensuring that a transaction does not abort satisfies the semantic requirements of both the irreversible code and the transaction block.

  Using irrevocable execution techniques [84, 27, 143, 173, 168, 190, 140] is the most straightforward approach for code in this category. These techniques generally ensure that an irrevocable transaction always commits. To guarantee that a transaction commits, it should always be chosen to be the committing transaction against transactions that access common data (i.e., other transactions that access common data abort if data consistency is violated). Hence, providing irrevocable execution requires an adaptation of the TM design. Since this section explains the language and compiler level support required for irrevocable execution rather than the TM design techniques, we defer the details on different irrevocability techniques to the related section (**Section A.4.6**).

  There exist, however, some irrevocable execution techniques applied at higher levels than TM code. TxLinux [157] uses the integration technique and adapts the whole operating system code (which accounts for the major part of external code) such

that it is compatible with transactional execution. TxLinux replaces the spinlocks protecting all the critical sections in Linux with `cxspinlocks` that dynamically choose to run the critical section either in a transaction or using traditional spinlocks. A system call invoked within a transaction results in the execution of its critical sections in transactions. However, if a critical section running in a transaction encounters an irreversible code the transaction is aborted and restarted in an irrevocable mode by acquiring a spinlock, thus by reverting to mutual exclusion. The isolation between critical sections (whether they acquire a spinlock or run in a transaction) is ensured by always adding the spinlock to the corresponding transaction's data-set that is monitored for accesses performed by concurrent transactions. Since a critical section that runs in mutual exclusion writes into the spinlock, transactions that are running critical sections protected by the same spinlock detect the concurrent accesses and abort (critical sections running in mutual exclusion have priority above transactional ones). Meanwhile, concurrent accesses between transactions are managed by the TM automatically. The study of Ziarek *et al.* [197] that aims to allow the use of both locks and transactions in the same code follows a similar approach for irreversible code: normally both critical sections protected by locks (lock-based critical sections) and transaction blocks execute transactionally but they fall back to non-transactional execution (fallback mode) when encountering irreversible code. The isolation of lock-based critical sections and transaction blocks is ensured through the use of SLA semantics, i.e., both lock-based critical sections and transaction blocks are always mutually excluded with respect to a single global lock regardless of their execution mode (transactional or fallback).

Although irrevocable execution enables the inclusion of irreversible code in a transaction block, it has two important drawbacks. First, the condition that an irrevocable transaction always commits decreases the concurrency among transactions, since (at least) part of concurrent transactions need to wait for the irrevocable transaction to commit. Second, the use of irrevocable execution brings a semantic limitation to transaction blocks: a programmer cannot perform an explicit abort of a transaction that has executed an irreversible code.

Due to these drawbacks some other solutions has been proposed in the literature. The TIC programming model [167] proposes such an alternative solution through the `expose-establish` construct explained in **Section 2.5.3.3**. As it does for task synchronization, the model wraps external code in functions and imperatively execute them in an `expose` blocks. With the TIC model, if the irrevocable code that is in the

`expose` block executes (the execution of the `expose` block is non-transactional), the top transaction is committed, as well as the effects of the irreversible external code is never reversed. Interestingly, the TIC model overcomes both limitations of irrevocable execution, especially because it allows transaction to abort after the execution of irreversible code. Such an abort results in the re-execution of the `establish` statement and the bottom transaction (leaving the top transaction and the irreversible code untouched). Although this raises the limitations of irrevocable execution, it requires the programmer to provide a piece of code which ensures that data is consistent with the commit of the top transaction. Since the approach actually splits a transaction into two, it is not always evident to provide an `establish` statement allowing the semantics of a single transaction. However, the model ensures that a programmer is aware of this fact because s/he is forced to enclose irreversible code in an `expose` block (and to provide its accompanying `establish` statement).

Since applying solutions only to specific types of external code does not provide full support for the use of external code in transactions, some studies apply the integration approach (using a mixture of the above techniques) and provide solutions where external code can be used in transaction blocks much like any external code can be used in non-transactional code. xCalls [184] is a replacement to standard Linux system library providing transactional versions of system calls. Executing external code in transaction blocks requires only replacing standard system library calls with calls to their corresponding functions in xCalls library. An important asset of xCalls is its capacity of returning errors from library functions, which does not exist in other solutions. A similar but more transparent work is performed by Miletić *et al.* [132] where the `dietlibc` library is adapted such that its functions can be called both from transactional and non-transactional code in the same manner (even without changing the function calls). While these studies apply the techniques for including external code into a transaction in the user space, Rossbach *et al.* take another approach and adapt the operating system itself to be compatible to transactional execution. Their first attempt in this direction is TxLinux [157] where the original Linux kernel is modified so that all its critical sections can be used both in transactional or non-transactional-code. However, TxLinux does not exclude irrevocable execution of transactions since the kernel code still includes irreversible code and, when needed, critical sections abort transactional execution and execute using mutual exclusion to ensure irrevocable execution. Moreover, by allowing locking and transactions to co-exist in the system, TxLinux re-introduces the deadlock problem. Their second attempt, TxOS [149], overcomes also these issues and allows the operating system services to execute fully transactionally if enclosed in a so-called *operating system transaction*. Operating system transactions provide transactional seman-

tics only for system state and not for application state. For an application to use external code in a transaction block, an operating system transaction should be started with the first system call of the transaction (by means of compiler or run-time support). To provide transactional semantics, operating system transactions commit or abort together with the application level transaction represented by the transaction block.

One important point to notice is that some solutions allowing to use external code in transaction blocks are not composable. One example is the deferral technique, which can not be used if the result of the deferred external code is required by some other statement in the transaction. Transactional nesting is also a threat for composability of some techniques. Deferral is again impossible to use if transactional nesting is allowed (revealed by Miletić *et al.* [132]). A more serious problem is that irrevocable execution is practically difficult to use in a nesting context since a transaction that starts executing in irrevocable mode should not be aborted by the user. A programmer can easily insert an explicit transaction abort in a transaction block without being aware of the fact the block calls a library function implemented using some external code and, hence, needs to execute in irrevocable mode. Such composability problems show that the interaction of techniques to include external code with the surrounding code should still be studied to obtain a fully working solution.

2.5.4 Transaction block semantics in use

As challenges and existing solutions for transaction block semantics suggest (see **Sections** 2.5.1-2.5.3), a single type of transactional behavior for a transaction block is not enough to satisfy all possible requirements of a transaction block as a language construct. For example, if a transaction block encloses irrevocable code, its semantics should be different from the semantics of a transaction block without any irrevocable code. Hence, the code a transaction block encloses determines the semantics the transaction block should exhibit. In this section, we describe commonly used transactional semantics for TM-based programming.

2.5.4.1 Basic transactional behavior

We say that a transaction block exhibits **basic transactional behavior** (or **basic transactional semantics**) if it satisfies opacity. In other words, a transaction block should guarantee atomicity and isolation for the statements it encloses. This behavior is generally the default for state-of-the-art TM implementations.

Although we limit our definition, the ideal basic transactional semantics are ones where a transaction block guarantees strong isolation and non-blocking progress alongside opacity.

However, practically most transaction block implementations to date can only guarantee weak isolation and obstruction-free progress together with opacity (e.g., Intel STM compiler [3], Sun C++ compiler [5] and Dresden TM compiler (DTMC) [1]).

As long as the statements enclosed in a transaction block require otherwise (e.g., there is no irrevocable code inside the transaction block), the transaction block generally behaves according to the basic transactional behavior, as this is the most intuitive semantics for a transaction block.

2.5.4.2 Irrevocable transactional behavior

The basic transactional behavior is not always enough for programming languages because the effects of irrevocable statements (such as I/O accesses, system calls or synchronization actions) cannot be rolled back. The use of irrevocable statements in transactional code requires instead the **irrevocable transactional behavior**. These semantics enforce opacity with the additional requirement that the transaction block can be executed only once. In other words, under irrevocable transactional behavior, a transaction block is executed such that whenever it conflicts with another transaction block, it is the other block that is aborted. In order to avoid the possibility that two transaction blocks running under irrevocable transactional behavior conflict, only one such transaction block is allowed to run at a time, i.e., such transaction blocks execute in a mutually excluded manner. As for basic transactional behavior, practical irrevocable transactional behavior do not provide strong isolation and non-blocking progress although it would have been desirable. The isolation provided by irrevocable transactional behavior is based on weak isolation but it is stronger since transaction blocks running under irrevocable transaction behavior need to run in mutual exclusion. Again due to the existence of mutual exclusion, irrevocable transactional behavior introduces blocking to the system, reducing the progress guarantee with respect to basic transactional behavior.

2.5.4.3 Mutual exclusion behavior

All the challenges described in **Section 2.5.1** made the definition of a clean transaction block semantics difficult. This fact has led some researchers to go back to the mutual exclusion behavior for transaction block semantics by giving up the transactional nature of the block. These semantics enforce the atomicity and isolation guarantees a lock-based code would provide while the programmer does not need to manage locks to obtain such semantics (i.e., the programmer only encloses statements within a transaction block). The most widespread mutual exclusion semantics are SLA, where the transaction block behaves

as if it acquired a single lock visible to all other threads. The advantage of using SLA semantics for a transaction block is that programmers can reason about the behavior of the transaction block in terms of the lock-based semantics they already know. This also implies that irrevocable statements can be used as any other statements in transaction blocks with SLA semantics. However, going back to the lock-based semantics reintroduces the correctness and scalability problems that are resolved by transactional execution. Moreover, using SLA semantics still requires the programmers to ensure that his/her program is race-free (we can think of race-free code as the lock-based equivalent of weakly-isolated code).

2.5.5 Memory models and transaction block semantics

A memory model describes whether an execution of program is legal in terms of data consistency, i.e., it sets the rules that define whether values observed by each thread of a program are valid at all times [67]. Since validity of observed values is closely related to the ordering of read and write accesses performed by threads and the effect of introducing transaction blocks in a language mainly has mainly an effect on such orderings, in this section, we focus our attention on the ordering constraints a memory model imposes on a program rather than the whole memory model.

Writing a program that can be explained by the memory model results into data-race-free programs (i.e., a program with no data races). In a data-race-free program each read from a memory location sees the value written by the last write ordered by the *happens-before* relation [12]. Hence, the ordering constraints of a memory model is based on the happened-before relation.

2.5.5.1 Overview of existing memory models

Fortunately, all the ordering constraints of a memory model can be described in terms of the happens-before relation. The memory model of a program can be summarized using the happens-before relation as follows [12][1] :

- If a statement A appears before another statement B in the code of the same thread, we say that A happens before B (this is also known as program order).
- If A and B are synchronization actions appearing in different threads, these synchronization actions are ordered according to their specific ordering rules (e.g., a lock acquisition happens before a lock release). If due to the nature of synchronization actions A happens before B then by transitive closure the following is true:

[1] This memory model summary is generally valid for most common programming languages such as C++ and Java.

- for any statement C that appears before A in the code of same thread, we say that C happens before B (even if C and B are actions belonging to different threads).
- for any statement D that appears after B in the code, we say that A happens before D (even if A and D are actions belonging to different threads).

2.5.5.2 Memory model extensions for transactions

Transactional behavior has important implications on the ordering constraints imposed by the memory model. Introducing transactions into a programming language actually introduces two new synchronization actions. We call them `starttx` and `endtx`. Each transaction should start with a `starttx` and terminate with an `endtx`, i.e., a transaction can be described only by its corresponding `starttx` and `endtx` pair. `starttx` and `endtx` extends the memory model of a language with the following additional ordering constraints:

- All statements between the `starttx` and `endtx` of the same transaction happen before the `endtx`, and the `starttx` happens before all the statements between the `starttx` and `endtx`.
- A `starttx` on thread T_1 is either the first `starttx` or an `endtx` on another thread T_2 happens before the `starttx` of T_1. In other words, if the `starttx` of T_2 happens before the `starttx` of T_1, the `endtx` of T_2 also happens before `starttx` of T_1.

With the above constraints the happens-before relation implicitly defines an order where a statement S of a transaction T_1 executed by one thread cannot be ordered in between `starttx` and `endtx` of another transaction T_2 on another thread, thus S cannot appear to interleave statements of T_2 (note that the constraint does not exclude nesting since the rules apply on `starttx` and `endtx` actions of concurrent threads).

All the above ordering constraints describe only the fundamental rules a memory model should incorporate for transactional execution. However, there are additional issues (such as the ordering between transactional and non-transactional addresses or the ordering violations that should be avoided due to compiler optimizations) that needs to be treated for a complete memory model. Those are out of the scope of this thesis. Further information on such issues can be found in the book Transactional Memory (2nd edition) [90] or in several papers by Dan Grossman [74, 135].

2.6 Transaction Performance

Many researchers agree that using a TM for data synchronization instead of locks brings a tremendous advantage in the practice of programming correct concurrent applications.

However, this comes with the price of overhead induced by the TM. For the advantages of TM in programming to be valuable, the code using TM should also be fast and efficient. This is especially an important issue for STMs, since an STM executes totally in software and its contribution to application execution time is significant. Mechanisms that compensate for or even hide this overhead is the ultimate goal in the field of STM research. There are two metrics used in TM field to evaluate TM overhead:

- **Throughput:** Throughput is a metric of performance traditionally used in TM to measure the number of transactions a TM commits per time unit. Throughput effectively reflects the ability of a TM to perform useful work (in the case of TM useful work is obtained when a commit occurs). While throughput partially depends on the application workload, the overhead a TM introduces in executing a transaction plays an important role in the achieved throughput and, hence, it is an important metric to assess TM performance.
- **Commit-abort ratio:** The *commit-abort ratio*, which we denote by τ, is the ratio of the number of committing transactions over the total number of complete transactions (committed or aborted) [70]. This metric captures the notion of success of a TM by giving the percentage of transactions that the TM committed versus the total number of transactions the TM attempted to commit. That is, the commit-abort ratio is an important measure of *achievable* concurrency for TM performance, especially from a theoretical point-of-view.

  Although throughput is the most widely used performance metric, it is not sufficient to identify the cause of TM efficiency because it gives information only about commit performance. Nevertheless, evaluating how likely a TM aborts transactions is a crucial issue since aborting can be very costly. First, this cost depends on the efforts wasted in executing the transaction before aborting it: a long transaction is generally costly to retry. Second, abort side-effects might be dramatic for performance: take, as an example, an aborting transaction that has previously forced several other transactions to also abort; this transaction may create further conflicts upon retry.

  Hence, it is not possible to understand TM performance only by observing commit performance: one TM may be efficient either because it aborts very few transactions or because it retries transactions very rapidly. The commit-abort ratio is complementary to the throughput since it determines whether a TM is simply fast or whether it is capable of committing most of its transactions. As with throughput, part of the aborts experienced in an execution is the result of the workload the application represents but the rest is due to the TM implementation (which can unnecessarily abort

some transactions just for the sake of implementation simplicity) and commit-abort ratio helps to determine the amount of unnecessary aborts a TM implementation performs. The usefulness of the commit-abort ratio is demonstrated by Gramoli *et al.* [70] where, apart from complementing throughput, this metric allows classifying different TM designs in terms of how much a design is close to an ideal one where all aborts are due to consistency violations (and not due to TM design decisions).

The above metrics are useful to assess performance at a given parallelism, i.e., for a given number of executing threads. However, with concurrent programming another dimension in performance is the behavior of program execution under changing parallelism. Two major indicators of TM performance with respect to varying number of threads are *speed-up* and *scalability*:

- **Speed-up** is defined as the gain in performance achieved by parallelizing a sequential task. In the case of TM, this converts to the gain in throughput of committed transactions with respect to a sequential version of a program. A subtle point that needs clarification is that the sequential version of a program should not be considered as the TM-based version of the program running with a single thread, but rather a version of the program that has no data synchronization and performs all the task sequentially on a single thread. Due to the overhead TM incurs, the TM-based version of the program running with a single thread can be significantly slower than a sequential version of the same program (especially when the TM is an STM). Hence, real performance gain obtained by increasing the parallelism can be observed only by comparing throughput with respect to the sequential version. A recent publication by Dragojević *et al.* [46] point this issue out and demonstrate that STMs can largely outperform sequential code despite all the overhead they incur.
- **Scalability**, in general, describes the ability to improve throughput or capacity of a parallel system when additional computing resources (such as CPUs, memory, storage or I/O bandwidth) are introduced [147]. For TMs this generally converts to the ability to improve transaction throughput as the number of executing threads increases. Scalability indicates *how much more work* can be done by a TM by adding more potential work (potential work is introduced by increasing the number of threads) to the system. Ideally, the preference is that a TM always produces more work (have higher transaction throughput) with increasing number of threads. However, practically, scalability is bound by the contention of data sharing introduced by the application. Examples of scalability measurements can be found in nearly all TM related work, while good examples contrasting scalability of different realistic applications are by Minh *et al.* [133], Dragojević *et al.* [46], Kestor *et al.* [109] and Zyulkyarov *et al.* [201].

2.7 TM implementations

Up to this point, we have discussed the semantic and performance requirements of transactions. In this section, we give examples of TM implementations that satisfy the widely-used transaction semantic and performance requirements. The TMs we present here cover only the STM implementations used in our studies throughout the thesis but the broad range of these STMs allows us also to introduce the major aspects of TM design. More details on STM implementations and the mechanisms used in STM designs are further explained in **Appendix A**.

DSTM [97] is one of the first STM implementations. It targets object-oriented programs and controls the accesses to objects within transactions (object-based STM). For each object that is to be used in a transaction (actual object), the programmer should instantiate a transactional object from the actual object and use the transactional object instead of the actual object to access the object within transactions. A transactional object is mainly a data structure that allows accessing the following information about the actual object: *(i)* the owner transaction information (ownership record), *(ii)* the state of the actual object when it was first accessed by the transaction (initial state), and *(iii)* the state of the object where the modifications performed by the transaction on the actual object are applied (tentative state). The data structure provides the access to all this information through separate pointers (one for the ownership, one for the initial state and one for the tentative state). DSTM acquires objects with the first write access of the transaction to the object (eager acquire). As long as the transaction that owns an object is not committed, other transactions observe the initial state of the object (whether a transaction is committed or not is stored in a transaction status field within the data structure representing the transaction). Whenever the transaction that owns the object is committed, the tentative state of the object becomes effective. The commit operation of a transaction is a single atomic operation (on the transaction status field) and when this operation occurs the tentative states of all the objects for which the transaction has ownership become effective all together (atomic multi-word update). Such mechanism allows DSTM to update multiple objects atomically in a non-blocking manner (non-blocking STM). A transaction can not always make its updates effective due to two types of actions (conflicts) performed by other transactions: *(i)* other transactions may commit during the execution of the transaction, invalidating the values previously read by the transaction (invalidation), or *(ii)* concurrent transactions may be competing with the current transaction to access common data (contention). DSTM (as any TM) should detect such cases (conflict detection) and perform necessary actions (e.g., aborting a transaction) so that a subset of the transactions involved in a conflict can make

their updates effective (conflict resolution)[1] . The detection of invalidations is performed by validation, the operation ensuring that all the data read by the transaction is still at its initial state. Conflict resolution for invalidation occurs if validation fails. The resolution action taken for invalidation is the abort of the transaction executing the validation (the other conflicting transactions are already committed). The detection of contention occurs during the execution of transactional operations (read, write or commit). DSTM reduces the overhead of contention by hiding the reads performed by a transaction from others (invisible reads). Conflict resolution of a contention is managed by a contention manager module. This module resolves conflict by deciding which of the concurrent transactions should be aborted or delayed temporarily so that transactions are not in conflict after the resolution. The contention manager of DSTM simply aborts the transaction that has already acquired the object, leaving the object non-acquired (transactions later compete to acquire the object). With all the above-mentioned mechanisms, DSTM ensures opacity (see **Section 2.4.4.1**) as correctness guarantee and obstruction freedom (see **Section 2.4.5**) as progress guarantee (it incorporates a simple contention manager in order to ensure progress, while the rest of the algorithm is obstruction-free).

WSTM [89] is another early TM implementation and it controls the accesses to memory words rather than objects (word-based STM). Apart from ownership records, it stores a table mapping memory word addresses to the corresponding ownership records. In order to break the dependency of the number of stored ownership records to the number of possible memory words, this map table is organized as a hash-map. This, of course, may require that the ownership records of two different memory words are obtained from the same hash-map entry, requiring each entry to be a list of ownership records. Unlike DSTM, this STM acquires ownerships of memory words only during the commit operation (lazy acquire). This STM stores the initial state of words in their original locations, and their tentative states in private copies. As DSTM, WSTM is a non-blocking STM since the single atomic update of the transaction status into committed state allows the tentative states of all acquired memory words to be effective all together (hence, other transactions observe the tentative states from that moment on). A subtle issue WSTM faces during commit is that the tentative states produced by a transaction are stored in private copies and they need to be copied on the original locations of memory words after the transaction status has been successfully set to committed state (redo-log STM). During this copy process any read access on the acquired memory words are provided from the private copies rather than the original locations of memory words. For conflict detection WSTM associates a version number to each memory word. Each time a memory word is updated during a successful

[1] For a detailed explanation of conflict, conflict detection and conflict resolution see **Section A.4.1**.

commit the version number related to the word is incremented and when the commit is terminated the ownership record is replaced with the actual version number. Since the version numbers allow only tracking the presence of updates, this STM, as DSTM, supports invisible reads approach for conflict detection. This scheme allows the reuse of the hash-map table to be used both for storing ownership records and version numbers. Unlike DSTM, WSTM allows a transaction to access tentative state of a memory word stored for another transaction T, even if T's transaction status is not set to committed. This allows transactions to read invalid values of a memory word. Even if such a transaction would eventually abort, such invalid values can cause incorrect executions due to issues such as infinite loops or divide-by-zero errors [79]. In order to avoid such situations, WSTM allows an explicit call for validation (to be executed before each read access). Another alternative for avoiding such incorrect executions is to use sandboxing together with WSTM. With its above features, WSTM ensures strict serializability (see **Section 2.4.4.1**) for correctness. As for progress it guarantees lock-freedom (see **Section 2.4.5**) because a transaction detecting a conflict helps the other to commit before itself in order to avoid the two transactions to compete repeatedly for common memory words.

SXM [78] is mostly an adaptation of DSTM for the C# language (whereas DSTM is in Java). However, SXM also adds a flexible contention manager on top of the DSTM algorithm that can switch between existing contention managers schemes at run-time. Such adaptability of contention managers allows this STM to use the best performing contention manager for the workload the application represents (even when the workload changes at run-time). Another particularity of SXM is that it allows other transactions to know about the read accesses a transaction performs (visible reads). With such a scheme conflicts between accesses (read or write) to some shared data item are signaled to transactions the moment they occur, whereas in the original DSTM implementation some of the conflicts can go unnoticed until commit. Since SXM differs mostly from DSTM only in the way conflicts are detected and resolved it also ensures opacity as DSTM. Also, again as in DSTM, SXM bases the progress on a contention manager and it is obstruction-free.

TL2 [42] is a well-known STM especially due to its low execution overhead. The STM achieves such low overhead thanks to two mechanisms: *(i)* using locks to acquire ownerships (lock-based STM), and *(ii)* using a global counter (sometimes called the global clock) to record the commits performed by the STM. The STM avoids the serialization of transactions due to early acquisition of ownerships by applying lazy acquire technique as in WSTM[1] . For locking to allow atomicity of multiple updates, TL2 locks the data

[1]While lazy acquire technique reduces the TM overhead for ownership acquisition, in some cases it requires a transaction to execute completely before detecting that one of its first read operations has been

items according to two-phase-locking [72]. The global counter is used to identify different versions of updated data items. The value of the counter stored as a version corresponds to the commit of the transaction updated the data item (and each commit corresponds to a unique counter value which is called a timestamp). The use of such global counter allows transactions to efficiently detect at commit time whether a read value has been modified, thus reduces significantly the overhead required for validation (time-based STM). Different global counter managements have been proposed to enhance scalability by minimizing the contention on this global counter, e.g., gv5 and gv6 [116]. Other than its distinguishing features explained below, TL2 uses redo logging for storing tentative states and invisible reads as part of its conflict detection. It ensures opacity as correctness guarantee while its progress guarantee is blocking due to the use of locks (see **Section 2.4.5**).

RSTM [127, 38] is a highly configurable transactional memory library that includes multiple STM implementations. There are two major implementations: object-based and word-based. The object-based STM takes DSTM as example but enhances it in many respects. The implementation reorganizes the transactional object data structure for efficient access and allows the configurability of the STM for using eager/lazy acquire and visible/invisible reader mechanisms. It is a redo-log STM incorporating a global counter based fast conflict detection scheme (simpler than that of TL2) to minimize the overhead of validation. RSTM proposes many different contention managers and use, by default, the Polka contention manager [170] resolving contention in favor of higher priority transactions and delaying lower priority transactions according to an exponential back-off. This object-based implementation ensures opacity and obstruction-freedom. The word-based implementation of RSTM resembles TL2 and it is also configurable as the object-based implementation. As with TL2 it ensures opacity and blocking progress.

TinySTM [57] is another time-based TM implementation using locks for acquiring ownerships. TinySTM improves the use of global counter scheme of TL2 and can obtain even lower overheads than TL2 in certain workloads. TL2 assumes that for the values read by the transaction to correspond to the same snapshot, the timestamps for all the read data items should be below (or equal to) the timestamp of the first read data item. But, actually, the global counter may be incremented by committing transactions accessing disjoint data items and, hence, in general transactions are valid for a given range of timestamp values (validity range) instead of a single timestamp value. TinySTM determines this validity range and do not unnecessarily abort transactions (while TL2 would) if all read data items have timestamps lying in this validity range. As RSTM, TinySTM can be configurable in many

invalidated by another transaction. Thus, in the end, the effectiveness of the mechanism also depends on the workload.

ways. It can be parameterized to either defer the write to commit-time as in TL2 (redo-log STM), we refer to this version of TinySTM as *write-back* and denote it by WB, or to execute immediately the writes in-memory (undo-log STM), we refer to this version of TinySTM as *write-through* and denote it by WT. TinySTM can also be configured to use both lazy and eager acquire as locking strategies. Due to the use of locks for acquisition these strategies are called *commit-time locking* (denoted by CTL) and *encounter-time locking* (denoted by ETL) respectively. TinySTM can be compiled to use one of several contention managers such as passive (aborts itself on conflict), timeout (delays itself until a timeout, consecutive delays can be according exponential back-off) or priority (aborts a conflicting transaction if it has lower priority). The default contention manager is passive. TinySTM can be considered as a successor of TL2 improving the commit throughput mainly through the uses of validity ranges and, hence, its correctness and progress guarantees are the same as that of TL2 (opacity and blocking).

SwissTM [47] is a C++ implementation based on TinySTM such that it uses a global counter and validity ranges. SwissTM combines the encounter-time and commit-time locking strategies for efficient conflict resolution. It resolves write-write conflicts eagerly (when encountered) and read-write conflicts lazily (at commit time). Eager write-write conflict resolution avoids executing the whole transaction before rolling back, while lazy read-write conflict resolution avoids a common type of aborts (that is not necessary for correctness) where a read address is overwritten by a concurrent transaction before commit-time. Like SXM, SwissTM also comprises an adaptive contention manager that uses the passive contention manager ideal for small transactions and the greedy contention manager (prioritizing older transactions over younger ones) better suited for long transactions. Since SwissTM mainly improves the conflict resolution applied by TinySTM (both by mixing locking strategy and adding adaptive contention management), the guarantees it offers are the same as TinySTM, i.e., it ensures opacity for correctness and is blocking due to the use of locks for ownership acquisition.

TSTM [18] is an object-based, obstruction-free STM that supports serializability (see **Section 2.4.4.1**) instead of opacity or strict serializability, which are the correctness guarantees commonly ensured by STMs. In general, the overhead to verify serializability is significant and can lead an STM to be unscalable (and even it may result in very poor speed-up). However, TSTM proposes an efficient method to detect whether serializability is violated or not. Instead of using a graph to track conflicts, it assigns a ticket (called *serializability order number*) to each concurrent transaction to determine the order in which the transactions would be executed in an equivalent sequential history. If all concurrent transactions can be assigned different ticket numbers serializability is ensured, i.e., an equivalent

sequential history can be found. This approach produces serializable executions, however, it can, in some cases, abort even if an equivalent sequential history exists. Such aborts are artifacts of the design and is a decision given to improve performance of the STM. Other than its conflict detection and resolution approaches, TSTM's design is similar to RSTM (e.g., in the way it makes the updates effective).

$\mathcal{E}$-STM [58] aims at providing a different correctness guarantee called elastic opacity (see **Section 2.4.4.1**) in order to make TM execution more efficient. Since for some applications opacity is too strong and results in overhead that is unnecessary, elastic opacity proposes to relax the atomicity requirement of transactions and provide a more efficient solution. In elastic opacity, atomicity is relaxed such that only write operations and couples of consecutive operations executed by the same transaction are atomic. More precisely, within a transaction, all write operations plus the read that precede them (in case there is one) looks like a regular transaction (a transaction that ensures opacity). Similarly, all couples of consecutive operations look like regular transactions as well. Elastic opacity comes together with a new transactional model where two types of transactions can co-exist in the same execution: regular and elastic transactions. Regular transactions are traditional transaction that ensure opacity, while elastic transactions ensure only elastic-opacity. **$\mathcal{E}$-STM** is built upon ETL version of TinySTM and, hence, it is a blocking STM.

Although not illustrating the complete picture of STM design space, the STMs presented above represent well the commonly preferred semantic and performance properties in use. First, we observe that most STMs ensure opacity. This correctness guarantee has been pointed to be the most suitable for TMs by Guerraoui and Kapałka [79] and the research community seems to adopt it. Second, for progress guarantee there is no dominant choice: both blocking and non-blocking STMs have been studied by many researchers. Although blocking behavior is generally not desired, numerous STMs still prefer such progress guarantee since the use of locks seems to result in efficient STMs. Among the non-blocking STMs, it is easy to observe that nearly all non-blocking STMs guarantee obstruction-freedom. This shows that the design of such a TM is simple and efficient. When we analyze STM designs from the perspective of performance we observe that, among the STMs discussed above, the ones that are known to be the most efficient are blocking. In such STMs, the key design issues that allow the efficiency of the STM are about conflict detection (use of timestamps) and conflict resolution (contention management strategies). Based on these observations, the current STM research suggests that simple and efficient conflict detection and resolution mechanisms significantly improve STM performance. Additionally, the efficiency of blocking STMs confirms that progress guarantees stronger than the blocking progress result in less efficient designs.

2.8 Summary

In this chapter, we introduce different aspects of TM-based programming. First we motivate the transaction abstraction by presenting it as a concurrent programming language tool and by depicting its advantages over using locks. Then, we focus on the programming aspect of TM-based programming. In this part, we describe the semantics of the transaction abstraction provided by a TM (transaction semantics) as well as the semantics of the language construct providing the transaction abstraction (transaction block semantics). For transaction semantics, we describe the proposed correctness and progress guarantees a transaction should ensure. Although there is no agreed guarantee for transaction semantics, later in the chapter, while discussing TM implementations, we observe that in practice the widely accepted correctness guarantee is opacity, i.e., the execution of transactions is such that an equivalent sequential order of transactions can be found and no transaction executes based on inconsistent state at any time. Although opacity is a weak isolation guarantee, which is the not the ideal transaction semantics for correctness (strong isolation would be the ideal semantics), the overhead of ensuring a stronger guarantee is the major reason for the common use of opacity. For progress, the current trends indicate two major dominating guarantees: blocking (lock-based TMs) and obstruction-free. Blocking TMs are common in practice due to their efficiency, while TMs targeting non-blocking progress prefer obstruction-freedom due to simplicity of its design.

Although practically the semantics used for transactions is more or less fixed, different semantics for transactions should still be evaluated to decide which one fits the best for application execution. Moreover, TM design is a complicated task and ensuring the desired semantics is generally a hard task. In order to overcome these issues designers require tools that they could use to evaluate semantics of the transaction abstraction they provide. In **Chapter 4** we address this issue and propose a tool, TMunit, that is tailored for this purpose.

Although transaction semantics is the base of the semantics of a transactional language construct (transaction block), the interaction of such language construct with other language constructs and features is mainly the subject of transactional block semantics. In this chapter, we also present the major problems a transaction block would have in interacting other language constructs and features and analyze proposed solutions for different classes of such incompatible language constructs, namely, memory allocation code, synchronization actions, external code and exceptions. Although solutions exist for each separate construct or feature, very few studies integrate them together within a language. Later in **Chapter 5**, we provide a language extension specification for transaction blocks, taking both transac-

tion and transaction block semantics into account and proposing an integrated solution for different classes of incompatible constructs. We complete this specification by depicting how the specification can be implemented in **Chapter 6** using our second tool TMJava.

Among the different classes of incompatible constructs with which a transaction block should interact, a general solution that applies to all but exceptions is the use of irrevocability, i.e., running a transaction such that it can not be aborted by others. For exceptions, irrevocability is not an option because they are not part of normal program execution and it is not acceptable to run transaction using irrevocability (which introduces a significant overhead) just because a transaction block can potentially raise an exception. Hence, in this thesis, we have analyzed existing work to support exceptions in detail and provided the programmer (through the language extension we propose in **Chapter 5**) all the solutions that are required to support exceptions within transaction blocks. We have also figured out that multi-threaded applications have further exception related requirements compared to single-threaded ones. Exceptions in multi-threaded applications may need to be handled in cooperation by multiple threads and, in general, there is no support for such functionality in popular programming languages. This advanced functionality is treated in **Chapter 7** and is embodied within an augmented transaction block construct (atomic box) presented as part of our language extension.

At the end of the chapter, we focus on TM implementations and their performance. Using example TM implementations, we describe key design mechanisms that make different implementations efficient. However, generally no TM implementation to date can yet perform better than any other for all workloads. Hence, it is important to identify the workloads where different design choices perform poorly. While such a study have barely been done, in **Chapter 4** we evaluate different TMs under a variety of workloads to perform such identification. Additionally, apart from observing TM implementations through experiments, we analyze STM algorithms and identify some design decisions that allow us to classify different STM implementations. We present this classification in the next chapter.

Chapter 3

Qualitative Classification of STM Designs

3.1 Introduction

STM design incorporates many design decisions, resulting in a large STM design space (see **Appendix A**). Finding a winning combination of design decisions is not an easy task and, hence, the quest for such a combination is still an active and hot research topic. Such research requires the comparison of many design alternatives in different circumstances. In this chapter, we aim at providing a classification of STM designs, grouping several design decisions together, in order to narrow down the possible design alternatives to be compared. A quantitative comparison of designs is always possible, but taking into account the diversity of mechanisms proposed for STMs, the implementation effort to compare multitude of combinations may not be practical. Hence, a good solution is first to classify designs qualitatively and then compare representatives of each class. In this chapter, we introduce our effort in providing such a qualitative classification.

In general, it is easier to compare designs using metrics, since metrics allow to reason about the quality of the design in terms of what the metric represents (e.g., speed, efficiency, energy consumption etc.). Most studies in STMs measure the throughput metric to evaluate STM designs (see **Section 2.6**). This metric is valuable as it gives an idea of how efficient an STM is, but, per se, it does not always provide a good understanding about *why* an STM design is efficient. One STM may be efficient either because it aborts very few transactions or because it retries transactions very rapidly. Hence, in order to have a better understanding of the efficiency of STMs we need metrics that are complementary to throughput. In this

chapter, we introduce such a novel metric, *commit-abort ratio* (which is associated to a novel notion called *input acceptance*) and classify major STM designs based on this metric.

The chapter is organized as follows. In **Section 3.2** we start by illustrating that most STMs abort in cases where application correctness would have been ensured without aborting (unnecessary aborts). We then introduce the commit-abort ratio and the associated notion input acceptance in **Sections 3.3** and **3.4** respectively and explain how they are related to unnecessary aborts. In **Section 3.5** we explain how we classify STM designs according to the unnecessary aborts they generate and then present the different design classes in **Sections 3.5.1** through **3.5.5**. We explain how the presented classes are related in **Section 3.6** and conclude the chapter with **Section 3.7**.

3.2 Unnecessary aborts

As explained in **Section 2.4.4.1**, the basic rationale in ensuring the correctness of a TM is to check whether the current execution is equivalent to a sequential history, i.e., an execution where transactions execute one after the other without interruption. Among the TM correctness guarantees the one that allows the largest range of sequential histories is serializability. Since, in general, supporting serializability is enough for correctness of transactional execution, in this chapter, we define what a correct execution is on the basis of serializability, i.e., we say that *consistency is satisfied* if the history generated by TM is allowed by serializability (and otherwise we say that *consistency is violated*).

When we observe TM implementations, we see that TMs support stronger correctness guarantees than serializability, i.e., TMs support correctness guarantees that allow only a subset of cases allowed by serializability. This implies that TM implementations would abort for cases allowed by serializability but not by their own correctness guarantee (a TM aborts a transaction when it encounters a case that its correctness guarantee does not allow). We call aborts induced by such cases unnecessary aborts. More formally, we say that an abort is an **unnecessary abort** if it is performed under a case satisfying consistency (i.e., under a case allowed by serializability).

Figure 3.1 illustrates an example of an unnecessary abort. In the figure, two transactions execute concurrently so that their operations are interleaved and their interleaving is explicitly shown on the figure. The read operation at thread p_2 could return indifferently the new value of x or the overwritten value without violating consistency. However, as depicted in the figure, numerous STMs would abort in such a situation.

Many STMs produce unnecessary aborts for sake of simplicity of design [97, 57, 89, 42, 78]. This simplicity is mostly needed to obtain efficient STMs that do not introduce large

Figure 3.1: An input pattern for which numerous STMs unnecessarily abort. In this example, we assume that the transaction detecting the conflict aborts (see **Section 3.5** for the definition of conflict).

overheads to the application execution. In general, the cost of verifying that serializability is ensured is significant and, hence, STMs avoid supporting serializability completely; instead they support stronger correctness guarantees. This shows that there is a tradeoff between the efficiency of design and the amount of unnecessary aborts generated by the STM. The classification presented in this chapter allows us to group STM designs in terms of unnecessary aborts they produce and, hence, helps in better understanding this tradeoff.

3.3 Commit-abort ratio

Unnecessary aborts are artifacts of STM designs rather than the result of contention induced by a workload. Measuring the amount of aborts of different STMs under the same workload conditions indicates which STM produces more unnecessary aborts. Being able to assess the amount of unnecessary aborts produced by STMs is important since this evaluation helps in understanding the reason of STM efficiency.

Measuring the number of aborts, by itself, is also crucial since aborts can be very costly. This cost depends on the efforts wasted in executing the transaction before aborting it: typically, a long transaction will be generally costly to retry. Additionally, aborts may have significant side-effects for performance: for example, an aborting transaction that has previously forced several other transactions to also abort may create further conflicts upon retry.

In the light of these observations, we argue that a metric incorporating the number of aborts occurring in the execution is valuable in understanding STM efficiency. Thus, we introduce **commit-abort ratio** (see also in **Section 2.6**). The *commit-abort ratio*, denoted by τ, is the ratio of the number of committing transactions over the total number of complete transactions (committed or aborted). This metric captures the notion of success of an STM by giving the percentage of transactions that the STM committed versus the total number of transactions the STM attempted to commit. That is, the commit-abort ratio is an important measure of *achievable concurrency* for STM performance, especially from a theoretical point-of-view.

3.4 Input acceptance

Since our aim is to classify STMs qualitatively, we would like to use the commit-abort ratio to assess whether an STM produces unnecessary aborts (aborts whose absence would not violate consistency). For this purpose, we define a new notion which we call *input acceptance*. In this section, we give a formal definition of input acceptance.

As mentioned in **Section 3.3**, detecting unnecessary aborts is possible only by exposing different STMs under the same workload conditions. To define clearly what we mean by workload conditions, we first introduce a formalization of workload, and then present the definition of input acceptance.

An *input event* is either a start request, an operation call on a shared variable, or a commit request. Here, we only admit read and write operations and all operations are part of a transaction. For a given transaction t, we denote its start and commit requests as s_t and c_t, respectively. A read call performed by transaction t on x is denoted as $r(x)_t$ (or r_t^x for short). Similarly, a write call by t on x is expressed as $w(x)_t$ (or w_t^x for short). Whenever we refer to a memory access operation by transaction t (regardless of whether the access is a read or write), we denote the operation as $\pi(x)_t$ (π_t^x for short). Similarly, to refer to any operation (regardless of whether it is a read, write, start or commit operation) by a transaction t we use the notation π_t. The notation π stands for any operation performed by any transaction. The values read and written are of no interest in the input definition and they are omitted from the notations of input events.

An **input pattern** $\mathcal{P}$ of an STM is a (totally ordered) sequence of input events. The associated order corresponds intuitively to the real-time order in the sense that one event is ordered before another if and only if its execution precedes the other in time, and for the sake of simplicity we assume that no two distinct events occur at the same time. An **input class** $\mathcal{C}$ is a (potentially infinite) set of input patterns.

We can consider an input pattern as a word whose alphabet contains input events and an input class as a language defined over the alphabet of possible events. We use regular expressions to represent the possible input patterns of a class. In our regular expressions, parentheses, '(' and ')', are used to group a set of events. The star notation, '$*$', indicates the Kleene closure and applies to the preceding set of events. The complement operator, '$\neg$', indicates any event except the following set. Finally, the choice notation, '$|$', denotes the occurrence of either the preceding or the following set of events. Operators are ordered by priority as $\neg, *, |$.

We say that an STM *accepts* an input pattern if it commits all of its transactions, i.e., its commit-abort ratio is $\tau = 1$. More generally, we say that an STM *does not accept* an

input class if it accepts no pattern of this class. In other words, the STM does not accept a class if for each of its patterns, the STM aborts at least one transaction, i.e., $\tau < 1$. The classes that an STM accepts determine its **input acceptance**.

3.5 Classification of STM designs based on input acceptance

The criterion we used for classifying different STM designs is the classes of unnecessary aborts they allow. To reason about different classes of unnecessary aborts, we first need to understand the abort mechanisms in STMs.

Aborting is the major mechanism to ensure correctness in TMs. A TM aborts a transaction that would cause the violation of consistency (in the sense explained in **Section 3.2**). In order to detect that a consistency violation occurs, TMs use the notion of conflict. A conflict occurs, by definition, when two concurrent transactions access the same data and at least one of the concurrent accesses is a write. A conflict that fits to this definition does not always violate consistency, i.e., there can be several conflicts in a history allowed by serializability. Hence, it is not always necessary to abort a transaction upon a conflict. However, for simplifying designs most TMs usually abort a transaction whenever a conflict is encountered. In order to prevent a TM from aborting unnecessarily, a commonly applied method is to mask conflicts not violating consistency, i.e., the TM does not detect the masked conflicts. Hence, the unnecessary aborts generated by a TM depend on the masking mechanisms it uses.

There are two major design mechanisms that are used for masking conflicts:

- visibility of read or write accesses, and
- enforcing a commit order on transactions.

Visibility of a read or write access lets the other threads know the access as soon as the access is performed. Introducing visibility for read and writes (named *visible reads* and *visible writes* respectively) implies that the design protects the accessed data item from other threads (by setting a flag or by locking the data item), i.e., a write by a concurrent transaction to the same data item results in a conflict. Not having the visibility for an access should however be interpreted differently for read and write accesses. For reads not having visibility (*invisible reads* for short) means concurrent transactions are not at all aware of reads performed by other transactions. However, for writes not having visibility (*invisible writes* for short) means that the effect of a write is delayed until transaction commit and concurrent transactions are not aware of the writes until they become effective with a commit.

Enforcing a commit order on transactions can be considered as finding an equivalent sequential history that corresponds to the commit order. In other words, by using this mechanism, the STM directly checks whether serializability is satisfied. Ideally, a commit order can be found by constructing a cycle-free conflict graph [23]. Simpler solutions exist as applied in TSTM (see **Section 2.7**), but such solutions generally can not find all the commit orders constructed by the conflict graph approach, thus, may generate unnecessary aborts.

Based on these two design mechanisms (visibility and enforcement of commit order), we identified five designs shared by seven STMs. The STM designs that we consider always provide a consistent view of the memory to the application and guarantee serializability (see **Section 2.4.4.1**). They may or may not be strict serializable (see **Section 2.4.4.1**); this is typically not important from an application programmer's perspective (although it has some impact on the implementation of the STM). Our designs are:

1. **Visible read (VWVR):** This design adopts both visible reads and visible writes and is used for instance by SXM [78];

2. **Visible write (VWIR):** This design adopts visible writes and invisible reads and is used for instance by DSTM [97] and TinySTM [57];

3. **Invisible write (IWIR):** This design, used by WSTM [89] and by TL2 [42], adopts both invisible reads and invisible writes;

4. **Commit-time relaxation (CTR):** This design, used in TSTM [18], allows to order transactions independently from the time a commit request is received;

5. **Real-time relaxation (RTR):** This design orders transactions as CTR design but relaxes the constraint that if a transaction t_1 ends before another transaction t_2 starts, then all the operations of t_1 must precede operations of t_2.

For each of the 5 designs, we compare their input acceptance upper-bound. Upper-bound stands here for the limited amount of input the design accepts: the more input it accepts, the higher the upper-bound. The resulting design classification is confirmed experimentally on realistic workloads, which is discussed later in **Section 4.7**. Below, we explain the designs and their upper-bounds.

3.5.1 VWVR design

This set of STM designs, called *VWVR*, has visible writes and visible reads and shows its acceptance limitation by defining a class of input patterns that this design never accepts.

It turns out that common input patterns are not accepted by this design. For a classical example of write-after-read pattern by two transactions, consider the example proposed in **Figure 3.2**. If a transaction t_2 writes a variable that has already been read by another transaction t_1 that is still active, then a conflict is detected by t_2 while writing. This leads to resolving the conflict. As stated in the following theorem, an input class including this pattern is not accepted by this design.

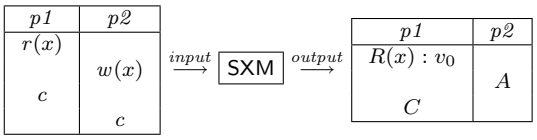


Figure 3.2: An input pattern for which SXM produces a commit-abort ratio of $\tau = 0.5$ (transaction of $p2$ aborts upon writing).

Theorem 1. There is no STM implementing VWVR design that accepts any input pattern of the following class:

$$\mathcal{C}1 = \pi^*(\pi_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* \pi_i^x)\pi^*, \text{ for any } i \neq j.$$

Proof. The proof of this impossibility relies on the existence of two sub-patterns, of which at least one is common to any pattern of class $\mathcal{C}1$ and that is not accepted by any VWVR STM. Consider the input pattern $\mathcal{P}1 = \pi(x)_1 w(x)_2$ and $\mathcal{P}1' = w(x)_1 \pi(x)_2$.

First, since a write operation on variable x verifies that neither a write operation nor a read operation is accessing x and aborts a transaction if this verification fails, $\mathcal{C}1$ does not accept $\mathcal{P}1$. Second, since both read and write operations on variable x verify that x is not currently written and abort a transaction if the verification fails, $\mathcal{C}1$ does not accept $\mathcal{P}1'$. That is, neither $\mathcal{P}1$ nor $\mathcal{P}1'$ are accepted by $\mathcal{C}1$.

Finally, observe that adding any event to $\mathcal{P}1$ or $\mathcal{P}1'$ (with the restriction that none of the events added between the two operations of $\mathcal{P}1$ nor $\mathcal{P}1'$ is a c_1) produces a pattern of $\mathcal{C}1$ that is not accepted by VWVR STMs for the same reason as above. As a result, class $\mathcal{C}1$ is not accepted by VWVR STMs. □

3.5.2 VWIR design

This set of STM designs, called *VWIR*, has visible writes and invisible reads and is similar to DSTM [97] and TinySTM [57] with a contention manager that aborts the transaction detecting a conflict. Common input patterns are not accepted by this design. Consider the input pattern depicted in **Figure 3.3** that may arise for instance when concurrent operations (searches, insertions) are executed on a linked list.

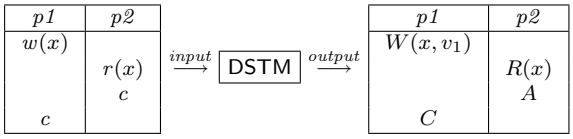


Figure 3.3: A simple input pattern for which DSTM produces a commit-abort ratio of $\tau = 0.5$ (transaction of $p2$ aborts).

This is a classical example of read-after-write pattern by two transactions, with the written value being visible and uncommitted. If a transaction t_2 reads a variable previously modified by another transaction t_1 that is still active, then a conflict is detected by t_2 while reading. In any case, this leads to resolving the conflict: while in this design the transaction t_2 aborts due to this conflict, any alternative contention manager aborts one of the current transactions. As stated in the following theorem, an input class including this pattern is not accepted by this design.

Theorem 2. There is no STM implementing VWIR design that accepts any input pattern of the following class:

$$\mathcal{C}2 = \pi^*(r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \mid w_j^x \neg c_j^* r_i^x)\pi^*, \text{ for any } i \neq j.$$

Proof. The proof is similar to the proof of Theorem 1 but with the following patterns: $\mathcal{P}2 = r(x)_1 w(x)_2 c_2$ and $\mathcal{P}2' = w(x)_1 r(x)_2$.

Since in $\mathcal{P}2$, t_2 writes and commits the value of x after the time at which t_1 reads x and before the time at which t_1 commits, t_1 fails in validating right before commit-time and aborts. As a result, $\mathcal{P}2$ is not accepted by $\mathcal{C}2$. Since in $\mathcal{P}2'$, t_2 reads the value of x after the time at which t_1 writes x and before the time at which t_1 commits, the read operation fails because t_2 knows that t_1 is still the writer of the object. As a result, $\mathcal{P}2'$ is not accepted by $\mathcal{C}2$.

Next, observe that adding events to $\mathcal{P}2$ or $\mathcal{P}2'$ (with the restriction that none of the events added between the operations of $\mathcal{P}2$ nor $\mathcal{P}2'$ is a c_1) results in a pattern of $\mathcal{C}2$ that is not accepted by VWIR STMs for the same reason as above. □

As mentioned earlier, this input class captures realistic workloads composed of common read and update transactions.

3.5.3 IWIR design

The IWIR design accepts patterns of the preceding classes, i.e., for which the previous impossibility results do not hold. Nevertheless, we do not claim that all patterns of $\mathcal{C}1$ or $\mathcal{C}2$ are accepted by this design. This design, inspired by WSTM [89] and TL2 [42],

uses invisible writes and invisible reads with a lazy acquire technique that postpones effects until commit-time, thus it is called *IWIR*. While a main constraint of STMs is that a read must return without being postponed, STMs allow us to postpone a write operation, thus delaying its visibility. The idea differs from the previous designs due to the invisibility of writes: while modifications are recorded at write-time in the write-set, these modifications are made visible not earlier than commit-time.

Even the IWIR design does not accept some very common input patterns, as mentioned in the introduction and as depicted in **Figure 3.1**. This is a classical example of transaction writing a value that is later read. Such a pattern arises, for example, when performing concurrent operations on a linked list. The following theorem gives a set of input patterns that are not accepted by STMs of the IWIR design.

Theorem 3. There is no STM implementing IWIR design that accepts any input pattern of the following class:

$$\mathcal{C}3 = \pi^*(r_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* r_i^x) \neg c_i^* c_j \pi^*, \text{ for any } i \neq j.$$

Proof. In this proof we consider the following two patterns $\mathcal{P}3 = r(x)_i w(x)_j c_j$ and $\mathcal{P}3' = w(x)_j r(x)_i c_j$ of $\mathcal{C}3$. We show that each of these patterns is not accepted.

First, consider the input pattern $\mathcal{P}3$, and assume by contradiction that its two transactions commit. Upon invocation of $r(x)_i$, transaction i records the variable in its read-set for later validation. At the time t_j commits, the variable x is updated with the new value written by t_j. Since t_i has not committed yet when the write becomes visible, upon committing, t_i fails in validating its read-set leading to an abort.

Second, consider the input pattern $\mathcal{P}3'$, and assume by contradiction that the two transactions commit. Since writes are invisible and $r(x)_i$ occurs before c_j, the value written by t_j is not read by t_i. That is, $\mathcal{P}3'$ and $\mathcal{P}3$ becomes indistinguishable from t_i standpoint. As above, upon committing, t_i fails in validating leading to an abort.

Clearly, adding any sequence of operations between the three events of $\mathcal{P}3$ and $\mathcal{P}3'$ (except that none of transactions commit before both the read and write operations are performed and that the transaction performing the read operation does not commit before the transaction that performs the write) would lead also to non-accepted patterns. Since all possible patterns of $\mathcal{C}3$ contain one of these two sub-patterns, input class $\mathcal{C}3$ is not accepted by IWIR STMs. □

Note that this impossibility result also holds for the VWVR and VWIR designs, since $\mathcal{C}3$ is a subset of $\mathcal{C}1$ and $\mathcal{C}2$.

3.5.4 CTR design

The following design has, at its core, a technique that makes as if the commit occurred earlier than the time the commit request was received. In this sense, this design relaxes the commit time and we call it *Commit-Time Relaxation (CTR)*. To this end, the STM uses scalar clocks that determine the serialization order of transactions. This design is inspired by the recently proposed TSTM [18] in its single-version mode.

TSTM is claimed to achieve conflict-serializability, however, it does not accept all possible conflict-serializations. **Figure 3.4** (center and left-hand side) presents an input pattern that TSTM does not accept since transactions choose their clock depending on the last committed version of the object they access: in this example, transactions of $p2$ and $p3$ choose the same clock and force the transaction of $p1$ to abort. This pattern typically happens when a long transaction t runs concurrently with short transactions that update the variables read by t. The following theorem generalizes this result by showing that STMs implementing CTR design does not accept a new input class.

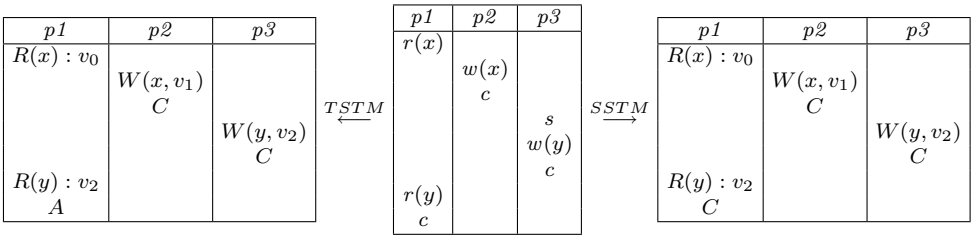


Figure 3.4: An input pattern (in the center) that TSTM does not accept as described on the left-hand side. The commit-abort ratio obtained for TSTM is $\tau = \frac{2}{3}$ (transactions of $p2$ and $p3$ commit but transaction of $p1$ aborts). In contrast, the Serializable Software Transactional Memory (SSTM) presented in **Appendix B** accepts it (the output of SSTM, on the right-hand side, shows a commit-abort ratio of 1).

Theorem 4. There is no STM implementing CTR design that accepts any input pattern of the following class:

$$\mathcal{C}4 = (\neg w^x)^* r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \neg c_i^* s_k \neg (c_i \mid c_k \mid r_k^x)^* w_k^y \neg (c_i \mid c_k \mid r_k^x)^* c_k \neg c_i^* r_i^y \pi^*, \text{ for any disctinct } i, j, \text{ and } k.$$

Proof. The proof relies on the existence of a sub-pattern $\mathcal{P}4$ common to any pattern of $\mathcal{C}4$ that is not accepted by the CTR design. Let $\mathcal{P}4$ be $r(x)_i w(x)_j c_j s_k w(y)_k c_k r(y)_i$. First, observe that when t_j commits, it chooses clock n, where n is the number of threads and it upper-bounds the clock of t_i to $n-1$. Second, when t_k commits it sets its clock to n so that t_i sets its lower-bound to n too, when reading y. Consequently, t_i has a larger lower-bound n than its upper-bound $n-1$, that is, t_i aborts upon reading y.

Next, we show that for any other pattern of $\mathcal{C}4$, t_i aborts for the same reason. By the definition of $\mathcal{C}4$, variable x cannot be written before $\mathcal{P}4$ in any pattern of $\mathcal{C}4$. As a result, the upper-bound of t_i cannot be larger than $n-1$. Since t_k does not read, while committing, t_k cannot choose a lower clock than n. Hence, when t_i performs its second read, it sets its lower-bound to n or to a larger value than n, and t_i aborts similarly as above. □

Observe that we use the notation s_k in this class definition to prevent transactions t_j and t_k from being concurrent. Similar to previous class definitions, the restrictions brought to the events that could be added between the events of $\mathcal{C}4$ are such that *(i)* t_k can never perform a read and *(ii)* any transaction does not commit before they are explicitly expressed by the class definition (as also seen in **Figure 3.4**.)

3.5.5 RTR design

This design, called *Real-Time Relaxation (RTR)*, presents a technique that relaxes the real-time order requirement. The real-time order requires that given two transactions t_1 and t_2, if t_1 ends before t_2 starts, then t_1 must be ordered before t_2. The design presented here outputs only serializable histories but does not preserve real-time order. More precisely, it outputs non real-time ordered histories as we can see in **Figure 3.4** (center and right-hand side). These outputs result from inputs that cannot be accepted by any STM ensuring real-time order (including all STMs that are opaque or strict serializable). **Figure 3.4** (center and right-hand side) presents an input pattern that SSTM accepts while other STMs that ensure real-time order do not accept. This is illustrated by the non-acceptance of the same pattern by TSTM, in **Figure 3.4** (center and left-hand side).

We propose a Serializable Software Transactional Memory, namely *SSTM*, that implements the RTR design (the algorithm for SSTM is present in **Appendix B.1**). SSTM is an STM with a high commit-abort ratio: SSTM accepts all patterns presented so far (including the ones of **Figures** 3.1, 3.2, 3.3, and 3.4). Moreover, SSTM is conflict-serializable[1] but neither opaque nor strict serializable, and it avoids cascading abort, since whenever a transaction t_1 reads a value from another transaction t_2, t_2 has already committed [23]. Finally, SSTM is also fully decentralized, i.e., it does not use global parameters as opposed to other serializable STMs [138, 18] that may experience congestion when scaling to large numbers of cores.

Tracking all conflicts is known to be a difficult task [79] while it is easy to check strict serializability in a composed manner [100], and SSTM may suffer from the induced memory

[1]The proof that SSTM is conflict-serializable is presented in **Appendix B.2**.

overhead. TSTM presented, however, encouragingly low overhead when tracking a subpart of the conflicts [18] SSTM track. Even though SSTM is not expected to be the fastest STM on today's architectures, we believe that hardware support may help tracking these predominant conflicts in a near future, and its design would benefit from this, as it presents already a higher input acceptance than other designs.

3.6 Comparison of STM design classes

Section 3.5 gives some impossibility results on the input acceptance by identifying input classes. Here, we use this classification to compare input acceptance of STM designs.

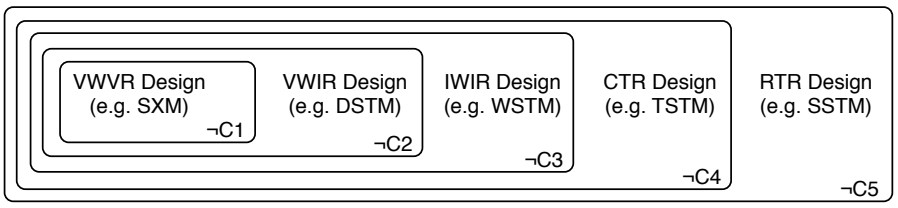


Figure 3.5: Hierarchization of classes. The VWVR design accepts no input patterns of the presented classes, the VWIR design accepts inputs that are not in classes ranging from $\mathcal{C}2$ to $\mathcal{C}4$, the IWIR design accepts inputs that are neither in $\mathcal{C}3$ nor in $\mathcal{C}4$, the CTR design accepts input patterns only outside $\mathcal{C}4$. Finally, we have not yet identified serializable patterns not accepted by the RTR design.

Looking at the class definitions, we identify interesting dependencies. Let $\mathcal{C}0 = \pi*$ be a special class that represents all possible patterns, and let $\mathcal{C}5 = \emptyset$ be the empty class. Observe that any pattern of class $\mathcal{C}4$ is also a pattern of classes $\mathcal{C}0$, $\mathcal{C}1$, $\mathcal{C}2$, and $\mathcal{C}3$, and any pattern of class $\mathcal{C}3$ is also a pattern of classes $\mathcal{C}0$, $\mathcal{C}1$, and $\mathcal{C}2$. For instance, as stated in Theorem 2, STMs implementing the VWIR design (like DSTM) do not accept $\mathcal{C}2$ but $\mathcal{C}5 \subseteq \mathcal{C}4 \subseteq \mathcal{C}3 \subseteq \mathcal{C}2$, hence DSTM accepts none of classes $\mathcal{C}2$ to $\mathcal{C}5$. To represent that a STM accepts only patterns that are outside a class, we draw the sets $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, $\neg\mathcal{C}4$, and $\neg\mathcal{C}5$ that represent $\mathcal{C}0 \setminus \mathcal{C}1$, $\mathcal{C}0 \setminus \mathcal{C}2$, $\mathcal{C}0 \setminus \mathcal{C}3$, $\mathcal{C}0 \setminus \mathcal{C}4$, and $\mathcal{C}0 \setminus \mathcal{C}5$, respectively. We omit to represent $\neg\mathcal{C}0$ since according to our definition it would be $\emptyset$.

Given this hierarchy, we are able to draw the input acceptance of VWVR, VWIR, IWIR, CTR, and RTR designs restricted to patterns that are in $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, $\neg\mathcal{C}4$, and $\neg\mathcal{C}5$, respectively. Observe that we do not propose patterns that are not accepted by SSTM since our first goal is to differentiate designs among each other, however, we could think

of a non-serializable pattern that would not be accepted by SSTM. The hierarchy shown in **Figure 3.5** compares the input acceptance of the STM designs.

3.7 Conclusion

In this chapter, we have presented a qualitative classification of major STM designs based on the unnecessary aborts they generate. Knowing which design generates which class of unnecessary aborts is important since such aborts may generate additional contention which are not due to the workload but just because the design has chosen to abort. Using such knowledge, an STM designer can take the necessary measures to prevent the desired classes of unnecessary aborts according to the performance needs of the applications the STM targets.

Together with the classification, this chapter also contributes in describing a new metric, commit-abort ratio, complementing throughput in the comprehension of why an STM is efficient. An STM can be efficient because it aborts rarely or because it aborts transactions fast. While this can not be deduced merely using throughput, the commit-abort ratio illustrates the reason of efficiency clearly: high commit-abort ratio indicates that an STM aborts rarely, while a low commit-abort ratio points to an STM that aborts fast.

Achieving high commit-abort ratio may require complex algorithms that jeopardize the overhead the STM introduces to the application. SSTM, the STM we propose for obtaining the least number of unnecessary aborts, while offering a high commit-abort ratio, is visibly more complex than the other designs just by looking at its algorithm (see **Appendix B.1**). However, taking TSTM [18] as an example we can think that methods that are both efficient and generating a minimal amount of unnecessary abort may be possible. Additionally, the significant overhead of an STM is due to its execution in software and moving the STM to hardware (i.e., using an HTM) the overhead can mostly be hidden. In such a case, an algorithm such as that of SSTM can be more appropriate than other design alternatives since it has a high commit-abort ratio (introducing less contention due to unnecessary aborts). Although, these are our conjectures, we expect that the work presented in this chapter encourages further research on finding the best tradeoff between design simplicity and high commit-abort ratio.

Chapter 4

Testing Semantics and Performance of TMs: TMunit

4.1 Introduction

The transaction abstraction provided by TMs is meant to simplify the way programmers design their multi-threaded applications. However, this is only true if the semantics of the transaction abstraction are clear for the programmer. Unfortunately, transaction semantics in a multi-threaded program are not yet fully clarified. Evidences of this fact are the recent studies by researchers from industry and academia devoted to important open questions on transaction semantics. Abadi *et al.* [7] studied a question crucial for the compliance of transactional code with non-transactional code: *How should a transaction behave in presence of concurrent non-transactional accesses (in a weak-isolation model)?* Gramoli *et al.* [69] have questioned the necessity of aborts with different TM correctness criteria: *Should a transaction abort even though its commit would not violate consistency*? The question may seem more performance oriented, however defining when a transaction aborts has implications on the transaction guarantee provided to the programmer. A programmer should know exactly what transaction guarantee is provided to him/her. Menon *et al.* [130] raised another issue concerning the TM guarantees when the memory model is not defined: *What result could we expect from a TM implementation when the memory model of the application language relaxes the program order of the code running on each thread?*

All the above questions mainly deal with the data synchronization aspect of TMs. There are yet many questions to be answered about the interaction of TM semantics with other programming language features such as other synchronization constructs, exceptions, I/O accesses, use of legacy code, use of kernel libraries etc. [167, 11, 184].

All these issues and the research conducted to answer them show that there is still a need to explore different aspects of TM semantics. Without such an exploration it will not be possible to obtain a clear definition of TM semantics and until then it is difficult to expect TMs are widely accepted for multi-threaded programming.

Another issue that needs to be covered for wide acceptability of TMs is the correctness of TM implementations. The simplicity provided by the transaction abstraction shifts part of the difficulty of synchronization from application programming to TM implementation. This is because, underneath the transaction abstraction TM actually automates the data synchronization between threads, which is deemed a complicated task. However, if the future programmers need to program using the transaction abstraction, the implementations providing this abstraction should be correct and robust. Thus, efficient ways to verify TM implementations are required for proposing correct TMs to programmers. Without solid TM implementations, the transaction abstraction cannot be provided, leading a failure to use TM for programming.

Last but not least, the overhead a TM introduces has a significant impact on its acceptability. This is especially important for STMs that have a significant overhead in the application execution time. Hence, evaluating TM performance is also critical for TM designs.

All the above issues represent important challenges in designing TM implementations. While it is simpler to address them one by one, these issues are not orthogonal. For example, while we introduce an element in TM design to improve performance, we can easily harm correctness and even inadvertently modify the guarantee the TM ensures. Hence, TM designers need tools to understand the implications of the TM design decisions to all aspects of the implementation.

4.1.1 Contributions

In this chapter, we present a domain specific language that is designed such that one can represent many of the crucial issues about TM semantics, correctness or performance in a concise and intuitive way rather than requiring to write complete programs to express these different situations[1]. This representation is then interpreted by the accompanying tool, TMUNIT, that reproduces the issues at run-time and apply it to a specific TM implementation to observe the response of the implementation to the issue. Notably, TMUNIT allows the user to design both deterministic and non-deterministic tests. Deterministic tests are especially useful for unit tests and comprehension of TM semantics while non-deterministic

[1]While we use the term "TM" for generality, we only consider here *software* transactional memories (STMs).

tests introduce randomization allowing concurrency and, hence, are important to assess TM performance. The combination of the domain specific language with TMUNIT permits the testing of both the semantic and performance aspects of TM implementations.

4.1.1.1 Deterministic tests

Most previous studies discussing semantic properties of TMs indicate the essential role of specific interleavings of conflicting operations in transactions. Various examples have been described as code fragments that may expose anomalies when transactional operations are executed in a certain order [74, 166, 7, 130, 79, 69, 155, 120, 90] but no tests of real TMs have been reported.

Consider, for instance, the well-known problem of dirty reads. On the left-hand side of **Figure 4.1**, the first thread executes a transaction (represented as an `atomic` block) that updates the same location x twice while the second thread concurrently reads location x. Since the transaction should appear as if it were executed atomically, it should not be possible to have $y = 1$. This would correspond to a dirty read of location x by the second thread. Observe that, when both threads execute in parallel, some interleavings of operations may never produce dirty reads. To test appropriately that a given TM avoids this problem, we propose, on the right-hand side of **Figure 4.1**, the specification of a dirty read unit test. The interleaving of the transactions is defined in a schedule that enforces the read operation of x by the second thread to occur between the two atomic write operations of the first thread. The invariant checks whether a dirty read occurs in this specific schedule. Such a test language specification is of crucial importance for testing TM behaviors in particular situations, notably in scenarios with concurrent transactional and non-transactional accesses.

Initially: x=0, y=0

Thread 1	Thread 2

atomic {
x=1;
 	
x=2;
}
	
	y=x;

Definitions: x = 0; y = 0;
Transactions: T := W(x,1), @L, W(x,2);
Schedules: S := T@L, { y=x }, T;
Invariants: [y != 1];

Figure 4.1: Dirty read test: a simple pathological scenario that may lead to a dirty read ($y = 1$) on the left-hand side, and the corresponding specification to test if the TM avoids the dirty read on the right-hand side. Note that $y = x$ is executed outside transactions and we define a label $@L$ in transaction T to specify the interleaving of operations in the schedule S.

Testing concurrent programs in a reproducible manner is not an easy problem [182, 50, 137]. One might think of recording the events of the execution at run-time to replay them later on. Unfortunately such recording may directly affect the way events are interleaved. Exhaustive testing of concurrent executions is also tedious, when practical, as the number of executions to test grows exponentially with the number of events each thread executes. Conversely, the amount of possible executions where threads are interleaved at the level of transactional operations is much more reasonable, which makes them testable. Actually sequential specification of TMs use this level of interleaving to describe TM semantics, assuming that transactional operations occur instantaneously in time with respect to each other [90]. Hence, it should be enough to test TM semantics with executions represented as interleavings of transactional operations.

Even though we believe that TM developers are skilled programmers that are used to avoiding lower-level concurrent programming issues, they also need tools to test whether their design behaves correctly. The deterministic tests that can be designed by TMUNIT can also be used for this purpose because, TMUNIT allows TM designers to define interleaving points (which corresponds to the label `@L` in **Figure 4.1**) even in the TM source code. This way, it is possible to design deterministic tests where the statements of transactional operations are interleaved (e.g., by defining a schedule like the schedule S of **Figure 4.1** but using the interleaving points introduced in TM source code instead of labels of transaction descriptions). Although we do not think it is useful for exhaustive testing, we argue that this should be very useful for TM developers to test corner cases of their design, or even to generate scenarios that would help debugging.

4.1.1.2 Performance tests

Our domain specific language is flexible enough to describe data access patterns in transactions, as well as to specify the transaction execution patterns in threads. We call such pattern specifications **performance tests**. Contrary to deterministic tests, in performance tests randomization can be introduced to express complex data access patterns. Using the expressive power of the language we developed a test-suite for TMUNIT composed of tests for very specific data access patterns (see **Section 4.6.2**) and executions that closely mimic micro benchmarks (see **Section 4.6.1**).

Like traditional TM evaluation frameworks, TMUNIT runs a given TM on some, possibly randomized, performance tests and records the performance statistics. The specified workload can be executed by dynamically interpreting the workload specification and map-

ping transactional accesses to an underlying TM, or for best performance, by generating a corresponding program to be compiled into a standalone application.

Thanks to the abstract interface provided by TMUNIT a specific workload expressed in the domain specific language can be run using different TM implementations, giving the opportunity to compare performance of TM designs. Taking advantage of this interface we were able to rapidly perform numerous tests on 6 different TM implementations: RSTM [127], SwissTM [47], TinySTM [57], TL2 [42], WSTM [89] and $\mathcal{E}$-STM [58].

4.1.2 Roadmap

This chapter is organized to show different testing aspects of TMs that are simplified by the domain specific language and the tool (TMUNIT). We first present related work on the subject and then we present the tool and the language. The rest of the chapter illustrates how the tool can be used to test the semantic, correctness and performance issues of TMs.

4.2 Related work: Testing concurrent programs

4.2.1 Testing semantics

In this section, we overview work on testing semantics and correctness of concurrent applications. These studies can be categorized mainly in two groups; *(i)* randomized tests that explore different possible interleavings in a non-deterministic manner to identify concurrency defects, and *(ii)* deterministic tests that can be replayed to identify the causes of potential issues.

The tightest related body of work relies on the dynamic testing technique that collects information about concurrency defects by executing the software itself as opposed to the static technique (see [54, 83] for details and references) that analyzes source code to extract information about possible defects such as data races, atomicity violations or deadlocks. Since TMUNIT also follows the dynamic testing technique, we discard the static techniques as being out of scope. In what follows, we focus on dynamic testing techniques.

4.2.1.1 Randomized tests

All the dynamic testing techniques apply an instrumentation method to collect information from an executing program. Part of those studies perform non-deterministic testing in the sense that the program instrumentation does not control the schedule of threads and each execution of the program results in a different schedule (we call such test randomized tests). Some randomized testing approaches target the detection of data races [159, 185,

141, 150, 195, 52], while some other target the detection of atomicity violations [61, 121, 193, 187, 123]. Since the tested schedules with such randomized approaches are limited, existing frameworks [177, 21] try to increase the schedule coverage by inserting sleep or yield functions to produce scheduling variations.

All the above cited work, perform tests for concurrent programs in general. TM specific testing and verification has not yet been studied as much. An important study in this direction is by Manovit *et al.* [125] where pseudo-random tests are used to verify whether the execution matches the consistency specified by the TM.

Unfortunately, randomized tests are inherently irreproducible, hence, they better apply to test a program as a whole rather than to identify precise problematic scenarios. In contrast, one goal of TMUNIT is to identify the pathological schedules that make TM fail or violate its semantics. Hence, the reproducibility of the testing is crucial for our needs.

4.2.1.2 Deterministic tests

A pioneer work on deterministic testing of concurrent programs is by Carver et al. [182], which is based on generating deterministic schedules and replaying them for testing. In their approach, the interleaving points of the schedules are the synchronization operations (e.g., mutex or semaphore accesses) and, hence, they can generate a schedule using a sequence of synchronization operations of the tested program.

Since this pioneer work, the advances in capturing possible thread schedules of a concurrent program allowed automatic generation and exploration of schedules where each specific schedule can be replayed deterministically for analyzing concurrency defects. Different proposals generally differ in the size of schedule space they analyze and in the nature of schedules they choose to analyze. ConTest [50] generates different schedules based on randomization or some coverage metric (such as covering different concurrency bug patterns). ExitBlock [28] limits itself to terminating lock-based programs and seeks to reduce the size of the schedule space by considering schedules where interleavings occur only at the points where a lock is released. CHESS [137] is another study that models all synchronization operations as interleaving points, and restricts the schedule search space by *(i)* using fair schedules and *(ii)* by limiting number of pre-emptions that occur in a schedule. RichTest [115] is a hybrid technique that is partially deterministic and partially non-deterministic. This work starts generating a schedule in a non-deterministic manner, stores this schedule in a vector time-stamped trace and generates all relevant interleavings based on this schedule. The approach tries to limit the schedule space using schedule symmetries and partial-order reductions. Another hybrid approach proposes to limit schedule space by analyzing the

source code to find out thread schedule equivalence classes and perform test execution only on a single member of a given equivalance class [34].

Verisoft [66] proposes to use systematic state-space exploration (model-checking) instead of scheduling space exploration. In this approach, the state-space corresponds to the combined behavior of all concurrent components of the program, and thanks to this model Verisoft applies the model-checker directly to the program (in contrast with other model-checking alternatives that perform checking on a separate model of the program). This approach eliminates the redundant state transitions based on independence of transitions.

Unlike aforementioned proposals that generate and test the schedule search space of concurrent programs, we focus on schedule search space of TM programs. More precisely, our goal is not to detect races on a TM metadata[1] but rather detecting data races between data items accessed through TMs. Hence, our approach inherently limits the number of schedules to explore which makes deterministic testing feasible. It is noteworthy that verification of TM correctness [75, 76, 77, 142] is based on a formal specification of TM algorithms and not on the actual TM code which makes the testing process more difficult. Other testing tools [94, 202] help on the step-by-step debugging of TMs but are out of the scope of this thesis.

Finally, we should mention MultithreadedTC [151] and ConAn [119] as unit testing frameworks that use an external clock, similar to the scheduler thread of TMUNIT, to synchronize threads and enforce deterministic schedules for Java programs. MultithreadedTC requires that the tests are written in Java, thus the final test code is still program-like and is not as succinct as the input files of TMUNIT. Also MultithreadedTC organizes test codes such that the code of each thread is given separately. This way the final schedule is defined by the combination of thread codes whereas TMUNIT can express the schedule simply as a series of events, which is easier to design. ConAn expresses the tests in a dedicated script language and organizes the tests such that the user expresses the events that are triggered at each time slot. Contrary to ConAn the schedules in TMUNIT are expressed without dependence to time: it is enough to give the order of events as they should occur.

4.2.2 Testing performance

Shared-memory management in concurrent programs have been extensively evaluated as demonstrated by the numerous multithreaded benchmarks such as SPLASH-2 [191], PARSEC [24], SPEComp [17]. Among these benchmarks SPEComp is specialized on high performance computing applications, while SPLASH-2 and PARSEC include applications

[1] Although we can even achieve designing schedules to analyze race on TM metadata, as explained in **Section 4.3.3.3**, this is not our primary objective.

from different domains. In addition, there exist other multithreaded benchmark suites specialized to a given application domain like BioParallel [107] dedicated to bioinformatics, ALPBench [124] dedicated to multimedia and MineBench [139] dedicated to data mining. The sole work we know of that attempted to port such applications to be used with transactional memory, is by Chung et al. [36], however, it uses lock elision which is reported to be unsafe [26, 36].

In contrast with the aforementioned benchmarks, TM benchmarks are very appealing as they can test any TM as long as it fulfills the given interface requirements. TM micro-benchmarks that are based on common data-structure have been used to evaluate TMs at the early stages. TL2 has been implemented on red-black trees [42] while TinySTM has been implemented on linked-lists [57]. DSTM has been implemented on both linked-list and red-black trees [97]. Microbench is a micro-benchmark suite that compares recent lock-free and lock-based implementation of common data structures (including skip list, hash table) to some pluggable TM [58]. Those benchmarks test few different type of transactions that modify or search the data-structure.

As further attempts to mimic realistic settings, more complex TM benchmarks have been proposed. STMBench7 [81] extends the OO7 benchmark that was used to evaluate databases. This benchmark executes several types of workloads that access a large graph of updatable elements. Generally, the transactions it generates are more complex than in micro-benchmarks as the transactions can be very long without necessarily accessing common elements of the data structure. As a side-effect, it consumes more memory than micro-benchmarks. Haskell STM benchmark suite [148] provides both small realistic applications and several micro-benchmarks written in Haskell. Wormbench [200] intends to provide synthetic workloads that can stress specific design and implementation aspects of TM systems including TM library, code instrumentation and compiler optimizations.

Finally TM macro-benchmarks are higher level benchmarks that often integrates real-world applications. The most widely used (especially for STMs) macro-benchmark suite STAMP [133] comprises eight different parallel applications as, for example, an online multi-threaded reservation service or a Delaunay triangulation algorithm. Lee-TM [16] is a benchmark suite based on the Lee's routing algorithm and provides implementations of the algorithm in different granularities for both lock-based and transactional versions. QuakeTM [65] and Atomic Quake [201] respectively provide a coarse-grained and a fine-grained parallelized transactional version of a multiplayer game server. RMS-TM [109] provides 4 benchmarks with nested transactions, memory management operations and I/O calls inside transactions from recognition, mining and synthesis domains.

Although those macro-benchmarks are invaluable for TM developers, they are limited by the number and type of applications available: extending the benchmark space requires to fully implement new applications.

Recently, Demsky and Dash [41] proposed a discrete event simulator that generates transaction and thread descriptions based on transaction events (such as start, commit, read and write) and used it to explore contention manager performance. However, the simulator generates either random input, where the description of transaction contents cannot be controlled by the user, or converts a real execution trace to its internal representation. In this sense, compared to TMUNIT, the simulator provide little flexibility in defining workloads. Also, the simulator has its embedded (and generic) TM design. Hence, it does not allow testing the same workload with different TM implementations. Furthermore, the main focus of the simulator is assessing contention manager performance.

4.3 TMunit

In this section, we describe TMUNIT, the testing framework we have developed to rapidly write performance and semantic tests for TMs.

4.3.1 Overview

Here, we describe the main components of TMUNIT whose interactions are depicted in **Figure 4.2**. TMUNIT executes a synthetic workload written in a domain-specific language (configuration file), on a dedicated TM, and records performance statistics and test results. To this end, TMUNIT uses a parser to transform the workload into an executable. This parser is written using the `lex` and `bison` tools. Depending on the choice of the user, it can either output an *interpreted* automaton, to execute the workload dynamically, or a *generated* automaton, to reduce the runtime overheads.

4.3.1.1 Automata

The interpreted automaton executes based on a data structure representing the workload. Typically, this data structure comprises a set of linked lists, each representing a transaction whose nodes are the operations to execute. This data structure allows loops and conditional executions.

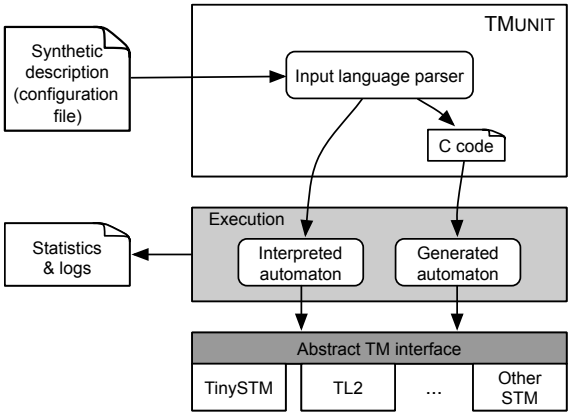


Figure 4.2: Architectural overview of TMUNIT.

The generated automaton is the translation of the configuration file into source files in a specific programming language.[1] These source files can then be compiled and executed as a stand-alone application.

While the interpreted automaton can be used to interpret and execute the configuration file in a one-step process, the generated automaton executes with negligible overheads, as shown later in **Section 4.6.1**. In contrast, the interpreted automaton is more convenient for execution of simple unit tests, e.g., when testing TM semantics.

4.3.1.2 Execution

The automata can be executed in two different modes: the *schedule mode* and the *parallel mode*. In the schedule mode, the execution corresponds to the sequence of transactional operations described in the schedule definition of the given workload. Each transaction is executed in a separate thread and only one thread is active at a time. This mode is mostly convenient for unit tests (see **Section 4.3.3**). TMUNIT can also automatically generate schedules and execute them. In such a case, TMUNIT will generate schedules that correspond to all possible interleavings of the transactional operations (where each transaction runs on a separate thread). In the parallel mode, the execution is performed according to a thread specification described in the given workload (see **Section 4.5** for thread specifications) In this execution mode threads execute concurrently and, hence, can be used to test performance related issues (see **Section 4.6**).

[1] Currently, only the C language is supported for generated automaton.

4.3.1.3 Abstract TM interface

Each automaton communicates with the underlying TM using a standard TM interface to initialize the transactional memory and thread data structures and to delimit transactions, e.g., `start` and `commit`, but also to execute transactional operations, e.g., `read` and `write`. TMUNIT has generic calls for these operations and the collection of these generic calls form its abstract TM interface. To plug a new TM to TMUNIT a user simply needs to map a transactional operation of this TM to its corresponding generic TMUNIT call (using a header file), and to recompile TMUNIT.

Although this thesis work focuses on the results obtained with STMs, the interface is generic enough to support HTMs as well. TMunit has been successfully adapted to be used with AMD's Advanced Synchronization Facility (ASF) [6] instruction set, which provides HTM support. The adaptation relied essentially on replacing software barriers with hardware barriers. TMUNIT has been used to test and identify bugs in ASF.

4.3.2 Simplicity and expressiveness

The language has been designed to be simple enough to specify transactions and schedules in an abstract way as usually found in academic papers [166, 7, 130, 79, 69, 155, 120] and expressive enough to reproduce classical transactional benchmarks using multiple data structure accesses.

```
1 T1 := R(x), R(y), W(x), W(y);                          // R = read, W= write
2 T2 := R(x,_a), R(y,_b), W(x,_a-10), W(y,_b+10);    // _a, _b = thread locals
```

Listing 4.1: Two sample transactions.

Listing 4.1 illustrates how simple it is to specify basic workloads. The first transaction, T_1, reads two memory locations before updating them (note that transaction beginning and commit are implicit). Memory locations are designated by symbolic addresses that are mapped to shared memory by TMUNIT. Here, we are not interested in the value read or written, i.e., we are only interested in possible conflicts. In contrast, T_2 stores the values read in local variables and writes updated values to shared memory, similar to a transfer between bank accounts. One can specify far more sophisticated behavior in transactions, as will be discussed next.

A workload (unit test or performance test) is written as a configuration file divided into six sections:

1. The *properties* section presents the execution settings and parameters.

2. The *definitions* section specifies the variables and constants.
3. The *transactions* section defines the operations that compose each transaction, using a simple but sufficiently powerful language.
4. The *threads* section specifies each thread as a transaction pattern.
5. The *schedules* section describes specific executions with a pre-determined interleaving of operations.
6. The *invariants* section specifies assertions that must be valid at each step of a schedule.

4.3.3 Unit test specification

Here, we consider a specific kind of test especially suited for TMs: they represent a deterministic scenario of a parallel execution. As motivated in the introduction, there is a crucial need for unit testing TMs to outline problems due to certain interleavings of conflicting operations. Note that unit tests in TMUNIT can include interleavings with non-transactional accesses, which allows testing strong atomicity support of TMs (as in **Figure 4.1**). Listing 4.2 illustrates our domain-specific language on the *zombie transactions* example of [7] (we have actually reproduced the equivalent variant of the zombie transactions example as sent by Dan Grossman on the *tm-languages* mailing list on July 1, 2008). The write to z by T_2 on Line 6 is dead code under single-lock semantics and should not happen. However, some TM implementations with eager update might perform the write and undo it later, causing the assertion to fail. Such a unit test can help determining the interference of one transaction to the other, unveiling semantic properties of TM implementations.

```
1  Definitions:                                    // variables and constants
2    y = 0; x = 0; z = 0;                  // shared variables, initially all 0
3
4  Transactions:                                 // specification of transactions
5    T1 := W(x,1), @L1, W(y,1);                        // W = write, @L1 = label
6    T2 := {? [ R(x) != R(y) ] : W(z,1) };     // R = read, {?:} = if statement
7
8  Schedules:                                      // specification of schedules
9    S := T1@L1, T2, T1;               // execute T1 until L1, then T2, finish T1
10
11 Invariants:                                       // invariants to fulfill
12   [z != 1];                                       // unprotected read of z
```

Listing 4.2: Unit test for zombie transactions [7].

4.3.3.1 Operations and transactions.

We assume a single address space of bounded size. Threads can only communicate by writing to, and reading from, the shared address space. We denote reads by `R` and writes by `W`. These two operations can only be applied to shared memory variables. Variables are defined in the *definitions* section and are either integers (of the size of a memory word) or arrays of integers. One can also use thread-local variables. Their name must start by an underscore symbol '_' and is scoped at the level of the transaction (they can be referred to as `<tx-name>:<var>` to avoid ambiguity). Identifier names with only capital letters are considered as constants and their value cannot be modified. A read operation accesses a shared variable, or an entry in a shared array. The read operation returns the content of the shared variable as provided by the underlying TM. The result can optionally be recorded in a thread-local variable. We denote this by `R(<sh-var>)` or `R(<sh-var>, <loc-var>)`. Similarly, a write operation accesses a shared variable `W(<sh-var>)` to write a value that can be optionally specified `W(<sh-var>, <val>)`. We refer to shared variable accesses via read and write operations as *protected accesses*, and to direct shared variable accesses (e.g., `x = 0`) as *unprotected accesses*. TMUNIT supports arithmetic expressions involving numbers, variables, random values, arithmetic operators, and parentheses that yield an integer value.

Each transaction is given a unique name and represents a finite sequence of operations, delimited by commas, implicitly started by a "begin" statement and ended by a "commit"[1] . It is possible to explicitly abort a transaction by using notation `A` to implement sophisticated test scenarios. Inside a transaction and between operations, labels can be specified by `@<label>` and local variables can be assigned values. In Listing 4.2, T_1 contains two operations (Line 5) while T_2 contains one operation and an *if* statement with two operations (Line 6). Label `@L1` is used in T_1's definition as a marker for specifying the schedule as explained below.

4.3.3.2 Schedules and assertions.

Schedules specify a pre-defined interleaving of the transactional operations for testing or debugging a TM. If no schedules are predefined, all different schedules of transactional operations will be automatically generated and tested, which may take time. If schedules are specified they are defined in the *schedules* section (multiple schedules can be specified

[1] Transaction start and commits are not expressed in the input, but they are visible in the execution output: `S` denotes that a transaction start operation has been executed and returned, `Try C` denotes that a transaction commit is about to be invoked and `C` denotes that a transaction commit has been successfully executed and returned.

in the same file) and at each execution of TMUNIT only one of the specified schedules is run (the schedule to run can be given as a run-time parameter). A schedule specifies the execution order of the transactional operations using `<tx-name>@<label>` to indicate that `<tx-name>` executes alone until label `@<label>`. If no label is specified, the transaction executes until the end. Note that, during the execution of a schedule, each transaction executes in its own thread, but only one thread is active at each step of the schedule. This is a design choice in order to provide repeatable schedules and unit tests. Multiple schedules can be specified but only one will be executed at a time; the schedule to execute can be specified using command-line parameters.

The scheduler acts as a sequential process executing transactional operations in turn. In schedule execution mode, TMUNIT creates a thread per each defined transaction and an additional *scheduler* thread. The scheduler thread performs the switching between threads thanks to the barriers that correspond to the labels in the transaction definitions. There are also two additional barriers; one is used only for the scheduler and the other is an initial common barrier for all the threads except the scheduler. The barriers are used to pass a token between the threads and at a given time there is only one thread that owns the token. Initially, the scheduler thread owns the token and it passes the token to the first scheduled transaction. The thread owning the token executes until it encounters the next barrier, which corresponds to a label in the configuration file, and then passes the token back to the scheduler thread. If the thread encounters no barriers/labels it executes until the end of the corresponding transaction before passing the token to the scheduler. Upon receipt, the scheduler thread passes the token to the next scheduled transaction's thread. This continues until the end of the defined schedule.

A schedule can also include direct shared memory accesses in its specification through variable assignments. An example of such assignment is the `y=x` assignment illustrated in the schedule of **Figure 4.1**. Assignments of this type are performed by the scheduler thread and the result of the assignment is immediately visible by all threads. Such assignments are useful to generate interleavings resulting in transactional data races (races between transactional and non-transactional code) and, hence, allow us to verify whether a TM supports strong isolation or not.

Invariants and assertions define tests that the execution must pass. Assertions are boolean expressions and can be specified in transactions or schedules as `[<bool-expr>]`. Invariants are assertions that are automatically evaluated at each step of a schedule. If an assertion evaluates to false or if an invariant is violated during the execution, then the test fails. The program prints an error message. The evaluation context of variables within boolean expressions used in assertions and invariants are as follows.

- Local variables are evaluated in the context of the transaction they are used in.
- Unprotected accesses, like `[z != 1]` (Line 12 of Listing 4.2), are evaluated by directly reading the memory location, i.e., without using the underlying TM.
- Protected accesses, like `[R(x) != 1]`, evaluate in a dedicated transaction that performs only the protected access.

An example schedule for the "zombie transactions" scenario [7] is presented at Line 9 of Listing 4.2. In this schedule, transaction T_1 executes up to label L_1, and then transaction T_2 runs before $T1$ resumes. As a result, this schedule forces T_2 to read x between the two writes of T_1. If T_2 reads a dirty value 1 (due to the TM implementation), it will update z (Line 6) and the invariant (Line 12) will be violated, leading to the failure of the test.

4.3.3.3 Increasing interleaving granularity

With the use of transactional operations, schedules and assertions in specifying unit tests one can only define test executions where transactional operations appear as if they were atomic (i.e., the implementations of transactional operations are never interrupted). While this is enough to test the semantic properties of the sequential specification of a TM, TM developers may need to test different cases where the internal implementations of transactional operations are interleaved. Executions with such interleavings are useful to verify correctness of TM design (e.g., to ensure that transactional operations act as if they occurred atomically) or even to find some bugs.

Obtaining such interleavings is not more difficult than generating an ordinary schedule (as defined in terms of transactional operations). A TM developer wishing to design such a test needs to include a header file provided by TMUNIT and introduce label names (with the use of a macro defined in the included header) to the points of the code where the transactional operations will be interleaved. The schedule specification in the configuration file of TMUNIT is capable of using the named labels introduced in the code as interleaving points in the execution of a transaction. Hence, the desired interleavings can be expressed in the schedule, resulting in a test, a TM developer can use for verifying behavior of concurrently executing transactional operations. Although such tests do not indicate where a bug is, a developer can easily try the interleavings s/he suspects to be related with the bug and see the result of the execution with TMUNIT without any coding effort. This is useful since a minimal modification (only introducing labels) allows the developer to obtain a deterministic schedule to test the interleaving s/he desires whereas otherwise s/he needs to insert code into his/her implementation in order to obtain the same deterministic schedule.

4.4 Testing semantic properties with TMunit

In this section, we present the tests that constitute our semantic test suite. This test suite, although is of moderate size, shows the variety of tests that can be obtained using the domain specific language. Current set of tests include tests on issues such as TM guarantees and consistency, isolation and liveness. We elaborate these tests in the upcoming sections. Also a summary of the results is presented in **Table 4.1** at the end of the section.

4.4.1 Safety tests

4.4.1.1 Consistency tests

The safety tests are interesting to better understand the consistency criterion ensured by TMs. Some TMs ensure opacity, some other ensure serializability. As pointed out in [69], however, some serializable TMs may also be opaque.

Here, we chose four safety tests from the literature. The *opacity* test (1) presented in **Figure 4.3** (top) comes from [79] (Fig. 1) and was used to illustrate the difference between strong isolation and opacity requirements. The *strict serializability* test (2) in **Figure 4.3** (middle) and taken from Figure 2 of [69] (Fig. 2), aims at showing that a serializable TM may not be strict serializable. The *serializability* test (3) depicted in **Figure 4.3** (bottom) and coming from Figure 2 of [155] (Fig. 2) exhibits that a TM is not serializable. Finally, the *SLA* test (4) illustrated in **Figure 4.4** and given in [130] (Fig. 11) indicates how single-lock-atomicity (SLA) [130] can be violated if the TM does not synchronize transactions accessing disjoint data.

TL2, TinySTM, RSTM, and SwissTM pass all the critical tests we proposed except the SLA test. Although this does not prove that they ensure the corresponding criterion, it simply shows that those TMs do not violate consistency in these specific scenarios. For the tests, $\mathcal{E}$-STM has been restricted to its elastic transaction implementation, and $\mathcal{E}$-STM fails to pass the strict serializability and serializability tests[1] . Since opacity is known to be strictly stronger than these two criteria we deduce that $\mathcal{E}$-STM violates also opacity. This was expected as the elastic transactions weaken intentionally the normal transactions for high level operations. Thus, testing the strict serializability of an integer set rather than the read/write would reveal that $\mathcal{E}$-STM ensures strict serializability, but at a higher level of semantics.

[1]Note that elastic transactions allow running both regular and elastic transactions in the same execution and if the test was applied for regular transactions of using $\mathcal{E}$-STM, these tests would have passed.

Opacity test

```
1  Definitions:
2  x=0; y=0; _t=0;
3  Transactions:
4  T1 := W(x,1);
5  T2 := R(x), @L, R(y,_t);
6  T3 := W(x,2), W(y,2);
7  Schedules:
8  S := T1, T2@L, T3, T2;
9  Invariants:
10 [_t!=2];
```

```
...
    [Th3:T3] W(y)
    [Th3:T3] W(y,2)
    [Th3:T3] Try C
    [Th3:T3] C
[Th2:T2] R(y)
[Th2:T2] R(y,2)
Invariant '[_t!=2]' failed.
```

Strict serializability test

```
1  Definitions:
2  x=0; y=0;
3  Transactions:
4  T1 := R(x), @L, R(y);
5  T2 := W(x);
6  T3 := W(y);
7  Schedules:
8  S := T1@L, T2, T3, T1;
9  Invariants:
10 [No-abort];
```

```
...
        [Th3:T3] S
        [Th3:T3] W(y)
        [Th3:T3] W(y,23264)
        [Th3:T3] Try C
        [Th3:T3] C
[Th1:T1] R(y)
[Th1:T1] A
Invariant NO_ABORT fails.
```

Serializability test

```
1  Definitions:
2  x=0; y=0; z=0; t=0;
3  Transactions:
4  T1 := W(x), W(y);
5  T2 := W(z), W(t);
6  T3 := R(z), @L2, W(y);
7  TL := R(x), R(y), @L1, R(z),
     W(t);
8  Schedules:
9  S := TL@L1, T3@L2, T1, T2, T3,
     TL;
10 Invariants:
11 [No-abort] ;
```

```
...
[Th2:T2] W(t)
[Th2:T2] W(t,27225)
[Th2:T2] Try C
[Th2:T2] C
    [Th3:T3] W(y)
    [Th3:T3] W(y,23264)
    [Th3:T3] Try C
    [Th3:T3] A
Invariant NO_ABORT fails.
```

Figure 4.3: (Top) The opacity test for which WSTM fails while other TMs succeed. WSTM trace appears on the right. **(Middle)** The strict serializability violation test for which all TMs pass (except $\mathcal{E}$-STM). The tests terminate with an abort failure which indicates strict serializability is not violated. The trace for TL2 appears on the right. **(Bottom)** The serializability violation test that all TMs pass (except $\mathcal{E}$-STM). The tests terminate with an abort failure indicating that serializability is not violated. The trace for TL2 appears on the right.

Unexpectedly, however, WSTM successfully passes the strict serializability and serializability tests but not the opacity test which indicates that WSTM is not opaque. The

SLA test

```
1  Definitions:
2   x =0; y =0; z=0;
3   t1=0; t2=0;
4  _u =0;
5  Transactions:
6   T1 := W(z,1);
7   T2 := R(x,_u), W(t1,_u), @L,
      W(y,1);
8  Schedules:
9   S:= T2@L, {x = 1}, T1,{t2 = y},
10     T2, [!((t1==0) && (t2==0))]
11 Invariants:
12  [No-abort];
```

```
...
-------  x = 1  -------
[Th1:T1] S
[Th1:T1] W(z)
[Th1:T1] W(z,1)
[Th1:T1] Try C
[Th1:T1] C
-------  t2 = 0  -------
   [Th2:T2] W(y)
   [Th2:T2] W(y,1)
   [Th2:T2] Try C
   [Th2:T2] C
Assertion [!((t1==0)&&(t2==0))]
not satisfied.
```

Figure 4.4: The SLA violation test for which all TMs fail. The tests terminate with the violation of the assertion in the schedule which indicates that SLA is violated. Right-hand side shows the trace obtained from running TL2.

detailed TMUNIT trace gave us some information about the reason of opacity violation: opacity as opposed to strict serializability requires that no transaction (even though it aborts) can see the result of the modification of another concurrent transaction. As shown in the *interference* test of **Figure 4.6** (top), WSTM allows this to happen. The design of WSTM intentionally separates the isolation from the transactional memory abstraction, however, the programmer can explicitly validate to avoid isolation issue or can assume sandboxing that will prevent isolation errors like division-by-zero. This observation raises the question whether isolation should be part of the transactional memory semantics, as recently suggested by opacity and virtual-world consistency (see **Section 2.4.4.1**).

4.4.1.2 Isolation tests

We tested five critical scenarios among which three include non-transactional accesses. All considered TM implementations used here ensure *weak isolation*: transactions appear as if they were isolated with respect to each other but not with respect to non-transactional accesses (we did not use the fences of RSTM that would counteract our schedules).

Specifically, we performed the following isolation tests. The *publication* test (5), as described in **Figure 4.5** (top), outlines a possible difference between the TM semantics and the lock semantics of the Java memory model [130]. The *dirty-read* test (6) discussed in the introduction is specified in **Figure 4.5** (bottom) and the *zombie transaction* (7) test has been specified in Listing 4.2. The *interference* test (8), described in **Figure 4.6** (top),

Publication test

```
1  Definitions:
2   data=42; ready=0;
3   val=0 ; _tmp=0;
4  Transactions:
5   T1 := W(ready, 1);
6   T2 := R(data,_tmp), @L,
7     {? [R(ready)==1] : W(val,_tmp)};
8  Schedules:
9    S := T2@L, {data=1}, T1, T2;
10 Invariants:
11   [val!=42];
12   [No-abort];
```

```
...
-------  data = 1  -------
[Th1:T1] S
[Th1:T1] W(ready)
[Th1:T1] W(ready,1)
[Th1:T1] Try C
[Th1:T1] C
    [Th2:T2] W(val)
    [Th2:T2] W(val,42)
    [Th2:T2] Try C
    [Th2:T2] C
Invariant '[val!=42]' failed.
```

Dirty-read test

```
1  Definitions:
2    x=0; y=0;
3  Transactions:
4    T := W(x,1), @L, W(x,2);
5  Schedules:
6    S := T@L, {y=x}, T;
7  Invariants:
8    [y!=1];
```

```
[Th1:T] S
[Th1:T] W(x)
[Th1:T] W(x,1)
[Th1:T]:L
-------  y = 1  -------
Invariant '[y!=1]' failed.
```

Figure 4.5: (Top) The publication test where all TMs fail. Right-hand side shows the trace for SwissTM. **(Bottom)** The dirty-read test where only TinySTM's WT variant fails because both transactional and non-transactional writes immediately modify the memory. Right-hand side shows the trace for TinySTM-WT.

outlines a possible interference between two transactions. Finally, in **Figure 4.6** (bottom), the *granularity* test (9) raises issues relying on the granularity of TM accesses when a coarse-granularity may have side-effect on locations that were not intentionally accessed.

As expected [130], all TMs fail the publication test (5). The reason is simply that none of the considered TMs can ensure isolation when non-transactional code accesses shared data.

Only TinySTM-WT fails the dirty-read test (6), other TMs succeed. This is due to the write-through strategy with which updates are directly written to memory when encountering a write operation and can potentially be reverted upon abort. Note that the other TMs might also exhibit this problem if an unprotected read occurs while a transaction is committing, but this scenario is not specified by our schedule.

All TMs pass successfully the test of zombie-transaction (7). First, all the TMs that use the write-back strategy defer modification to commit time so that transaction T_2 reads

Interference test

```
1 Definitions:
2   x=0; y=0; _v=0; _w=0;
3 Transactions:
4   T1 := R(x,_v), @L, R(y,_w);
5   T2 := W(x,1), W(y,1);
6 Schedules:
7   S := T1@L, T2, T1, [_v==_w];
```

```
...
[Th1:T1] R(y)
[Th1:T1] R(y,1)
[Th1:T1] Try C
[Th1:T1] A
[Th1:T1] Terminates
Assertion [_v==_w] fails.
```

Granularity test

```
1 Definitions:
2   arr[0..1]=0;
3 Transactions:
4   T1 := W(arr[0],1), @L;
5 Schedules:
6   S := T1@L,{arr[1]=1},
7      T1, [ arr[1] == 1];
```

```
[Th1:T1] S
[Th1:T1] W(arr[0])
[Th1:T1] W(arr[0],1)
[Th1:T1]:L
-------  arr[1] = 1  -------
[Th1:T1] Try C
[Th1:T1] C
Interruption in Schedule:
Assertion [arr[1]==1] not satisfied.
```

Figure 4.6: (Top) The interference test for which only WSTM, the trace of which is on the right side, fails. **(Bottom)** The granularity test where only object-based RSTM fails. The trace for object-based RSTM is also given on the right side.

value 0 for x. Second, TinySTM-WT aborts immediately T_2 when trying to read x; hence the read does not return and the invariant is not violated.

The role of *interference* test (8) given in **Figure 4.6** (top) is to check whether a writing transaction can interfere with a concurrently reading transaction. For TL2, TinySTM, RSTM, SwissTM, and $\mathcal{E}$-STM interference was prohibited, meaning that the write could not interfere with an ongoing reading transaction, even if this ongoing transaction eventually aborts. Conversely, WSTM allows interference and makes a transaction read a concurrently written value before aborting. This interference is the reason why WSTM violates opacity and virtual-world consistency (this violation has been detected in **Section 4.4.1.1**), since opacity and virtual-world consistency both require that strict serializability be satisfied and that transactions do not interfere with aborting transactions. As a result, if the user of WSTM is not aware of this subtle characteristics, then his/her application may behave unexpectedly: T_1 observing that x is different from y may provoke an infinite loop, a division-by-zero or an irrevocable external event, like missile firing [72].

A last isolation test we have specified is a *granularity* test (9) depicted in **Figure 4.6** (bottom) similar to some of [166]. We obtained the following results by running this test: *(i)* no problem occurs for WSTM, TinySTM, TL2, SwissTM, word-based RSTM, and $\mathcal{E}$-STM; *(ii)* the problem occurs only for object-based RSTM. As expected, word-based STMs do not

suffer from this issue as they do not buffer the whole array to roll it back. In contrast, the object based version of RSTM, which accesses the memory with the granularity of objects, fails the test. This is due to the fact that we consider the whole array as a single object instead of considering that all of its elements are individual objects.

4.4.2 Liveness tests

We test the six TM implementations on three unnecessary abort tests. The *write-during-read-only* test (10) is presented in **Figure 4.7** (top). Word-based RSTM and TL2 fail this test while other TMs succeed. The reason for the success of TinySTM and SwissTM is because they both use the LSA validation extension mechanism (see **Section 2.4.4.1**) while word-based RSTM and TL2 do not. $\mathcal{E}$-STM would have aborted only if T_2 would also write x in addition to y otherwise it considers that T_1 is serialized after T_2 and passes the test.

Write-during-read-only test

```
1 Definitions:
2   x=0; y=0;
3 Transactions:
4   T1 := R(x), @L, R(y);
5   T2 := W(y,1);
6 Schedules:
7   S := T1@L, T2, T1;
8 Invariants:
9   [No-abort];
```

```
...
    [Th2:T2] S
    [Th2:T2] W(y)
    [Th2:T2] W(y,1)
    [Th2:T2] Try C
    [Th2:T2] C
[Th1:T1] R(y)
[Th1:T1] A
Invariant NO_ABORT fails.
```

Invisible-write test

```
1 Definitions:
2 x=0; y=0;
3 Transactions:
4  T1 := W(x), @L;
5  T2 := R(x);
6 Schedules:
7  S := T1@L, T2, T1;
8 Invariants:
9  [No-abort];
```

```
[Th1:T1] S
[Th1:T1] W(x)
[Th1:T1] W(x,14666)
[Th1:T1]:L
   [Th2:T2] S
   [Th2:T2] R(x)
   [Th2:T2] A
Invariant NO_ABORT fails.
```

Figure 4.7: (Top) Write-during-read-only test for which TL2 and word-based RSTM abort while other TMs commit (TMUNIT gives the same trace for TL2 and RSTM, represented on the right-hand side). **(Bottom)** Invisible-write test where TinySTM's ETL and WT variants as well as $\mathcal{E}$-STM fail to commit both transactions without aborting. Right-hand side shows the trace for TinySTM-ETL.

The *invisible-write* test (11) given in **Figure 4.7** (bottom) comes from the [69] (Fig. 1) and checks whether unnecessary aborts may occur when the write operation is made visible. TinySTM-ETL and $\mathcal{E}$-STM fail the *invisible-write* test because they lock the to-be-written address eagerly as soon as a write occurs leading to a conflict later on. TinySTM-WT also aborts since the write is made effective with the write operation of T_1.

```
1 Definitions:
2   arr[0..1]=0;
3 Transactions:
4   T1 := W(arr[0],1), @L;
5   T2 := W(arr[1],2);
6 Schedules:
7   S := T1@L, T2, T1;
8 Invariants:
9   [No-abort];
```

False-sharing test

```
[Th1:T1] S
[Th1:T1] W(arr[0])
[Th1:T1] W(arr[0],1)
[Th1:T1]:L
  [Th2:T2] S
  [Th2:T2] W(arr[1])
  [Th2:T2] A
Invariant NO_ABORT fails.
```

Figure 4.8: The false-sharing test for which SwissTM, $\mathcal{E}$-STM and word-based RSTM abort while the other TMs commit (the trace of SwissTM is given on the right-hand side).

The *false-sharing* test (12) described in **Figure 4.8** accesses consecutive memory locations. Having a lock protect multiple consecutive addresses can have the undesirable consequence of producing false sharing, i.e., two threads accessing distinct memory locations can conflict because they share the same lock. As shown by this test, SwissTM, $\mathcal{E}$-STM and word-based RSTM abort, meaning they use a common lock on (at least) two consecutive addresses. Unlike other STMs, they are subject to false sharing.

The last liveness test is the *virtual-world* test (13) illustrated in **Figure 4.9**. Virtual-world consistency is weaker than opacity (see **Section 2.4.4.1**) because it allows aborting transactions to have a view of the system that is different from the view of other transactions (committed or aborted). The *virtual-world* test indicates an execution in which a transaction T_1 has to abort to ensure opacity while T_1 could commit without violating virtual-world consistency. Hence, when considering virtual-world consistency definition the abort of T_1 is unnecessary. We can see that all considered TMs fail this test as they unnecessarily abort T_1. We are not aware of any TM library that could ensure virtual-world consistency without ensuring opacity, and the efficiency of such an implementation remains an open question.

4.5 Performance test specification with TMunit

Performance tests are generally longer than unit tests since they execute more complex specifications to measure the performance of a TM. More precisely, they use randomization

```
1  Definitions:
2   _SIZE = 3;
3   m[0 .. _SIZE] = 0;
4  Transactions:
5    T1 := R(m[0]),@L1,W(m[1]);
6    T2 := W(m[0]);
7    T3 := W(m[0]), W(m[2]);
8    T4 := R(m[0]),
       R(m[1]),@L2,R(m[2]);
9  Schedules:
10   S := T1@L1, T2, T4@L2, T3,T4,T1;
11 Invariants:
12   [T1:No-abort];
```

Virtual-world test

```
...
        [Th4:T4] R(m[2])
        [Th4:T4] A
        [Th4:T4] Terminates
[Th1:T1] W(m[1])
[Th1:T1] W(m[1],1640212158)
[Th1:T1] Try C
[Th1:T1] A
Invariant T1:NO_ABORT fails.
```

Figure 4.9: The virtual-world test indicating whether an unnecessary abort happens in transaction T1. All TMs fail this test because they all abort T1 as if committing it would violate safety (even though T4 has to abort). The trace obtained running the test with TL2 is given on the right.

#	Test name	TL2	TinySTM			RSTM		SwissTM	WSTM	$\mathcal{E}$-STM
			ETL	CTL	WT	word	object			
	Safety tests									
1	opacity	✓	✓	✓	✓	✓	✓	✓	×	✓
2	strict serializability	✓	✓	✓	✓	✓	✓	✓	✓	×
3	serializability	✓	✓	✓	✓	✓	✓	✓	✓	×
4	SLA	×	×	×	×	×	×	×	×	×
	Isolation tests									
5	publication issue	×	×	×	×	×	×	×	×	×
6	dirty-read	✓	✓	✓	×	✓	✓	✓	✓	✓
7	zombie-transaction	✓	✓	✓	✓	✓	✓	✓	✓	✓
8	interference	✓	✓	✓	✓	✓	✓	✓	×	✓
9	granularity	✓	✓	✓	✓	✓	×	✓	✓	✓
	Liveness tests									
10	write-during-read-only	×	✓	✓	✓	×	✓	✓	✓	✓
11	invisible-write	✓	×	✓	×	✓	✓	✓	✓	×
12	false-sharing	✓	✓	✓	✓	×	✓	×	✓	×
13	virtual-world	×	×	×	×	×	×	×	×	×

Table 4.1: Results of our semantic test-suite obtained with the six TMs. Failures are denoted by the cross '×' while successes are denoted by the check-mark '✓'.

and loops to test a large set of schedules. Note that these specifications do not define schedules, thus for performance tests threads run concurrently instead of one thread at a time (as forced by schedule definitions). Here, we present additional language features on

```
1  Properties:                                                    // global properties
2    RandomSeed = 1;                                            // use random seed for RNG
3    ReadOnlyHint = 1;                                          // tag read-only transactions
4    Timeout = 10*1000*1000;                                    // maximum test duration (us)
5
6  Definitions:                                                   // variables and constants
7    SIZE = 4096;                                               // size of the list (constant)
8    m[0 .. 2*SIZE+1] = 0;                                      // memory range for list nodes
9    NB = <1 .. SIZE>;                                  // random value (constant) in range 1...SIZE
10   T2:_f = 0;                                        // flag to alternate between adds and removes
11   T2:_v = 0;                                                 // position of last added value
12
13 Transactions:                                               // specification of transactions
14   T1 := {# k = [0 .. NB-1] : R(m[2*k]), R(m[2*k+1]) },    // search element
15         R(m[2*NB]) ;
16   T2 := {? [_f == 0] :                                                  // add/remove element
17           {# k = [0 .. NB-1] : R(m[2*k]), R(m[2*k+1]) },      // add element
18           R(m[2*NB]), W(m[2*NB-1]),
19           { _f = 1, _v = NB }
20         |
21           {# k = [0 .. _v-1] : R(m[2*k]), R(m[2*k+1]) },  // remove element
22           R(m[2*_v]), R(m[2*_v+1]),
23           W(m[2*_v-1]), W(m[2*_v]), W(m[2*_v+1]),
24           { _f = 0 }
25         } ;
26
27 Threads:                                                      // specification of threads
28   P1, P2 := < T1 : 80% | T2 : 20% >*;
```

Listing 4.3: Complete specification of the sorted linked list micro-benchmark.

a slightly more complex example that corresponds to the complete specification of a widely used micro-benchmark: a sorted linked list.

The resulting TMUNIT benchmark is 28 lines of code written in our language (see Listing 4.3) while it was originally more than 1000 lines of code written in C (as it is available in the TinySTM distribution). This is mainly due to the unnecessary specification of threads and statistics management that are automatically handled by TMUNIT. This simple TMUNIT version is experimentally compared to the original benchmark in **Section 4.6.1**.

4.5.1 Randomness and loops

To implement realistic performance tests, our language provides constructs such as random executions and loops. Randomness is provided by special constructs `<min..max>` that evaluate to an integer value chosen uniformly at random between `min` and `max` (inclusive).

This random expression notation can appear everywhere a number is expected. For instance in Listing 4.3, constant `NB` at Line 9 represents an arbitrary element of a linked list of size 4096. Note that "random constants" are evaluated once at the beginning of each transaction, i.e., they get a different, immutable value for every transaction execution.

Transactions may include loops that repeat a predetermined number of times and loops that execute until a conditions becomes true. The former type of loop is illustrated in Listing 4.3 Line 14, where transaction T_1 repeatedly reads addresses representing nodes in the linked list (each node has two data items: a value (read using `R(m[2*k])`) and a pointer to the next node (read using `R(m[2*k+1])`). This transaction mimics the search for a random element in a linked list, with a number of iterations determined by the random constant `NB`. The last operation correspond to the read of the searched value (or the first larger value in case it is not found).

Conditional execution is another important mechanism to specify realistic workloads. Our language supports a generalized form of if-then-else statement. Conditional expressions may depend on the state of variables and constants. For instance, in Listing 4.3, transaction T_2 uses a flag $\_f$ to alternatively add or remove an element. If the flag is 0 then a new element is added (Lines 17–19); otherwise the last inserted element is removed (Lines 21–24). This approach is used by linked list micro-benchmarks to maintain the size of the list almost constant during the whole experiment. Note that the reason there is a single write (`W(m[2*NB-1])` at Line 18) upon node insertion is that the new node is not shared until commit time; in contrast there are three writes upon removal (the writes at Line 23) because one needs to detect concurrent accesses to the removed node, which can be achieved by overwriting it. This specification closely mimics the behavior of a custom linked list micro-benchmark with the notable exception of the placement of the node data in memory (deterministic vs. unpredictable placement); yet, as we shall see in **Section 4.6.1**, this difference does not affect the results of performance tests.

4.5.2 Threads and transaction patterns

The *threads* section specifies the combination of transactions that will execute in the context of each thread. Unlike transactions, threads may have infinite length and are defined as patterns using a syntax close to regular expressions. Each thread that executes at runtime must be defined. By default, the benchmark will execute one instance of each thread but command-line parameters can be used to indicate which threads to start and their number of instances (threads are referred to by their name). Multiple thread names can share the same specification. A thread definition may include repetitions (fixed, random,

or unbounded), execution of one out of several transactions chosen at random with predetermined probabilities, sequences and grouping of transactions. As an example, Listing 4.3 presents two threads P_1 and P_2 that both execute transaction T_1 with probability 80% and transaction T_2 with probability 20% in an endless loop. Such experiments are interrupted after a specified timeout or with a signal.

4.6 Analyzing TM commit performance

In this section, we present the performance results obtained with TMUNIT on a 4-Quad-Core AMD Opteron Processor 8354 running at 2.2Ghz (16 cores). We tested TinySTM, TL2, word-based RSTM, and SwissTM. We did not include WSTM in the performance graphs because of a problem encountered with porting the inline assembly code to the target architecture. This issue did not prevent us from executing the performance-insensitive semantic tests of **Section 4.4** on another architecture.

In the upcoming sections, we first depict that the performance of an existing benchmark code written by hand and the code generated by TMUNIT for the same benchmark in the proposed language are close enough to identify performance related issues in benchmarks. Next, we investigate TM performance in extreme scenarios and under different contention manager policies.

4.6.1 Validating TMunit code generation

Here, we compare an existing micro-benchmark, a linked list implementation of an integer set, to its corresponding TMUNIT specification (see Listing 4.3). The benchmark initially inserts a given number of elements in the linked list. Then, each thread starts executing and performs a series of *search* and *update* transactions (alternating inserts and removals to maintain the size of the list roughly unchanged during the whole execution) according to a given probability. A common problem in performance tests is the overhead introduced by the evaluation framework. As mentioned in **Section 4.3.1.1** and to avoid this overhead, TMUNIT can translate the specification into C code to be compiled before execution. Here, we motivate this choice by comparing the results obtained using the generated technique against the results of the existing benchmark written in C code "by hand", denoted by native.

Figure 4.10 presents the throughput (top) and the abort rate (bottom) of the linked list with an update transaction probability of 20% and with a size of 4096 elements. A first observation is that the generated version presents results (commit and abort rates) very

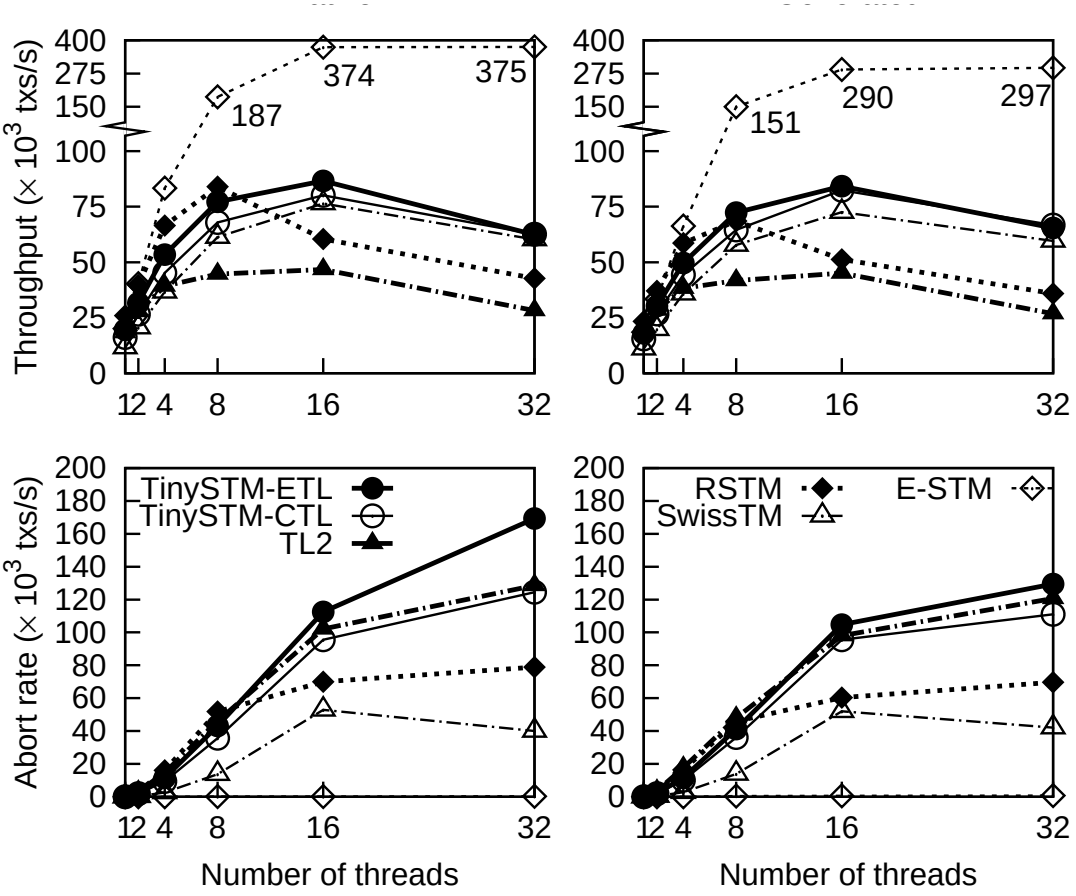


Figure 4.10: Performance comparison of the native intset benchmark **(left)** and the intset TMUNIT specification **(right)**. The commit rate is shown above and the abort rate below. Note that in order to distinguish curves other than $\mathcal{E}$-STM curve, the graphs for commit rates have different scales above and below 100 thousand tx/s.

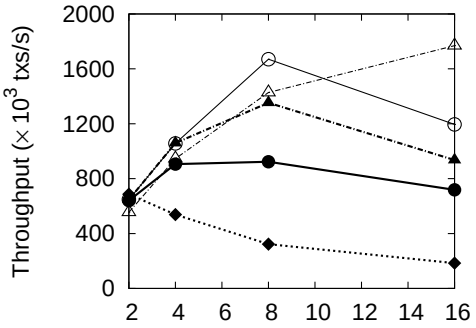
similar to the results of the hand-crafted benchmark. We have also executed the workload on the interpreted automaton and observed a 65% decrease in throughput with respect to the throughput of hand-crafted benchmark. This clearly motivates the need for the generated automaton.

$\mathcal{E}$-STM has also been tested here because the integer set operations can benefit from the elastic transactions. Actually, **Figure 4.10** confirms that $\mathcal{E}$-STM performs best. $\mathcal{E}$-STM performance is followed by TinySTM-ETL, TinySTM-CTL and SwissTM, respectively. An interesting observation is that word-based RSTM throughput gets significantly lower than other STMs when the number of threads increases. The authors of RSTM have hinted as a possible reason the non-scalable libstdc++ exception mechanism used to trigger a roll-back

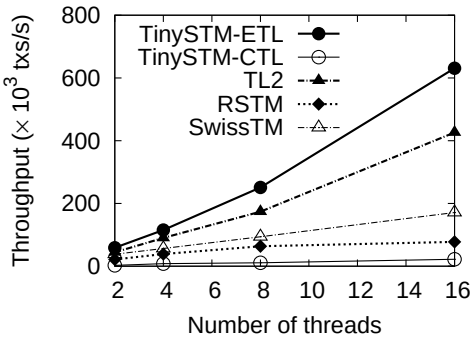
upon abort. The reason why TL2's performance is lower than TinySTM can be explained by the differences in timestamp management: TL2 does not perform dynamic snapshot extension (see **Section A.4.1.4** for details). We can clearly see that $\mathcal{E}$-STM is not subject to contention as opposed to other STMs (even for more threads than processors there is no performance drop) as it uses elastic transactions that ensure the minimum guarantees to satisfy the correctness of integer set operation.

4.6.2 Extreme scenarios

A powerful feature of TMUNIT is that it allows to quickly design workload to test specific scenarios or seldom exercised functionalities of the TMs. Here, we investigate the response of TMs in the face of extreme workloads that highlight the differences in TM designs. $\mathcal{E}$-STM is not tested here as there is no need for using elastic transactions in these tests and its normal transactions would have the same performance as TinySTM-ETL. These tests demonstrate that TM performance relies tightly on the workload used.




```
Definitions:
  SIZE=1024;
  arr[1..SIZE+100]=0;
  AD=<1..SIZE>;
Transactions:
  T := W(arr[AD]),
   {# k=[1..100]; R(arr[AD+k])};
Threads:
  P1, ..., P16 := T*
```




```
Definitions:
  N=1000;
  arr[1..16*N]=0;
Transactions:
  T1:={#k=[1..N]; W(arr[k])};
  T2:={#k=[N+1..N*2];W(arr[k])};
Threads:
  P1:=T1*;
  P2:=T2*; ...
```

Figure 4.11: (Top) Performance tests with write-once-read-many transactions and **(bottom)** with disjoint-writes transactions.

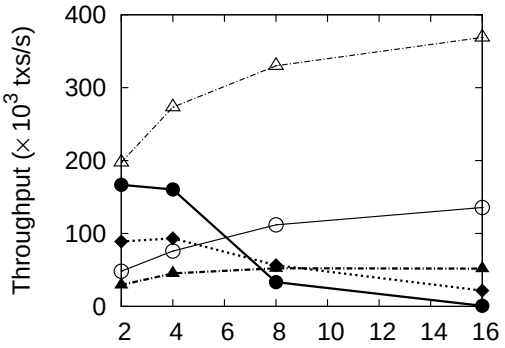
In **Figure 4.11** (top), all threads execute a transaction composed of one write followed by a series of reads. This performance test is made such that the reads executed by one

thread may access the same address as the one already written by another thread. This read-after-write pattern is expected to emphasize the circumstances in which commit-time locking is better suited than encounter-time locking. As expected, TinySTM-CTL and TL2 present better throughput than TinySTM-ETL. SwissTM takes advantage of the extra version that can be accessed while a memory location is locked. Word-based RSTM executes slower than other STMs, despite using a commit-time locking strategy, again due to the scalability problems of libstdc++ that was pointed out in **Section 4.6.1**.

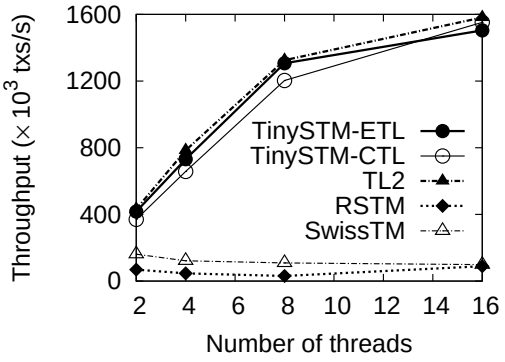
Figure 4.11 (bottom) shows a scenario that highlights the cost of writes without contention. Threads perform series of writes to disjoint memory regions, which implies that there are no conflicts. One can observe that TinySTM-ETL scales best. TL2 suffers from using commit-time locking: it needs to check for every write whether the memory location has already been written by the same transaction, which requires a traversal of the write set. To limit the cost of this check, TL2 uses bloom filters but the overhead is still not negligible. SwissTM performs a double locking of the entries in the write set and, hence, is penalized when transactions write many memory locations. RSTM again shows scalability problems due to libstdc++. Finally, TinySTM-CTL suffers from the same problem as TL2 but was executed without the bloom filter optimization.

In **Figure 4.12** (top), all threads execute a transaction composed of multiple write operations followed by multiple reads. In this performance test, operations executed by each transaction access consecutive addresses and generate much contention—a scenario where TM is typically less efficient than locking and contention management has great importance. Interestingly, SwissTM scales significantly better than the other TMs. To better understand the reason for this behavior, we have also experimented with TinySTM-ETL when (1) allowing transactions to read the previous version of locked memory locations by peeking into the write set of the lock owner, as in multi-version LSA (TinySTM-1v), and (2) activating TinySTM's built-in "priority" contention manager (TinySTM-CM). Each of these mechanisms provide noticeable improvement on this extreme workload. The remaining optimizations consist in choosing the right tuning parameters, as was studied in [57]. In particular, having each lock protect a set of consecutive memory locations (typically the size of a cache line) improves the performance because it causes fewer cache invalidations and reduces the number of compare-and-swap operations, as the lock needs to be acquired only once for several consecutive memory addresses.

To observe the influence of having a lock to protect multiple consecutive addresses (i.e., false-sharing as mentioned in **Section 4.4.2**), we have created a workload in which threads can only conflict if there is false sharing. In **Figure 4.12** (bottom) each thread reads a series of consecutive addresses and writes a single, distinct address. To avoid




```
1  Definitions:
2    SIZE=1024;
3    arr[1..SIZE+100]=0;
4    AD1=<1..SIZE>;
5    AD2=<1..SIZE>;
6  Transactions:
7    T :=
8     {# k=[1..100]; W(arr[AD1+k])},
9     {# k=[1..100]; R(arr[AD2+k])};
10 Threads:
11   P1, ..., P16 := T*
```



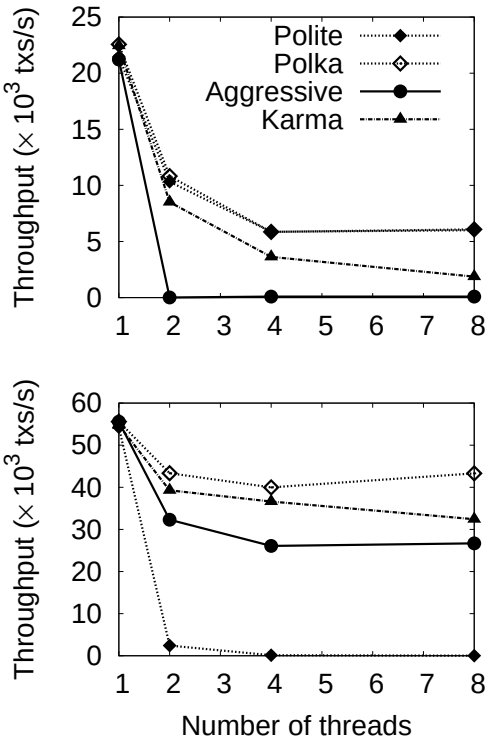

```
1  Definitions:
2    SIZE=256;
3    arr[1..SIZE]=0;
4  Transactions:
5    T1 := {#k=[0..15] {#j=[1..15];
6     R(arr[k*16+j])}}; W(arr[0]);
7    T2 := {#k=[0..15] {#j=[1..15];
8     R(arr[k*16+j])}}; W(arr[16]);
9  Threads:
10   P1 := T1*;
11   P2 := T2*; ...
```

Figure 4.12: (Top) Performance tests with write-many-read-many transactions and **(bottom)** with false-sharing transactions.

undesirable memory effects, each write accesses an address on a distinct cache line next to addresses read by the other threads. This example may trigger false sharing: upon writing, threads acquire more addresses than necessary, including addresses that have been read by concurrent transactions, and produce unnecessary aborts. Indeed, we observe that both SwissTM and word-based RSTM are subject to false sharing, while other implementations are not.

4.6.3 Testing contention manager policy

In this section, we present the performance impact of contention management policy. A Contention Manager (CM) is a module that indicates how to solve conflicts depending on which transactions should take resolution actions and which should continue. The reader is referred to **Section A.5.2** for more information about contention management and different contention managers.




```
Definitions:
  SIZE = 256;
  m[0..SIZE-1] = 0;
Transactions:
  T :=
    {# k=[0..SIZE-1]:
    R(m[k]), W(m[k]) };
Threads:
Th := T*;
```

```
Definitions:
 SIZE=1024;
 arr[1..SIZE+100]= 0;
 AD1=<1..SIZE>;
 AD2=<1..SIZE>;
Transactions:
 T :=
   {# k=[1..100]: W(arr[AD1+k])},
   {# k=[1..100]: R(arr[AD2+k])};
Threads:
 P1, ..., P16 := T*;
```

Figure 4.13: Performance variation of contention managers depending on the workload. **(Top)** long read-write workload. **(Bottom)** write-many-read-many workload.

4.6.3.1 Performance variations

Here we test several CMs we could find in the literature. To this end, we generated a code compatible with the object-based version of RSTM [38] where each access to a memory word was treated as an access to an object. RSTM is implemented so that it can be easily parameterized to run with one of these CMs. Additionally, for the experiments presented in this section, RSTM is configured to use eager acquirement strategy (whereas in the previous sections it is used with its default configuration where lazy acquirement strategy is used). The results we obtained on an 8-core Intel Xeon CPU X5365 running at 3.00GHz are depicted in **Figure 4.13**.

In the long read-write workload of **Figure 4.13** (top) every memory location accessed is a source of conflict and all transactions access the locations in the same order. So a backing off policy, as in Polite and Polka, provides a first-come-first-commit kind of ordering between transactions and allow progress. Similarly, but less efficiently, prioritizing transactions according to past work done as in Karma also ensures progress. In contrast, the Aggressive policy results in continuous aborts and cannot provide any progress.

In the write-many-read-many workload depicted in **Figure 4.13** (bottom), transactions forcing other transactions to backoff may conflict later on and backoff in turn, hence blocking for a while. Since the workload is designed to be highly contended, such situations are most likely, thus a simple backoff policy like Polite slows performance down. In contrast, Polka, which incorporates an effort-based priority scheme as in Karma, copes with this problem and performs well. Although simple, Aggressive also avoids blocking in this case. As with the previous workload Karma performs reasonably well but still not achieving the best performance among other CMs.

4.6.3.2 Livelocks

As claimed in [170], the Passive CM may suffer livelocks. We experiment the existence of this issue by providing a dedicated workload that can be reproduced easily on other TM/CM using TMUNIT.

The *bank-benchmark* test, specified in **Figure 4.14** (top), uses a classical scenario where one thread computes the sum of the balances of 1024 to 8192 accounts of a bank (long read-only transaction) while the other threads concurrently perform transfers (short update transactions). In TM designs with invisible reads and no fair contention management, updates conflicting with long read-only transactions may lead to failed validation. The problem is illustrated in **Figure 4.14** (middle), where short *transfer* transactions prevent the long *balance* transactions from committing. **Figure 4.14** (bottom left) shows the result when using Passive CM in TinySTM while **Figure 4.14** (bottom right) presents the result with the built-in priority-based contention manager. With Passive CM, throughput drops to 0 when the number of threads performing transfers reaches 3. We inspected the corresponding TMUNIT trace to make sure that the cause was the problem described in **Figure 4.14** (middle). As expected, when using the Retry CM, the throughput is almost independent of the number of threads performing transfers. Unlike Passive CM, Retry CM ensures progress.

To conclude, we have experimentally demonstrated the initial thoughts under which some CM are more progress-friendly than others. Thanks to TMUNIT these scenarios are easily reproducible for further testings and may outline similar liveness issues in future implementations.

```
1  Definitions:                                          // variables and constants
2  % NB = 8192;                                        // number of accounts (constant)
3  % a[1 .. NB] = 0;                                     // memory range for accounts
4  % SRC = <1 .. NB>;                                 // random value (constant) in range 1...NB
5  % DST = <1 .. NB>;                                 // random value (constant) in range 1...NB
6  %
7  %Transactions:                                      // specification of transactions
8  % T1 := R(a[SRC]), R(a[DST]), W(a[SRC]), W(a[DST]) ;                  // transfer
9  % T2 := {# k = [1 .. NB] : R(a[k]) } ;                             // compute balance
10 %
11 %Threads:                                           // specification of threads
12 % P1 := T2*;
13 % P2, P3, P4, P5, P6, P7, P8 := T1*;
```

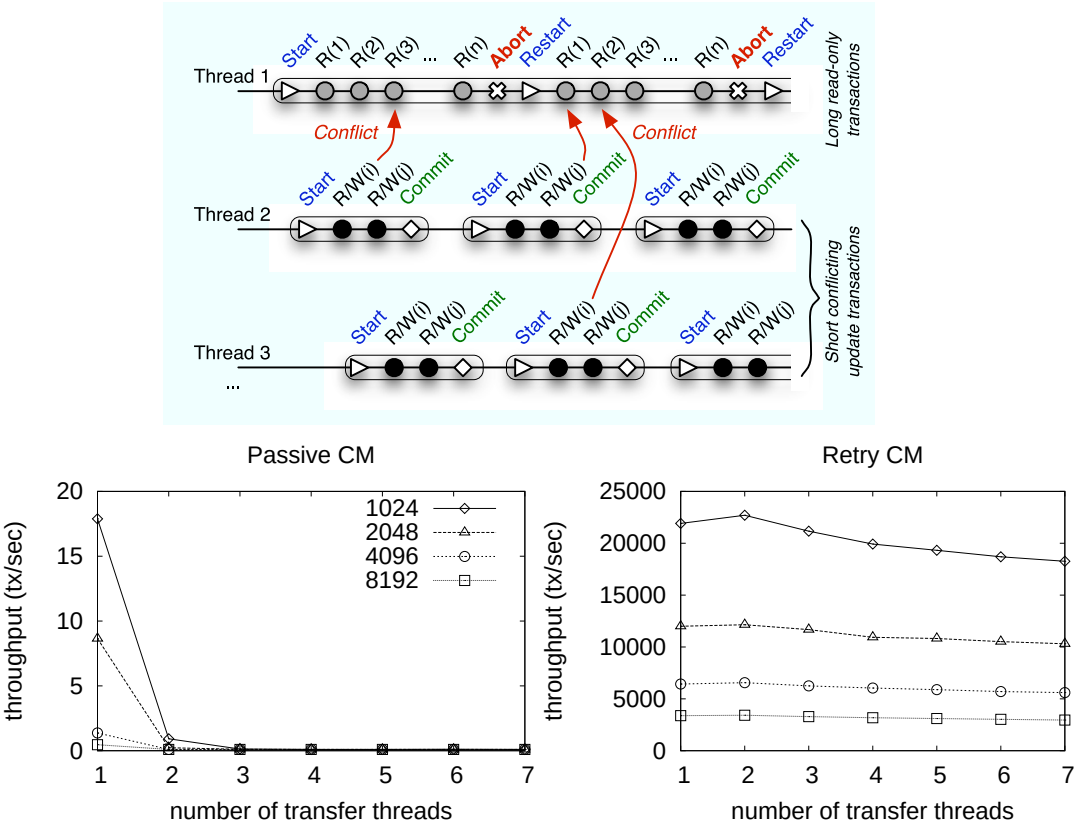


Figure 4.14: **(top)** Specification of the bank benchmark that exhibits lack of progress (for NB=8192 accounts and 8 threads). **(middle)** Illustration of bank account benchmark suffering from the lack of progress when Passive CM is used. **(bottom)** The throughput for Passive and Retry CMs (for 1024 to 8192 bank accounts) appear on the bottom left- and right-hand side respectively.

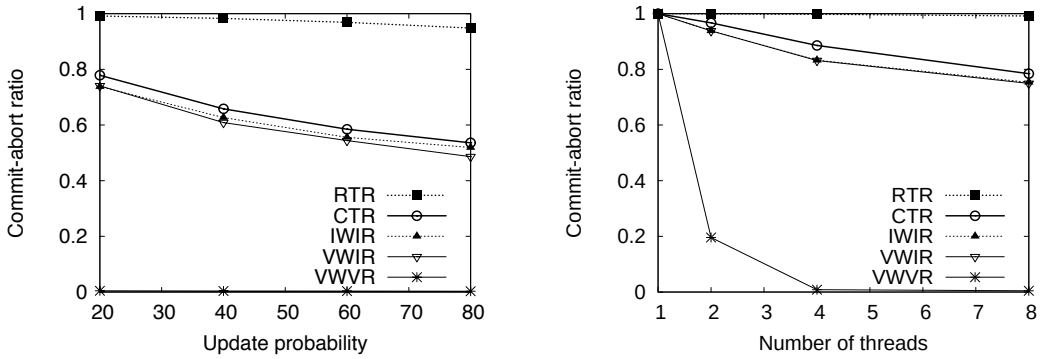


Figure 4.15: Comparison of average commit-abort ratio of the various designs on a 256 element linked list: (left) with 8 threads as a function of the update probability; (right) with a 20% update probability as a function of the number of threads.

4.7 Experimental validation of input acceptance bounds

The design classification presented in **Section 3.5** relies on upper-bounds of input acceptance. Since we ignore how tight these upper-bounds are, some design lower-bounds may not reflect the obtained comparison. To make sure that we did not omit important classes of input patterns, we validated experimentally our theoretical comparison of TM designs using TMunit. In order to perform the validation experimentally we have implemented all the VWVR, VWIR, IWIR, CTR and RTR designs and tested them on the same benchmark we have used in **Section 4.6.1**(integer set linked-list benchmark for a linked-list of size 256).

For all experiments, the benchmark is executed on an 8-core Intel Xeon machine. The first series of experiments, presented in **Figure 4.15** (left), compare the input acceptance under high contention on 8 threads. At first glance, the commit-abort ratio decreases as the update probability increases. As one can expect, a larger update probability increases the probability that two transactions conflict, thus the number of aborts in all designs. Second, we can see that the higher a design in the hierarchy of **Figure 3.5**, the higher its commit-abort ratio (thus the higher its input acceptance). This clearly confirms our theoretical results. The commit-abort ratio of design VWVR is close to zero because VWVR aborts preferably small transactions each time a write-after-read pattern occurs. More surprisingly, the update probability affects much less the acceptance of the RTR design than any other design. That is, additional contention results mostly in conflicts that are unnecessary to resolve.

We have performed a second series of experiments to analyze the scalability of each design. These experiments are similar to the first ones with a fixed update probability of 20% and a variable number of threads. The results are depicted in **Figure 4.15** (right). This figure clearly illustrates that the acceptance of RTR design scales well with the number of threads, while the other designs have a decreasing acceptance as the number of threads increases. This result indicates how well RTR design copes with conflicts that span transactions of multiple threads.

4.8 Summary

In this chapter, we have introduced our domain specific language and the tool TMUNIT that enables the use of this language. We have described the syntax and semantics of the language and have explained how the language can be used to specify unit and performance tests. We have seen how to describe schedules as part of unit tests, to enable deterministic testing of the specified test case. We have also seen the language elements that allow more complex behaviors such as randomization, loop execution and conditional execution, used mainly to describe performance tests. Then we have introduced our test suite composed of both semantic and performance tests. As it can be noticed the content of the test suite allows the testing of diverse aspects of TM design encompassing both semantic and performance issues.

As a consequence of applying our test suite on 6 different STMs we have *(i)* demonstrated which TMs fail on pathological scenarios related to the use of non-transactional code [130]; *(ii)* shown that WSTM violates opacity if no explicit validation is performed by the programmer, and *(iii)* illustrated a livelock execution due to the use of Passive CM [170] *(iv)* briefly compared some well known contention managers on highly contended workloads *(v)* depicted how the performance of TM implementations vary under some canonical test cases. The latter test cases are useful especially if the access pattern of a workload fits into one of these canonical test cases. In such cases, we can predict the application performance of a TM using the results of these tests. Since performance of TM implementations can be contrasted on the same workload, this can also help programmers to choose the best TM that fits the application purposes. Of course the same applies for workloads, that can be generated by the users and to represent some other data access patterns.

The current release of TMUNIT including its companion test suite and documentation is available online `http://www.tmware.org/tmunit`.

Chapter 5

Language Extension API

Although very important in providing the transaction abstraction as a programming construct, a TM (i.e., the runtime support for transactional memory) per se, is not enough to offer the complete programming language support required for TM-based programming. Up to recently the transaction support at the language level has been mostly provided in terms of software libraries. However, providing the transaction abstraction using TM libraries in multi-threaded sofware hampers the development process since:

- it is cumbersome and time consuming (the programmer needs to specify all the transactional accesses through library calls),
- code is hard to read (simple memory accesses that need to be transactional become all library calls),
- it is not portable (the written software stays specific to the TM library for which it is written).

A more convenient way to code transaction abstraction in programs is to have transaction constructs available as programming language constructs. With such a language support all the above issues can be addressed since the mapping between the behavioral representation and the actual library based transactional code is done automatically by compile-time and/or run-time tools for the programmer.

The objective of this chapter is describing a collection of transactional language constructs for the Java language and present it as a language extension specification (can also be considered as an API) to a programmer who would like to use transaction abstraction in his/her program. The ideal syntax to express a transaction as part of a programming language would be a block of code (we call such a block *transaction block*) in which the program statements constituting the block execute according to a specified transactional behavior. The syntax of such an ideal transaction block is simple: it encloses code as would

any other traditional block (e.g., function body, `if` statement body, loop body etc.). Despite its simple syntax, the semantics of a transaction block do not always fit the semantics of the statements that could be enclosed within, introducing difficulties of interoperability of the ideal transaction block with other language constructs. For example, the use of irrevocable language constructs (such as I/O accesses, systems calls, synchronization primitives) disallows the rollback-and-restart behavior of transaction constructs. Another example is the necessity to support different types of exceptions in a transaction: some exceptions can be handled only if the transaction commits its changes before raising the exception, while for some others it is better to rollback the transaction in order to preserve application consistency.

In short, the ideal transaction block is not flexible enough to meet all the necessary semantic requirements that are expected from a language support for transactional behavior. A complete transactional language support requires a combination of language constructs together with a transaction block.

The specification presented in this chapter describes the syntax and the corresponding semantics of such a collection of language constructs and proposes those constructs as a language extension. The specification of a similar extension for C/C++ languages has been performed recently by the Transactional Memory Specification Drafting Group (composed of members from Intel, Sun and IBM) [12]. However, there is no standardization process for Java TM extensions like for C/C++. In this sense, the language constructs proposed in this chapter constitute the first transactional language extension for Java.

In defining our language specification we have loosely followed the language constructs proposed in C/C++ and adapted them to the Java language[1]. Meanwhile, the C/C++ specification focuses on the synchronization aspect of the transaction abstraction and tries to cover the language constructs required to support this aspect. Our Java language specification goes beyond the synchronization aspect and aims at providing language constructs supporting additional aspects the transaction abstraction can offer. To that end, the presented specification contributes to the description of transactional language constructs by including transactional control flow and exception handling constructs that are not present in the C/C++ specification.

Roadmap: We start by detailing the fundamental transactional constructs in **Section 5.1**. We then explain the types of transactional guarantees proposed by our Java language

[1]Since the C/C++ specification serves as a reference for these constructs, a summary of this specification is provided as an appendix (**Appendix C**) to the present dissertation. The constructs proposed in the current specification are always contrasted with the C++ specification by referencing the corresponding section in **Appendix C**.

specifications in Section **Section 5.2**. **Section 5.3** illustrates the requirements for using functions within transactional code. We describe the support for control flow and nesting respectively in **Section 5.4** and **Section 5.5**. We finish by explaining the exception handling support in **Section 5.6**.

5.1 Fundamental transactional constructs

Our Java language extension specification allows only a single language construct to be executed in a transaction: *compound statements*. We call compound statements that exhibit transactional behavior **transaction statements**. The keyword that allows performing the execution of a compound statement as a transaction statement is the keyword __transaction. Hence, the syntax for a transaction statement is[1]:

`__transaction` *compound-statement*

We do not specifically provide a transactional behavior for expressions or for functions as in C++ language (see **Section C.2** for such language constructs). This is a deliberate choice based on the following reasoning:

- Supporting transactional behavior for mere expressions is nothing but a syntactic sugar for transaction statements enclosing simple language statements (see **Section C.2** to observe how a transaction expression can be written in terms of a transaction statement).
- Functions can be used both in transactional and non-transactional code even in the same program. The fact that a function is to be executed in a transaction is mainly decided by the caller of the function and the caller code does this performing the call in a transaction statement (which may enclose also other statements). Hence, it is perfectly possible for a compiler to decide whether a function should support transactional behavior or not. The approach we take in our language extension specification is to simplify the use of language extention and exempt the programmer from the need to indicate whether a function is (or can be) transactional.

[1]Hereafter when we refer to transaction block, it would correspond to the syntactic code block that follows the `__transaction` keyword (we interchangeably call the same block either "transaction block" or "`__transaction` block"). The transaction statement can have syntactically more components and we use the term "transaction statement" to refer to all these components.

5.2 Types of transactional guarantees

The use of different language statements may require different transactional behavior and hence a transactional language extension must offer the possibility to choose between several transactional behaviors. With our language specification the programmer expresses his/her choice of transactional behavior with a parameter to the `__transaction` keyword. The transactional guarantees provided to the programmer and their syntax are as follows:

- **Atomic transaction:** An atomic transaction executes according to basic transactional behavior (see **Section 2.5.4.1**) as long as it does not execute irrevocable code. In the case where the transaction executes irrevocable code, its behavior follows irrevocable transactional behavior (see **Section 2.5.4.2**). Atomic transaction guarantee is the default for the proposed language extension. The syntax for declaring an atomic transaction has two equivalent forms which are as follows:

  `__transaction` *compound-statement*, or
  `__transaction(atomic)` *compound-statement*

  When compared to the C++ language extension, our atomic transaction guarantee corresponds to the relaxed transaction guarantee of C++ specification (see **Section C.3**). We find this guarantee simpler to use since the programmer does not need to reason whether there will be an irrevocable code execution inside the transaction or not. However, a programmer willing to use an atomic transaction that has the potential of performing irrevocable execution is responsible for indicating what code is irrevocable within the transaction block. An atomic transaction can detect that it is executing an irrevocable code if *(i)* it has a nested irrevocable transaction (see next transactional guarantee for irrevocable transaction), or *(ii)* it is executing a function marked with the `@Irrevocable` annotation (see **Section 5.3**). A programmer should use one of these two irrevocable behavior constructs to indicate potential irrevocable behavior. Hence, the programmer should enclose irrevocable statements either in irrevocable transactions or in member functions which they mark with `@Irrevocable` annotation. A failure to do so will result in inconsistencies due to irrevocable statement executions.
- **Irrevocable transaction:** An irrevocable transaction executes only according to irrevocable transactional behaviour (see **Section 2.5.4.2** for details), i.e., the transaction is not allowed to roll back. The syntax for declaring an irrevocable transaction is:

  `__transaction(irrevocable)` *compound-statement*

With the irrevocable transaction guarantee we provide the programmer more control on the transactional behavior (rather than just the default behavior). The programmer may like to use this guarantee mainly for two purposes: (*i*) if the defined transaction statement encloses statements that are irrevocable, or (*ii*) for speeding up transaction execution by giving a hint to the compiler right at the beginning of the transaction (it may take a while for an atomic transaction to switch from basic transactional behavior to irrevocable transactional behavior when the necessity occurs).

- **Elastic transaction:** An elastic transaction executes according to elastic opacity guarantee instead of opacity guarantee (i.e., basic transactional behavior). In other words, the atomicity property of an elastic transactions is such that, the transaction as a whole does not need to appear as atomic, whereas all couples of consecutive operations or the operations delimited by write operations in a transaction need to appear as atomic (see **Section 2.4.4.1** for details). The syntax for declaring an elastic transaction is:

  `__transaction(elastic)` *compound-statement*

  Elastic transactions are especially useful for providing efficiently executing transaction blocks. However, the content of the transaction should allow more concurrency than regular opacity guarantee requires (refer to **Section 2.4.4.1** for the concurrency requirements of a transactional code under which elastic opacity can be safely used). As long as these requirements are met, the programmer can think of using elastic transactions to obtain faster transactional executions.

5.3 Function usage in transactional code

The Java language extension specification simplifies the usage of functions in transaction statements. The programmer is exempted from the task of explicitly assigning attributes to functions to express whether the function is to be used in transactional code or not (as it is in the C++ specification **Section C.4**). This task is delegated to the language extension support. However, if a user defined class function includes irrevocable code, this should be indicated to the compiler for the correct execution of a potential call to the function from transactional code. Indicating that a function contains irrevocable code is done by adding the `@Irrevocable` annotation to the function signature as follows:

```
@Irrevocable void f()
```

In case of class inheritance, member functions preserve their irrevocable semantics except for overridden functions or interface member function implementations (this is because the overridden or implemented functions do not need to perform the required functionality using irrevocable code).

5.4 Constructs for control flow in transactions

The control flow is an important feature of a programming language. The transactional nature of transaction blocks has implications on the behaviors of regular control flow keywords and the exception throw that already exist in the Java language. Additionally, being part of the language, the transaction abstraction introduces new control flow behaviors tied to the transactional behavior. We, hence, present three types of control flow constructs related to transactions aiming to satisfy all these requirements: regular, transactional and exceptional control flow.

5.4.1 Regular control flow constructs

The semantics for usual control flow statements are kept as close as possible to their original semantics. The `break`, `return` and `continue` statements can be used to transfer control out of a transaction while `switch` statements must not be used to transfer control into a transaction statement. With these requirements our language specification follows closely the C++ specification (see **Section C.5.1**).

5.4.2 Transactional control flow constructs

The Java language extension tries to take advantage of transactional behaviour for the use of programmer in different possible ways. With this objective it proposes two types of control flow constructs: *alternative execution paths*, and novel transactional control flow keywords.

5.4.2.1 Alternative execution paths

The Java language extension specification introduces a language construct, which does not exist in the C++ specification, to enrich transactional control flow: **alternative execution paths**. An alternative execution path can be considered to correspond to a `case` statement of a `switch-case` statement that resides in a transaction statement. The main difference is that an alternative execution path comes with an additional transactional semantics: at any point inside an alternative execution path the modifications performed by the enclosing

transaction statement can be rolled back and the transaction statement can be re-executed using another alternative execution path.

Alternative execution paths are useful mainly in two kinds of situations. The first situation is when there are alternative ways to perform some actions provided in the transaction statement (e.g., in graceful degradation where a specific service can be given with degraded functionalities and each degradation is represented as an alternative). In such a case, each of the alternatives correspond to a different alternative execution path in the transaction statement. If one alternative cannot be completed (e.g., due to some error, exception or a nonfulfillment of a programmer defined condition) re-execution of the transaction statement using another alternative can be performed by rolling back the effects of the uncompleted alternative. The second situation where alternative execution paths are useful is speculative execution. If a thread is required to execute one task among a bunch of speculative tasks and needs to switch between one task to the other upon misspeculation, alternative execution paths propose a natural way to provide this behavior. The transactional nature of alternative execution paths matches the requirements speculative task execution upon misspeculation detection: when a misspeculation is detected the effect of the current speculative task is to be rolled back before executing another speculative task.

The syntax for alternative execution paths introduces two new keywords to the language; `either` and `or`. Each of the keywords express different code blocks in which an alternative is specified. The block starting with the `either` keyword corresponds to the very first alternative. Each of the code blocks starting with an `or` keyword corresponds to one of the alternatives that are after the first alternative. The syntax of alternative execution path language construct is presented in **Figure 5.1**.

5.4.2.2 Transactional control keywords

The Java language extension introduces transactional control keywords to give the programmer control on transaction execution. Three new keywords are introduced for this purpose: `__transaction_retry`, `__transaction_next`, and `__transaction_cancel`.

The `__transaction_cancel` keyword (i.e., the cancel statement) is the same as in the C++ specification (see in **Section C.5.2**) and is used to abort a transaction statement (i.e., cancel all the effects of the transaction statement) upon a serious error where the software should stop execution immediately. Other keywords give more control to the programmer in determining the control flow of a transaction. `__transaction_retry` aborts the transaction statement and re-executes it from the beginning. It is useful in handling temporarily raised exceptions or situations that are generated due to temporary conditions such as data

```
1  // Statements preceding the transaction statement
2  ...
3  __transaction{                                          // transaction statement
4  ...                                                     //*** Code Region A ***//
5        either{                                     // Code for the 1st alternative
6                                                          //*** Code Region B ***//
7       }
8       or{                                          // Code for the 2nd alternative
9                                                          //*** Code Region C ***//
10      }
11      ...
12      or{                                         // Code for the (n)th alternative
13
14      }
15 ...
16 }
17 // Statements following the transaction statement
18 ...                                                     //*** Code Region D ***//
```

Figure 5.1: The syntax of the alternative execution path construct. The construct is composed of the `either` and `or` blocks (lines 5-14).

races. The `__transaction_next` is similar to `__transaction_retry` but re-executes the transaction statement with the next alternative execution path. It is mainly useful within alternative execution paths especially if used for the purpose of graceful degradation or speculative execution.

These keywords are meaningful only in transaction statements, but their semantics inside or outside an alternative execution path may have slight differences. The semantics of each of the keywords according to their location in a transaction statement are described in **Table 5.1**.

To concretize the behavior associated to each the keyword, example usage of the keywords over the code regions marked on **Figure 5.1** is shown by **Table 5.2**.

5.4.2.3 Exceptional control flow constructs

The usual mechanism to control the flow of an application that raised an exception is to use a `try-catch` block. With the use of transactions this mechanism does not change. The specification, however, defines the way a transaction throws an exception and proposes two types of exceptional throw behavior in transactions (exactly as in **Section C.5.3** of the C++ specification): *commit-and-throw* and *abort-and-throw*. Both behaviors cause a transaction to throw the desired exception. However, the behaviors differ in the visibility

Keyword	Only in transaction statement	In alternative execution path
__transaction_retry	Rollback and re-execution of the transaction statement	Rollback and re-execution of the transaction statement using the same alternative execution path
__transaction_next	Rollback and passing of the control to code following the transaction statement	Rollback and re-execution of the transaction statement using the next alternative execution path[1]
__transaction_cancel	Rollback and passing of the control to code following the transaction statement	Rollback and passing of the control to code following the transaction statement

Table 5.1: The semantics of transactional control flow keywords proposed by our Java language extension according to their location in a transaction statement.

Keyword	Behavior upon keyword execution in region A	Behavior upon keyword execution in region B
__transaction_retry	Rollback and re-execute region A	Rollback and execute code in region A and then B
__transaction_next	Rollback and execute code in region D	Rollback and execute code in region A and then C
__transaction_cancel	Rollback and execute code in region D	Rollback and execute code in region D

Table 5.2: Example illustrating the semantics of transactional control flow keywords using the code sample in **Figure 5.1**.

of the effects of the transaction when the exception is thrown out of the transaction. With commit-and-throw behavior the effects of the transaction up to the point where the exception is raised are made visible to other threads with a commit before throwing the exception. In other words, commit-and-throw allows partial execution of transactions. The abort-and-throw behavior, however, rolls back all the effects of the transaction and only then throws the exception.

[1]The selection of the next alternative is done in a round-robin manner, i.e., if the `__transaction_next` keyword appear in the last `or` block (i.e., the very last alternative), in the next re-execution of the transaction block, the very first alternative enclosed in the `either` block will be chosen to be executed.

Since commit-and-throw behavior allows the partial effects of a transaction to be visible to other threads at the time an exception is raised, the atomicity guarantee of the transaction will be violated upon an exception. Abort-and-throw behavior avoids such a problem by throwing the exception only after aborting the transaction. However, since with abort-and-throw the exception is thrown *after* the transaction is aborted, the exception object cannot carry information about the actions performed in the transaction and hence exceptions that can be thrown inside a transaction are limited: *only a single type of exception*, `CancelException`, *can be raised for abort-and-throw semantics*. The programmer, however, can give the class of a valid exception class as a parameter to the `CancelException` as a means to convey the reason of the exception outside the transaction.

The commit-and-throw behavior is provided by the existing Java `throw` statement (as in **Section C.5.3** of the C++ specification), hence the syntax for commit-and-throw is simply:

`throw` *throw-expression*

To specify the abort-and-throw behavior, the `__transaction_cancel` keyword should be prepended to the existing Java `throw` statement. However, since only `CancelException` can be thrown with abort-and-throw its syntax becomes:

`__transaction_cancel throw CancelException(`*ExceptionClass*`)`

Apart from the exception throw behavior, transaction statements do not possess any exception specification in the Java language specification as opposed to C++ specification (see **Section C.5.3**).

5.5 Transactional nesting support

5.5.1 Basic nesting

Syntactically, the nesting support in the specification is of two types: *lexical* and *dynamic*. **Lexical nesting** is where a transaction statement takes place inside transaction statements as code. **Dynamic nesting** is the one where the nested transaction statement is not directly visible inside the outer transaction statement, however, one of the statements inside the outer transaction statement actually causes another transaction to execute. An example of dynamic nesting is a transaction statement that contains a call to a function that further encloses a transaction statement in its body; the nesting is not observable from the transaction statement code that contains the function call, however, at run-time

the inner transaction statement should be nested in the outer one (with the help of the underlying TM implementation).

The Java language extension specification allows syntactically both lexical and dynamic nesting of atomic transaction inside atomic transactions (as in **Section C.6** of the C++ specification). No nesting is allowed in irrevocable or elastic transactions. Contrary to the C++ specification where closed nesting is supported, only flat nesting of an atomic transaction in another atomic transaction is supported in the Java language extension (when the nesting is flat the abort of the inner transaction aborts also the outer transaction)[1] .

5.5.2 Nesting and transactional control flow

As only flat nesting of transactions is supported, `__transaction_cancel` keyword by itself causes the automatic abort of the outermost transaction.

5.5.3 Nesting and exceptional control flow

As for the control flow, since only flat nesting is supported by the specification, the commit-and-throw and abort-and-throw behaviors apply to the outermost transaction. For commit-and-throw, all the transactions in the nesting (including the outermost transaction) commit at the point where the exception is raised. An abort-and-throw causes the outermost atomic transaction to abort (thus aborting all the inner transactions) and raise the specified exception for further propagation. In short, the syntax of exceptional control flow does not change with nesting.

5.6 Exception handling support

In this section, we first discuss the exception handling support provided for transaction blocks and then introduce our novel language extension syntax for enhanced exception handling in multi-threaded applications.

[1] Although closed nesting corresponds to the nesting behavior a programmer expects, many TMs actually support flat nesting and, hence, we chose to support flat nesting in our specification. The reason for TM implementations to prefer flat nesting is that semantically it is close to closed nesting behavior (it behaves the same for committing transactions, which is what programmers care in general) and it introduces less overhead to the TM implementation. The decreased overhead allow flat nesting to perform faster for committing transactions (hence for the fast path) compared to a TM implementation supporting closed nesting. Clearly, however, flat nesting performs worse than close nesting when transactions abort.

5.6.1 Support for transaction blocks

Up to this section among the exception related features, we have seen how the exceptional control flow could be controlled by the programmer, but the handling of exceptions was not studied. The approach taken with the C++ specification, although not mentioned explicitly, is to use the existing `try-catch` mechanism for the handling of exceptions raised in transaction blocks. In the C++ specification both commit-and-throw and abort-and-throw behaviors raise an exception of a known type as in any other `try` block, with the limitation that an abort-and-throw can only throw exception types which are limited to primitive and enumerated types (see **Section C.5.3**). Hence, by inserting a transaction block into a `try` block, exceptions raised in the transaction block can be handled by the `catch` blocks of the wrapping `try` block.

The above approach satisfies the requirements of a transaction block for exception handling in an elegant manner since no modification is needed in the language for handling exceptions. That is why we also follow a similar approach in our specification. The only difference is that for abort-and-throw a different exception, `CancelException` is raised instead of primitive or enumerated typed exceptions of C++. Actually, considering that exception handling of exceptions raised according to commit-and-throw and abort-and-throw should be different, we think that attributing a separate exception class for abort-an-throw simplifies exception handling design of the application.

5.6.2 Enhanced exception handling capabilities

As it has been presented, the C++ specification defines the exception handling support of C++ only to satisfy exception handling needs of its new transactional constructs. The point of view in the C++ specification is that the transaction abstraction is used mainly as a data synchronization mechanism. We argue that by considering the transaction abstraction as a tool in the design of a programming language, it is possible to improve aspects other than data synchronization in a language. We exemplify this on exception handling of Java language and we propose to use a transaction block not only for data synchronization, but also for exception handling purposes.

In this section, we only overview the syntax and semantics of this *augmented* transaction block (which will later be named atomic box) in order to cover the complete language extension specification in this chapter. This novel language extension is mainly presented in **Chapter 7** where its syntax and semantics are described in detail, its usefulness is illustrated on different examples, its implementation is given in detail and its performance is analyzed.

Our objective of using a transaction block for exception handling is to preserve data consistency. In general, when an exception is raised in an application, a subtask of the application is left unfinished and, hence, some data in the application is left in an inconsistent state. For a sequential application such inconsistency only causes an abrupt and improper termination, while for multi-threaded applications it may cause incorrect executions, since an exception can be raised in a thread handling shared data, leaving shared data in inconsistent state either permanently or temporarily until the correction of inconsistency (if ever such correction exists).

The inconsistency created by a raised exception can easily be avoided with the transactional execution of `try` block: a transaction has the ability to rollback, hence, it allows to go back to the consistent state before the execution of the `catch` block. Conveniently, the abort-and-throw semantics provide automatically the rollback before the exception is raised out of the transaction. Setting the abort-and-throw semantics for transactional `try` blocks as the default behavior, prevents the application from executing on inconsistent state not only for handled exceptions but also unhandled exceptions.

To provide the above functionality we augment the `__transaction` block of our language extension with an optional `recover` block serving as a special type of `catch` block to the `__transaction` block. When the `__transaction` block is used without the `recover` block, it will keep the semantics defined as before. However, if used with the optional `recover` block, the `__transaction` block will become a transactional `try` block where `recover` block is its only `catch` block that catches any exception (whether raised according to commit-and-throw or abort-and-throw semantics) not handled in the `__transaction` block. Note, however, that the contents of the `recover` block are executed only after the transaction represented by the `__transaction` block is rolled back.

Our language extension goes even further and allows naming `__transaction` blocks with parameters to the `__transaction` keyword. Such naming is used to describe the set of `__transaction` blocks that should act together in a coordinated manner on an exception not handled in any of the `__transaction` blocks constituting the set. We call such a set of transaction blocks an **atomic box**. Raising of an unhandled exception in `__transaction` block of an atomic box results in the roll back of all the `__transaction` blocks of the atomic box, ensuring that the application state is always consistent.

The complete syntax for such `__transaction` block incorporating the above functionalities is:

```
__transaction [ (<type>)|("name", <handlingContext>) ]
{ S }
[ recover(CancelException <exceptionName>)
{ S' } ]
```

where `__transaction` and `recover` are keywords, S and S' are sequences of statements (that may include, among other statements, the transactional control flow keywords introduced by our language extension[1]), `<type>` is the parameter specifying the transactional guarantee (atomic, irrevocable or elastic) explained in **Section 5.2**, `name` is a string that associates the `__transaction` to the atomic box it belongs to, `<handlingContext>` is a keyword describing which `recover` blocks will execute for performing recovery in the associated atomic box and the `<exceptionName>` is the parameter of the `recover` keyword. Optional parameters and structures are enclosed in square brackets: `__transaction` may have no parameter and the block `recover` is optional. Note that, the parameters taken by the `__transaction` keyword can either be the `<type>` parameter or the `name,<handlingContext>` pair. The `<type>` parameter can only be used when the `recover` block is not used (our implementation does not allow elastic or irrevocable transaction guarantees for the `__transaction` block when used with a `recover` block), whereas the `name,<handlingContext>` pair can only be used when there is a `recover` block (without the `recover` block this parameter pair is meaningless). The above syntax description results in three forms which we describe as follows:

- **transaction form:** This is the syntax where `__transaction` block is used neither with the optional parameter pair `name,<handlingContext>` nor with the `recover` block (the `<type>` parameter is, however, allowed). This form corresponds to the language extension of the `__transaction` block where there is no built in exception handling support.
- **transactional try form:** This is the syntax where the `recover` block is used with the `__transaction` block without the optional parameter pair `name,<handlingContext>` (the `<type>` parameter is not allowed in this form). This form corresponds to the augmented `__transaction` block where the attached `recover` block serves as a `catch` block. In this syntax, irrevocable language statements in the `__transaction` block are forbidden since this would be in contradiction with the ability to rollback before performing recovery actions in a `recover` block.

[1] For the current implementation, an exception for the sequence of statements S is that when a `__transaction` block is used with an accompanying `recover` block, S can not include irrevocable statements.

- **atomic box form:** This is the syntax where `__transaction` block is used both with the optional parameter pair `name,<handlingContext>` and the `recover` block. This form is similar to transactional try form but with the use of optional parameters the full potential of the language extension is enabled. With this form atomic boxes are defined and the members of this atomic box can handle exceptions in a coordinated manner using their codes in the respective `recover` blocks. As with the transactional try form irrevocable language statements in the `__transaction` block are forbidden with this form.

The transactional try and atomic box forms of `__transaction` block propose a cleaner exception handling semantics compared to the transaction form. These two novel forms allow to separate cleanly the exceptions requiring commit-and-throw and abort-and-throw semantics:

- Exceptions requiring commit-and-throw semantics are handled in `catch` blocks appearing inside the `__transaction` block. This implies that such exception handling code is executed in transactional context which makes sense since the exceptional state of the system is valid only in transactional context. In other words, if an exception requiring commit-and-throw semantics is raised in a transaction, it needs to be handled in the transaction block. This way if the transaction commits, the corresponding exception handling actions are also committed, otherwise neither the transaction nor its exception handling actions appear as executed.
- Exceptions requiring abort-and-throw semantics are handled in the `recover` block. The `recover` block is meant to be executed only after the transaction has aborted, thus its content is executed outside transactional context. For abort-and-throw semantics this also makes sense, since the `recover` block should modify the system state permanently either for a later re-execution of the `__transaction` block (on a corrected state) or for continuing with the code following the transaction block.

An additional advantage of the transactional try and atomic box forms is that they propose a safer solution for unhandled exceptions: an exception not handled in a `__transaction` block is raised according to abort-and-throw semantics and will be treated in the `recover` block even if the code raising it inside the transaction required commit-and-throw semantics. This is a deliberate choice and is done to preserve data consistency through the rollback of the transaction in case of unhandled exceptions.

5.7 Summary

In this chapter, we have presented our complete Java language extension specification for transactional behavior. This specification allows the programmer to use the transaction abstraction as an exception handling and transactional control flow support as well as a synchronization mechanism. The specification is also contrasted wherever possible with the existing C++ specification [12] (also discussed in **Appendix** C).

The syntax of the extension is kept as simple as possible while still providing flexibility. The syntax allows multiple transactional semantics for a transaction block in a simple manner which makes it very easy to extend the specification with even further transactional semantics. Also the syntax requires the minimum effort from the programmer to use irrevocable statements in transaction blocks. Alternative execution paths are proposed as part of the specification, which introduces simple-to-use control flow possibilities to the programmer to provide alternative solutions to a given functionality. Alternative execution paths also provide a natural environment to support speculative execution. The specification allows the programmer to control whether a transaction should be committed or aborted upon exception raise. Moreover, it proposes safe coordinated exception handling mechanism over multiple threads. This mechanism provides safety since rollback ability of the transaction abstraction can bring the application's shared state to a consistent safe state upon an exception. The mechanism also allows coordination of multiple threads to handle an exception concerning shared or global state.

With all the powerful features provided together in a simple syntax, we hope our language extension can be useful at least for inspiring other similar specifications, if not used as is.

The implementation study we have performed for the specification explained in this chapter is presented in **Chapters** 6 and 7. **Chapter 6** presents the implementation details related to major part of the specification except the enhanced exception handling related issues (i.e., atomic boxes). The implementation for the atomic box related syntax and semantics is presented in **Chapter 7**. A realization of the full specification merging the implementations from **Chapter 6** and **Chapter 7** has been left as future work.

Chapter 6

Automatic Transactification of the TM Language Extension

The language specification presented in **Chapter 5** is described at the level of programming language, hence, it is simple and intuitive for the programmer to use this language specification to introduce transactional behavior into his/her program. However, the actual transactional behavior is implemented by a TM library that has as interface a collection of function calls (e.g., start, commit, read, write etc.). Hence, a necessary step for integrating TMs in a programming language is **transactification**, i.e., the transformation of a code written according to the language extension specification to one composed of TM library calls.

For the initial TM prototypes transactification was generally performed manually by programmers. The integration of TMs to a programming language requires, however, that transactification is done automatically. In this chapter, we present our automatic transactification solution and detail how it is applied to the language extension specification presented for the Java language in **Chapter 5**. In describing our solution we mainly focus on STMs, while most concepts are applicable also to HTMs (in case of HTMs, library calls correspond to special instructions, or short functions using these special instructions).

The organization of the chapter is the following. We first describe existing automatic transactification methods. This is followed by the solution we propose for automatic transactification in the Java language. We then explain the implementation details of our solution.

6.1 Automatic transactification methods

Four types of approaches for automatic transactification exist in the literature:

- **Native compiler transactification:** This is the most natural transactification approach where the compiler interprets the transactional constructs and generates the corresponding code directly. This approach has been adopted mostly for the C/C++ languages. Examples of the approach are Intel-STM compiler [140, 3], Sun C++ Compiler with TM extensions [5], Dresden TM Compiler [35, 1] and gcc-tm [160, 2]. All these compilers follow the draft specification for C++ [12] (which is similar to the language specification proposed in **Chapter 5** except for transactional control flow and exception handling features).
- **Source-to-source transformation:** The input code is preprocessed and the newly introduced transactional constructs are replaced with some code that represents the semantics of the described transactional constructs but using only existing programming language statements. The first known effort that has performed such transactification is AtomJava [101]. However, AtomJava produces a lock-based code that has atomicity semantics rather than using optimistic concurrency as TMs do. It introduces an extra instance field to every class and relies on this field to maintain an object-based lock schema. This schema is shown to reduce lock access overhead, but imposes a memory overhead which impacts not only transactional objects but all the application objects. Source-to-source transformation simplifies development of TM based programs while it introduces difficulties to the programmer to debug the original code. Although the objective is to replace Java monitors with transactions, Ziarek *et al.* [197] also uses the source-to-source transformation for their implementation.
- **Bytecode instrumentation:** The input code in this approach does not contain any new programming language constructs, but rather special annotations expressing transactional behavior. These approaches transform Java bytecode at load time (using BCEL/ASM-like libraries, or using Aspect-Oriented Programming (AOP)). Examples of this approach are LSA-STM [153], Deuce [111] and Multiverse [4]. This approach is elegant in the sense that no compiler modification is needed, yet the transactional behavior is available to the programmer. The downside is the lack of flexibility in the possible transactional language features since additional behavior is provided through annotations only.
- **Just-in-time compilation:** It delays the transactification of the code inside a transaction until runtime. The transaction start and commit are expressed with dedicated function calls. These calls can be replaced by macros in C/C++ to give the impression that a transaction code is enclosed in a transactional block marked with a language level keyword. The function calls allow a just-in-time compiler to switch mode such that in a transaction it augments the code on-the-fly in order to gener-

ate transactified code; and it does compilation without modifying/adding any code outside the transaction. This approach has been used by JudoSTM [143]. The advantage of this approach is the possibility to alter behavior at runtime and thus optimize transactional execution. One limitation with this approach, however, is that it is language dependent (it uses macros to provide the transactional block specification in the C/C++ language) and is not flexible in the transactional language features that can be provided. Moreover, an important downside that comes with just-in-time compilation is the runtime overhead required for instrumentation, especially if there are more concurrent threads than cores.

6.2 Automatic transactification with Java

Our objective for automatic transactification is to have both a simple implementation and a flexible language extension (for easy modification in programming language level constructs). The flexibility of language extension is desired in our case because it allows to introduce *(i)* new keywords (or syntactic elements) to support TM at the language level to achieve finer transaction granularity, and *(ii)* more sophisticated constructs to control transactional execution (e.g., for retries, control flow, exceptions etc.). This is how the language specification presented in **Chapter 5** could be implemented.

To meet our objective, we performed transactification using a combination of two methods described in **Section 6.1**: bytecode instrumentation and source-to-source transformation. Bytecode instrumentation was used to generate code allowing transactional behavior while source-to-source transformation served to fill the flexibility gap between the features allowed by the bytecode instrumentation and the language constructs we needed to provide for our language specification. In the realization of this combined approach, we preferred to use an existing instrumentation framework, namely Deuce [111], for bytecode instrumentation and we have developed our own front-end tool TMJava for source-to-source transformation. In the rest of this section we first give the overview of the complete transactification process, then we describe the implementation details of each of the two tools.

6.2.1 Transactification overview

The complete process of transactification is performed in two steps. In the first step, TMJava transforms the input source code such that the resulting code can be instrumented by Deuce. In the second step, Deuce instruments the code so that transactional behavior is possible.

6. AUTOMATIC TRANSACTIFICATION OF THE TM LANGUAGE EXTENSION

Deuce can only provide transactional behavior at the granularity of methods. It executes a Java class method in transactional context if the method is annotated with an `@Atomic` annotation. Since transactional programming language constructs are not limited to methods, the task of the TMJava front-end tool is to analyze the input code, find the constructs requiring transactional behavior and restructure the code such that the code requiring transactional behavior is moved into a method annotated with `@Atomic` annotation.

```
1  public int transferAll(Account[] src, Account dst)
2  {
3    int total = 0;
4    for (Account acc : src) {
5      __transaction {
6        int amount = acc.balance();
7        acc.withdraw(amount);
8        dst.deposit(amount);
9        total += amount;
10     }
11   }
12   return total;
13 }
14
15
16
17
18
19
20
21
22
23
24
```

Listing 6.1: Code illustrating a transaction expressed as a code block (the `atomic` block) rather than as a method.

```
public int transferAll(Account[] src, Account dst)
{
    int total = 0;
    for (Account acc : src) {


      total = transferAll_ab1(total, dst, acc);



    }
    return total;
  }
  @Atomic
  private final int transferAll_ab1(int total,
                                    Account dst,
                                    Account acc)
{
    int amount = acc.balance();
    acc.withdraw(amount);
    dst.deposit(amount);
    total += amount;
    return total;
  }
```

Listing 6.2: Code of **Listing 6.1** transformed by TMJava for Deuce.

The example code presented in **Listing 6.1** illustrates a case where a transactional behavior is required for a language construct other than a method. In this example, one wants to transfer the balance of a set of accounts to another account. Transfer operations must be atomic, but using a single transaction for all transfers would produce an unnecessarily long transaction, and would increase the risk of conflicts with concurrent operations. In contrast, using one transaction per transfer would reduce the likelihood of aborts, and allow for better concurrency. Note that the local variable `total`, which holds the total amount transferred, must be accessed transactionally, with its previous value restored upon transactional abort. Hence, all the necessary code is enclosed conveniently in an `atomic` block.

The fundamental restructuring performed by TMJava can be visualized by observing how **Listing 6.1** has been transformed to **Listing 6.2**. The transactional code block has been moved into a new method, while the code block is replaced with a call to this new method.

6.3 Deuce

Deuce [111] is a novel open-source Java framework for transactional memory. It requires little effort from its user for the transactification process. By default, Deuce uses an original locking design that detects conflicts at the level of individual fields without a significant increase in the memory footprint (no extra data is added to any of the classes) and therefore there is no GC overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs.

Deuce has been heavily optimized for efficiency. While there is still room for improvements, the performance evaluations of the authors on several high-end machines (up to 128 hardware threads on a 16-core Sun Niagara-based machine and a 96 core Azul machine) demonstrate that it scales well and show that it outperforms the main competing compiler and JVM-independent Java STM, the DSTM2 framework [96], in many cases by two orders of magnitude. Below we discuss briefly its architecture, its programming interface and the instrumentation it performs.

6.3.1 Architecture

Deuce consists of a framework that has both a load-time and a run-time component. **Figure 6.1** illustrates these components, the Deuce Java agent and the Deuce runtime, and their location in the software stack.

1. **Java agent:** This is the load-time component of Deuce that intercepts classes during class loading but before their first instance is created in the virtual machine. The

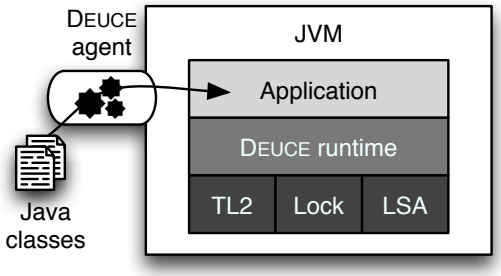


Figure 6.1: Components of the Deuce architecture and their respective location in the software stack.

component is based on an efficient Java bytecode instrumentation tool, ASM [25], to perform the instrumentation provided by the framework. During instrumentation, new fields and methods are added, and the code of transactional methods is transformed. The resulting code (the application layer in **Figure 6.1**) is the actual code running on top of the run-time component of the framework.

2. **Deuce runtime:** This is the run-time component of the framework that orchestrates the interactions between transactions executed by the application and the underlying STM implementation that provides the actual transactional behavior (STM implementations are shown as the lowest layer in the stack in **Figure 6.1**). The interface provided by the Deuce runtime component makes the framework architecture easily extensible, allowing researchers to integrate and test their own TM algorithms. The Deuce distribution comes with the implementations of two widely known STMs, TL2 [42] and LSA [153], compliant to the interface proposed by Deuce runtime.

6.3.2 Programming interface

The architecture explained above demonstrates the non-intrusive nature of Deuce: the components of the framework perform no modifications to the Java virtual machine (JVM) nor any extensions to the language are necessary. Deuce only provides transactional behaviour at the level of methods. Hence, the interface it proposes to the programmer is mainly composed of a set of annotations to be used by class methods that determine the method behavior in a transactional execution. There are three annotations used in Deuce with that objective:

- **@Atomic:** The method executes, by default, in a transaction that has weak isolation semantics (the semantics depend on the STM library used underneath but is usually the semantics of opacity (see **Section 2.4.4.1**)). The annotation can take two parameters: `retries=<NumOfReplies>` and `metainf="InfoString"`. Both parameters require a value to be associated each of them (the `<NumOfReplies>` and `"InfoString"` respectively). The `retries` parameter sets a limit on the number of consecutive retries of a transaction, if the limit is exceeded the transaction aborts and throws a special exception. The `metainf` parameter is mainly used to provide a property for the transaction to execute accordingly. Currently, a valid `"InfoString"` value is `elastic` meaning that the transaction will execute according to elastic transactional behavior (i.e., the transaction will provide the guarantee of elastic opacity described in **Section 2.4.4.1**).

- **@Irrevocable:** With this annotation the underlying TM is informed of the fact that the transaction is to be executed in irrevocable mode, hence the transaction cannot be aborted. Also, Deuce assumes this behavior automatically for native methods (even if those methods are not explicitly annotated).
- **@Unsafe:** The method executes out of any transactional context and accesses directly the memory addresses of the object fields used in the method, thus being more efficient than accessing the transactional copy that is under TM control. However, the execution of such a method executed in parallel with a transactional method accessing same object fields is prone to data races. That is why if its use is required, this annotation should be used with great care.

Apart from the annotations, Deuce adopts simple design decisions for exception handling and nesting of transactions. Deuce defines two special exceptions: `TransactionException` and `AbortTransactionException`. The former of the exceptions causes the transaction to roll back and retry, while the latter causes the transaction to rollback without any retries. Any other exception raised in a method (marked with the `@Atomic` annotation) causes the transaction to commit up to the point where the exception has been raised.

The nesting scheme used by Deuce is flat nesting; i.e., if an inner transaction aborts, the outer transaction also aborts.

6.3.3 Instrumentation

Deuce performs the following instrumentation operations on the Java bytecode at load-time.

- For every field, a constant is added to keep the relative location of the field in the class for fast access.
- For every field, 2 accessors are added (getter and setter).
- For every class, a constant is added to keep the reference to the class definition and allow fast accesses to static fields.
- Every (non-`@Atomic`) method is duplicated to provide an atomic and a non-atomic version.
- For every `@Atomic` method, an atomic version is created and the original version is replaced with a retry loop calling the atomic version in the context of a new transaction.

6.4 TMJava

As previously discussed, Deuce provides the integration of an STM implementation into Java programs using bytecode instrumentation as long as the Java program has a structure

suitable for Deuce (method-level transaction granularity, with transactions declared using annotations). Therefore, we developed, as part of this thesis, the TMJava transformation front-end tool (pre-compiler), to transform the source code of a program written according to the Java language extension specification given in **Chapter 5** into a program that is suitable to be instrumented by Deuce. In this section, we present the design of TMJava as well as the details of its transformation.

6.4.1 Design

TMJava is based on the *JastAdd* extensible Java compiler [51]. It makes use of JastAdd's ability to add new language constructs to Java language, to introduce the transactional constructs described in **Chapter 5**. To that end, TMJava is an extension only to the front-end of JastAdd and does not modify the backend. The front-end is by itself enough to generate a new source code from the input code. Hence, TMJava acts as a pre-compiler rather than a full compiler because it does not generate an intermediate representation or an object code from the input code.

TMJava uses only the following two features of JastAdd front-end:

- **AST generation:** The front-end can parse the input Java source code, which can also include constructs of a language extension, and generate an abstract syntax tree (AST) that corresponds to this source code. The extension of TMJava that is applied to the front-end is designed such that each transaction block of the language specification corresponds to an AST node. This way, once an AST is generated it is easy to spot the constructs proposed by the language specification.
- **Code regeneration:** The front-end can generate the source code corresponding to an AST. This feature of JastAdd is used to generate the transformed output code.

The execution flow of TMJava is composed of 5 consecutive steps:

1. The JastAdd front-end performs the AST generation.
2. TMJava performs a first pass on the AST to spot the locations of AST nodes that correspond to the transaction blocks of the code and save their locations for further processing.
3. TMJava performs an analysis pass over all the AST nodes that correspond to transaction blocks to find out what modifications to apply.
4. TMJava performs the modifications on the AST as indicated by the analysis pass (the modifications are performed per transaction block). An example of typical modification can be explained using the example in **Listing 6.1**. In this example, the AST

node that corresponds the transaction block (lines 5 - 10) would be replaced by a new AST node which corresponds to the function call to the method `transferAll_ab1` at line 7 of **Listing 6.2**. At the same time the body of the transaction block is copied as the body of the new generated method `transferAll_ab1` (line 15 of **Listing 6.2**), which constitutes a new AST node that is added to the AST (as a new method of the same class). At the end this step, all the modifications for all transaction blocks are applied and a modified AST is obtained.

5. The JastAdd front-end performs the code regeneration for the modified AST obtained in the previous step.

6.4.2 Transformation

In this section, we describe the implementation of code restructuring performed by TMJava. The section is divided into subsections each of which contains the transformation performed for a given language aspect or construct.

6.4.2.1 Transaction block

The transformation of a transaction block is the fundamental processing performed by the front-end tool. This transformation maps transaction blocks in the extended language into annotated Deuce methods. The transformation consists of *(i)* generating new annotated methods (using the `@Atomic` annotation) whose bodies are the content of the transaction blocks, and *(ii)* replacing the transaction blocks with calls to the generated annotated methods *(iii)* inserting some wrapper code before and after the call to generated methods if necessary. An example of this transformation is illustrated in **Listings** 6.1 and 6.2 where the steps *(i)* and *(ii)* are obvious and the assignment of the result of `transferAll_ab1` method is part of the wrapper code of step *(iii)*.

Moving a transaction block into a method body is not trivial as we need to take into account several issues, notably:

1. Variables and objects that are accessible inside the scope of the transaction block in the original code should also be available inside the scope of the method that corresponds to the transaction block in the generated code.

2. Modifications performed inside a transaction block on variables and objects that are defined outside the scope of the block, should be visible outside the transaction block. Hence, such modifications should also be perceived in the generated code

when returned from the method that corresponds to the transaction block of the original code.

To solve these issues, variables that are defined outside the transaction block, but are used and/or modified inside the transaction block, are passed as parameters to the corresponding annotated method. If a variable is of reference type the variable is passed to the method as is. As modifications of reference types will also be visible outside the transaction block there is no need to return such variables from the annotated method. For primitive types, however, Java only supports parameter passing by-value. Hence, variables of primitive types modified inside the transaction block are copied onto arrays (an array for each primitive type used), the arrays are passed as parameters to the annotated method and once the method has returned the array elements are copied back to the variables. As a simple optimization (to avoid the overhead of generating an array object), in the case a single primitive type variable being modified, the variable is passed as is to the annotated method and its updated value is simply returned and assigned to the variable. An example of how parameters are passed to the annotated method can be seen in the definition of the annotated method `transferAll_ab1` (lines 15-24 in **Listing 6.2**). The reference type variables are passed as is. The primitive type variable `total` is subject to the optimization so it is also passed as is, but its modified value is returned by the method on line 23.

6.4.2.2 Irrevocability

The support for irrevocability in the language extension is provided by two features: the `irrevocable` parameter passed to the `__transaction` keyword and the `@Irrevocable` annotation that is used for methods. These two features do not require code restructuring but merely the following modifications:

- For supporting the `irrevocable` parameter passed to the `__transaction` keyword, it is enough to replace the `@Atomic` annotation with the `@Irrevocable` annotation for the generated annotated method that corresponds to the transaction block.
- An irrevocable method is instrumented accordingly by Deuce using the `@Irrevocable` annotation. If a method is marked with this annotation no transformation is required to indicate the irrevocable nature of the method; the method specification is already Deuce compatible.

6.4.2.3 Control flow

Below we detail the transformations performed both for regular, transactional and exceptional control flow constructs described in **Section 5.4**.

Regular control flow: For regular control flow we consider only the control keywords `return`, `continue` and `break`. In the immediate scope of a transaction block (i.e., if the control flow keywords do not relate to a nested construct such as a loop), these keywords evoke the commit of the transaction. However, if the scope of these keywords is larger than the transaction, once committed the behavior expected from the keyword should be valid also out of the transaction. In order to provide this behavior we perform the following transformation:

- A position pointer field (an integer) is added to the class where the transformation is performed. This position pointer is meant to store the position (in the transaction) of the regular control flow keyword that caused the transaction to commit. The position value is generated by enumerating the regular control flow keywords in the transaction in their order of appearance in the transaction block code.
- The enumeration in the previous item causes a nested `if-then-else` statement to be added after the call to the Deuce function generated for the transaction block. In this nested `if-then-else` statement the conditions that are checked are the position values of the regular control statements (the position of the currently active control statement is held in the position pointer). The body corresponding to each position value in the nested `if-then-else` statement is the regular control statement for this position value.
- Each regular control statement encountered in the transaction block is replaced with two statements: *(i)* assignment of the position value of the control statement to the generated position pointer, and *(ii)* a `return` statement.

An example of this transformation is shown **Listings** 6.3 and 6.4. The code to be transformed is similar to the code in **Listing 6.1** except that there are two `if` statements before and after the line where the amount stored in the source account is obtained (lines 11 and 16 of **Listing 6.3** respectively). The first `if` statement (line 11) checks whether access to the account is granted (using the `credentials_ok()` method) and if access is denied quits the `for` loop using the `break` keyword in order to cease the transfers. The second `if` statement (line 16) ensures that the source account has an amount higher than zero before before any transfer is done. If, however, amount is not a positive value, it skips the rest of the iteration using the `continue` keyword in order to read the next source account.

The transformed code for the example in **Listing 6.3** is illustrated in **Listing 6.4**. The code generated by the transformation is shown in red. The three transformation items previously explained are as follows:

```
1
2
3
4
5  public int transferAll(Account[] src, Account dst)
6  {
7    int total = 0;
8    for (Account acc : src) {
9
10     __transaction{
11       if ( !acc.credentials_ok() )
12         break;
13
14       int amount = acc.balance();
15
16       if ( amount <= 0 )
17         continue;
18
19       acc.withdraw(amount);
20       dst.deposit(amount);
21       total += amount;
22     }
23
24
25
26
27
28
29   }
30   return total;
31 }
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
```

Listing 6.3: Code illustrating a transaction enclosing regular control keywords whose scopes exceed the transaction boundaries.

```
1
2  // Class has a new field;
3  int commitPos_transferAll_ab1;
4
5  public int transferAll(Account[] src, Account dst)
6  {
7      int total = 0;
8      for (Account acc : src) {
9
10       commitPos_transferAll_ab1=-1;
11
12
13
14
15
16       total = transferAll_ab1(total, dst, acc);
17
18
19
20
21
22       if(commitPos_transferAll_ab1 >= 0){
23         if( commitPos_transferAll_ab1 == 0);
24           break;
25         else
26           if( commitPos_transferAll_ab1 == 1)
27             continue;
28       }
29     }
30     return total;
31   }
32   @Atomic
33   private final int transferAll_ab1(int total,
34                                     Account dst,
35                                     Account acc)
36 {
37     if ( !acc.credentials_ok() ){
38       commitPos_transferAll_ab1=0;
39       return total;
40     }
41
42     int amount = acc.balance();
43
44     if ( amount <= 0 ){
45       commitPos_transferAll_ab1=1;
46       return total;
47     }
48
49     acc.withdraw(amount);
50     dst.deposit(amount);
51     total += amount;
52     return total;
53   }
```

Listing 6.4: Code of **Listing 6.3** transformed by TMJava for Deuce.

- The position pointer field added to the class is seen at line 3. The name of the position pointer is constructed using the prefix `commitPos_` and the name of the method corresponding to the transaction block (i.e., `transferAll_ab1`). The range of values this field can store is $\{-1, 0, 1\}$ in this example. The field is set to the value -1 in line 10 before the `transferAll_ab1` call and if this value stays unchanged after the `transferAll_ab1` method returns, this means that the method has completed normally without using any of the control keywords. If a condition in one of `if` statements of the transaction is true, the value of the position pointer will be 0 or 1 after the method returned. 0 indicates that the `break` caused the return while 1 corresponds to the case where `continue` keyword resulted in the return.
- The nested `if-then-else` statement (lines 22-28) is generated after the call to `transferAll_ab1` method. The outermost `if` statement checks whether the method `transferAll_ab1` has returned using a control keyword. If this is the case, then it checks the value of the position pointer field (the `commitPos_transferAll_ab1` variable) in lines 23 and 26 and executes the corresponding control keyword, i.e., it executes a `break` if position pointer field is 0 (line 24) and a `continue` if the field is 1 (line 27).
- The control keywords used in the transaction block are replaced with code storing the position of the keyword in the position pointer field and returning from the method. The `break` keyword under the access grant condition is replaced by lines 38-39 where the position pointer field is set to 0 with a following `return` statement. The code replacement for the `continue` keyword is the same (lines 45-46) with the exception that the position pointer field is set to the value 1 to indicate the difference in the keyword used and its position in the transaction block.

Alternative execution paths: An alternative execution path corresponds to one of the `either`, or `or` blocks given to define the alternative paths. Since the execution of all of these blocks is done according to transactional behavior the content of these blocks need to be executed in the Deuce function that corresponds to the original transaction block. The `either-or` structure of a transaction is transformed into a `switch-case` statement, where each of `either` and `or` blocks become a `case` of the `switch-case` statement. The mapping of the blocks to the `cases` is done as follows: `either` and `or` blocks are enumerated in order of their appearance and the number that corresponds to each block becomes the `case` value which enables the execution of the corresponding `either` or `or` block.

Transactional control flow: The transactional control flow is provided by the 3 keywords `__transaction_retry`, `__transaction_next` and `__transaction_cancel`. The transformation required for these keywords is composed of the following modifications:

- An object that counts the number of retries and that is excluded from transactification. This object is instantiated before the Deuce function that corresponds to the transaction is called and it is passed as a parameter to the Deuce function. The fact that it is excluded from transactification allows to follow the status of the control flow between transaction re-executions.
- A condition check at the very beginning of the Deuce function. This check executes each time the transaction executes and compares the current retry number to the maximum possible retry number (the maximum possible retry number is any value bigger than the number of alternative execution paths in the transaction). If the retry number is at the maximum possible value or bigger the Deuce function returns immediately. This allows the `__transaction_cancel` behavior, i.e., it implements the rollback and passing of the control flow to the statement that follows the transaction. If the retry number is smaller than the maximum possible value, then the transaction is executed. The retry number in this case corresponds to the `case` statement value of an alternative execution path that should be executed for the current transaction execution.
- Each of the encountered keywords is replaced with a piece of code as follows:
  - `__transaction_retry`: It is simply replaced with a transaction abort. It does not change the retry number stored so that the transaction can be re-executed exactly with the same retry number (hence exactly with the same alternative execution path).
  - `__transaction_next`: It is replaced with two statements: *(i)* a statement incrementing the stored retry number *(ii)* a transaction abort. This way the retry number is modified for the next transaction execution. If the transaction is to be re-executed, it will be executed with another alternative execution path that corresponds to the new retry number (because the switch statement will use the value of this retry number).
  - `__transaction_cancel`: The replacement is the same as `__transaction_next` except that instead of incrementing, the retry number is set to its maximum value. This way, the re-execution of the transaction is prevented after the abort.

Exceptional control flow: For the exceptional control flow, two behaviors need to be provided: commit-and-throw and abort-and-throw (see **Section 5.4.2.3**). Since Deuce in-

strumentation provides the commit-and-throw behavior by design, no changes are necessary for this behavior. However, the abort-and-throw behavior requires modifications. Since the abort-and-throw behavior is equivalent to the behavior of a `__transaction_cancel` upon an exception, all the modifications necessary for the `__transaction_cancel` keyword needs to be performed also for abort-and-throw. Additional modifications required are as follows:

- A `CancelException` object is instantiated before the call to the Deuce function that replaces the original transaction block and the same object is passed as a parameter to the Deuce function. The object is also excluded from transactification, since it will need to carry the type of raised exception out of the transaction.
- The `throw` statement that performs the abort-and-throw is replaced with a statement that stores the exception class given to the abort-and-throw statement in the `CancelException` object (see **Section 5.4.2.3** for the abort-and-throw statement syntax). Of course, this store statement is followed by some code that replaces a `__transaction_cancel` statement so that the transaction is not executed again.
- Right after the call to the Deuce function that replaces the original transaction block the `CancelException` object is checked to see if it has stored an exception, and if it is the case the `CancelException` object is thrown to propagate the exception.

6.4.2.4 Nesting

Since in the current specification the nesting support provided is flat nesting, the only transformation required for lexically nested transactions (see **Section 5.5.1**) is to replace an inner transaction with an anonymous block (dynamic nesting is handled by Deuce). If an inner transaction is defined to have irrevocable or elastic transactional behavior, the outer transaction behavior is changed to irrevocable or elastic behavior respectively. If the outer transaction has already irrevocable or elastic behavior, or if there are multiple inner transactions with irrevocable or elastic behavior in the outer transaction, the final behavior of the outer transaction becomes the highest priority behavior among all the available behaviors. The priority order between the behaviors is defined as atomic, elastic and irrevocable, where the highest priority is assigned to irrevocable.

6.4.2.5 Exception handling

As described in **Section 5.6.1** there is no language extension feature required to support exception handling of `__transaction` blocks. Conversely, atomic boxes and the corresponding enhanced exception handling constructs introduced in **Section 5.6.2** require an implementation. However, since the atomic boxes represent an original extension, all the

details about them, including the implementation (see **Section 7.6**) is presented in **Chapter 7**.

6.5 Summary

In this chapter, we presented our automatic transactification process which is composed of two steps: code restructuring (also called source-to-source transformation) and bytecode instrumentation. We have explained the role of each step, and focused on our tool TMJava, that performs the code restructuring and that has been developed as part of the thesis work (the other step is performed by a different tool, Deuce, developed at Tel Aviv University). In the transactification process the insertion of the transactional code is performed by the bytecode instrumentation step, however, this tool can produce transactional code only for properly annotated class methods. The role of the TMJava tool is to restructure the code such that all code that needs to execute in a transaction is mapped into a method(s) with the necessary annotation. We also describe how TMJava generates code to support irrevocability, control flow (regular, transactional or exceptional control flow), alternative execution paths, and transactional nesting.

As it performs code restructuring, TMJava is the tool that enables the specification that is described in **Chapter 5**. Based on an extensible Java compiler, JastAdd, TMJava is a flexible language extension specification implementation. Modifications or enhancements on the specification of **Chapter 5** can be easily adapted to TMJava. This shows the power of our two step automatic transactification process: allowing flexibility on the language specification side while providing a simple methodology (bytecode instrumentation based on method annotations) for generating transactional code.

Chapter 7

Atomic Boxes: Coordinated Exception Handling with TM

The transaction abstraction implemented by TM is mostly considered as a means to provide data synchronization. However, the transaction concept is more powerful than just being a data synchronization aid and has the potential to be useful on other aspects of concurrent programming. This chapter illustrates how to make use of the transaction abstraction provided by an underlying TM to improve failure recovery of multi-threaded applications. Since detection of failures and their handling is performed through the exception handling mechanism in Java, our approach is to introduce a transaction abstraction to enhance the Java exception handling.

We introduce the transaction abstraction as a language construct as part of our language extension described in **Chapter 5** and we present it as follows. First, we describe the problem for which the language construct could be useful in **Section 7.1**. We then give an overview of the solution proposed by the language construct in **Section 7.2**. **Section 7.3** presents closely related work. **Section 7.4** presents the example that is used subsequently to illustrate our language construct. **Section 7.5** describes the syntax and semantics of the language constructs for coordinated exception handling. **Section 7.6** presents the implementation details of coordinated exception handling in terms of code transformations (as part of TMJava). **Section 7.7** compares the performance we obtained against the competing alternative approach and **Section 7.8** summarizes the chapter.

7.1 Problem

Exceptions and exception handling mechanisms are effective means in redirecting the control flow of an error-prone sequential program before it executes on an inconsistent state of the system. This fact has led to extensive studies on exception handling mechanisms and their being tailored to work well with sequential programs. At the same time, a recent survey on 32 sequential applications presents the general picture on the exception handling usage by the programmers and reports that even though more than 4% of the total source code is dedicated to it, exception handling is neglected in most of the cases: either terminating the program or ignoring the exception [29]. This result shows that sequential programs are generally developed by using exceptions as a means to terminate programs in a convenient way and inconsistencies resulting from exceptional situations are not really handled.

In concurrent programs, however, an exception raised by one thread interrupts the execution of the thread's task, but this situation does not prevent other threads from continuing to execute their own tasks. This implies that an exception raised on a thread can bring some shared data to inconsistent state (at least temporarily until the exception is handled) and since other threads are not blocked upon the exception (they are unaware of the raised exception) they can access this inconsistent shared state. However, such an exception (one whose cause affect multiple threads due to shared state) should ideally be detected by all the threads that operate on the same shared state. Two solutions to the problem can be considered: *(i)* the program should be brought back to a consistent state by handling the exception, or *(ii)* all the affected threads (or even the whole application) should be terminated to avoid execution on inconsistent shared states. Since there are no widespread mechanisms that allow any of the solutions, it is up to programmers to devise a solution for such cases. In other words, compared to sequential programs where treating exceptions is barely considered, for concurrent programs handling exceptions should be part of the main application design and development in order not to jeopardize the application correctness.

Consider the code in **Figure 7.1** (inspired by a similar example in [176]) that illustrates how easily the above-mentioned inconsistency problem can appear in concurrent programs written using regular programming language constructs, i.e., without transactions. The figure presents a naive implementation of a classifier program where multiple threads concurrently evaluate nodes from the `unclassifiedNodes` list, process them, and move them to the target class using the `assignToClass` method. Note that we assume both the `unclassifiedNodes` list and the target classes `class[N]` are shared by all threads.

```
1  Class AlgorithmA {
2    int N; // number of classes
3    List unclassifiedNodes;
4    Set class[N];
5     ...
6     public void assignToClass(int srcPos, int targetClass) {
7       synchronized(this) {
8         Node selectedNode = unclassifiedNodes.remove(srcPos);
9         selectedNode.transform();
10        class[targetClass].add(selectedNode);
11      }
12   }
13 }
```

Figure 7.1: A concurrent code that may end up in an inconsistent state if an exception is raised while the selected node's representation is being transformed (in the `selectedNode.transform()` function) as required by the target class object.

If an exception is raised on line 9 and is not handled, the system reaches an inconsistent shared state: the `selectedNode` gets lost as it is neither in the `unclassifiedNodes` nor in its target class. For correct execution of the program, the exception should be handled and this should be performed before any of the other threads, unaware of the raised exception, access either the `unclassifiedNodes` list or the target class which are inconsistent. Hence, the handling of the exception should take the existence of concurrent threads into account.

This example, albeit naive, clearly shows that exception handling becomes a first class design consideration in development of correct concurrent programs. And, needless to say, with the mainstream computer hardware becoming multi-core, concurrent programming is about to become mainstream too. This fact highlights the need for solutions that will simplify concurrent programming under exceptional situations.

7.2 Overview of the proposed solution

Recently, Jacobs and Piessens proposed *failbox* as a mechanism to prevent the system from running in an inconsistent state [106]. The key idea is that if one thread raises an exception in a failbox, any other thread is prevented from executing in the same failbox. Instead of letting the system run in an inconsistent state, a failbox simply halts all concurrent threads accessing the same failbox. However, failboxes neither revert the system to a consistent state nor help the programmer recover from the error. Thus, resuming a program that has generated an inconsistent state is not possible with this approach. Since concurrent programming will be soon mainstream, we argue that an exception handling solution should be complete and allow both terminating and resuming the program upon raised exceptions.

In this chapter, we propose a novel language construct, `__transaction-recover`, as part of our language extension described in **Chapter 5**, that supports coordinated exception handling by providing a `recover` block attached to a transactionally executing `__transaction` block. Our `__transaction-recover` construct differs radically from the failbox extension, as it reverts the system to a consistent state upon an exception to enable recovery through coordinated exception handling. Hence, `__transaction` blocks do not only propagate the exceptions to concurrent threads of the system (as failboxes do), but also allow these threads to recover from this exception in a coordinated manner.

The programmer uses a `__transaction` block to demarcate blocks of code that should remain in a consistent state even upon exception raise. The `__transaction` block guarantees failure atomicity either by executing all its content or by reverting its modifications. Failure atomicity allows the programmer to handle exceptions. For example, by replacing `synchronized` with `__transaction` block in Figure 7.1, the inconsistency problem can be solved: the `__transaction` block reverts all its changes including the modification of `unclassifiedNodes`.

`__transaction` blocks can also be named to relate blocks of code that are dependent on each other such that if one of the blocks fails, the inconsistent state resulting from its execution is visible by each of the other related blocks. We call each such code block enclosed inside a `__transaction` block a *dependent* `__transaction` block (if an `__transaction` block is nested in another, the inner `__transaction` block is also a dependent `__transaction` block for the outer `__transaction` block). We call such a set of dependent `__transaction` blocks an **atomic box**.

Our `__transaction` block implementation is based on TM (as far as we know it is the first language construct that benefits from memory transactions for handling exceptions in concurrent programs). A `__transaction` block is a kind of augmented transaction block which not only aborts for memory access conflicts that occur in the block but also upon a raised exception inside a dependent `__transaction` block of the atomic box.

If an exception is raised inside a `__transaction` block, all threads executing in *dependent* `__transaction` blocks (or, equivalently, in the corresponding atomic box) stop their execution and rollback their changes. Then, execution continues in a `recover` block, analogous to a `catch` block, by one or all the affected threads, as specified by the programmer. Typically, the `recover` block aims at correcting the condition that caused the exception and/or reconfiguring the system before redirecting the control flow, restarting for example the execution of the `__transaction` blocks. Our `__transaction-recover` construct therefore provides the simplicity of a `try-catch`, but for coordinated exception handling in multi-threaded applications.

We find the simplicity of the construct important as it makes the handling of exceptions that have implications on multiple threads as simple as handling exceptions in sequential programs. Also, the fact that the shared states are accessed and modified in specific blocks of code in a program (in critical regions for lock-based programs and in transactions for TM-based programs), makes the use construct of the construct intuitive since the programmer does not need to reason further to locate dependent `__transaction` blocks. As a result, we believe that the proposed construct is a strong candidate for enhancing the exception handling in the future concurrent programs.

In order to evaluate the performance of our approach, we compared experimentally the `__transaction-recover` construct against failboxes, which only stop threads running in the same failbox without rolling back state changes. Our results indicate that `__transaction` blocks that comprise up to few hundreds memory accesses execute $2\times$ faster than failboxes in normal executions, where no exceptions are raised, and $15\times$ faster than failboxes to handle exceptions. We also tested extreme cases where a `__transaction` block executes thousands of memory accesses, in which case the cumulated overhead of TM accesses may result in lower performance than long failboxes. Besides illustrating that TM is a promising paradigm for failure atomicity and strong exception safety, our evaluation indicates that the atomic box mechanism is efficient compared to similar techniques providing weaker guarantees.

7.3 Related work

Concurrent exception handling. Thanks to their ability to rollback and their isolation from the other parts of the program, atomic transactions have been used for concurrent handling of exceptions since the eighties [175]. Transactions by themselves have been considered useful only for *competitive concurrency* where concurrently executing threads execute separately, unaware of each other, but access common resources. This type of concurrency is the primary target of our approach.

Although it can also be used for independent concurrent executions, the simplest form of competitive concurrency is the *future* construct. A future is a thread that is spawned at runtime, asynchronously performing some calculation concurrent to the program and yielding a value that is passed back to the program at the first point where it is needed. A future without any further support is unsafe under competitive concurrency, since shared accesses are not synchronized. Welc *et al.*[188] overcome this issue by object versioning and task revocation. Their solution also allows exceptions to be propagated out of a future (making it visible in the program) at the point where the future is spawned.

One early approach that targets handling exception under competitive concurrency is Arche [105]. This approach allows the propagation of exceptions from one thread to the other through synchronization constructs and represent multiple simultaneously raised exceptions with a single exception and terminate concurrent execution without rolling back changes. Clearly this approach, not adopting the transaction abstraction, does not solve the inconsistency problems our approach can handle.

A classical alternative to avoid inconsistencies in portions of programs generating competitive concurrency consists in encapsulating the associated code in transactions. Argus [118], Venari/ML [82] and Transactional Drago [108] all initiate a transaction on a single thread (within which multiple threads can be spawned) and allow an exception that cannot be resolved on a local thread to abort the transaction, passing the exception to the context where the transaction has been initiated. OMTT [110] allows existing threads to join a transaction but still propagate exceptions to a context outside the transaction. In these approaches, exceptions concerning all the competing threads result in the rollback of the transaction and the propagation of the exception outside the transaction. In contrast, our approach allows threads to (cooperatively) handle such exceptions, instead of directly propagating the exception outside of the transaction scope.

The secondary target of our approach is *cooperative concurrency* that occurs when multiple threads communicate to perform a common task. The mainstream solution for cooperative concurrency is coordinated atomic (CA) actions that propose to complement transactions with conversations to provide coordinated error recovery. This approach applies to distributed objects like clients and databases in a message passing context [192], e.g., the systems surveyed in [31] whose distributed modules are presented in [20]. In contrast, our approach targets modern multi-core architectures thus benefiting from shared memory to coordinate efficiently the recovery among concurrent threads. For example, our approach shares the concept of guarded isolated regions for multi-party interactions from [199] without requiring a manager to synchronize the distributed interaction participants. Furthermore, a programmer needs to include a significant amount of code to construct the CA action structure in his/her program using frameworks specifically designed for this purpose [31, 60], whereas in our approach the programmer can simply relate dependent code regions of an atomic box using built-in language constructs and their parameters.

A more recent checkpointing abstraction for concurrent ML, called *stabilizers*, monitors message receipts and memory updates to help recovering from errors [196]. Stabilizers can cope with transient errors but do not allow coordinated exception handlers to encompass permanent errors.

Failboxes [106] ensure cooperative *detection* of exceptions in Java. A thread that raises an exception while executing the code encapsulated in a failbox sends a signal to the concurrent threads that are also executing in the same failbox. Upon receipt of this signal an exception is raised so that all threads can terminate, which ensures that no thread keeps running on a possible inconsistent shared state. A failbox does not provide coordinated exception handling because the inconsistent state produced by the error cannot be reverted, hence the system has no other solutions but stopping. One could use failboxes to stop the entire concurrent program and restart it manually, however, restarting the program from the beginning may not prevent the same exception from occurring again. In contrast, `__transaction` blocks automatically rollback their changes upon exceptions and let the programmer define recovery handlers to remedy the cause of an exception and redirect the control flow.

Transactional memory. Transactional (atomic) blocks have been suggested as a potential solution for exception handling. Shinnar et al. [165] proposed a `try_all` block for C#, which is basically a `try` block capable of undoing the actions performed inside the block. Cabral and Marques [30] similarly propose to augment the `try` block with transactional semantics (using TM as underlying mechanism) to allow the retry of a `try` block when necessary. Other work proposed richer atomic block constructs that build upon TM and that help with exception handling [88, 91, 59]. However, all the existing implementations for the above work focus on sequential executions, hence being unable to cope with coordinated exception handling. When a thread raises an exception, it can either rollback or propagate the exception. If the exception is not caught correctly, the thread may stop and leave the memory in a corrupted state that other threads may access.

7.4 A running example

In this section, we introduce an example code (**Figure 7.2**) which we later use to explain different aspects of atomic boxes. The example represents a multi-threaded application with a shared task queue `taskQueue` from which threads get tasks to process. All threads execute the same code. Once a thread obtains a task, it first performs pre-computation work (getting necessary inputs and configuring the task accordingly) in the `prepare` method. The execution of the task is performed in the `execute` method of the thread, by calling sequentially the `process` and `generateOutput` methods of the task. We assume that `generateOutput` can add new tasks in the `taskQueue`.

```
1   public void run() {
2     Task task = null;
3     while(true) {
4       synchronized(taskQueue) {
5         task = taskQueue.remove();
6       }
7       if (task == null) break;
8       prepare(task);
9       execute(task);
10    }
11  }
12
13  public void prepare(Task task) {
14    task.getInput();
15    task.configure();
16  }
17
18  // No exception handling
19  public void execute(Task task) {
20    task.process();
21    task.generateOutput();
22  }
```

Figure 7.2: A simple example where multiple threads process tasks from a common task queue and that would benefit from concurrent exception handling.

In what follows, we mainly focus on the `execute` method of the thread. The code of the method is given without any exception handling. The traditional approach would be to use a `try-catch` statement enclosing the content of the `execute` method. However, when an exception is caught, one cannot easily determine at what point the execution of the method was interrupted and hence, in general, it is difficult to revert to the state at the beginning of the method. In such a case the `task` object could stay in an inconsistent state, possibly even affecting the state shared with other threads, and it would not be possible to simply put the task back into the `taskQueue` for later re-processing. The loss of a task might require other threads to reconfigure, or to stop execution altogether for safety or performance reasons: shared state may be inconsistent, incomplete processing would be worthless. We see in the next section using this example how atomic boxes prevent the loss of the task and how they allow the cause of the exception to be corrected and the coordination of threads to be used for program recovery.

7.5 Syntax and semantics

Our language extension deals mainly with code blocks that are dependent on each other in the sense that if a statement in one of the blocks raises an exception not handled within

the block, none of the other code blocks should continue executing. We call such blocks **dependent blocks** . An **atomic box** is a group of dependent blocks that can act together to recover from an exception raised in at least one of the dependent blocks. In order to express an atomic box, each dependent block belonging to an atomic box is enclosed inside a new Java statement, `__transaction-recover`. The fact that `__transaction-recover` statements belong to the same atomic box is specified by assigning them the same name. The name of an atomic box is assigned to a `__transaction-recover` statement as a parameter. If `__transaction-recover` statements of the same atomic box execute on different threads, these threads are said to be executing in the same atomic box.

An atomic box can be descendent of another atomic box, which means that the atomic box is dependent on the parent atomic box. Relating an atomic box as a descendant of another atomic box is achieved by assigning a descendant name in the hierarchical naming space.

The complete syntax of the our `__transaction-recover` and its resulting three forms, namely *transaction form*, *transaction try form* and *atomic box form* has been described in **Section 5.6.2**, but we repeat it here for convenience:

```
__transaction [ (<type>)|("name", <handlingContext>) ]
 { S }
[ recover(CancelException <exceptionName>)
 { S' } ]
```

In the above syntax, a `__transaction` block encloses a dependent block of the application, while the `recover` block specifies how exceptions not caught in the `__transaction` block are handled. If an unhandled exception is raised in a `__transaction` block, we say that the `__transaction` block fails. A `__transaction-recover` statement provides the convenience of `try-catch` to a dependent block with the following notable differences:

- *Failure atomicity:* A `__transaction` block of an `__transaction-recover` statement can be rolled back, i.e., either the contents of the `__transaction` block performs all of its modifications successfully (thus none of the `__transaction` blocks that belong to the same atomic box fail at any point), or the `__transaction` block acts as if it has not performed any modifications. The failure atomicity property of the `__transaction` block is possible because a `__transaction` block is executed inside transactional context.
- *Dependency-safety:* An atomic box ensures **dependency-safety**; i.e., if a statement fails by raising an exception, none of the statements that depend on the failing statement continue executing. The dependency relation between statements is es-

tablished by naming __transaction-recover statements with a common name (or with names of descendents). The dependency-safety is ensured by two properties of the __transaction-recover statement: *i)* A __transaction block executes in a transaction, thus its execution is isolated from all dependent code in the system until it commits. In other words, none of the dependent blocks see the effects of each other as long as they do not commit. *ii)* If an exception is not handled in a __transaction block it rolls back its changes and recovery actions are taken only after *(1)* all the __transaction blocks of an atomic box that have not started committing are rolled back and *(2)* all the __transaction blocks of the atomic box that have already started committing safely terminate their commits. These two properties make it impossible for a dependent block to see partial modifications of another dependent block that is in an inconsistent state.

- *Coordinated exception handling:* A try-catch statement offers a recovery from exception only for the thread on which the exception occurs. The __transaction-recover statement allows the programmer to inform concurrently executing threads of an exception raised in one of the other threads. Moreover, through the recover block of the __transaction-recover statement it is possible to recover from that exception in a coordinated manner. Note that the coordination is possible among recover blocks because they do not execute inside transactional context.
- Last, a __transaction block and its associated recover block can include try-catch statements to handle exceptions raised in their context.

We distinguish two different modes of operation in a __transaction-recover statement: *normal mode* and *failure mode*. The normal mode is associated with __transaction block and the failure mode is associated with the recover block. A __transaction block executes in normal mode, i.e., a __transaction block executes as long as no exceptions are raised or until an exception raised inside __transaction block propagates outside of the block. Note that if the code inside __transaction block raises an exception, and this exception is caught in the block itself, the __transaction block still executes in normal mode.

When an exception is propagated out of __transaction block boundaries (i.e., when an unhandled exception is raised in the __transaction block), the __transaction block is said to fail and its __transaction-recover statement switches to failure mode. The failure model of the __transaction-recover statement is such that when the block __transaction block fails, its associated atomic box also fails (because the atomic box acts as a single entity upon an exception). Thus, all the __transaction-recover statements

associated to the atomic box switch to failure mode upon the failure of a __transaction block. The failure of a __transaction block also triggers the failure of the descendent atomic boxes.

In the failure mode, all the threads that execute in the atomic box coordinate together. They wait for each other to ensure that all the associated __transaction-recover statements switch to failure mode and all the __transaction blocks are rolled back. Then they perform recovery actions as specified by the parameters of the __transaction block where the exception is raised. After the recovery actions are terminated all the threads decide locally how to redirect their local control flow. There are three options in redirecting the control flow at the end of recovery: restarting, continuing with the statement that comes after __transaction-recover statement, or raising an exception (i.e., abrupt termination). The first two options are provided through the transactional control flow keywords of our language extension (__transaction_retry and __transaction_cancel respectively), while raising an exception is done by the usual throw statement.

In the rest of this section, we detail the constructs for normal and failure modes, the control flow keywords, and the nesting of atomic boxes. We also discuss the semantics of the __transaction-recover statement under concurrently raised exceptions.

7.5.1 Normal mode constructs

The only normal mode construct introduced by our language extension is the __transaction block. A __transaction block encloses a dependent block of an atomic box. The block is part of the application code and the fact that it is enclosed in a __transaction block does not modify its functionality except for exception handling. In other words, as long as no exception is propagated out of the dependent block, there is no difference in terms of correctness of the application to have the block in a __transaction block or not. However, inserting the code in a __transaction block increases safety and provides a means for handling exceptions across multiple threads.

Although the functionality of the code inserted in a __transaction block is not modified, a __transaction block has different semantics compared to traditional blocks: __transaction block executes in transactional context. That way, the modifications performed by the code inside the __transaction block are only guaranteed to be effective if the __transaction block successfully terminates (i.e., if it successfully commits without switching to failure mode). Otherwise, none of the modifications performed in the context of the __transaction block are visible by code outside the __transaction block. Therefore, the code in an __transaction block executes atomically and in isolation.

The transactional nature of the `__transaction` block normal execution does not have effect on the correctness of enclosed code but has implications on its execution time. As the transactional execution is provided by an underlying transactional memory (TM) runtime, it incurs two types of latency overhead: *i)* data accesses in the `__transaction` block are under the control of TM and will be slower than bare data accesses. *ii)* in multi-threaded code if different `__transaction` blocks concurrently perform accesses on shared data in a way that inconsistencies would occur, a `__transaction` block may be aborted, rolled back and restarted, which adds extra latency to its execution.

As explained in **Section 5.6.2**, there are two forms of the `__transaction-recover` that can be used for handling of exceptions: *transactional try form* and *atomic box form*.

The transactional try form of a `__transaction` block is considered as an indication that the block is the only block in an atomic box, and thus it does not have any dependencies on other parts of the code. For such `__transaction` block the exception handling is done locally without any coordination with any other `__transaction` block. Hence, this form is suitable for exception handling in single-threaded applications as well as handling of exceptions for code blocks of multi-threaded applications that do not have any implications on other running threads.

As an example of such scenario, assume that an `OutOfMemoryError` is raised during the execution of the `execute` method of **Figure 7.2**. If for the running multi-threaded application, it is known that most of the tasks have small memory footprint but occasionally some tasks can have large memory footprint (but never exceeding the heap size allocated by the JVM), it is possible to clean up some resources and wait for a while before restarting execution. This would solve the problem as it allows to free memory until a task with a possibly large footprint finishes executing. Using the transactional try form of `__transaction` block, the code for this solution would be as in **Figure 7.3** (the syntax for the `recover` block is explained shortly).

Note that by enclosing the lines 3 and 4 of **Figure 7.3** with either a `try-catch` block or a failbox it is not possible to provide the suggested recovery since the state of the `task` object cannot be rolled back to its initial state. While the failbox approach cannot be used for a recovery in general, in this situation using `try-catch` block could have provided recovery, but this solution requires complicated exception handling code to be inserted into normal program code (which is exactly what is avoided using a `try-catch` block), whereas the transactional try form of `__transaction` block proposes a solution without changing the normal program code (only wrapping it around).

Contrary to the transactional try form of the `__transaction` block, the atomic box form implies that upon failure of the `__transaction` block the exception handling should

```
1   public void execute(Task task) {
2     __transaction {
3       task.process();
4       task.generateOutput();
5     } recover(CancelException e) {
6       if(e.getCauseClass() == OutOfMemoryError.getClass()){
7         // Back off (sleep) upon OutOfMemoryError
8         backOff();
9       }
10      // Implicit restart
11    }
12  }
```

Figure 7.3: Local recovery for an `OutOfMemoryError` using the transactional try form of `__transaction` block.

be coordinated across the atomic box. This form serves mostly in handling exceptions in multi-threaded applications.

We can slightly change the conditions to the example for which `__transaction` block provided a solution in **Figure 7.3** and generate a different scenario. Let us assume that in the example there are not many solutions for solving the `OutOfMemoryError` and the programmer simply wants to stop all the threads when such an exception is raised. The code that provides this solution would be as in **Figure 7.4**.

```
1   public void execute(Task task) {
2     __transaction("killAll", all) {
3       task.process();
4       task.generateOutput();
5     } recover(CancelException e) {
6       if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7         // Upon OutOfMemoryError, propagate to terminate thread
8         throw e;
9       }
10    }
11  }
```

Figure 7.4: Coordinated termination upon an `OutOfMemoryError`. The atomic box form of `__transaction` block can be used to provide such recovery.

Note that all the threads are running the same code. The code in **Figure 7.4** uses the atomic box form of `__transaction` block. The `<handlingContext>` parameter is given as `all`, which means that when the `OutOfMemoryError` is raised on one thread, all the threads running in the atomic box will execute their `recover` blocks. In the `recover` block an exception is raised so that the currently executing thread dies (since the threads are

assumed to be running the code in **Figure 7.2**, the exception will not be caught and each thread will be terminated). This solution is again not possible with a `try-catch` statement. Since the objective in this example is to stop the application, the failbox approach would also work: one could enclose the content of the `execute` method in an `enter` block, which would specify that the code enters a failbox common to all threads.

We can also think about a variant of the above example that cannot be resolved using the failbox approach. Let us assume that, as the `task` object can configure itself before execution, it is also possible to reconfigure it to perform the same job using less memory but slower (e.g., by disabling an object pool). In such a case, the atomic box form of the `__transaction` block allows us to resolve the problem with the code in **Figure 7.5** (again only by changing the content of the execute method). This solution is possible with the atomic box form of `__transaction` block since the `__transaction-recover` statement including the `__transaction` block provides failure atomicity and coordinated exception handling. The failure atomicity property of the `__transaction-recover` statement allows the modifications of the execution inside the `__transaction` block to be rolled back, thus the task object can be reverted to a consistent state where it can be reconfigured. The coordinated exception handling provided by the `__transaction-recover` statement allows the same behavior to be performed on all threads in a synchronized way and remedy the problem in a single step.

7.5.2 Failure mode constructs

Since an atomic box corresponds to a set of dependent blocks, when an `__transaction` block fails, its associated atomic box also fails. We call the atomic box that fails upon the failure of an `__transaction` block an *active* atomic box. An **active atomic box** is

```
1   public void execute(Task task) {
2     __transaction("reconfigure", all) {
3       task.process();
4       task.generateOutput();
5     } recover(CancelException e) {
6       if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7         // Upon OutOfMemoryError, reconfigure and restart
8         task.reconfigure();
9       }
10    }
11  }
```

Figure 7.5: Coordinated recovery to reconfigure tasks (for decreasing their memory footprint) upon `OutOfMemoryError`.

defined as the set of __transaction blocks of the same atomic box that have started executing and that have not yet started committing. This set is defined as long as at least one __transaction block executes in the atomic box.

We argue that in terms of failure it is enough to consider an active atomic box rather than all the statically defined atomic box to ensure dependency-safety and failure atomicity. Since __transaction blocks that have started committing are guaranteed not to execute on any inconsistent state that can be generated by the __transaction blocks of the active atomic box (__transaction blocks execute in isolation), their exclusion does not harm dependency-safety. Moreover, the consistency of data is ensured as long as the commit of __transaction blocks that have started committing are allowed to finish before the __transaction blocks of the active atomic box start performing recovery actions. Such synchronization eliminates the need for __transaction blocks that have already started committing to rollback upon the rollback of an active atomic box. Hence, it is safe to provide failure atomicity only for an active atomic box.

To have a better understanding of the concept of active atomic box consider the solution proposed in **Figure 7.4**. For this solution if we think that the tasks executed by all of the threads have more or less the same load, the threads will generally be executing the execute method at about the same time. However, if we think of a scenario where tasks have variable load, this may not be true. So when the OutOfMemoryError is raised, some threads may be executing in the __transaction block, while some others may be still committing the __transaction block in the execute method and some others maybe fetching a new task from the taskQueue (these threads have not yet entered in a __transaction block). In such a case, the proposed solution may not stop all the threads since not all may be executing in the active atomic box when the OutOfMemoryError is raised. However, for these non-terminated threads the execution continues safely; threads that were committing while the exception is raised in the active atomic box do not have any more dependence on the __transaction blocks of the atomic box, and threads that have not yet entered execution in the atomic box may not raise an OutOfMemoryError if there is enough memory once the threads of the active atomic box get killed. Even if an OutOfMemoryError is again raised, this will be resolved by the active atomic box defined at the time of the second exception. Hence, we see that by applying the failure atomicity and dependency-safety only on the active atomic box it is also possible to provide safe executions.

The failure of an active atomic box results in the following coordinated behavior in the __transaction blocks that constitute the active atomic box:

1. __transaction blocks that constitute an active atomic box switch to failure mode. This triggers the coordinated failure behavior of the atomic box. Entry to the active

atomic box is forbidden for any thread during failure mode. If such a thread exits, it waits to join the active atomic box until the current active atomic box switches back to normal mode.

2. All the `__transaction` blocks that switch to failure mode automatically rollback. At the same time all `__transaction` blocks that have started committing terminate their commit.

3. All the threads executing in an active atomic box are notified of a special exception `CancelException` (the structure of this exception is explained later).

4. All the threads executing in an active atomic box wait for each other to make sure that they all rolled back and received the `CancelException` notification. The threads in the active atomic box also wait for threads running a `__transaction` block that has already started committing, to finish their commit operation (which may not succeed and trigger an abort).

5. All the `__transaction` blocks that constitute an active atomic box perform the recovery actions in the associated `recover` blocks according to the `CancelException` they receive.

6. All the threads executing in an active atomic box wait for each other to terminate their recovery actions. Once all recovery actions are terminated each of the threads executing in the active atomic box decide locally how to redirect their control after failure.

The `CancelException`: The structure of the `CancelException` that is notified to all the threads in the active atomic box is as follows:

```
public class CancelException{
  Class causeClass;
  String message;
  Thread source;
  String aboxName;
  int handlingContext;
  // Methods omitted...
}
```

where the `causeClass` field stores the class of the exception raised by the `__transaction` block that initially failed (initiator `__transaction` block), the `message`

field is the message of the original exception, the `source` field is the reference to the `Thread` object executing the initiator `__transaction` block, `aboxName` is the name of the failing atomic box and `handlingContext` is an integer value that defines which of the corresponding `recover` blocks associated to the atomic box will be executed. The value of the `handlingContext` corresponds to the `<handlingContext>` parameter of the initiator `__transaction` block (the details for the values of `handlingContext` are explained below together with the `recover` block). Note that the `CancelException` stores the class of the original exception object that initiated the atomic box failure rather than its reference. This is a deliberate choice since the original exception object can include references to other objects that are allocated inside the initiator `__transaction` block and that will be invalidated by the rollback performed upon the failure of the atomic box.

The `recover` block: A `recover` block encloses recovery actions to be executed when the `__transaction` block to which it is associated fails. Since the `recover` block is related to failure of an atomic box, it is only part of failure mode execution. Note also that the `recover` block does not execute in a transactional context; it always executes after its corresponding `__transaction` block rolls back (if not the recovery actions performed in the `recover` block will also be rolled back making it difficult to perform modifications in system state for recovery). The decision of whether the `recover` block will be executed depends on the `handlingContext` field of `CancelException` sent by the initiator `__transaction` block. Two values exist for the parameter `handlingContext`: `local` and `all`. With the `local` option, only the `recover` block of the initiator `__transaction` block will be executed, other threads will not execute any recovery action (even if the `recover` block has been specified for threads other than the initiator). If the `all` option is chosen all the threads executing in the atomic box execute their respective `recover` blocks.

Whichever of the `handlingContext` options is chosen, once the `recover` block executions are terminated each of the threads executing in the atomic box take their own control flow decision. If the `handlingContext` parameter has the value `local`, the initiator `__transaction` block redirects the control flow according the control flow keyword used in its `recover` block (for the control flow keywords see **Section 5.4.2.2**). All the other threads in the atomic box re-execute their respective `__transaction` blocks. If the `handlingContext` parameter has the value `all`, each of the threads redirects the control flow according the control flow keyword used in its respective `recover` block.

The syntax of the `recover` block can be described as follows:

recover(CancelException exceptionName) { S }

where the `exceptionName` is the name of the `CancelException` notified to all the threads upon failure of an atomic box. The exception parameter of the `recover` block is expected to be of type `CancelException` and providing an exception of another type will produce a compiler error.

Having analyzed most of the properties of the normal and failure modes, it would be appropriate to analyze the mechanisms described above in an example. At this point, we can use another variant of the running example of **Figure 7.2** with an `OutOfMemoryError` being raised during the execution of the `execute` method. Suppose, in this case, that the programmer knows that s/he is using too many threads and if the heap allocated by the JVM is not sufficient, it would be enough for him/her to kill only some of the worker threads. This would effectively handle the exception while keeping the parallelism of thread execution at a reasonable level. Since the programmer would not know the size of the memory allocated in advance s/he can choose to implement the solution in **Figure 7.6** using the atomic boxes.

```
1  public void execute(Task task) {
2    __transaction("killSome", local) {
3      task.process();
4      task.generateOutput();
5    } recover(CancelException e) {
6      if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7        // Upon OutOfMemoryError, propagate to terminate local thread
8        throw e;
9      }
10   }
11 }
```

Figure 7.6: Coordinated recovery to decrease the memory used by the multi-threaded application by only killing some of the threads upon `OutOfMemoryError`.

The solution shown in Figure 7.6 is the same as the code in **Figure 7.4** except that the value of the `<handlingContext>` parameter is set to `local` instead of `all`. With this change each time an `OutOfMemoryError` is raised only the thread raising the exception executes the `throw` statement and kills itself. This solution works better than a simple `try-catch` because with the `try-catch` solution multiple threads could raise the same exception at the same time and, being unaware of the exceptions raised in other threads, all of these threads would kill themselves leaving a smaller number of threads running in the system, rather than gradually decreasing the amount of concurrency. Gradual decrease is possible thanks to the coordinated nature of the exception handling: coordination causes the threads to abort their `__transaction` blocks (instead of killing themselves) and restart

execution after the initiator __transaction block's thread is killed. Thanks to the failure atomicity provided by atomic boxes, this can safely be repeated as many times as required until the required number of threads are killed.

7.5.3 Redirecting control flow after recovery

The transactional control keywords __transaction_retry, and __transaction_cancel defined in **Section 5.4.2.2** are valid for providing the desired control flow after the actions in the recover block are performed. These keywords are to be used mainly inside recover blocks but they can also be used with similar semantics in the __transaction blocks. The only difference of using the keywords in a __transaction block is that they immediately fail the __transaction block (and respectively also the active atomic box) and they behave as a recover block that has no other recovery actions but only the specified keyword. Thus, the existence of these keywords in the __transaction block will just serve as a shortcut to a case where the atomic block has failed and we execute only a __transaction_cancel or __transaction_retry inside the recover block.

If no control flow keyword is provided, upon exit, the recover block implicitly re-executes the associated __transaction block. This behavior is natural to adopt for the programmer since the default behavior of a __transaction block is also re-execution for other issues such as synchronization of data shared among __transaction blocks. (providing different default behaviors for different cases would complicate the use of the __transaction block). Additionally, this approach is safe since the members of the atomic box are all rolled back before recovery. Default re-execution behavior can lead to liveness issues, but the responsibility to resolve such issues is left to the programmer, since it is the programmer who performs the recovery (using recover blocks) and knows the state of the system after the recovery (at the end of recover blocks).

If the default control flow behavior of the recover block is not suitable for the program, the programmer can control where the program is redirected by using the transactional control keywords: explicit re-execution of the associated __transaction block can be requested using the__transaction_retry keyword, while passing the control to the statement following the recover block is provided by using __transaction_cancel. Note that with a __transaction_cancel keyword, the effect of a __transaction block is as if it has never executed. The reason is that the failure of the __transaction block has caused the rollback of the modifications performed within.

The use of throw statement inside recover block will quit the recover block and propagate the exception in the context of the statement following the recover block.

With a `throw` statement, again the atomic box appear as if it has never executed. Similarly if an unhandled exception is raised in recovery action code enclosed in an `recover` block, the behavior is the same as an explicit `throw` statement.

As explained in **Section 5.4.1**, any already existing regular control flow keyword that quits a block (i.e., `continue`, `break` and `return`) does not change semantics when used in an `__transaction` block. When used inside a `__transaction` block (and not used inside a nested block such as a loop) these keywords imply immediate commit of the tentative modifications up to the point of occurrence of the keyword and pass the control to the target destination outside the `__transaction` and `recover` blocks. If these control keywords are used inside a `recover` block, they behave exactly the same way as in the `__transaction` block except that, since the `__transaction` block is rolled back, none of the effects of the `__transaction` block are visible (but of course the modifications inside the `recover` block are effective).

The use of a `throw` statement inside the `__transaction` block raises an exception in the block as in plain Java. If the exception is handled inside the `__transaction` block the behavior of the `throw` statement is unchanged. However, if the exception is not handled in the `__transaction` block, the `__transaction` block (and the corresponding active atomic box) switches to failure mode.

7.5.4 Nesting of atomic boxes

The failure of an `__transaction` block can also trigger the failure of an atomic box other than the one it belongs to. If the failing atomic box is parent of another atomic box, when the parent atomic box fails, the child atomic box also fails, thus both the parent and the child atomic boxes switch to failure mode. In contrast, when a child atomic box fails, its parent atomic box does not fail, thus the child atomic box switches to failure mode, while the parent atomic box does not.

The fact that atomic boxes have ascendants or descendants is reflected by a hierarchical naming of `__transaction` blocks. The `name` parameter of an `__transaction` block can be a string of the form `x.y.z` following the naming convention of Java package names.

7.5.5 Resolution of concurrently raised exceptions

Up to this point we have considered only the case where a single `__transaction` block initiates an atomic box failure. If an exception needs to be treated by a `__transaction` block, this is most probably because the exception concerns all the threads executing in the atomic box. So it is not surprising to expect that multiple `__transaction` blocks raise

the same exception and fail the atomic box. It is also perfectly possible that different `__transaction` blocks of the same atomic box, concurrently raise different exceptions and cause the atomic box to fail.

The atomic box takes a very simple approach to resolve concurrently raised exceptions thanks to its failure atomicity property: an atomic box allows only one exception (the first one to be caught) to be treated in failure mode and ignore all the rest of the concurrently raised exceptions during failure mode. In other words, the atomic box does not consider all the concurrently raised exceptions together. By handling one exception and removing its cause before re-execution, one may avoid reoccurrence of other concurrent exceptions. During re-execution, if the causes of the concurrently raised exceptions are not removed they will again manifest and fail the atomic box. They will thus be treated during re-execution.

As can be noticed, among other advantages, the atomic box approach brings an elegant solution to the concurrent exception handling problem thanks to its failure atomicity property. Actually, the solution presented in **Figure 7.6** is a good example illustrating the resolution of concurrently raised exceptions. In this example, other than the coordinated nature of the exception handling, it is the simple concurrent exception handling approach taken by atomic boxes that allows only the required number of threads to be killed.

7.6 Atomic boxes implementation

Our atomic boxes implementation is currently at a preliminary stage but covers the fundamentals of the atomic box semantics explained in **Section 7.5**. Our implementation is designed as part of the TMJava front-end tool (see **Chapter 6**). As such, the implementation consists of transforming the source code into plain Java code where the `__transaction` and `recover` blocks are appropriately wrapped and augmented with code allowing the required semantics.

A `__transaction` block is mapped to a Deuce function annotated with an `@Atomic` as is done for transaction blocks without the atomic box semantics (explained in **Section 6.4.2.1**). The `recover` block is also mapped to a function but that is explicitly marked such that it does not execute in transactional context (the reason for an explicit indication is that the function is called from inside a transactional context for which Deuce automatically instruments a transactional version and performs a call to the transactional version instead of non-instrumented version, which we would like to avoid in this case). The body of a `__transaction` block is placed into the corresponding Deuce function by enclosing it in a `try` block followed by a single `catch` block catching all `Throwables`. If an unhandled exception is propagated out of the body of a `__transaction` block, this `catch`

block catches the exception and sets the status of the atomic box to failure (also records the raised exception inside a `CancelException` for all the members of the atomic box to see the raised exception).

To describe an atomic box, we define an `AtomicBox` class that contains, a field indicating the status of the atomic box (failure or normal execution mode), a `CancelException` member storing the exception that caused the failure of the atomic box[1], a counter indicating the number of `__transaction` blocks that are in the active atomic box (to be used by barriers synchronizing threads; one before beginning recovery and one right after the recovery) and a second counter indicating the number of `__transaction` blocks that have started committing but has not finished it yet. Since atomic boxes are defined statically in code using the name parameter of the `__transaction` block, they are allocated and constructed at the beginning of the program and they are used as shared objects throughout the program (except if the `__transaction` blockis not named; in this case the atomic box is local to a thread). As with the function that corresponds to the `recover` block of the `__transaction-recover` statement, the `AtomicBox` objects are accessed directly, and not through the TM, i.e., `AtomicBox` object accesses are excluded from transactification. This is required since `AtomicBox` objects carry information out of aborted transactions.

The Deuce function that corresponds to a `__transaction` block contains some more code before and after the `try-catch` construct that encloses the contents of the `__transaction` block. At the beginning of the function and before returning from the function (these points corresponds to start and commit of a transaction), this additional code checks whether the atomic box to which the current `__transaction` block belongs is in failure mode. If this code finds out that the atomic box is in failure mode, it aborts the transaction for a restart. If it is the code at the beginning the Deuce function that observes the failure of the atomic box, it calls the function that corresponds to the `recover` block by passing the `CancelException` member of the atomic box to the function. The return value of this function is used to retry the `__transaction` block or to quit the function (thus to apply `__transaction_cancel` semantics). If the function throws an exception the exception is propagated out of the Deuce function as well.

The function that corresponds to the `recover` block executes the contents of the `recover` block, but passes through a preceding and succeeding barrier, to ensure the synchronization of threads as explained in the steps 4 and 6 of the list of steps describing the coordinated behavior upon failure of an atomic box (see **Section 7.5.2**). Between the two barriers, the function checks its `CancelException` parameter's `handlingContext` field to

[1] a single exception storage is enough since only one exception is handled even when there are concurrently raised exceptions (see **Section 7.5.5**)

see if it has to execute the `recover` block contents. If the `handlingContext` field has the value `local` it skips all statements in the `recover` block and enters the second barrier for the second synchronization of recovering threads (if the `recover` block corresponds to the initiator `__transaction` block, the `recover` block contents are executed regardless of `handlingContext` field value). By default the function returns a value such that the caller Deuce function retries the `__transaction` block. If the contents of the `recover` block requires, the return value is changed so that the caller Deuce function quits to provide `__transaction_cancel` semantics. Also, if the `recover` block raises an exception, a flag is set to indicate this, and the exception is raised only after the second barrier is crossed.

Since our implementation is at a preliminary stage, the design explained above is partially realized. Especially, the features required for the correct execution of atomic box form of a `__transaction` block are not complete. For this reason, the examples in **Figures** 7.4-7.6 could not be tested with the current implementation. This, however, did not prevent us to perform the evaluations on the performance of the design (with respect to the performance of failboxes) that we present in the next section.

7.7 Evaluation

We compare our atomic box solution against failbox [106] on an Intel Core2 CPU running at 2.13GHz. It has 8-way associative L1 caches of 32KB and an 8-way associative L2 cache of 2MB. For atomic box we implemented the extension to TMJava as explained in Section 7.6 whereas for failboxes we reused the original code from [106].

7.7.1 Producer-consumer example

Our first experiments consist of a simple producer-consumer application, where one thread pushes an item onto a shared stack while another pops the topmost item from the same stack. For the sake of evaluation, the stack `push()` method raises an exception if adding the new item to the stack would exceed its capacity. We evaluated two versions of the same program: one using failbox, the other using our atomic box. The execution times of these two versions have been evaluated in normal cases (where we fill the stack prior to execution such that no exceptions are raised) and for handling exceptions (where we try to push an item to an already full stack). Results are averaged over 100 executions.

Tables 7.1 and 7.2 report the minimum, maximum and average execution time in microseconds, respectively without and with exceptions. On the one hand, we observe that our solution executes about $2\times$ faster (on average) than failboxes in normal executions.

	min	max	average
atomic box	7.27	11.67	8.92
failbox	15.70	34.97	18.58
speedup of atomic box	1.34	4.81	2.08

Table 7.1: Execution times of atomic box and failbox (in microseconds) on a multi-threaded producer-consumer application when no exception is raised.

This is due to a cache effect observed with the failbox approach. Each time a failbox is entered a shared variable is checked to verify whether it has failed. Since this experiment requires very frequent entries to a failbox by multiple threads the failbox entries are serialized. Our implementation does not suffer from this problem since the check for the failure of an atomic box does not need to be verified often (an atomic box is executed in isolation from other code).

On the other hand, our solution performs more than $15\times$ faster (on average) than failboxes to handle exceptions. We conjecture that it is due to the fact that the failbox approach uses the `interrupt` mechanism to communicate the exception on one thread to the other threads. The atomic box approach communicates over the shared memory, resulting in a faster notification. It is worth mentioning that our atomic boxes allow both `push()` and `pop()` methods to recover from an exception, allowing the program to resume, while failbox simply stops the program upon the first raised exception. Considering this desirable behavior and the observed overhead, atomic box clearly represents a promising approach.

	min	max	average
`__transaction` block	1.40	2.62	2.22
failbox	32.167	47.23	34.55
speedup of `__transaction` block	12.28	33.74	15.7

Table 7.2: Execution times of atomic box and failbox (in microseconds) on a multi-threaded producer-consumer application when exceptions are raised.

7.7.2 Sorting examples

Our second experiments rely on two single-threaded sorting applications (quick-sort and bubble-sort) coded in 3 ways: *(i)* using plain Java (with no extensions), *(ii)* inside failboxes, and *(iii)* inside `__transaction` block. The plain Java version is used to measure the inherent

overhead of failbox and atomic box versions. The sort is performed inside a function and the application can choose to run either a *quick-sort* or a *bubble-sort* function.

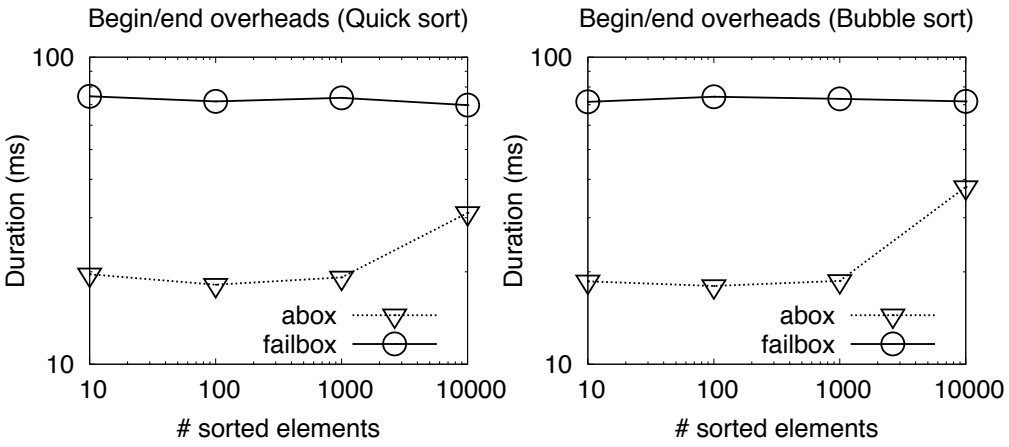


Figure 7.7: Comparison of the overhead produced when starting and terminating an atomic box (mentioned as abox) and a failbox (note the logarithmic scales on both axes).

Figure 7.7 through **Figure 7.9** depict the performance of failbox and atomic box on quick-sort (left column) and bubble-sort (right column). **Figure 7.7** compares the execution overhead due to entering and leaving a `__transaction` block block or a failbox (we call this the *begin/end overhead*). **Figure 7.8** shows the execution time performance of atomic box and failbox executions without the *begin/end overhead*. **Figure 7.9** depicts the total execution time performance of atomic box and failbox. The execution time performance depicted in figures 7.8 and 7.9 are given as the slowdown with respect to the performance of the plain Java version, which does not have any begin/end overhead. Each point in the graphs corresponds to the average of 10 runs.

The results show that although the failbox approach performs as well as plain Java inside the failbox, its begin/end overhead is quite high. We attribute this high overhead of the failbox approach to the memory allocation performed to generate a new failbox (be it a child or a new failbox) before entering the failbox. **Figure 7.9** also illustrates that atomic box perform better than the failbox approach for input arrays of up to about 1000 elements. This demonstrates that our atomic box implementation, although using transactions to sort array elements, performs well even compared to simpler approaches that do not roll back state changes.

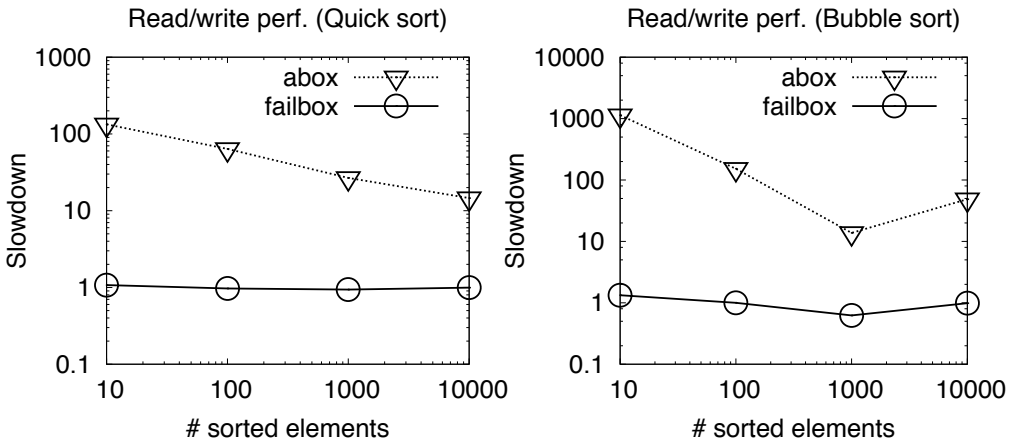


Figure 7.8: Comparison of the overhead due to accessing the shared memory in atomic box (mentioned as abox) and failbox (note the logarithmic scales on both axes).

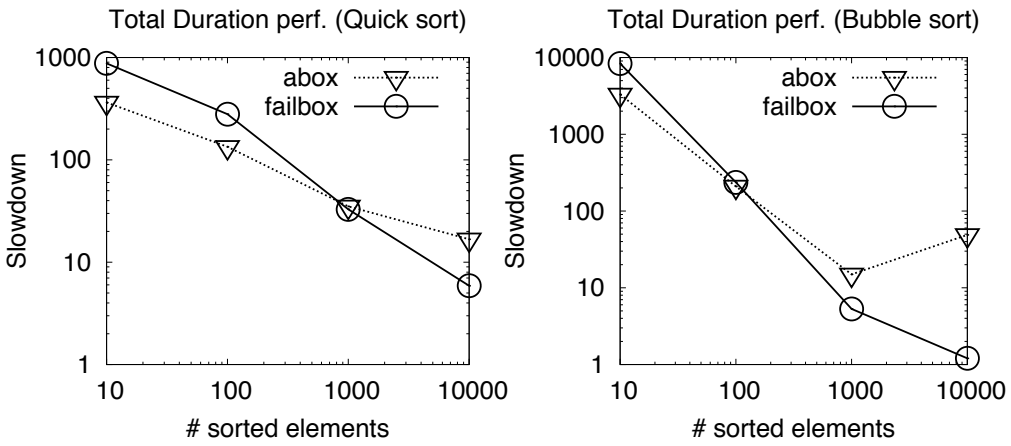


Figure 7.9: Comparison of the total duration time of atomic box (mentioned as abox) and failbox (note the logarithmic scales on both axes).

7.8 Summary and conclusions

In this chapter, we introduced language constructs for concurrent exception handling, more specifically for handling exceptions of multi-threaded software in a concurrent manner. The key novelty of the proposed approach is to ensure that any inconsistent shared state resulting from an exception cannot be accessed by concurrent threads, allowing the programmer to safely define concurrent exception handlers. The alternative failbox [106] language construct that prevents threads from running on inconsistent states simply stops all threads. Letting

the programmer define concurrent exception handlers allows us to recover rather than stop. Using the recovery option, the programmer can remedy the cause of an exception and retry the concurrent execution.

We have covered in detail the syntax and semantics of the proposed language constructs, the `__transaction-recover` and its accompanying constructs, as well as presenting an overview of the implementation for supporting the atomic box approach. Our implementation has been realized as an extension to TMJava (see **Chapter 6**) that converts `__transaction` blocks into code that uses an underlying STM runtime. Our preliminary evaluations indicate that the overhead of the code we introduce for exception handling and transactification is low such that when encompassing up to hundreds of elements, `__transaction` blocks execute twice faster than failboxes. These evaluations demonstrate that our approach is quite efficient in preventing concurrent threads accessing inconsistent shared state, with the added advantage of handling exceptions in a coordinated manner among threads.

Chapter 8

Conclusion

Before the beginning of the millennium, concurrent programming was mainly confined to a restricted community of researchers and engineers. This fact has abruptly changed in the past decade with mainstream processor architectures converting to multi-cores. Today, concurrent programming has become a necessity even for mainstream application programmers.

This paradigm shift in programming requires programmers to start designing applications such that they have as many concurrent components as possible to take advantage of the hardware parallelism. Thus, existing sequential algorithms need to be rethought and partitioned to work in a concurrent manner whenever possible. This is one of the tough tasks awaiting the IT community in the near future. However, before even accomplishing this task, we have another problem to overcome: finding ways to code correct concurrent programs in acceptable time frames. If we fail to do that we cannot keep up with the current high demands on software development. That is why, the IT community is currently investigating ways to solve this problem. Transactional Memory (TM) has been proposed as a significant step in this investigation and research on TM is evolving rapidly to integrate TM into current languages, so that it can be actively used for developing concurrent programs.

The objective of this current work was to contribute as much as possible to this integration process. To that end, in this thesis, we have studied extensively the comprehension of TM semantics both at the run-time and compile-time. Our work has also resulted into tools that allow researchers and engineers to explore different transactional semantics easily, without neglecting the performance overhead associated with the semantics. We have also explored different ways to exploit the transaction abstraction other than data synchronization and we have demonstrated one such approach that allows coordinated exception handling among multiple threads.

We believe that the research conducted for this thesis work has important implications in TM research and concurrent programming in general. TM research has gone a long way, but it is still not widely accepted as a concurrent programming tool, mainly due to its run-time performance being considered unsatisfactory (this is especially true for STM). However, the transaction abstraction provided by a TM provides the simplicity a programmer would need to design concurrent applications in time frames close to the ones of sequential applications. Hence, it is certain that important research will be carried out to find solutions to the performance efficiency of TMs. Such research will need to introduce new mechanisms to TM designs and TMs proposed by this research should still satisfy semantics required by languages and their performance advantages should be demonstrated. Moreover, whatever the mechanisms a TM incorporate, its integration to a programming language is a must for its widespread use. The tools provided in this thesis allows the quick satisfaction of all these requirements (semantics, performance and integration), thus these tools are crucial for demonstrating capabilities of TMs and, as a consequence, their being widely accepted.

In the rest of this section, we detail the outcomes of our research and present possible future directions.

8.1 Outcomes

The research we conducted during this thesis work, resulted in outcomes on both levels of support the TM-based programming requires: run-time and compile-time.

8.1.1 Run-time support outcomes

With the high interest shown in STM over the last years, there is a large body of work performed to improve STM designs especially to obtain efficiency in performance. The classification presented in **Chapter 3** provides a good picture of the STM design space in terms of abort performance. With this classification we observe clearly that simpler designs provoke more aborts than necessary while more complex and costly designs (mostly) abort when application correctness can no more preserved. Neither the simplest nor the most complex design prove to be the most efficient and our classification provides a spectrum of STM designs illustrating both the extreme cases of simplicity and effectiveness as well as the mid-way alternatives between them. This way, we expect the classification to help finding the sweet spot in STM design space where the balance between simplicity and effectiveness results in an STM with optimal efficiency.

In **Chapter 4** we have presented the first to date test-suite for transactional memory. This test-suite is completed with a testing framework for TMs, TMUNIT. With this test-suite and TMUNIT we offer TM researchers a unique tool that allows *(i)* testing and debugging of TM designs, *(ii)* experimenting both semantic and performance aspects of TMs thanks to a novel domain-specific language, and *(iii)* comparing semantic and performance characteristics of TMs thanks to an abstract TM interface. TMUNIT is available online (`http://tmware.org/tmunit`) and comes with its test-suite. To show the effectiveness of TMUNIT, we have performed initial experiments on five TMs and various CMs to compare their behaviors with theoretical expectations. We identified TMs violating SLA (single-lock-atomicity) and opacity and CMs violating progressiveness.

Apart from the fact that TM developers can use TMUNIT for verifying the behavior and performance of their TM, we envisage other uses. For example, the application developer can verify if his/her assumptions are satisfied by the underlying TM. This could be used to select the most efficient TM variant that still satisfies the application requirements.

8.1.2 Compile-time support outcomes

In **Chapter 5**, we provide the first language extension for Java introducing necessary language constructs related to the transaction abstraction. With the specified language extension Java can support *(i)* automated data synchronization among concurrent threads, *(ii)* transactional control flow, and *(iii)* enhanced exception handling. Although a Java derivative (the Atomos programming language [33]) and a language extension specification for C++ exits, the specification we provide for the Java language offers a larger set of features to the programmer with a simple syntax, in particular for transactional control flow and exception handling.

Our work also resulted in the TMJava pre-compiler, explained in **Chapter 6**, that effectively implements the transactification of the syntax described by our language specification. The TMJava implementation, being based on the extensible Java compiler Jastadd, can easily be modified or extended to incorporate new features into the language extension.

Finally, as part of our language specification we have also defined novel constructs, namely atomic boxes (in **Chapter 7**), for concurrent exception handling; a way to handle exceptions in a concurrent manner for multi-threaded software. These constructs ensure that any inconsistent shared state due to some exception cannot be accessed by concurrent threads. The key novelty of these constructs, however, is the ability of defining concurrent exception handlers which in return allow all threads concerned with the exception to participate recovery. The alternative failbox [106] language construct that prevents threads from

running with inconsistent states simply stops all threads. On the contrary, atomic boxes prevent the access to inconsistent shared state signaled by an exception not by stopping execution but by automatically bringing the application back to consistent state so that the application can continue running safely. It is based on this guarantee that atomic boxes further allow the definition of concurrent exception handlers and recovery from exceptions on multiple threads. Consequently, atomic boxes give the programmer the ability to control the concurrent application as a whole (rather than controlling a single thread) upon an exception concerning the whole application. This way, the programmer can remedy the cause of an exception and let the application continue execution (e.g., by retrying the last actions performed before the exception was raised).

8.2 Future directions

The outcomes we have obtained at the end of this thesis indicate several paths for improvements and new directions.

The classification we presented in **Chapter 3** should serve as a good starting point to better understand the tradeoff between design simplicity and effectiveness. By describing the different classes of unnecessary aborts, the classification illustrates important access patterns an STM design should consider. An interesting research direction that could be taken is to identify which of these patterns occur more frequently in real applications and, hence, determine which of them should imperatively be supported to obtain efficient STMs. Additionally, identifying which of these access patterns occur more frequently at run-time could be used for adapting (or reconfiguring) an STM to support different combinations of these patterns according to changing workload and result in more efficient executions compared to a non-adaptive STM.

We also think that the versatile nature of TMUNIT can help improving TM research in many aspects. We enumerate some of them below:

1. An extension that can be added to TMUNIT is the inclusion of additional operations that correspond to special statements; such as memory allocation/deallocation, irrevocable statements, task synchronization primitives, raising of exceptions, statements specific to transactions (e.g., the transactional control flow statements explained in **Section 5.4.2.2**). This will allow much richer semantic properties of STM designs to be explored, and new pathological scenarios to be discovered.

2. TMUNIT and the associated domain-specific language can be extended to design tests based on objects rather than memory locations. A natural follow-up of this

extension will be the code generation for object-based STMs. This will allow the testing of a larger set of STM designs with TMUNIT.

3. TMUNIT code generation can be elaborated to generate code in C++ and Java (including the transactional language extensions). This extension together with the previous two extensions, will allow the design of test cases where the performance of the transactional language constructs are evaluated. Moreover, such generated code can be used to rapidly design test cases for the Deuce bytecode instrumentation tool (see **Chapter 6**).

4. TMUNIT can be extended such that the labels that serve as interleaving points in deterministic schedules can be inserted in any multi-threaded program (as is done for TM library code in **Section 4.3.3.3**) and the desired schedule of the program to be expressed in a TMUNIT configuration file. This will allow the semantics of transactional language constructs (either the C++ or Java language extension constructs) to be tested especially in interesting cases such as *(i)* the interaction of transactional language constructs, and *(ii)* the interaction of code including transactional constructs with non-transactional code. Of course, such an extension also allows any multi-threaded code to be tested under deterministic scenarios.

5. Currently the semantic tests of TMUNIT are limited such that only one thread can be executing at a time. We also propose extending the schedule expressions such that multiple threads can execute desired portions of their code (e.g., delimited by labels) at the same time. This will allow programmers to design semantic tests with TMUNIT where parts of the STM design is of blocking nature (e.g., support for publication, privatization or irrevocability). Together with the previous extension, any semantic property of any multi-threaded code can be tested.

6. The deterministic scheduling engine of TMUNIT can be improved to simulate operating system schedulers and allow TM designers to explore different contention management policies under different system loads.

7. Currently TMUNIT only accepts configuration files written in the associated domain-specific language as its input. An alternative input format can be trace files (especially ones including transactional accesses) that represent memory accesses of real applications. An extraction facility can be added to TMUNIT for the collection of such traces from real application executions. Such traces can be used as realistic workloads on which the performance of all TMs that are adapted to TMUNIT's abstract

interface can be compared. Moreover, an analysis on these traces can also reveal frequently used memory access patterns in multi-threaded applications, which can later be added to the test-suite for the comparison of different TMs under such memory access patterns.

As for the integration of language constructs for TM into programming languages, there are still unresolved issues concerned with the interaction of irrevocable statements with transactional constructs and of transactional constructs with other task synchronization primitives. With the current TMJava implementation and the existing framework, it is possible to easily try our different syntax possibilities and test the interaction issues on real examples. Apart from this, our novel transactional control flow and exception handling constructs have proved that there is still room to exploit the transaction abstraction in programming languages.

The preliminary evaluations of our experiments with our atomic box solution for concurrent exception handling indicate that the overhead of our transactified code is low such that when encompassing up to hundreds of elements, `__transaction` blocks execute twice as fast as failboxes [106], the closest alternative solution. This result implies that the transactional memory overhead does not significantly impact the concurrent exception handling and it should encourage further research in this direction. The atomic box approach could for example benefit from ongoing progress in hardware and hybrid transactional memory to reduce this overhead further, as our current implementation is purely software based. Even though there is a long road before integrating such language constructs in Java, we believe that exploring transactional memory as a building block for concurrent exception handling will raise new interesting research challenges and offer new possibilities for programmers. To that end, it will be enriching to test our atomic box language constructs in large applications in order to *(i)* further analyze the performance of atomic boxes and *(ii)* experience the simplicity and expressiveness of these constructs.

Appendices

Appendix A

A Survey of STM Designs

Since Shavit and Touitou [164] initiated the STM research in 1995, there has been a large body of published work resulting in a variety of STM designs. With the growing size of research performed on STMs, one can easily lose sight of the big picture and fail to identify the ideas that make an STM design efficient. In order to avoid this, in this chapter, we delve into the large body of STM research, qualitatively analyze STM designs and organize their components in a comprehensive manner.

Sections A.2–A.5 present the result of our qualitative analysis where we explain the data structures and mechanisms STM designs use to fulfill their functionality. For ease of comprehension, mechanisms are classified by their purposes in STM design, specifically by the property of STM they ensure. Since STMs target being efficient, it is possible to observe that some mechanisms serve multiple purposes at the same time. In such cases, we explain the notions of the mechanisms that relate to the analyzed property and, hence, the same mechanism can appear in different sections with their different aspects.

A.1 Overview of STM designs

An STM design aims to implement the transactional semantics in terms of the transactional properties discussed in **Section 2.3**. Considering an STM from a design perspective, it is also possible to consider this transaction semantics as an atomic `read-modify-write` operation performed on multiple data items (memory words or objects). While atomic `read-modify-write` on a single data item (generally a memory word) is readily available as a hardware instruction, providing the same semantics on multiple data items cannot be performed as a single step (such as a hardware instruction) in current computer systems. Hence, in considering STM functionality as a `read-modify-write` operation, each of its

read, modify and write components should be viewed as phases rather than single step sub-operations:

- The read phase of an STM corresponds to achieving an atomic snapshot of data items read by the transaction. This snapshot is taken such that it is valid throughout the transaction and thus ensures that all the data items read by a transaction are consistent.
- The modify phase corresponds to the actual calculation performed by the transaction. This phase is the least complex one since calculation is performed on the processor and can be performed on local data.
- The write phase corresponds to an atomic update performed on multiple data items.

Of course the read-modify-write abstraction we presented so far is not enough to explain the functionality of STM designs. STM designs have the following particularities on top of being a read-modify-write operation:

- The read, modify and write phases of STM designs are not successive phases but rather phases that are overlapped and that span significant fractions of transaction execution time.
- The set of data items read and modified by a transaction can be different (it may be possible to say that all data that a transaction updates should at least be read once, however it is not possible to claim that all data items read by a transaction should be finally updated by the transaction).
- STM functionality is speculative in nature. Hence, all the 3 phases can be aborted at any time (all together) and restarted from the beginning.
- STMs are supposed to guarantee a certain level of progress, which should be higher than the one ensured by lock-based code. While some STMs use lock acquisition and release in their implementation and are prone to problems due to blocking (see **Section 2.2**), there exists STMs that provide stronger progress guarantees such as obstruction-freedom or lock-freedom (see **Section 2.4.5**).

It is possible to relate the `read-modify-write` operation view of a transaction with the classical transaction semantics describing it in terms of atomicity and isolation properties (see **Section 2.3**). Here, we assume atomicity and isolation describe complementary interference properties of transactions as explained in **Section 2.4.1**. In other words, we assume that atomicity enforces that the effects of a transaction T does not interfere in the execution of other transactions until its commit, while isolation enforces that the effects of other transactions do not interfere in the execution of T until they commit. When described this way, atomicity property of a transaction is satisfied by the implementation of the write

phase and the isolation property is satisfied by the implementation of the read phase (as mentioned before the modify phase can be performed locally so no synchronization issues are to be considered for this phase).

The transaction semantics described above require an STM to incorporate appropriate data structures and mechanisms. In the following sections we analyze all these design components (data structures and mechanisms). First, in **Section A.2**, we describe data structures that STM designs adopt. Then we explain the mechanisms that ensure atomicity and isolation in **Sections A.3** and **A.4** respectively. Mechanisms that ensure STM progress are presented in **Section A.5**.

A.2 STM data structures: Metadata

In a concurrent system, while multiple threads can concurrently access a shared data item for reading, only one thread can be allowed to write to the same item at a given time. In other words, a thread should guard a data item it modifies from other threads by obtaining exclusive access to the item (obtaining this exclusive access is also called *acquiring* the ownership of the data item) and such thread is usually called the *owner* of the data item. STMs follows the same principle; they delegate the ownership to transactions instead of threads, and store the ownership information of a data item in data-specific metadata. We call this metadata **ownership record**[1] (orec for short) and the transaction that has the exclusive access to the data item through the orec the **owner transaction**.

Apart from managing ownerships for data items, STMs need to support rollback, a particular property of the transactional execution. Due to this rollback notion, a transaction needs to know at least two states of the data items it accesses: the state at the moment the transaction first accesses the item, **initial state**, and the state during the transaction execution, **tentative state**. The initial state is required to restore the data item upon rollback, and the tentative state is needed to store the modifications on the data item that should eventually be made effective in the program memory upon a successful commit.

While an STM should store such **dual state** (initial and tentative states) for each data item it modifies, it should further be able to keep track of modifications performed by other transactions on the same data items. For this purpose, STMs use the notion of versions. A **version** is a valid state of a data item that is accessible by all threads, i.e., it corresponds to a state of the data item that is made effective by a commit. This implies that a data

[1]While STM papers may consider only part of all ownership related information as ownership record, we prefer to capture all ownership related information under this name.

item transitions from one version to the other as a result of a transaction commit. In that sense, a version can also be called **committed state**.

Each STM makes use of a collection of data structures, usually named **STM metadata** (or metadata for short), to store information it requires. Orecs together with the the dual state and the version information of data items constitute the data-specific information a transaction requires to manage. We call the collection of such information **data-specific metadata**. Naturally, STMs cannot manage transactions only by using the information they store for each data item. STMs also need to store data to represent transactions, i.e., to describe the state and the view of a transaction. The metadata STMs use to represent transactions is called **transaction-specific metadata**.

Transaction-specific metadata is generally assembled in a data structure named **transaction descriptor**. This data structure has at least a field indicating the status of the transaction which can be in states such as *committed*, *active* (i.e., not yet committed) and *aborted*[1] . Another usual part of a transaction descriptor is composed of the **read-set** and the **write-set**. These sets represent respectively the read-only and modified data items a transaction has accessed so far. The sets can either store pointers to data items or the data items themselves depending on the organization of STM design. Transactions that have an empty write set are named **reader transactions** (or read-only transactions) and are distinguished from other transactions since they are generally much simpler to manage. The rest of the transactions, i.e., transactions having an non-empty write set, are called **writer transactions**.

Metadata is out-of-band information that is not part of the application being synchronized using STM. Hence, it represents *(i)* an artificial increase in the memory footprint of the application, and *(ii)* a performance overhead since STM needs to access and manage it for providing data synchronization. This intervening nature of metadata imposes a careful organization. One consideration that needs attention is the effect of metadata organization on the cache performance: the metadata organization should not harm the locality of the application data. Simple solutions are shrinking the metadata size and organizing metadata in the memory in order to preserve application data locality (e.g., metadata related to each shared data could be on the same cache line as the data). A second consideration is the efficiency of accessing metadata. Solutions to this are *(i)* placing metadata in a contiguous memory location as the data, whenever it is possible, or *(ii)* providing fast access to metadata through simple mappings between the data location and the corresponding metadata location. Although considerations on the metadata organization may seem to be

[1] the 3 states of a transaction status is the minimal number of states observed for STMs. Some STMs may require more states due to their design requirements.

minor implementation details, it was recently shown by Dalessandro et al. [40], that this organization directly effects the performance overhead incurred by STMs and, hence, is a critical design issue. Below, we analyze different aspects of STM metadata under the above considerations.

A.2.1 Unit of shared data

A fundamental decision in STM metadata organization is the unit of shared data for which versions will be tracked by the STM. Since multi-threaded programs access data of diverse sizes and types, generally STMs simplify their designs by defining a single unit of shared data. Some STMs support a memory word (**word-based STM**s, e.g., [89, 42, 57, 40]) or a fixed sized memory region (**block-based STM**s, e.g. [42, 47]) as a shared data unit while others use program level objects (**object-based STM**s, e.g. [97, 63, 53, 127]). Although rare, there also exist STMs that can support both words or objects but not during the same execution (i.e., at run-time the STM uses one or the other data unit but the decision of the data unit type can be given prior to run-time) [42, 158].

STMs track versions for each unit of shared data accessed by transactions and, hence, the granularity of the chosen unit has a significant impact on performance. With object-based STMs, the number of versions to track is kept low, however since a version is used to track a memory region rather than a memory word, accesses by different threads to different fields of the same shared object will give a misleading impression that there is a version change for the exact same location in memory, while semantically different data items are accessed. This phenomenon is just a side-effect of the unit granularity. The same problem exists with block-based STMs with the additional difference that the memory regions tracked by block-based STMs may not correspond exactly to object locations and unrelated object accesses may also give the impression that the version of the same data item has changed. Word-based STMs track words and are not prone to this problem. However, since they track words, the number of versions to track can explode. Hence, they generally use hash tables to map several addresses to a hash table entry and track the hash table entry (and not the distinct memory words that map to the entry) for version changes. With this optimization (for memory space), all the words that correspond to the same hash table entry, although not having any relationship at the programming level, can be perceived as going under a version change even if only one of the words has actually been modified, just as a side-effect of the data structure used.

Spear et al. [127] suggest that using object-based STM can be more suitable since the metadata for an object needs to be fetched only once in the lifetime of the transaction,

while the metadata needs to be fetched at each read or write access for word-based STMs. However, a quantitative comparison of word-based and object-based STMs by Saha et al. [158] depicts that neither choice outperforms the other in a significant extent [1].

A.2.2 Accessing metadata

Each of the read and write accesses inside a transaction requires that the STM consults metadata corresponding to the accessed data items in order to know about ownerships or to track versions. There are two approaches proposed for accessing metadata:

- *Mapping:* The STM stores a map data structure that maps the address of an accessed data item to its metadata. This approach is mostly used by word-based (or block-based) STMs. The mapping between the address of a word and the corresponding metadata is generally simple for allowing efficient access. A recurring map used for metadata is a hash table. The hashing function generally uses simple modulo arithmetic to transform the input address to a slot in the hash table. The use of hash table results in a single metadata entry to be shared by multiple data items. Such metadata sharing may complicate version tracking for data mapped to the same metadata entry. However, the hashing function is generally organized such that the frequency of such sharing is highly reduced (e.g., by using a large modulo number).
- *Indirection:* Instead of accessing the data directly, the STM accesses pass through some other data structure allocated for each accessed data item. This data structures allow the access to all metadata information as well as the original data itself. This approach is preferred by object-based STMs since it is not possible to define a simple mapping function between object addresses (determined at run-time) and corresponding metadata. This data structure is also convenient for object-oriented programming since it can be organized as an object and the programmer can access this structure instead of accessing the object directly. Such data structure is named **object header**[127]. There are two approaches in the way an object header allows access to the original data: either object header has a field pointing to the original data object [97, 63] or it wraps around the original object by adding up metadata information [53, 127, 181]. In recent STMs, the latter approach is preferred since it takes advantage of cache locality by concatenating the original data and the metadata. This locality is important since an access to data requires generally both metadata and original data accesses.

[1] The comparison is based on benchmarks stressing hash table, binary tree and linked-list data structures. While for linked-list word-based STM perform by about 20% better, for other data structures it performs about 10-20% worse with respect to object-based STM.

A.2.3 Storing dual state

Since STM needs to store dual state for each accessed data item, metadata should be organized to include dual states. For efficiency in memory use, STM designs usually reuse the original location of the data item (thus not the memory used for metadata) to store one of the dual states[1] . The dual state not stored in the original location of the data item is part of the metadata and we call this state **logged state**. In general a list (hereafter called a *log*) is used in metadata to maintain this logged state. If the stored logged state corresponds to an initial state, the list is called an **undo-log** (when the transaction rolls back all the modifications performed on the original location can be undone using this log), otherwise it is called a **redo-log** (when the transaction is being committed all the logged states are actually transferred to the original location, in a way, modifications are redone during commit). There are two approaches in the design of logs: *per-transaction* and *per-data*.

- *Per-transaction logging* stores the logged state of data items accessed by a transaction as elements of a list. For undo-logs there is no need for storing read accesses, while for redo-logs read and write accesses are stored in separate logs. This approach is generally used in word-based STMs. It is also possible to see in some implementations that the undo- and redo-logs are merged with read and write sets for efficiency in accessing metadata.
- *Per-data logging* stores the logged state individually, i.e., the logged state is not an element of a log, but rather part of the metadata allocated for data. This approach is used mainly by object-based STMs where the metadata points to the logged state by a pointer. With this approach there is no need to separately maintain a list for undo- or redo-logs.

A.2.4 Managing ownerships

As explained in **Section A.2**, ownership-related information for data items is stored in orecs. The information an orec contains should hence include *(i)* the ownership status; i.e., whether the associated data item has an owner, and *(ii)* the current owner of the data item at the time the record is consulted (if the data item is owned). The ownership

[1] DSTM [97] and its successors do not totally fit to this description. However, DSTM uses the metadata to access the data item in both transactional and non-transactional code, hence the access to the original data is always done through metadata, contrary to many follow-up STM designs. Thus, although both dual states are part of metadata, one of the dual states is used for multiple purposes (both for transactional and non-transactional code accesses).

status can have two states: *acquired* (i.e., owned) and *released* (i.e., not owned)[1] . The current owner information usually serves to store the owner transaction rather than the owner thread since ownerships are managed by transactions in STMs. These two pieces of information (the ownership status and current owner) can be stored in the same memory word, either by attributing a special value to the released ownership status (a value that is invalid as a transaction identifier) or by using part of the word (i.e., a single bit) to indicate the ownership status.

Locks are also data structures widely used to manage ownerships and generally store similar content as an orec. This fact has led many STM designers to use locks as orecs[2]. Such STM designs are called **lock-based STM**s. The main difference in using a simple data structure or a lock as an orec is that the simple data structure can be modified by any transaction/thread at any time whereas a lock can be only released by the transaction/thread that has acquired the lock. This difference makes lock-based STMs blocking (because the release of an ownership depends on a single transaction/thread in the system) and, hence, the STM designs not using locks as orec are usually called **non-blocking STM**s.

Orecs are shared among transactions (thus among threads) and, hence, the read and write accesses on the orecs by concurrent threads should be synchronized. Since an orec can usually be fit in a single memory word, it is enough to perform atomic updates on ownership records to achieve such synchronization. Such an atomic update can be done by hardware instructions such as compare-and-swap (CAS). However, the use of such instructions is costly compared to a simple memory word update. In that sense, orecs and mechanisms using them should be designed carefully, so that the number of orec modifications is minimized.

A.2.5 Tracking versions

The isolation property of transactions requires that a transaction accesses always the same version (the initial state) of a data item throughout the transaction as long as the transaction itself does not modify the data item. However, between two accesses, the version of a data item stored in memory can be modified by other transactions and, if care is not taken, such changes can result in inconsistency in the transaction execution. To avoid such inconsistencies, an STM should track version changes of the data items and, hence,

[1] In some special situations, such as ensuring irrevocability (see **Section A.4.6**), an orec can also be used to guard read data items. However, the guarding required for read accesses is not as strict as the one for writes and, hence, in such situations, there can be two acquired states, one for each type of guarding.

[2] Since an orec and a lock are used for the same purpose, in considering the data structure managing ownership of data items we prefer to call this data structure an orec even if it is actually a lock.

should store some version information. The version information an STM stores can be of various types: version content, version location, version number, global commit counter and timestamps.

Version content is the most straightforward information to store for tracking versions of a data item. However, this information is not enough by itself due to the ABA problem. The **ABA problem** is the situation where between two samplings of the same data item yielding the same value A, another concurrent thread modifies the value of the location twice, first to a value B, then back to the original value A. Such concurrent writes go unnoticed by the sampling thread which deduces that the data item has not been modified. In order to avoid the ABA problem, an STM storing version content for tracking versions should ensure that no modifications has been performed on the data item between the two samplings of the data item (i.e., between the time the version information is stored until the time where this stored information is consulted for tracking). This approach is used by JudoSTM [143] and NOrec [40].

Version location information identifies different versions by their memory locations. It should be noted that the mapping between versions and memory locations is valid only if each version is stored in a unique (and separate) location in memory and the location is immutable. This approach suits well object-oriented STMs where shared data items are objects. In such STMs, both initial and tentative states of shared objects are stored as separate objects that are allocated by memory manager of the system. Since memory allocation provides immutable and unique location for each allocated object[1], each version of a data item can be identified by its memory location. Hence, only by comparing memory locations of different versions of a data item, it is possible to track versions. Some STM designs using this approach are DSTM [97], SXM [78] and RSTM [127].

Version number information associates a version number to each version of a data item. Tracking versions with this information boils down to compare the values of version numbers obtained for different accesses of the same data item. STMs using this approach (e.g., McRT-STM [158], Bartok STM [92], WSTM [89], Ennal's STM [53]) store a version number that indicates the number of modifications applied on the data itself. This kind of version information is also called **local version number**, in the sense that version numbers are stored and managed independently for each data item [90]. The per-data organization

[1]The memory location associated to a deleted object can of course be reused for memory allocation, but if a memory location is freed for use in memory allocation we assume that no valid version of any data item resides in that memory location, thus valid versions of data items are always stored in unique memory locations. In general, this is a valid assumption since STMs do not allow a data item to be deallocated if inconsistencies would occur due to accesses concurrent to the deallocation.

of version numbers has the advantage of allowing parallel accesses to version information via its distribution.

Contrary to other approaches, using version numbers requires additional management of the version information, i.e., the version number should be updated each time the data item is modified (however this update is generally cheap because it corresponds to an increment). Although such an additional operation should be performed, this approach is dominant for word-based STMs. Probably, the reason for this preference is that for word-based STMs data-specific metadata is already separate from the data item itself and metadata should be associated with the data item anyway. So the simplest information that would reveal different versions is adequate for word-based STMs.

Global commit counter information records the number of commit attempts of all writer transactions (even if the commit attempt does not succeed). The increase of the commit counter between two samplings of the same data is an indication that *some* data (not necessarily the one we sample) have been under modification because some commit have probably made modification of these data effective (here we use the expression *probably* because the commit counter also counts commit attempts that do not succeed and the increase in the commit counter is a false indication as far as version tracking is concerned). Such indication can be used to decide whether there is a need to track versions or not. If global commit counter indicates no changes, this helps to avoid further (and costly) investigation on whether a specific data item has been modified or not. Thus, the use of global commit counter as an additional version tracking information introduces a performance improvement to the version tracking mechanisms of STMs (as shown for RSTM [172]).

Some STM designs have noticed that the information in global commit counter can also be used in a more powerful way: the global commit counter is always updated at instants of the application execution where there is actually a valid version change (for the write-set elements of the committing transaction). In that sense, the global commit counter acts as a clock advancing, slower than the real time clock but, fast enough to keep up the version changes in the execution. STMs using global commit counter information with this interpretation call it a **global clock**. In general, in such STMs, a sampled value of the global clock is called a **timestamp**. We further define a timestamp as a **commit timestamp** of a transaction if it corresponds to the value of the global clock right after the global clock is updated to indicate the commit of the transaction.

Timestamp information is generally used as a replacement to version number information in STMs using a global clock. In other words, a timestamp value is stored for each data item (STMs using timestamps as a version information are called **time-based STM**s). As it is done for version numbers, timestamp information is updated each time a data item

is updated. More specifically, each time a transaction commits, the timestamp information of all the data items modified by the transaction is set to the commit timestamp of the transaction. This way, timestamp information stored for data items always reveals the timestamp at which the data item has been modified for the last time and, hence, such information can safely be used to track versions. Examples of STMs using this approach are TL2 [42], LSA-STM [153] , tinySTM [57] and SwissTM [47].

A.2.5.1 Versioned orecs (Versioned ownership records)

A significant number of STMs (especially word-based STMs) use a version information that can be stored in a single memory word, such as a version number or a timestamp. This fact has resulted into an optimized metadata organization where version information shares the same memory word with an orec (which also fits in a single memory location). Such metadata organization, which we call a **versioned orec** is based on the following observation[1] : *Shared data can be in two states: either owned by some transaction or not owned by any transaction. When data is owned, any transaction accessing the metadata needs to know the owner transaction to decide what step to take next. In the case where the data is not owned, a transaction requires to know which version of the data item it is accessing (i.e., it needs to consult the version related information).*

According to this observation, a versioned orec is organized as follows: since it is possible to indicate the state of shared data with a single bit (this is usually chosen to be the least significant bit), the rest of the word is used to store either owner transaction information or the version-related information depending on the state of the shared data.

By using a version orec, STMs speed up the way they access both the ownership state of data and the information they require according to the ownership state. Also, by putting several metadata information into the same memory word, the memory footprint of metadata is reduced, resulting in decreased cache pressure.

A.2.6 Managing transactions

The metadata that has been discussed until this point are related to the data items that are managed by transactions. However, apart from the data items, transactions should also be managed. Information that should be stored related to a transaction is mainly the transaction status and the read and write sets. The management of all this information is generally simple. However, it should be noted that the status of a transaction can be modified also by other transactions and, hence, needs to be accessed using synchronization.

[1] If the orec is actually a lock this metadata organization is called a **versioned lock** [90].

Since transaction status can easily fit in a single memory word (it can be determined using only several bits), accessing transaction status under concurrency is possible using atomic access instructions provided by hardware.

A.3 Mechanisms ensuring atomicity

Atomicity requires that the execution of a transaction is perceived as an indivisible step by other transactions. Since what other transactions perceive is only the updates performed by a transaction this implies that the updates to be performed by the transactions should be perceived by other transactions as if they were performed all at once. This behavior is possible thanks to the interaction of multiple mechanisms:

- Ownership management
- Storing updates
- Canceling updates
- Making updates effective

In the following subsections we discuss each of these mechanisms in more detail.

A.3.1 Ownership management

A transaction desiring to update a data item should first acquire the permission to update it, i.e., the transaction should acquire the ownership of the data item by modifying the corresponding orec. However, it is not enough for a transaction to acquire the ownership of a single data item to have the it updated on the memory. A transaction is allowed to make updates of its write-set elements effective only if it owns all the elements of the write-set (see **Section A.2** for write-set). There are two approaches a transaction can take to acquire the ownership of all the write-set elements: either acquire ownership during the first write access to the data item (**eager acquire**[1]) or acquire all ownerships during commit operation (**lazy acquire**[2]).

Both eager and lazy acquisition have their advantages and disadvantages. The major issue with the acquisition approach is the duration during which the ownership is held by a transaction. Obviously, eager acquire has tendency to hold ownerships longer than lazy acquire. Depending on where the write accesses occur within the lifetime of the transaction this duration can even span most of the transaction lifetime. The longer the duration the higher the probability that two transactions contend for the same data item for modifying

[1] Eager acquire approach is named as **encounter-time locking** (ETL) in lock-based STMs.
[2] Lazy acquire approach is named **commit-time locking** (CTL) in lock-based STMs.

it. Since an ownership can be held only by one transaction at a time, concurrency is harmed by holding an ownership for a long time. Hence, eager acquire approach can decrease the concurrency of transactions accessing common data while lazy acquire approach would allow higher concurrency. However, the early acquisition allows also some optimizations on how the updates for each data item are made effective. The resulting effect of the acquisition method on performance, thus, generally depends on the workload.

Once a transaction owns all the elements of its write-set, it should make sure that this situation continues to hold until it makes the modifications of the write-set elements effective on the memory. However, in non-blocking STMs, making sure that all write-set items are still owned is not always evident, because ownerships can be released upon the request of other transactions (an ownership can also be released by the owner transaction at any time, but a transaction willing to make its updates effective would of course avoid that). If another transaction requests to acquire some data item already owned by transaction T, it is possible that T is aborted. In non-blocking STMs, when T is aborted all its ownerships become invalid and other transactions can subsequently acquire the ownership that were previously owned by T. This implies that T can be sure that it still owns all the write-set items by verifying that its transaction status is not *aborted*. Consequently, in non-blocking STMs, transaction status determines whether a transaction still owns all its write-set elements.

Making sure that all write-set items are owned is simplified in lock-based STMs because the ownership records are locks, and once T acquires the lock for all write-set elements, it can be sure that no other transaction can attempt to make changes on the acquired data items, because in lock-based STMs only the transaction that owns a lock (hence only T) can release it. Hence, even if another transaction causes T to abort, it will not be capable of releasing the locks of T. Consequently, once T owns all the locks of its write-set elements, it can modify the corresponding data items safely until T itself releases the locks. In other words, for lock-based STMs, in the time window where T owns all the locks of its write-set elements, it is safe to make any changes on owned data items. This, however, does not mean a lock-based STM would not take into account that T is aborted during this time window. On the contrary, this time window will come to end upon an abort since T restores the initial states of all write-set elements and release all the associated ownerships.

A.3.2 Storing updates

An important decision in the STM design is the location where the tentative modifications performed by the transaction are stored. There are two major methods to store the tentative modifications: *direct* and *deferred update* methods[1] .

Direct update method applies modifications performed by a transaction directly on the shared data items. This implies that separate modifications that should become visible to other threads all at once are dispersed along the transaction execution. Since modifications are applied on the data items themselves, updates are automatically made ready after the last write access of the transaction. This allows the commit operation under direct update method to be lightweight. However, the same fact requires the direct update method to use eager acquire approach for holding exclusive ownership of its write-set items [90]. As a consequence, the concurrency between transactions is reduced with direct update method. Hence, this method performs well when shared data updates are seldom, i.e., when transactions commit most of the time. Although appealing for some kind of applications, few STMs apply direct update method [158, 92, 57, 126].

Deferred update method generates, unlike direct update, a private copy of each data item that needs to be modified (the collection of all these private copies constitutes the redo-log) and performs all the modifications on these private copies. The modifications are made effective only during commit: in lock-based STMs, the contents of private copies are written back over the actual data items during commit operation, while in non-blocking STMs, the private copies become the valid versions of the data upon commit. Since the effects of the private copies become effective only during commit, both eager or lazy acquire approaches can be used to obtain exclusive ownership of the write-set items. The possibility to use lazy acquire allows this update method to have increased concurrency among transactions. Thus, deferred update method (together with lazy acquire) is interesting for applications where there is frequent demand on updating shared data items. A large number of STMs use this update method probably because the advantage of using STM is more visible in such workloads when compared to lock-based code.

A particular performance critical issue in deferred update method is the need to access earlier modifications performed by the transaction for reading (i.e., a read access inside the transaction needs to return the modified value if the transaction has already tentatively modified the data). Such requirement implies that each time a read access is performed, the STM should check whether it has modified the data searching through its write-set. Taking into account that the ratio of reads is in general much larger than the ratio of writes

[1] It is common to call the methods storing tentative modifications update methods.

in applications, such requirement represents a substantial overhead for deferred update method. Several solutions exist to decrease the incurred overhead [90]:

- An auxiliary look-up table can provide a mapping from a data location to a previously stored redo-log entry [92, 90, 170].
- A summary of the write-set can be maintained and the redo-log is searched only if data being read is found in the summary. Common practice is to use a Bloom filter to store the summary of the write-set [90, 42, 57].

By applying modifications on data, the update method causes data items to transition from one version to the other. For this reason, update method is also called **version management** to refer to the way versions are transitioned. Direct and deferred update methods are, hence, also known as **early version management** and **lazy version management**, respectively.

A.3.3 Canceling updates

As long as isolation requirement of transactions is met, a transaction can run to completion without interruption, i.e., it makes its updates effective. However, in some cases the isolation requirement among transactions is not satisfied and at the moment this fact is discovered, one of the transactions for which isolation is violated should no longer execute. We call this interruption of transaction execution an **abort** and we say that the interrupted transaction is *aborted*.

An aborted transaction is generally in an intermediate state where it performed its updates only partially. Since leaving the transaction is such intermediate state is not safe (and would in general violate atomicity requirement), the transaction cancels all the updates it has performed so far, i.e., it restores its write-set elements to their initial states. This mechanism is called rollback and is possible thanks to the storage of both an initial state and a tentative state (as explained in **Section A.2**). With such dual state (for each modified data item) at disposal, a transaction can choose the state that is visible to other transactions and with a rollback, it chooses to show the initial state.

The implementation of rollback depends on how the dual state is stored, i.e., how the logging is performed for modified data items. If direct update method is used, the tentative modifications are performed directly on the data items, requiring a rollback to replace tentative modifications with the initial state of the data items stored in the undo-log. Hence, for an STM using direct update method rolling back is a costly operation. On deferred update method, however, a rollback is cheap because the modifications are

performed on private copies and rollback corresponds to simply discarding these private copies.

Since a transaction should never be partially completed, rollback is conceptually critical for ensuring atomicity. However, practically, the duration where a rollback is critical for ensuring transaction atomicity depends on the updated method used. With the direct update method, rollback is critical throughout all the transaction because with this method other transactions can be aware of performed modifications and before the commit is effective the modifications are only partial. With deferred update method, however, the tentative modifications of the transaction are hidden from other threads, so it is like the transaction stays in all the time in *rolled back state* and atomicity is safely ensured until the commit operation. Meanwhile, during the commit operation, the copying of the redo-log on the data items introduces a time window in which updates are partial on the write-set elements. Hence, for an STM using deferred update, rollback is critical for ensuring atomicity only during commit.

A.3.4 Making updates effective

In lock-based STMs, making the updates effective is possible thanks to the ownership management mechanism. As explained in **Section A.3.1**, when a transaction acquires the ownership of its write-set elements it can safely update all the write-set elements. Actually, in lock-based STMs, this holds even when a subset of the write-set is acquired, i.e., as long as the transaction releases all the ownerships it has acquired so far upon an abort, it is actually safe for the transaction to make the modifications only on the data items it has acquired so far. This is possible because a lock-based STM has the guarantee that an acquired lock can only be released by the owner transaction, so the transaction can rollback its modifications before releasing its acquired locks. This allows the lock-based STMs using direct update method to make the modification on data items effective during the write accesses. Lock-based STMs using deferred update method, however, need to wait for all the write-set elements to be acquired to start making the modifications on the data items effective. An important point to note is that it is not only the condition whether or not it is safe to make write-set elements effective that allows a commit[1] to be possible. A commit is possible if the isolation requirements of the transaction is still met (i.e., if none of its read-set elements is modified throughout the transactions) at the point where all write-set elements are acquired.

[1]Here we consider a commit as a successful termination of a transaction where the updates of all the write-set elements are made effective.

In non-blocking STMs, the acquisition of the ownerships does not give the guarantee that it is safe to update, since ownerships can be modified by any transaction. Hence, for non-blocking STMs, there is no time window in which the transaction can freely modify acquired data items. Instead, the key element allowing atomicity is update of the transaction status information. However, the use of transaction status in object-based and word-based non-blocking STMs are different.

For object-based non-blocking STMs (most of these STMs use deferred update method) the instantaneous update of all write-set elements of a transaction is tied to a single atomic update performed on the transaction status. In other words, as long as the transaction status is not *committed*, none of the updates of a transaction is perceived by other transactions. Upon setting the transaction status to *committed*, however, all of the updates to be performed by a transaction are instantaneously made effective. This instantaneous update effect on multiple data items can be considered as a multi-word CAS operation and this is possible thanks to the notion of *logical state* [90]: the **logical state** is the state of a data item that is visible by other transactions depending on the value of transaction status. The logical state implies the following interpretation: when the transaction status is *committed*, it is the tentative state of a data item that is visible to other transactions, while for other values of transaction status, the state that is made visible to other transaction is the initial state of the data item. Such interpretation is valid for all elements of the write-set and, hence, the modification of transaction status to the *committed* value immediately changes the logical state of all write-set elements to their tentative state. This technique has been first used in DSTM [97], while most object-based non-blocking STMs proposed after DSTM use the same approach but with a different metadata organization (e.g., [127, 181]).

The reason why the instantaneous update effect is possible with object-based non-blocking STMs is that an object can reside on any memory location in the heap. This allows both the modified version as well as the initial version of the same object to reside on a different memory locations and, hence, gives the possibility to use one or the other version as the version that is perceived by other transactions. However, in word-based STMs the memory location of the data item is fixed and is determined by the address of the item (i.e., a memory word). This immutability of location necessitates the updates on the data item to be applied on the data item itself. Thus, word-based non-blocking STMs (which also use deferred update method in general) perform the instantaneous update effect in a different manner. These STMs first acquire the ownerships of all their write-set elements and set their transaction status to *committed*. Once the transaction status becomes *committed*, other transactions accept the private copies of the committing transaction as the valid version.

This situation is of course transient until the transaction finishes to copy the private copies on to the data items. This approach has been successfully used in WSTM [89, 63].

A.4 Mechanisms ensuring isolation

As explained in **Section 2.4.1**, depending on the application using transaction blocks (i.e., whether the application should avoid transactional data races or not), the isolation requirement expected from a transaction may vary. One way of providing different isolation requirements for transactions is by guaranteeing them at the STM level (rather than at the programming language level). In this section, we present mechanisms that allow providing these isolation requirements.

As it can be expected, different isolation requirements necessitate different techniques to be applied and, hence, the discussion of mechanisms ensuring isolation requirements is organized in terms of different isolation guarantees that can be ensured by an STM. We start by explaining mechanisms used to ensure weak isolation (both for isolation of committed transactions and for isolation of all transactions), then introduce mechanisms for lock-based isolation guarantees (publication and privatization) and strong isolation. We terminate the section with techniques for ensuring irrevocability which has additional isolation requirements compared to previous isolation guarantees. Below, we discuss mechanisms for each isolation guarantee in a separate subsection.

A.4.1 Ensuring weak isolation of committed transactions

The STM mechanisms ensuring atomicity only determine how transactions apply modifications on shared data. However, for a transaction to commit successfully not only all its modifications should be visible to other transactions at once, but also the values of data it reads (and that is not modified yet by the transaction) should correspond to their initial states throughout all the transaction execution[1]. The latter defines the minimal isolation requirement of an STM implementation, because the fact the read values change during the transaction's execution means that the transaction sees external effects during its execution (which should be excluded to ensure isolation). We call this minimal isolation requirement **consistency** and if this requirement is violated we say that the *consistency is violated*.

Transactions ensure consistency by *(i)* establishing an agreement on the order in which transactions commit (this order is not necessarily the order of occurrence of commit operations of transactions in real-time), and *(ii)* aborting transactions that prevent establishing

[1]The read values can correspond to different values and still be valid in multi-version STMs but this is out of the scope of the thesis and discarded for simplicity of expression.

a commit order. The key concept that brings these two mechanisms together is called **conflict**. Informally, any event that has the potential to prevent the establishment of a commit order is a conflict. Formally, a conflict occurs if two concurrent transactions access the same data and at least one of the accesses of these transactions is a write. A conflict is just an indication that consistency could be violated: the absence of conflicts means that consistency is preserved, however, the presence of conflicts does not necessarily imply that consistency is violated.

The two mechanisms that ensure consistency can be designed using the concept of conflict. Establishment of an agreed commit order is provided by detecting conflicts and ordering the commits according to the order dependencies dictated by conflicts. Since absence of conflicts ensure consistency, **conflict detection** allows determining whether transactions can agree on a commit order. The second mechanism, i.e., eliminating transactions that prevent the establishment of a commit order, is effectively performed by **conflict resolution**. Conflict detection and conflict resolution together constitute the critical path of transactional execution since most of the STM overhead is due to these mechanisms. In the rest of this subsection, we discuss different aspects of conflict detection (visibility of reads and validation) as well as the mechanism for conflict detection and resolution.

A.4.1.1 Conflict detection and visibility

The simplest possible conflict detection that can be imagined is one which detects conflict at the instant they occur. With such conflict detection, just tracking the conflicts is enough to determine the consistency of a transaction. However, such an STM needs to know about all the accesses (both reads and writes) of all transactions that are executing in the system. In other words, such conflict detection implies the global *visibility* of all transactional read and write accesses.

Maintaining visibility of write accesses is trivial. Actually, write accesses are inherently visible since a successful write implies that data is acquired. Since acquisition of a data declares to other transactions, by definition, both the owner and the fact that the data is owned, visibility of write accesses comes for free (in terms of conflict detection mechanism). This allows both the read and write accesses to determine if they are involved in a conflict with a previous write access.

Since write visibility is ensured, it is the presence of global visibility for read data that additionally needs to be considered by an STM design. However, such visibility generally comes with a significant performance cost and this gives rise to the major tradeoff in conflict detection: visibility versus performance. As a result of this tradeoff, we observe

that there are three visibility approaches that have been proposed in the literature: *visible reads*, *semi-visible reads* and *invisible reads*.

Visible reads requires that all read accesses are observable by all transactions in the system. This knowledge allows a write access to see all the transactions that have previously read the data, hence to find out all the transactions that are in conflict with the writer transaction. Such approach can detect conflicts at the moment they occur by providing a simple way to detect read-write conflicts during write accesses. Detection of other conflicts is already possible thanks to acquisition: (other) read-write conflicts are detected during read accesses and write-write conflicts during write accesses. This approach has been taken by a version of DSTM [97]. Marathe *et al.* [127] and Spear *et al.* [172] also experienced with the mechanism and reported that it performs poorly due to high cost of metadata management and the cache invalidations caused by this metadata management.

Semi-visible reads proposes to keep only an indication that a data item has been read by some other transaction but does not reveal the transaction's identity. This way, it greatly reduces the metadata size required for visibility and relieves the incurred cache pressure. The idea is to use Scalable Non Zero Indicators (SNZI) to indicate whether a data has been read or not. Using this information, write accesses can also detect conflicts but cannot immediately perform a conflict resolution. They can however propagate this conflict information for later use, i.e., for a later validation (see **Section A.4.1.2** for validation). The approach is used by SkySTM [116].

Invisible reads requires that a transaction is aware of only its own read accesses. Spear *et al.* [172] have shown that using invisible reads is the most efficient way known for implementing conflict detection. Hence, most state-of-the-art STMs use this approach for conflict detection[1]. In the sequel, our discussion on conflict detection mechanisms assumes invisible reads approach unless otherwise is stated.

A.4.1.2 Validation mechanisms in conflict detection

One implication of avoiding visible reads approach is that some conflicts in which a transaction is involved in can go unnoticed since the transaction can perform only a partial conflict detection during write accesses (i.e., can detect only conflicts between two write accesses). This leads to the necessity of using an additional step, namely *validation*, in conflict detection to ensure consistency. **Validation** is an operation where a transaction checks whether each read-set element is still in its initial state. In other words, validation is the mechanism that performs the version tracking of read data items.

[1] Even most STMs designed prior to this result intuitively used this mechanism to avoid large metadata.

It can be noticed that, by definition, validation by itself verifies whether consistency is preserved. Hence, it is possible to use only validation for ensuring consistency. However, validation is generally a costly operation and it is preferred to use it as a *second chance* operation whenever conflict detection comes short in determining consistency violations. Validation can be performed at any point in the transactional execution and the most natural point to perform validation is during the commit operation after ensuring the write-set locations are acquired by the transaction. This way the conflict detection mechanism ensures to catch any consistency violations that are not noticed during the transaction execution (one example that applies this approach is TL2 [42]).

Even when used as a second chance operation, the cost of validation incurs an important overhead so there are different approaches to speed-up validation.

Incremental validation (a.k.a. full read-set validation) is the straightforward validation approach: it checks whether all the read-set elements still correspond to their initial states. This approach is said to be incremental since using this validation multiple times in a transaction implies checking progressively more and more read-set elements (validation may be needed multiple times in a transaction execution in order to ensure isolation and this is explained in **Section A.4.2**). This validation approach is costly due to the time spent and the size of metadata required.

Global commit counter based validation is a heuristic approach used by RSTM [172] where a global commit counter indicates the progress of committed transactions that perform a write. The aim of this counter is to avoid incremental validation if it is possible to do so. Whenever a validation is required, the approach checks whether the value of the global commit counter is the same as the one stored during the last successful validation. If the two values are the same, it means that there has been no transactions that modified any memory location so it is not necessary to perform a lengthy validation. This heuristic approach is simple and effective but has two disadvantages: *(i)* it allows unnecessary validations for cases where modifications indicated by the change in commit counter do not concern the transaction that performs the validation, *(ii)* each time a validation is required, the global commit counter needs to be read. Since the counter is shared, this harms the scalability of the approach.

Value-based validation is the validation operation in which each read-set element is compared against a log of actual values that have been encountered for its previous accesses in the transaction. The major difference of value-based validation is that it uses the content of read-set elements to track their versions. This validation approach is appealing since it relies on private metadata (rather than a shared global commit counter). Meanwhile, it is subject to ABA problem (see **Section A.2.5** for the ABA problem). Hence, STMs

that use value-based validation should make sure that during the validation no other writer transaction commits. This approach is used by JudoSTM [143] and NOrec [40].

A.4.1.3 Classification of conflict detection mechanisms

A formal classification of different possible conflict detection mechanisms has been proposed by Scott [162]. However, in practice, mainly three of these classes are used in conflict detection: *eager conflict detection*, *lazy conflict detection* and *mixed invalidation*.

Eager conflict detection detects conflicts at the earliest possible point. This earliest point of detection can either be the first attempt to access a data as well as the first validation operation which reveals the conflict. Since conflict resolution is applied to the detected conflict, its resolution does not wait until commit time. If the detected conflict is a read-write conflict, applying resolution right away is pessimistic since *(i)* the transaction that performs the read can commit before the transaction that writes, or *(ii)* the transaction that wins after the conflict resolution can later be aborted, unnecessarily aborting/delaying the looser transaction. Meanwhile, it is not possible to know at runtime which transaction is to commit first, so resolving the conflict early can also avoid wasting work. Examples of STMs that use this type of conflict detection are DSTM [97] and SXM [78].

Lazy conflict detection is the exact opposite of eager conflict detection: conflicts are detected at the latest possible point in the transaction execution, i.e., during commit operation. Such a detection mechanism avoids checking conflicts until commit operation and finds conflicts out using a validation operation. This type of conflict detection seizes the opportunities that could be missed by eager conflict detection by attempting to resolve the conflict early. However, for transactions that are doomed to abort (e.g., in the case of a write-write conflict among transactions) this approach allows wasted work to be performed.

Mixed invalidation detects only write-write conflicts at the earliest possible point in the lifetime of the transaction while it detects read-write conflicts at commit time. This mechanism takes advantage of opportunities missed by both eager and lazy conflict detection and avoid wasted work. For example, by detecting write-write conflicts early, it avoids wasted work of doomed transactions, and by waiting until commit time, it allows useful work to execute if both transactions could commit. Some STMs that adopt this mechanism are RSTM [127] and SwissTM [47].

A.4.1.4 Conflict detection mechanisms

Local versioned locks is a straightforward eager conflict detection mechanism. Each data item has an associated versioned lock (see **Section A.2.5.1** for details on versioned locks)

using version number as version tracking information. For each access to the data item, the mechanism first consults the versioned lock to understand whether there is conflict. For write accesses a conflict is notified only if the versioned lock is acquired by some other transaction (we call this condition the *ownership conflict condition*). For read accesses the notification of a conflict is done in one of the following cases (which we call *inconsistency conflict conditions*): *(i)* the data item is acquired by some other transaction, *(ii)* the data item is not acquired but the version number stored during a previous read access for the same data item has changed (the versioned lock indicates a different version number). During the commit operation, a validation is performed for all the data items in the read-set (after making sure that all the elements of the write-set are acquired) and the inconsistency conflict conditions are again used as indicators of a conflict. For the write-set elements the commit operation increments the version number recorded at the time of acquisition by one and stores it to the corresponding versioned lock. By storing a version number to the version lock, the ownership is released.

Time-based conflict detection mechanism uses again versioned locks for each data item but replaces version numbers with timestamps sampled from a global clock (see **Section A.2.5** for details on global clock and timestamps). The use of a global clock provides a total order on transaction commits and data item versions [154]. The mechanism is similar to local versioned locks with the following differences: a transaction stores a timestamp, the *read timestamp*, that is sampled at transaction start and this timestamp corresponds to the state of the data items at the beginning of the transaction. The commit operation also samples the global clock when committing is safe (i.e., when the locks for all write-set elements are acquired) and increments it by one (at the same time the global clock is incremented). This new value, which we can call *write timestamp*, determines the commit order of the transaction and is set as the timestamp of all write-set elements during commit. Since each transaction has a unique write timestamp, the version of a data item is uniquely identified by such write timestamps. The ownership and inconsistency conflict conditions of this mechanism (which serve to notify a conflict upon an access) are the same as that of local versioned lock except the method to determine the version change. To detect version change of a data item, this mechanism compares the timestamp stored in the versioned lock against the read timestamp (timestamp sampled at the beginning of the transaction). If the timestamp found in the versioned lock is greater than the read timestamp then the mechanism decides that the data has been modified by the commit of some other transaction during the execution of the transaction (thus a conflict is signaled). As in local versioned locks mechanism, a validation during commit checks again the inconsistency

conflict conditions on all read-set elements and reports that consistency is ensured only if no conflict is signaled. This conflict detection mechanism is adopted by TL2 [42].

The major disadvantage of time-based conflict detection mechanism is the need to access to a global clock value which is performed twice during a transaction. For such conflict detection mechanism to be scalable *(i)* the number of accesses to the global clock needs to be minimized, and *(ii)* the number of validations needs to be reduced.

Lazy snapshot isolation is an extension of time-based conflict detection, where instead of using a single read timestamp, a range of timestamp values for which all the read-set elements are valid is used. This range of timestamps is called the *validity range* of a transaction. The lower limit of this range is the same as the read timestamp of time-based validation, i.e., it corresponds to the state of the system at the beginning of the transaction. The upper limit of the range corresponds to the smallest timestamp where at least one of the read-set elements has been modified. In other words, the validity range of transaction T is the set of timestamp values for which some transactions have probably committed but their commits modified data items other than the read-set elements of T. This effectively reduces the number of validations required during read accesses, because transactions modifying disjoint data items since the sampling of the read timestamp do not cause the mechanism to notify a false conflict (whereas time-based conflict detection does). In this conflict detection mechanism the validity range is constructed as follows: when the observed timestamp of the read data item is not in the validity range (and is higher than its upper limit) a validation is performed and if this validation does not indicate any conflicts then the upper limit of the range is advanced to the timestamp observed for the data item. Some STMs using this conflict detection mechanism are LSA-STM [153] , tinySTM [57] and SwissTM [47].

Global versioned lock mechanism uses a single global versioned lock to be used by all transactions and conflict detection is heavly based on validation rather than other mechanisms. The global lock is used to have exclusive permission to execute a commit. This scheme can only work with deferred update STMs since the lock is acquired only at commit time and is used to commit modifications of elements of the write-set. The version number in the global versioned lock serves as a global commit counter, and each transaction samples the value in the versioned lock at the beginning of the transaction (snapshot). Each time a read access is performed value-based validation is performed and, if the validation passes, the snapshot is advanced. The same validation is performed also during commit. The validation is repeated until the global versioned lock does not change during validation. This makes sure that no other writer transaction committed during validation. Once a validation is performed with an unmodified global versioned lock, the lock can be acquired

with an atomic operation to perform modifications in the write-set elements (to avoid any commits to occur between the end of validation and the following acquisition of the lock, the acquisition of the lock and the detection that validation is valid are performed by the same CAS operation).

A.4.1.5 Conflict resolution

Once a conflict is detected, it should be resolved for the consistency of data items to be preserved. Resolving a conflict is possible only by removing the cause of the conflict standing in the way of a transaction. Since in the case of STMs the cause of a conflict is that there are two concurrent transactions contending for a common data item, the conflict should be resolved by enforcing only one transaction to own a shared data item at a given time. Two methods are possible to satisfy this requirement:

- **Ordering:** In this solution, two conflicting transactions are not allowed to execute concurrently. Hence, while one transaction resumes executing, the other either waits (so that the conflict does not occur during the waiting interval) or gets aborted to be re-executed.
- **Merging:** With this solution, the update responsibility of both of the conflicting transactions is passed over to one of the transactions, i.e., in a way the two transactions are merged into one (if not all the update responsibility of a transaction is passed to the other transactions, at least the update responsibility for the contended data item is passed).

In STMs the above resolution decision is generally given in a component called **contention manager**. Since the conflict resolution decision effects the progress of a transaction and the application running on top of the STM, we defer the discussion of contention managers to **Section A.5.2**.

A.4.2 Ensuring weak isolation of all transactions

Tentative changes performed during transaction execution are already protected from other transactions: with direct update method a transaction that acquires a data item for modification has exclusive access to it as long as the transaction is active (if acquisition becomes invalid the transaction is aborted), with deferred update method the tentative changes are only visible to the transaction itself. Hence, by design, write accesses are isolated from other transactions during transactional execution.

Read accesses, however, can observe tentative changes of other transactions if care is not taken. As it has been explained in **Section 2.4.4.1**, allowing the read accesses to

observe inconsistent values can cause anomalies such as infinite loops, access violations or divide-by-zero errors. Avoiding such problems requires the use of extra mechanisms to check the consistency of read values during transaction execution, more specifically during read accesses. We have seen in **Section A.4.1.2** that validation is the mechanism that reveals consistency violations. Hence, validation can also be used to ensure the consistency of read accesses. The validation to ensure consistency of read accesses is sometimes called **post-validation** [171].

The use of validation for ensuring consistency of read accesses is only required if reader visibility is omitted (see **Section A.4.1.1**). Under invisible reads (which is the commonly used approach), maintaining consistency during transactions require post-validation to be performed regularly at each read access, introducing an important additional overhead. At this point, the use of lightweight validation mechanisms becomes much more important in providing isolation compared to conflict detection. The mechanisms explained in **Section A.4.1.2**, except incremental validation, serves exactly this purpose: reducing the validation overhead by avoiding lengthy validations when it is not necessary (e.g., if there have been no modifications since the last read, there is no need for a validation). Each STM can choose the appropriate validation mechanism that fits into its design to decrease this additional validation overhead.

A.4.3 Ensuring publication safety

Ensuring weak isolation for all transactions (and not only committed transactions) is the minimal isolation guarantee expected for an STM. However, the STM can be designed to have higher isolation guarantees that provide the isolation of mutual exclusion. Ensuring publication is one step to raise the isolation of weakly isolated STM.

Before starting to present techniques to ensure publication, it is instructive to see the cause of the problem generated by the publication pattern. **Figure 2.2** illustrates an example of the publication pattern. The issue with the execution of this example code is that the transaction in *Thread 2* can speculatively read the contents of the shared object early (i.e., before checking that the object is shared). Such speculation actually generates a benign race even for lock-based programs and thus can be preferred by the programmer or by the compiler optimization step to increase execution speed. However, this optimization of speculative reads does not work well with weakly isolated STMs. Fortunately, it turns out that the problem can be solved by avoiding the speculative reads as shown in **Figure A.1**.

	Thread 1	Thread 2
1	ShObject = **new** Object();	**atomic**{
2	ShObject.x = val1;	if(published)
3	ShObject.y = val2;	z = ShObject.x + ShObject.y;
4		}
5	// synchronizing transaction	
6	**atomic**{	
7	published = true;	
8	}	

Figure A.1: Correctly synchronized publication using weakly isolated STM.

Taking the above example as a guideline, Menon *et al.* [130] suggest that weakly isolated STM can ensure publication safety as is as long as the following are applied to avoid early speculative reads:

- The STM does not introduce speculative reads of data inside a transaction[1].
- The programmer avoids data races even if they are benign under lock-based semantics. This way the speculative reads on lines 3 and 11 of **Figure 2.2** are automatically eliminated.
- The programmer uses compiler options that do not hoist memory operations inside transactions.

Menon *et al.* [130] also provide a *start linearization* mechanism for ensuring publication safety to free the programmer from assuring that the above conditions are met (which is not always obvious). The mechanism simply forces the transactions to commit in the order they start. They use the global commit counter metadata such that it serves as start number for the transaction and it is advanced at transaction starts instead of transaction commits. This way, transactions can determine an order between transaction starts. To ensure the start order for commits, a commit operation waits for other active transactions with a start number smaller than its own start number. The transactions can track such information thanks to a shared array storing the start numbers of the currently active transaction for each thread.

Dalessandro *et al.* [40] use value-based validation unconditionally at every transaction commit to ensure publication safety. Since value-based validation tracks actual value changes, it is able to detect non-transactional writes and hence can detect a race that can

[1]A deferred update STM can perform speculative reads if the granularity of data it reads from memory is larger than the granularity of the target variable/object. As a side effect of the granularity, a single access of the STM fetches both the target variable/object and another variable/object that needs to be accessed by the transaction. The reuse of this single access for the second variable/object in later accesses of the transaction is the cause of such speculative read.

occur during publication: the failure of value-based validation due to a publication race causes a transaction to abort, preventing it to read data items early.

A.4.4 Ensuring privatization safety

As with publication, ensuring privatization is a necessary step in providing lock-based isolation guarantees. Although publication safety can be avoided by eliminating data races at the source code, it is much more complex to guarantee privatization safety since the source of the data races depends on the STM implementation rather than the order of accesses inside transactions. Dalessandro *et al.* [40] tie this complexity to the fact that publication is performed by a single thread, whereas privatization can be performed by any thread.

A common particularity of all the privatization safety mechanism is that they are overly conservative. The reason is that all the mechanisms target *implicit privatization* [116] (where the programmer does not explicitly tell which transactions are privatized) and do not try to dynamically detect which transaction is a privatizing transaction (see **Section 2.4.3** for privatizing transactions). Targeting implicit privatization is righteous since transactions are expected to be transparent (i.e., removing any burden related to synchronization from the programmer) and explicitly marking privatizing transactions is error-prone [116]. However, failing to detect the privatizing transactions results into a mechanisms where all transactions (at least all writer transactions) need to behave as if they are all privatizing, hence the overhead of privatization is associated to all transactions (at least all writer transactions), significantly degrading STM performance. Yoo *et al.* [194] depict that performance degradation can attain up to 100% for ensuring privatization. We name these overly conservative privatization solutions **privatizer-agnostic** solutions. Unfortunately, all the privatization safety mechanisms proposed to date are privatizer-agnostic. Obviously, solutions that are not privatizer-agnostic would considerably decrease the overhead incurred by privatization safety since, in general, a very small portion of executed transactions in a program are privatizing.

The only proposed mechanism to date that ensures publication safety in all cases is called the *quiescence* mechanism. Other mechanisms proposed to ensure privatization safety most generally target only part of the problem (e.g., either delayed cleanup or delayed conflict detection, see **Section 2.4.3** for a description of these problems) or do not fully ensure privatization safety for both direct update or a deferred update STM. Hence, in this section we only discuss the quiescence mechanism and let the interested reader to consult the following publications for other approaches: [171, 128, 116, 130, 40, 143, 174].

Quiescence is a mechanism which consists of waiting right before terminating the commit operation for all concurrently executing transactions to commit or to abort and to finish their cleanup. Although allowing a restricted form of privatization, the first appearance of quiescence mechanisms were for memory deallocation *outside* transactions using pointers read *inside* transactions in unmanaged languages[1] [103, 43, 42]. These were followed by quiescence mechanism covering the complete privatization issue. The first of these mechanisms is by Wang *et al.* [186] where they use a linked list to find out the concurrent active transactions to wait for. The second of these mechanisms is named *transactional fence* by Spear *et al.* [171] and uses epochs to determine which are the concurrent active transactions. A committing transaction is permitted to terminate only when all the other active transactions in the same epoch have completed their work.

A.4.5 Ensuring strong isolation

The main issue in guaranteeing strong isolation is the ability to detect conflicts between transactional and non-transactional accesses (along side the conflicts among transactional accesses). Hence ensuring strong isolation converts to extending the STM conflict detection to catch also conflicts among transactional and non-transactional accesses. Providing such extension to conflict detection is difficult since normally STM has control only on the transactional regions of multi-threaded code, while strong isolation requires that STM has control over the entire program. In short, the conflict detection range of STMs needs to be extended to detect all conflicts both inside and outside transactions. STMs already introduce a significant overhead just to detect conflicts among transactions and increasing the conflict detection range over all the program naturally aggravates this overhead.

Although strong isolation is an important guarantee to be ensured by STMs, most to-date STMs ensure only weak isolation and approaches that ensure strong isolation are rare. Among the approaches that exist, there are two main strategies to extend the conflict detection range[2] : *(i)* inserting conflict detection code around non-transactional accesses using code instrumentation techniques, and *(ii)* extending STM conflict detection mecha-

[1] These studies can be seen as the first papers bringing the privatization issue into the attention of STM community.

[2] Here, we consider only the approaches that propose additional mechanisms to ensure strong isolation. There exist other approaches (namely static and dynamic separation [90]) that imposes a certain programming discipline to the programmer so that the written code does not include any data races between transactional and non-transactional accesses. These approaches allow the use of weakly isolated STMs to execute the code safely without the need for an additional mechanism to ensure strong isolation. The rationale of these approaches is the same as writing data-race-free programs to ensure sequential consistency with weaker underlying memory models.

nism to detect conflicts between transactional and non-transactional accesses at run-time. Below, we discuss mechanisms of each strategy separately.

A.4.5.1 Code instrumentation techniques

Code instrumentation techniques use an STM guaranteeing weak isolation as a building block and thanks to compiler analysis insert code around non-transactional accesses to convert such accesses to optimized short transactions. Thus, these techniques extend the conflict detection range of the STM by moving non-transactional accesses into transactions.

Hindman *et al.* [101] instrument, by their design, all the objects to introduce an additional lock field. All accesses to objects first check this lock to see whether the object is acquired by some other thread and access is granted only if no acquisition is present. This check is done whether or not an access is transactional. In this approach, the conflict detection is by design over all the program.

Lev and Maessen [117] follow a similar approach, but introduce an access mode field instead of a lock. If the access mode of an object is *local* (object is accessed by a single thread during its entire lifetime) then the object can be accessed directly. If the access mode of the object is *shared* (object can be accessed by multiple threads) then its access is performed in a transaction. This is made possible through special getter and setter methods instrumented by the approach and all accesses to the objects use those methods. Objects always start in local mode, and transition to shared mode when a variable known to be global or shared is updated using the object (i.e., when the local object is made open to be used by other threads). Once an object is in shared mode it stays in the same mode for the rest of its lifetime. This approach, hence, tracks effectively publication but not privatization.

Shpeisman *et al.* [166] improve Lev and Maessen's approach by abandoning object field modification and, instead, use orecs of STM to indicate access mode of an object. This way they decouple the object organization and STM instrumentation. They introduce more access modes for objects and embed this information into the orec together with the lock field indicating the acquisition. Thus, any non-transactional access first fetches the orec to learn about the access mode and then decides to access either directly or transactionally according to the mode. The tracking and transitioning of access modes of objects is performed as done by Lev and Maessen. This approach mainly improves the execution performance by performing a static compiler analysis to eliminate the instrumentation of non-transactional accesses that would never need it[1] . They perform two types of analysis

[1] Of course, the analysis discovers only the accesses that can be proven to be unnecessary statically.

to identify such accesses: *(i)* detection of thread-local objects, and *(ii)* a *not accessed in transactions* (NAIT) analysis that further identifies the non-transactional accesses that can be proven not to conflict with transactional accesses. The latter analysis is based on the following two observations: *(i)* a non-transactional write access does not need instrumentation if the target location is never accessed in a transaction, *(ii)* a non-transactional read access does not need instrumentation if the target location is never modified in a transaction.

Schneider *et al.* [161] perform a follow-up work on Shpeisman *et al.*'s approach using just-in-time (JIT) compilation techniques to avoid the static analyses (e.g., the NAIT analysis) being applied to the whole program. This is possible since JIT compilation can perform static analyses only on the code emitted but not yet run. This allows the JIT compiler to assume optimistically that non-transactional accesses do not need instrumentation. As the JIT compiler generates transactional code, it detects accesses that have dependency on the generated transactional code and convert the necessary non-transactional accesses to transactional ones.

A.4.5.2 Mechanisms extending STM conflict detection at run-time

Mechanisms extending STM conflict detection at run-time are rare. The major work that is in this category is by Abadi *et al.* [8]. This approach uses the already existing virtual memory mechanism to detect conflicts between transactional and non-transactional accesses. They organize the virtual memory space of the multi-threaded program such that each physical heap page is associated to two logical pages; one used by transactional and the other used by non-transactional accesses. The approach avoids instrumenting non-transactional accesses. Instead, transactional accesses change physical page access permissions before performing their accesses and release the permissions after the access. Hence, a conflict between transactional and non-transactional access generates a page fault and the conflict is resolved in the page fault handler. Using pages as unit of conflicting data, the mechanism results in high number of page faults, degrading the performance of STM drastically. To decrease the number of pages faults, the approach proposes rather complex auxiliary mechanisms:

1. Page permissions are modified lazily limiting the period of time where non-transactional accesses can conflict with transactional accesses.

2. Removing the accesses that do not need conflict detection out of the page ranges that trigger conflict detection. This can be done thanks to a NAIT analysis based on Shpeisman *et al.*'s approach together with identification of thread local accesses such as immutable data, language implementation related data (e.g., virtual tables, garbage collector data). Note that these steps need to be performed at compile time.

3. Dynamically replacing a non-transactional access that generates frequent page faults with a short transaction encapsulating the access. This way the number of page faults can be decreased significantly. However, this dynamic action requires patching the code at run-time (resulting in a self modifying code).

A.4.6 Ensuring irrevocability

The ability of a transaction to rollback greatly simplifies ensuring both atomicity and isolation. It helps ensuring atomicity since rolling back a transaction helps masking its partial modifications. It also helps ensuring isolation because if the execution of a transaction would violate isolation (due to a conflict with another transaction) this transaction can simply be rolled back and restarted. Hence, rollback is a critical transactional operation.

It is possible, however, that a transaction contains irreversible operations and this makes it impossible for the transaction to rollback completely. For such transactions the generally accepted solution is **irrevocability**, i.e., arranging the transaction commit order such that the transaction containing the irreversible operation is executed only once, i.e., it is never asked to be rolled back. Transactions for which irrevocability is applied are called **irrevocable transactions** [190] (such transactions are also called *inevitable* [168] or *unrestricted* transactions [27] by different researchers).

Irrevocability trivially ensures atomicity; the irrevocable transaction will execute and commit as any other transaction and will thus make its modifications effective at once. However, providing irrevocability requires that the level of isolation between the irrevocable transaction and the other transactions is raised. In a way, the level of isolation comes closer to the isolation provided by mutual exclusion. This can be explained as follows. The isolation guarantee of STMs requires that the execution of a transaction T is as if there were no other transaction running concurrent to T. To provide this, when there is a conflict between T and another transaction, the STM (more specifically the contention manager) is free to abort T to provide isolation. However, if T is an irrevocable transaction, it should not experience even the effects of conflicts, i.e., it should not abort (in the case of conflict it will be the transaction which is not irrevocable that needs to abort). In a way, the isolation of T with respect to other transaction is higher. This can also be interpreted as T's execution being guarded more strictly than the execution of other transactions.

Ideally, it is not desired to raise the isolation to the level provided by mutual exclusion, i.e., we would like to have concurrent transaction executions even under irrevocability. However, by its very nature, the concurrency we can obtain under irrevocability is limited. First of all, there can only be a single irrevocable transaction executing at a time in a

STM system (if not, it is not possible to guarantee that two irrevocable transactions do not conflict, and, hence, it is not possible to guarantee that there will not be the need to abort an irrevocable transaction) [168, 190]. Second, the data accessed by an irrevocable transaction (either for reading or writing) should not be modified by any transaction until the irrevocable transaction commits. Thus, any transaction whose write-set intersects with the union of read and write sets of an irrevocable transaction should not commit until the irrevocable transaction commits. These two requirements defines the maximum level of concurrency acceptable under irrevocability.

Being limited by the concurrency under irrevocability, STMs avoid any degradation of performance that irrevocability would incur when there are no irrevocable transactions in the system. In order to satisfy this requirement, the simplest solution is adopted: using different execution modes for normal and irrevocable transactions. In **normal execution mode**, a transaction can be aborted without any restrictions (thus concurrency limitations under irrevocability does not apply in this mode), while in **irrevocable execution mode** a transaction executes in such a way that when a conflict is encountered, it is always the irrevocable transaction that is chosen to continue executing (while the other conflicting transaction is aborted)[1] . The existence of two execution modes also separates the transactions into two: normal transactions (transactions executing in normal execution mode) and irrevocable transactions (transactions executing in irrevocable execution mode).

Taking the requirements and the facts about irrevocable execution into account, we can distinguish two types of mechanisms to be provided for ensuring irrevocability:

- **Guarding mechanisms:** Forbidding some transactions to continue executing/committing until an irrevocable transaction commits (in order not to cause conflicts) can be considered as guarding the irrevocable transaction form external effects. In this sense, mechanisms that allow the execution of an irrevocable transaction without being asked for any aborts is provided by guarding mechanism.
- **Transition mechanisms:** Adopting different modes of execution, an STM needs a mechanism to transition (or to switch) from one mode to the other. These mechanisms ensure that consistency of transactions is preserved during the transition from one mode to the other.

Below we describe these two types of mechanisms in more detail.

[1] Although the execution mode is attributed generally to a transaction, the effects of this mode is mostly experienced by the transactions other than the irrevocable transactions, thus the mode can be considered as system-wide rather than limited to the irrevocable transaction itself.

A.4.6.1 Guarding mechanisms

The guarding mechanisms for irrevocability can be classified in two groups: coarse-grained and fine-grained guarding.

Coarse-grained guarding mechanisms permit only a single transaction (i.e., the irrevocable transaction) to access or modify data items. This way, any transaction (except the ones already executing at the moment where the irrevocable transaction started) trying to perform an access is blocked until the irrevocable transaction commits (in a way such transactions are forbidden to perform transactional accesses for this duration). Several forms of coarse-grained guarding have been studied by Spear *et al.* [173, 168]. Two major forms of coarse-grained guarding is observed in their studies. The first form forbids normal transactions both to read or write to any data item (this approach is used by TCC [84] and Yang *et al.* [140]), while the second form forbids normal transactions only to write to any data item (this approach is mainly proposed by Spear *et al.* [173, 168]). While the first form, does not allow any concurrency between the irrevocable transaction and other transactions, the second form allows concurrency of the irrevocable transactions with read-only transactions.

Fine-grained guarding mechanisms, in contrast to coarse-grained guarding, protect each of the data items accessed by the irrevocable transaction (regardless of whether it is a read or written data item) by means of ownerships. These mechanisms aim at having concurrency between the irrevocable transaction and other transactions whose write-sets do not intersect with the read- or write-set of the irrevocable transaction. This concurrency is provided by making use of the following observation: any concurrent read that accesses to the read-set elements of the irrevocable transaction does not threaten the safe execution of the irrevocable transaction. Hence, the ownerships for read-set elements of an irrevocable transaction are protected only from any concurrent write accesses. In the case where a concurrent write access tries to acquire a read-set element of the irrevocable transaction, a conflict is generated (thanks to the acquisition of read-set elements by the irrevocable transaction), which is resolved in favor of the irrevocable transaction.

Two approaches for expressing the ownerships of read-set elements have been proposed in the literature. The first one (called Single read-only locks by Welc *et al.* [190] and Inevitable Read Locks by Spear *et al.* [168]) uses special orecs (more specifically read-locks) for the read-set elements of the irrevocable transaction in addition to orecs used for write-set elements[1] . The particularity of these orecs is that they cannot be acquired for writing by concurrent writer transactions, but concurrent read accesses to data items protected by such orecs are granted access. The second approach, proposed by Spear *et al.* [168], uses

[1] JudoSTM [143] and Blundell *et al.* [27] can similarly guard read-set elements but with the same type of orecs used for write-set elements. Hence, they experience decreased concurrency.

a Bloom filter to store a summary of the read-set elements of the irrevocable transaction and normal transactions cannot acquire a data item that hit in the filter. Conceptually, this approach is the same as the first approach except that instead of using separate orecs for each read-set element, a single filter is used.

A.4.6.2 Transition mechanisms

Transiting from normal to irrevocable execution mode is accomplished by the acquisition of two different types of ownerships:

- **Irrevocability mode ownership**: There can be only one transaction executing in irrevocable mode in the system at a given time. Hence, an irrevocable transaction should acquire a so-called **irrevocability token**. This token should be acquired using an atomic operation since there can be multiple transactions requesting to transition to irrevocable mode. If the irrevocability token is acquired by a transaction, other transactions requesting to transition to irrevocable mode need to wait until the token is released.
- **Accessed data ownership**: An irrevocable transaction needs to guard all of the data items it accesses from other transactions, thus during a transition it needs to acquire ownership for all the data it accessed so far (both read and written data items).

When performing a transition from normal to irrevocable mode, a transaction (that has already started executing) should perform the following steps to complete the transition:

- Acquire the irrevocability mode ownership,
- Acquire the accessed data ownership for data accessed by the transaction so far,
- Ensure that data items already read by the transaction are still valid (i.e., perform a full read-set validation).

If none of these steps fails, a transaction transitions to irrevocable mode, otherwise it aborts by releasing any ownerships that it has already acquired.

STMs generally allow a transaction to transition to irrevocable mode in the midst of its execution, when they are up to executing their first irreversible operation. A group of STMs opts for a simple solution to allow transition at any point: aborting the transaction and restarting it in irrevocable mode [27, 111]. The reason for such a choice is that it eliminates the need for the acquisition of all the data items accessed by the transaction so far. This approach is generally taken by STMs adopting coarse-grained guarding mechanisms because such guarding mechanism does not have a means to acquire ownership on specific data items. For the same reason, these mechanisms need to ensure that no other transaction

has ownership on any data (to allow the irrevocable transaction to acquire ownership of any data it desires). Hence, usually these mechanisms also require that the irrevocable transaction waits for other active transactions to commit/abort before it starts executing[1].

Although simple, restarting a transaction in irrevocable mode unconditionally requires the re-execution of all operations of the transaction up to the first irreversible operation. This can be a loss of time if both the irrevocability token and the data accessed by the transaction so far can be acquired the moment irrevocability transition is desired. Hence, approaches allowing control on ownerships, usually fine-grained guarding mechanisms (except for [140] which uses coarse-grained guarding), perform all of the three transition steps mentioned above. If one of the steps fails, these approaches also fall back to restarting the transaction after having released all the ownerships they acquired (both the accessed data ownerships and the irrevocability token).

Transition from irrevocable to normal execution simply requires releasing both types of acquired ownerships. Since an irrevocable transaction should release all these ownerships only after it makes its modifications visible to other transactions, the most natural place to perform this transition is the commit of the irrevocable transaction. To ensure correctness, the irrevocability token should be released after the data access ownerships.

A.5 Mechanisms ensuring progress

Mechanisms that ensure atomicity and isolation serve mainly to provide correct execution of STMs. However, it is generally desired that an STM, as a synchronization mechanism, also provides forward progress (see **Section 2.4.5**), i.e., it is free from liveness issues such as deadlocks, livelocks or starvation.

STMs generally target obstruction-freedom due to the simplicity of design it offers (see **Section 2.4.5** for obstruction-freedom). By its nature, obstruction-freedom decouples the design components providing correctness from the components ensuring progress [95, 97, 90]. The decoupling is realized as follows:

- The components that provide correctness should be wait-free as long as there is no synchronization conflict.
- The removal of synchronization conflict is delegated to a separate module.

The overall progress of an obstruction-free design dictates that progress is ensured for both parties of the decoupling: the components providing correctness provide wait-freedom

[1] While the first form of coarse-grained guarding requires that the irrevocable transaction waits for all currently active transactions to commit, it is enough to wait only for currently active writer transactions for the second form.

in the absence of conflict and the separate module removing synchronization conflicts handles the overall progress. In the case of STMs, the components providing the correctness are the collection of transactional operations (generally function calls) composed mainly of transactional read/writes, and transaction start/commit, while the separate module removing conflicts is the so-called contention manager[1] . Based on this separation, it is possible to analyze the progress of STM designs at two levels [80, 90] :

- **Progress of transactional operations[2] :** STM designs ensure that each of the functions corresponding to these transactional operations are wait-free. Note that the progress requirement of transactional operations is such that these operations are wait-free even under conflicts. Of course, this requires that the contention manager never blocks indefinitely.
- **Progress of transactions[3] :** When transactions conflict, the conflict resolution techniques explained in **Section A.4.1.5** are applied by the contention manager. It is important to note that it is the way the contention manager resolves the conflicts that determines the overall progress of the transactions. If the contention manager removes the conflict temporarily just not to be blocking the system indefinitely, it is possible that the STM is always live (i.e., the transaction operations are wait-free) but transactions never commit. The reason for such a situation is that the cause of the conflict is actually not removed. Hence, for the overall progress to be ensured, contention manager should take actions to remove the causes of the conflicts without blocking indefinitely.

Below, we describe the mechanisms that ensure progress at each of these two levels.

A.5.1 Ensuring progress of transactional operations

The major concern for the progress of transactional operations is that there is no waiting during the execution of a transactional operation. It is only the concurrent access to common data items that can cause a problem for progress (concurrent access to disjoint data is trivial and naturally wait-free). An STM protects the access to common data through the use of metadata, hence accessing common data items corresponds to accessing shared metadata within transactional operations. The metadata that is shared by STM designs is mainly the orecs and the transaction descriptors. Thus, as long as transactional

[1] When the contention manager strategy is very simple, such as the transaction deciding to kill itself upon a conflict, there is no visible contention manager in the STM design (probably because it is more efficient to do the design this way). However, logically this simple decision can always be attributed to a separate module.

[2] [90] names this level of progress *TM-level progress*

[3] [90] names this level of progress *transaction-level progress*

operations access the orecs and transaction descriptors without waiting (i.e., in finite time steps), the transactional operations are wait-free.

STMs use atomic read and writes provided by the hardware (hereafter we call these atomic reads/writes) to access both orecs and transaction descriptors. Since atomic reads/writes are by definition wait-free, the accesses to these shared metadata do not incur waiting. Intuitively, however, among the shared metadata access, it seems that the ownership acquisitions of shared data (i.e., modifications of orecs) could be a cause of waiting. Indeed, under some conditions there could be waiting incurred by ownership acquisition. We analyze the ownership acquisition process of STMs below to determine these conditions.

A transaction modifies an orec only if the record does not already point to an owner transaction. If a transaction (hereafter called the **victim transaction** [78]) is pointed by an the ownership record while another transaction (hereafter called the **attacker transaction** [78]) tries to acquire the data item represented by the record, this event generates a conflict. In that case, it is the contention manager that resolves the conflict. After the conflict is resolved there are two cases:

- The attacker transaction is aborted. In this case, the transaction rolls back and safely terminates the current transactional operation. So, the current transactional operation is wait-free (the termination of the transactional operation is of course followed by the restarting of the transaction, but this has no effect on the wait-freedom of the current transactional operation but rather on the overall progress of the transaction).
- The victim transaction is aborted. This implies that the orec can safely be released. For an obstruction-free STM, it is the attacker transaction that directly modifies the orec (using an atomic write) to release (and subsequently acquire) the ownership. For a lock-based STM, however, orecs are the locks and the attacker transaction cannot modify the lock because in lock-based STMs, it is only the owner transaction of a data item who can release the lock. Hence, the attacker transaction needs to wait for the victim transaction to perform its rollback where it will release the locks it owns. The rollback is eventually performed by the victim transaction (running any transactional operation would satisfy this), unless the thread running the victim transaction does not stop running indefinitely long.

In these series of actions described above for ownership acquisition there are two points where indefinite long waiting can be experienced:

- During the execution of the contention manager.

- The period between the moment where the victim transaction aborted, until it becomes aware of this and rolls itself back (only valid for lock-based STMs).

The first point will take a finite duration if the contention manager always returns a decision in finite time and this is generally true for STMs. Hence, for obstruction-free STMs the transactional operations are most generally wait-free.

For lock-based STMs, the second point is also source of waiting. This waiting delay is infinitely long only if the victim transaction's thread stop running indefinitely. For a transaction to stop running indefinitely it should either crash or should be de-scheduled from the running threads queue of the operating system and should be starving due to the operating system scheduler decision. Since practically these two conditions are unlikely, in most cases lock-based STMs do not incur indefinite waiting for the second point. Hence, under practical considerations (where a thread does not wait indefinitely long), a lock-based STM can provide the same progress guarantees as an obstruction-free STM.

Note, however, that the STMs considered in the above discussion are the ones ensuring the basic transactional properties of STMs, i.e., atomicity and weak isolation (among all transactions, i.e., not only among committed transactions). If an STM tries to ensure higher isolation guarantees such as publication, privatization, or irrevocability, it is possible that the proposed solutions for these additional guarantees involve blocking regardless of the fact that the STM is lock-based or not. So, when considering whether an STM ensures the wait-free progress of its transactional operations, one should be careful about whether the isolation guarantee ensured for the STM requires blocking.

A.5.2 Ensuring progress of transactions: Contention management

Ensuring progress of transactions implies that transactions executed by an STM eventually commit. The wait-freedom of transactional operations (explained in **Section A.5.1**) as well as the progress guarantee of transactions all depend on the contention manager of an STM. With this responsibility, a contention manager serves as a keystone in ensuring progress of transactions. Actually, due to the fact that a contention manager resolves conflicts, it also plays an important role in ensuring both atomicity (by resolving write-write conflicts) and isolation (by resolving read-write conflicts).

As explained in **Section A.4.1.5** contention managers resolve conflicts among two transactions using one of two major techniques; ordering or merging. We describe approaches for each solution in separate subsections.

A.5.2.1 Contention management using ordering

The ordering technique for STM contention management is the most widely used technique for contention managers. The aim of this technique is to avoid running two conflicting transactions (at least the portion of transaction codes that will generate a conflict) at the same time. To accomplish this, the technique either delays the attacker transaction (to order the attacker transaction after the victim transaction) or aborts the victim transaction (to order the victim transaction after the attacker transaction). Hence, the major decision to give is how to order the conflicting transactions (this decision determines which action to take, delaying or aborting). The delaying action of a contention manager can be considered as a threat to its non-blocking progress requirement but generally this action lasts only for a finite duration. Hence, generally the more critical decision determining progress is how to order the transactions.

On one hand, it is technically simple to take an ordering decision; the decision can be always the same or even random. The simplicity of the decision is crucial for a contention manager because the contention manager should be non-blocking. On the other hand, the simplest ordering decision does not always remove the cause of conflicts, or even worse can introduce new conflicts. Furthermore, the ordering decision depends also on the workload observed at decision time. Hence, it is not easy to find a decision strategy for contention management. This fact resulted in a variety of contention managers to be proposed. The major ordering decisions used by these proposals are as follows:

- **Static:** There are two static decisions that can be given: always let the victim transaction to execute first (*Passive* contention manager), or always let the attacker to be the first to execute (*Aggressive* contention manager).
- **Timeout:** In this strategy, the attacker transaction decides to let the victim run until a timeout is reached. The timeout is generally determined by a fixed number of delaying intervals. The delaying intervals can be constant (as in *Karma* contention manager) or exponentially growing (as in *Polite* and *Polka* contention managers).
- **Priority:** A simple way to order transactions is to assign them priorities. The priorities are not only determined for two conflicting transactions, but rather for all transactions. However, the priorities are used for the ordering of conflicting transactions. They can be set according to different metrics. One useful metric used for priorities in STMs is the amount of work already performed by a transaction and this is generally deduced from the number of accesses that a transaction has already performed (including accesses performed by the transaction before it aborts). The higher the number of accesses, the higher the priority of the transaction. Upon a conflict, the strategy

behaves as follows: if the attacker has a higher priority it aborts the victim, otherwise it falls back to timeout strategy. Together with the contention management strategy proposed by Spear *et al.* [170], examples of contention managers using this strategy are *Karma*, *Eruption* and *Polka*.

- **Timestamp:** Another method to order transaction is to assign timestamps to the first time a transaction starts executing (this implies when a transaction is aborted its timestamp is not updated). With these timestamp values the transactions are ordered such that the smaller the timestamp the earlier the transaction is in the order. As in priority scheme, the order determined by the timestamps is used mainly to order conflicting transactions only. Upon a conflict, the strategy behaves (in general) similar to priority strategy: if the attacker has a smaller timestamp it aborts the victim, otherwise it falls back to timeout strategy. Contention managers using this strategy are *Greedy*[1] and *Timestamp*. It should be noted that this strategy is easier to be used by STMs using time-based conflict detection where timestamps are already available (see **Section A.4.1.4** for time-based conflict detection). For other STMs using such strategy requires a timestamp to be added to each transaction's metadata.
- **Serialization:** The above strategies aim to order transaction executions but does not necessarily ensure that the conflicting transactions do not run concurrently once the conflict is resolved. In other words, when a victim transaction is aborted to resolve a conflict, it will restart executing, or when an attacker transaction is delayed, its delay is not decided according to the length of the victim transaction. Hence, it is perfectly possible that after the conflict resolution decision, the attacker and victim transactions run concurrently again at a later time. Serialization approach, instead, ensures that one transaction is executed after the other. One way to do this, is delaying an attacker transaction only until the victim transaction commits or aborts. Another approach which is taken by Dolev *et al.* [45] serialization through queuing, i.e., to queue conflicting transactions on the same thread. This is made possible by migrating a transaction from one thread to the other when the transaction is aborted. This approach ensures that the pair of transactions do not conflict again and they run one after the other.

Unfortunately, no contention manager approach gives the best result for all workloads. However, empirically *Polka* contention manager is found to be more efficient than others

[1] Greedy behaves differently than the default strategy when the attacker has a higher timestamp; instead of waiting for some workload independent time interval, it waits for the victim to commit, abort or to wait (i.e., Greedy can abort a victim if it is waiting for some other transaction).

in most cases [90]. A possible reason for this can be that it is a combination of two other contention managers: *Polite* and *Karma*. It has been also observed that switching from one contention manager to the other according to the workload conditions also gives good results [78, 47].

A.5.2.2 Contention management using merging

The main idea in the merging technique is to use helping; i.e. when an attacker transaction detects a conflict it performs the task of the victim transaction on behalf of the thread that started running the victim transaction. However, this is only possible if all the task to be done for a transaction is known while STMs are generally concerned with transactions where the data items to be accessed are not known a priori. Thus, the applicability of the merging technique is limited and it cannot be used as a standalone contention management technique. However, the technique is important in providing progress especially in non-blocking STM systems. Two approaches have been taken in the literature for merging:

- **Using read- and write-sets:** In this approach, helping is achieved literally by performing the accesses of another transaction on behalf another. Such approach was possible in early STMs where the read- and write-sets were assumed to be known a priori, however these STMs are out of the scope of this thesis [164, 134].
- **Using orecs:** This approach is also called **stealing** since it concerns an attacker transaction to transfer the ownership of a data item from the victim transaction onto itself. Again, this technique cannot be used without knowing what action should be taken for the stolen orec. However, in non-blocking STMs, while the STM is performing a cleanup[1] the action to take on the data item is known. Hence, if a conflict occurs while the victim transaction is performing a cleanup, the attacker can steal the ownerships of an orec and act as if the orec were part of its own write-set. This way, the attacker takes the responsibility of performing the cleanup operation corresponding to the given orec. However, managing stealing is complicated and requires careful handling of intricate cases [63, 126].

A.6 Conclusion

In this chapter, we have presented a thorough discussion of the data structures and mechanisms used in STM designs. This discussion has been carried out such that the purpose of

[1] The meaning of a cleanup changes according to the STM update method of an STM (see **Section A.3.2**): in deferred update STMs it corresponds to the writing back the updates onto the data items upon commit, while in direct update STMs it refers to restoring original values upon an abort.

each mechanism (e.g., which property of an STM the mechanism provides) and the relations between different mechanisms are clarified. The explanations provided for the mechanisms allow the reader to do the following:

- Having a good understanding of an overall STM design,
- Locating easily each mechanism in the design,
- Understanding what properties and other mechanisms of the STM will be effected by modifying a given mechanism,
- Having an insight of the effect of each mechanism on STM performance,
- Spotting the issues that need to be considered for improvement of STM designs.

We hope that, this chapter's content is useful for researchers and engineers who need to design an STM or enhance the performance of an existing STM design.

Appendix B

SSTM Algorithm

B.1 Algorithm pseudocode

Algorithm 1 (on the next page) presents the pseudocode of SSTM. In this code, functions are assumed to execute atomically for the sake of simplicity in the presentation.

During the execution of SSTM, a transaction records the accessed variables locally and registers itself as a potentially future conflicting transaction in the accessed variables. These records help SSTM keeping track of all potential conflicts. More precisely, a transaction t accessing variable x keeps track of all transactions that may both precede it and follow it. (respectively in $past\text{-}tx$ and $future\text{-}tx$). Only transactions that read and that are concurrent with t (namely, the active readers of t) can both precede and follow t. This is due to invisible writes that can only be observed by other transactions after commit. When detected, the preceding transactions are recorded in $t.past\text{-}tx$. transaction t). Transaction t detects those transactions either because they are in $x.active\text{-}readers$ (Line 38) or precede one of these (Line 37), or because they are in $x.write\text{-}fc$ (Lines 23 and 38) or precede one of these (Lines 22 and 37). Transaction t also keeps track of its succeeding transactions in $t.future\text{-}tx$ so that it can inform them as soon as it discovers a new preceding transaction. Hence, each transaction t' keeps up-to-date records of $t'.past\text{-}tx$ and $t'.future\text{-}tx$. Transaction t may abort for two reasons. First, if it appears to precede itself in the conflict graph (Lines 21 and 36). Second, if there exists a transaction that t precedes but that also precedes t (Lines 26 and 41). Finally, the *a-clean* function aims at garbage collecting all metadata associated with the current transaction if it aborts whereas the *c-clean* functions garbage collect only the metadata corresponding to the past committed transactions that have nothing in their past, as it is sure these transactions will not create a cycle in the conflict graph later.

Algorithm 1 SSTM – Serializable Software Transactional Memory

```
1:  State of transaction t:
2:    status ∈ {active, inactive}, initially active
3:    write-set ⊂ X × V, initially ∅
4:    read-set ⊂ X, initially ∅
5:    past-tx ⊂ T, initially ∅ // the previous tx in the conflict graph
6:    future-tx ⊂ T, initially ∅ // the next tx in the conflict graph

7:  State of shared variable x:
8:    write-fc ⊂ T, initially ∅ // the write future conflicts
9:    active-readers ⊂ T, initially ∅ // the active reader tx
10:   val ∈ V, initially the default value

11: write(x, v)_t:
12:   write-set ← (write-set \ {⟨x, ∗⟩}) ∪ {⟨x, v⟩}

13: read(x)_t:
14:   read-set ← read-set ∪ {x}
15:   if ⟨x, v'⟩ ∈ write-set then
16:     v ← v'
17:   else
18:     x.active-readers ← x.active-readers ∪ {t}
19:     for all t' in x.write-fc do
20:       for all t'' ∈ t'.past-tx do
21:         if t = t'' then abort()
22:         past-tx ← past-tx ∪ {t''}
23:       past-tx ← past-tx ∪ {t'}
24:     for all t' in past-tx do
25:       for all t'' ∈ future-tx do
26:         if t' = t'' then abort()
27:         t'.future-tx ← t'.future-tx ∪ {t''}
28:       t'.future-tx ← t'.future-tx ∪ {t}
29:     v ← x.val
30:   return v

31: commit()_t:
32:   for all ⟨x, v⟩ ∈ write-set do
33:     x.write-fc ← x.write-fc ∪ {t}
34:     for all t' ∈ x.active-readers ∪ x.write-fc do
35:       for all t'' ∈ t'.past-tx do
36:         if t = t'' then abort()
37:         past-tx ← past-tx ∪ {t''}
38:       if t ≠ t' then past-tx ← past-tx ∪ {t'}
39:     for all t' in past-tx do
40:       for all t'' ∈ future-tx do
41:         if t' = t'' then abort()
42:         t'.future-tx ← t'.future-tx ∪ {t''}
43:       t'.future-tx ← t'.future-tx ∪ {t}
44:   for all ⟨x, v⟩ ∈ write-set do
45:     x.val ← v
46:   status ← inactive
47:   c-clean()

48: abort()_t:
49:   status ← inactive
50:   a-clean()

51: a-clean()_t:
52:   for all x such that ⟨x, ∗⟩ ∈ write-set or x ∈ read-set do
53:     x.write-fc ← x.write-fc \ {t}
54:     x.active-readers ← x.active-readers \ {t}
55:   for all t' ∈ past-tx do
56:     t'.future-tx ← t'.future-tx \ {t}
57:   for all t' ∈ future-tx do
58:     t'.past-tx ← t'.past-tx \ {t}
59:   free(t)

60: c-clean()_t:
61:   for all x such that ⟨x, ∗⟩ ∈ read-set do
62:     x.active-readers ← x.active-readers \ {t}
63:   for all t' ∈ T do
64:     if t'.status = inactive and t'.past-tx = ∅ then
65:       past-tx ← past-tx \ {t'}
66:       for all t'' ∈ t'.future-tx do
67:         t''.past-tx ← t''.past-tx \ {t'}
68:       for all x such that ⟨x, ∗⟩ ∈ t'.write-set do
69:         x.write-fc ← x.write-fc \ {t'}
70:       free(t')
```

B.2 Correctness proof of SSTM

In this section, we show that SSTM, presented in **Section B.1**, is conflict-serializable, but neither opaque nor strict serializable.

Lemma 1. If there exists a conflict $p = t_0 \longrightarrow t_1$, t_0 and t_1 are both committed and $t_0.past\text{-}tx \neq \emptyset$ then $t_1 \in t_0.future\text{-}tx$.

Proof. Observe by definition that $t_0 \longrightarrow t_1$ holds only if there is a conflict between t_0 and t_1, and note that $t_0.past\text{-}tx \neq \emptyset$ prevents t_0 from being cleaned. There are two cases to consider whether the conflicting operations of t_0 is a write. Without loss of generality let x be the common location on which both transactions conflict.

First if t_0 writes x and commits, then t_0 adds itself to $x.write\text{-}fc$ at Line 33. Hence, if t_1 reads x afterwards, then it inserts t_0 in its $t_1.past\text{-}tx$ set at Line 23 and symmetrically

inserts itself in $t_0.\mathit{future\text{-}tx}$ at Line 28. Otherwise, if t_1 writes x afterwards, it inserts t_0 in its $t_1.\mathit{past\text{-}tx}$ set at Line 38 and symmetrically adds itself in $t_0.\mathit{future\text{-}tx}$ at Line 43.

Second if t_0 does not write but reads x before t_1 writes x, then t_0 adds itself to $x.\mathit{active\text{-}reader}$ at Line 18 so that t_1 adds it to $t_1.\mathit{past\text{-}tx}$ at Line 38. Again symmetrically, t_1 inserts itself into $t_0.\mathit{future\text{-}tx}$ at Line 43. The result follows. □

The next lemma shows that the relation, defined by set $t.\mathit{future\text{-}tx}$, between t and the transactions it contains is transitive. Transitivity is necessary to show that a cycle in the conflict graph exists only if a transaction t is in its own $t.\mathit{future\text{-}tx}$.

Lemma 2. Let t_0, t_1, t_2 be three committed transactions. If $t_2 \in t_1.\mathit{future\text{-}tx}$ and $t_1 \in t_0.\mathit{future\text{-}tx}$ then $t_2 \in t_0.\mathit{future\text{-}tx}$.

Proof. Let τ and τ' be the times at which the second operation of the conflict between t_0 and t_1 and the second operations of the conflict between t_1 and t_2 start, respectively. By the assumption of function atomicity, we know that $\tau \neq \tau'$, hence we focus on the two following cases.

In case $\tau' < \tau$, $t_2 \in t_1.\mathit{future\text{-}tx}$ and $t_1 \in t_2.\mathit{past\text{-}tx}$ at time τ. Hence, when the conflict between t_0 and t_1 happens by a read (resp. a write) of t_1, t_1 adds not only t_0 in its $\mathit{past\text{-}tx}$ at Line 23 (resp. at Line 38) and itself to $t_0.\mathit{future\text{-}tx}$ at Line 28 (resp. at Line 43) but also t_2 at Line 27 (resp. at Line 42), which belongs to its $t_1.\mathit{future\text{-}tx}$, to $t_0.\mathit{future\text{-}tx}$.

In case $\tau < \tau'$, $t_0 \in t_1.\mathit{past\text{-}tx}$ at time τ'. Assume t_2 conflicting operation is a read (resp. a write). Transactions t_0, which belongs to $t_1.\mathit{past\text{-}tx}$, and t_1 are inserted in $t_2.\mathit{past\text{-}tx}$ at Line 23 (resp. at Line 38), at time τ'. As a result, t_2 inserts itself to the $\mathit{future\text{-}tx}$ of both t_0 and t_1 at Line 28 (resp. at Line 43). □

Lemma 3. $t \notin t.\mathit{future\text{-}tx}$.

Proof. Assume that $t \in t.\mathit{future\text{-}tx}$ holds, we proceed by contradiction. Transaction t can only be inserted in $t.\mathit{future\text{-}tx}$ at Line 28 or at Line 43 because neither reaching Line 27 nor Line 42 with $t = t'$ is possible as transaction t would abort prior to that (Lines 26 and 41). As a result, t was already in $t.\mathit{past\text{-}tx}$ when Line 28 or 43 has been reached.

Now we show that t cannot be inserted in $t.\mathit{past\text{-}tx}$ leading to the contradiction. If t already belongs $t \in x.\mathit{write\text{-}fc}$, then this means that t is executing its commit and all its read operations are past, hence, there is no chance that t can be added to $t.\mathit{past\text{-}tx}$ during its read operation. Finally, during the execution of a write operation $\mathit{past\text{-}tx}$ remains unchanged, and during the execution of the commit t cannot be inserted into $t.\mathit{past\text{-}tx}$ because $t = t'$ (Line 38). □

B. SSTM ALGORITHM

The following corollary shows that for any history H of SSTM there is no cycle in the serialization graph $SG(H)$ of committing transactions.

Corollary 1. In all histories H of SSTM, there is no path $p = t_1 \longrightarrow ... \longrightarrow t_k \longrightarrow t_1$ such that all t_i commit $(0 < i \leq k)$.

Proof. By absurd, assume that this is possible. We show that this leads to a contradiction. First, by Lemma 1 we know that $p = t_1 \longrightarrow ... \longrightarrow t_k \longrightarrow t_1$ implies that $t_{i+1} \in t_i.\mathit{future\text{-}tx}$ for all i such that $0 < i \leq k-1$ and $t_1 \in t_k.\mathit{future\text{-}tx}$. Second, by the transitivity property of Lemma 2 we obtain that $t_i \in t_i.\mathit{future\text{-}tx}$ $(0 < i \leq k)$ which contradicts Lemma 3.

□

Theorem 1. SSTM is conflict-serializable.

Proof. The proof follows from the conjunction of Corollary 1 and Theorem 2.1 of [23]. □

SSTM is conflict-serializable, however, we have not shown yet that SSTM is neither opaque [79] nor strict serializable [100].

A simple counter-example is presented in Figure 3.4 (center and right-hand side). Clearly, the input (center) is accepted by SSTM resulting in the output on the right-hand side. This output is neither opaque nor linearizable. More precisely, let t_1, t_2, and t_3 be the transactions of p_1, p_2, and p_3, respectively. It is clear that $t_3 \xrightarrow{W} t_1$ and $t_1 \xrightarrow{R} t_2$ implying by transitivity that $t_3 \longrightarrow t_2$, however, because of the real-time precedence requirement common to both opacity and linearizability, $t_3 \not\longrightarrow t_2$ is necessary for the output to be opaque or linearizable. In contrast, this output is equivalent to the sequential execution $t_3 \longrightarrow t_1 \longrightarrow t_2$, thus it is conflict-serializable.

Appendix C

Summary of C++ Draft Specification of Transactional constructs

C.1 Introduction

This section summarizes the specification of C++ language extension for transactional memory support as defined by the Transactional Memory Specification Drafting Group (composed of members from Intel, Sun and IBM) [12]. The specification describes the transactional behavior and features of a compliant TM solution but does not specify how it should be implemented. The specification also expects that the constructs suggested within are well-defined only for programs with no data races, i.e., for programs where accesses to the same data on different threads can be ordered deterministically with respect to each other (see **Section ??** for details).

The specification describes the syntax and semantics of transactional C++ code. This summary describes the specification under the following organization. We describe the fundamental transactional constructs in **Section C.2**. We then explain the types of transactional guarantees proposed by the specification in **Section C.3**. We illustrate the requirements on functions for using them within transactional code in **Section C.4**. We finally describe the support of control flow and nesting for programs including transactional constructs in **Section C.5** and **Section C.6**, respectively.

C.2 Fundamental transactional constructs

The specification allows three language constructs to be executed in a transaction: *compound statements*, *expressions* and *functions*. Transactional compound statements are called **transaction statements**, transactional expressions are called **transaction expressions** and transactional functions are called **function transaction blocks**. The keyword that allows to perform the execution of these three language constructs in a transaction is `__transaction`. The syntax for each of the transactional constructs is as follows:

- **transaction statement:** `__transaction` *compound-statement*
- **transaction expression:** `__transaction` (*expression*)
- **function transaction block:** *function-signature* `__transaction` { *function-body* }

In general, it is common to associate a transaction to a so-called *atomic block*, where the beginning and the end of the block are the start and commit actions and the content is the set of actions for which the transaction guarantees are ensured. Among the three constructs described above, the transaction statement corresponds to an atomic block while the transaction expression and function transaction blocks can be seen as derivations of the transaction statement. A function transaction block is merely a reusable transaction statement, while a transaction expression can be considered as the transactional computation of an expression that is equivalent to the following transaction statement:

`__transaction` { T temp = *expression* }

Hence a transaction expression computes an expression in a transaction and stores the result of the expression in a temporary object.

C.3 Types of transactional guarantees

The `__transaction` keyword can be followed by one of two transactional attributes while declaring a construct transactional: `[[atomic]]` or `[[relaxed]]`. These attributes specify that following guarantee is ensured for described transactional construct:

- The `[[atomic]]` attribute requires that the described transaction executes according to the basic transactional behavior (see **Section 2.5.4.1**). This attribute forbids the use of irrevocable code (see **Section 2.5.4.2**) within the construct.
- The `[[relaxed]]` attribute requires that the described transaction executes according to irrevocable transactional behavior (see **Section 2.5.4.2**) only if it encloses

irrevocable code, otherwise it executes according to basic transactional behavior (see **Section 2.5.4.1**).

A transaction that is assigned an `[[atomic]]` attribute is called an **atomic transaction**, while a transaction assigned a `[[relaxed]]` attribute is called a **relaxed transaction**. Syntactically the attributes just need to follow the `__transaction` keyword to make the distinction between the different transactional guarantees as follows (although below the syntax is given for transaction statement the same construction applies to transaction expression and function transaction blocks):

- **atomic transaction:** `__transaction [[atomic]]` *compound-statement*
- **relaxed transaction:** `__transaction [[relaxed]]` *compound-statement*

The explanations of `[[atomic]]` and `[[relaxed]]` attributes imply that atomic transactions enforce stricter guarantees than relaxed transactions. It is desirable to control that this stricter guarantee is respected at compile time. The specification calls any code that can be enclosed inside a relaxed transaction but not inside an atomic transaction as *unsafe*. More specifically a statement that is used in a transaction is deemed unsafe if any of the following applies [1] :

- The statement is a transaction statement with the `[[outer]]`[2] or `[[relaxed]]` attribute.
- The statement performs any use of volatile object (initialization, assignment or a read).
- The statement is a function call that is
  - either not explicitly declared safe with the `[[transaction_safe]]` or the `[[transaction_may_cancel_outer]]` attributes (see **Section C.4** for these attributes),
  - or not a virtual function and but contains any of the unsafe statements defined above.

A statement is called safe if none of the above applies. According to this definition of safety, atomic transactions are transactions that contain only safe code. Since only relaxed transactions can contain irrevocable code, such code cannot reside in atomic transactions (by the first condition above). Hence, irrevocability mechanisms for transactional memory

[1] There is one more condition mentioned in the specification which is not presented here since it is a restriction on assembly code that can be used in a transaction and is out of the scope of this summary.

[2] The `[[outer]]` attribute only applies to transaction statements and is a nesting related attribute which is explained in **Section C.6**.

can only be used for implementations of relaxed transactions. Note that relaxed transactions provide basic transactional behavior if their content does not include irrevocable code. If a relaxed transaction encloses irrevocable code, it is only then the relaxed transaction provides irrevocable transactional behavior (where the basic transactional behavior guarantees are still ensured while it is only the concurrency of the execution that is hampered).

C.4 Function usage in transactional code

The usage of functions in atomic or relaxed transactions depends on whether the code they contain is safe. If a function contains only safe code it is said to be a *safe* function. A function can be declared safe using two attributes: `[[transaction_safe]]` and `[[transaction_may_cancel_outer]]`[1] . It is also possible to explicitly declare a function unsafe using the `[[transaction_unsafe]]` attribute. Function safety attributes are syntactically located in front of the function signature, e.g., the declaration of a function `f()` returning `void` can be marked with one of the above safety attributes as follows:

```
[[transaction_safe]] void f()
[[transaction_unsafe]] void f()
[[transaction_may_cancel_outer]] void f()
```

Function pointers can also be declared safe or unsafe with the same attributes. This is useful to forbid the assignment of an unsafe function or function pointer to a safe function pointer.

In case of class inheritance member functions preserve safety attributes declared for their base class. For virtual functions the overriding restrictions are as follows: A virtual function explicitly declared safe can only be overridden by a virtual function also explicitly declared safe (except that if the base class function is declared `[[transaction_safe]]`, the derived class function cannot be declared `[[transaction_may_cancel_outer]]` (while the reverse is possible)).

For simplicity of assigning attributes to functions, the specification allows classes to be marked with a function safety attribute (except the `[[transaction_may_cancel_outer]]` attribute). Such a class attribute acts as the default attribute for all the member functions declared in the class unless overridden explicitly by the member function declaration. Also class attributes do not apply to functions inherited from a base class or functions included in the class via the `using` declaration.

[1] The `[[transaction_may_cancel_outer]]` attribute is a control flow and nesting related attribute and is be explained in the corresponding sections (**Section C.5.2** and **Section C.6.2**).

If a function is not explicitly declared safe, it can still be inferred to be safe from its definition if it contains only safe statements. This is useful especially for template functions that do not take function safety attributes, because their safety can only be determined at compile time when the template parameters are known. In such a case, the compiler can decide the safety of the function analyzing the body of the function.

A final function attribute is the `[[transaction_callable]]` attribute but it has no safety implications. It is just a hint for the compiler to indicate that the function is intended to be called in relaxed transactions. Such an attribute can be used if the function is specifically written and optimized for use in relaxed transactions.

C.5 Constructs for control flow in transactions

We analyze three types of control flow related to transactions: regular, transactional and exceptional control flow. The regular and transactional control flow constructs are related to the normal execution while exceptional control flow constructs are related to constructs to be used upon exceptions.

C.5.1 Regular control flow constructs

The regular control flow statements `goto`, `break`, `return` and `continue` can also be used to transfer control out of a transaction. A limitation in normal execution control flow including transactions is that `goto` or `switch` statements must not be used to transfer control into a transaction statement.

C.5.2 Transactional control flow constructs

Apart from the usual control flow statements, transactions have an additional control flow mechanism: *aborting*. If the code requires, a transaction can abort, i.e., can roll-back all its modifications and transfer the control to the statement that follows the transaction. This can be done using a so-called *cancel statement*: the `__transaction_cancel` keyword (the keyword corresponds to the cancel statement by itself). A cancel statement is only allowed inside atomic transactions and only for transaction statements (and not for transaction expressions or function transaction blocks), i.e., the cancel statement should appear inside the transaction statement for which it would perform its roll-back. A programmer can use a cancel statement within a function only if the function is marked with a `[[transaction_may_cancel_outer]]` attribute. Details on this kind of usage is explained in **Section C.6.2**.

It should be noted that an aborted transaction, although not having an effect on the memory once aborted, is part of the program execution and is subject to data races, i.e. a transaction that is aborted using `__transaction_cancel` can still be a reason for a data race occurring in the application.

C.5.3 Exceptional control flow constructs

The specification defines the way a transaction throws an exception and proposes two types of exceptional throw behavior in transactions[1] : *commit-and-throw* and *abort-and-throw*. Both behaviors cause a transaction to throw the desired exception. However, the behaviors differ in the visibility of the effects of the transaction when the exception is thrown out of the transaction. With commit-and-throw behavior the effects of the transaction up to the point where the exception is raised are made visible to other threads with a commit before the exception is thrown. The abort-and-throw behavior, however, roll-backs all the effects of the transaction (also mentioned as cancellation in the specification) and only then throws the exception.

The commit-and-throw behavior is provided by the existing C++ `throw` statement. To specify the abort-and-throw behavior it is enough to prepend the `__transaction_cancel` keyword to the existing C++ `throw` statement. More explicitly, the syntax for each of the behaviors are as follows:

- **commit-and-throw:** `throw` *throw-exception*
- **abort-and-throw:** `__transaction_cancel throw` *throw-exception*

If an exception is to be raised inside a transaction, it requires special care to preserve application consistency because a transaction should guarantee atomicity. If the partial effects of a transaction are visible to other threads at the time it raises an exception, the atomicity guarantee of the transaction will be violated. The commit-and-throw behavior allows this while the abort-and-throw behavior avoid such a problem by throwing the exception only after aborting the transaction. However, for abort-and-throw, since the exception is thrown *after* the transaction is aborted, the exception object cannot carry information about the actions performed in the transaction and, hence, objects that can be thrown inside a transaction are limited to integral and enumerated types.

The specification also extends the exception specification of traditional functions to transaction statements and transaction expressions (the function transaction blocks already

[1] Here, by transaction we mean a transaction statement or a function transaction block. By its nature a transaction expression cannot include a separate throw statement, so it can only support one of the behaviors; namely commit-and-throw)

have the possibility to express throwable exceptions through the exception specification syntax allowed for traditional functions). The syntax for transaction statements and transaction expressions with an exception specification is as follows:

`__transaction` *tx-attrib* `throw(` *type-id-list* `)` *compound-statement*
`__transaction` *tx-attrib* `throw(` *type-id-list* `) (` *expression* `)`

In the above syntax the *tx-attrib* is one of the `[[atomic]]`, `[[relaxed]]` attributes (for a transaction statement *tx-attrib* can also be `[[outer]]`). The part that defines the exception specification is `throw(` *type-id-list* `)`. The *type-id-list* can be one of the following:

- a list of exception object types, e.g., `throw(int, char)`
- empty, i.e., `throw()`. Such syntax means that the transaction throws no exceptions.
- an ellipsis, i.e. `throw(...)`. This syntax means that the transaction can throw *any* exception.

Not specifying any exception specification is accepted to be equivalent to `throw(...)`. If a transaction that throws an exception other than in the exception specification list, this results in a call to `std:unexpected()`.

C.6 Transactional nesting support

C.6.1 Basic nesting

The C++ specification allows both lexical and dynamic nesting of transactions (see **Section 5.5.1** for the definition of lexical and dynamic nesting).

The specification allows nesting of atomic transaction inside atomic and relaxed transactions inside relaxed transactions. Atomic transactions can also be nested in relaxed transactions, but the reverse is forbidden since it would violate safety of atomic transactions.

Semantically, the C++ language extension supports closed nesting, i.e., the abort of an inner transaction does not result in the abort of an outer transaction. The abort of an outer transaction should be specified at the nesting level of the outer transaction (either lexically or dynamically).

C.6.2 Nesting and control flow

The control flow also has support for nesting of transactions. `__transaction_cancel` keyword by itself allows only aborting the innermost transaction, i.e., if the transaction

is nested in other transactions the outer transactions would not be aborted. The specification defines two attributes to be able to abort also outer transactions: `[[outer]]` attribute (for transaction statements and for the `__transaction_cancel` statement), and `[[transaction_may_cancel_outer]]` attribute (for functions).

When a group of nested transactions are aborted by the innermost transaction a chain of aborts occur to abort all the nested transactions in the reverse order to the nesting order (if the abort is explicitly requested to propagate out of the innermost transaction). The `[[outer]]` attribute is used to mark a transaction statement as the outermost transaction where the chain of abort stops. In other words, it specifies that an abort cannot be propagated further out of such transaction. This attribute can only be used by transaction statements and such statements are called *outer atomic transactions*. An outer atomic transaction is accepted to be unsafe by the specification. The syntax illustrating the declaration of an outer atomic transaction is as follows:

`__transaction [[outer]]` *compound-statement*

The `[[outer]]` attribute is also used together with the `__transaction_cancel` statement to form a *cancel-outer statement* as follows:

`__transaction_cancel [[outer]]`

Such a cancel-outer statement aborts the outermost atomic transaction of a group of nested transactions and passes the control to the statement following the outermost atomic transaction. The outermost atomic transaction is the outer atomic transaction that lexically or dynamically contains the transaction where the cancel-outer statement is executed. A cancel-outer statement is allowed to be contained either lexically inside an outer atomic transaction or inside a function that has the `[[transaction_may_cancel_outer]]` attribute.

The `[[transaction_may_cancel_outer]]` attribute is the only way for a function to use cancel-outer statement. It also provides a convenient way for a cancel-outer statement to cancel an outermost atomic transaction from anywhere (rather than being limited with the lexical scope of the outer atomic transaction where it resides in). A function specified with the `[[transaction_may_cancel_outer]]` attribute is called a *cancel-outer function*. A cancel-outer function is accepted to be safe and should not contain unsafe statements.

The location of the `[[transaction_may_cancel_outer]]` attribute in the declaration syntax of a function is the same as the location of the `[[transaction_safe]]` attribute. Since `[[transaction_may_cancel_outer]]` attribute declares a safe function it can nei-

ther be used with `[[transaction_unsafe]]` attribute (this will be a contradiction in the function safety) nor with the `[[transaction_safe]]` attribute (this will be redundant).

C.6.3 Nesting and exception propagation

As the cancel statement can precede a throw statement to first abort a transaction and then propagate an exception, a cancel-outer statement can also precede a throw statement:

`__transaction_cancel [[outer]] throw` *throw-expression*

The semantics associated with adding the `[[outer]]` keyword are that instead of the transaction that encloses the throw, it is the outermost atomic transaction marked with the `[[outer]]` attribute that is aborted and that raises the specified exception. As with cancel-outer statement, such a throw statement is only allowed either to be lexically contained inside an outer atomic transaction or to be inside a function that has the `[[transaction_may_cancel_outer]]` attribute.

References

[1] Dresden TM compiler. http://www.velox-project.eu/software/dtmc. 51, 132

[2] gcc-tm: GCC with TM support. http://www.velox-project.eu/software/gcc-tm. 132

[3] Intel C++ STM Compiler, prototype edition. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/. 51, 132

[4] Multiverse. http://multiverse.codehaus.org/. 43, 132

[5] Sun C++ compiler with TM extensions. See announcement on SkySTM Interest group: http://groups.google.com/group/skystm-interest. 51, 132

[6] Evaluation of the advanced synchronization facility (ASF), 2009. http://forums.amd.com/devblog/blogpost.cfm?threadid=118419&catid=317. 89

[7] Abadi, M., Birrell, A., Harris, T., and Isard, M. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Symposium on Principles of Programming Languages* (Jan. 2008), pp. 63–74. 17, 18, 29, 30, 79, 81, 89, 90, 93

[8] Abadi, M., Harris, T., and Mehrara, M. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Feb. 2009), pp. 185–196. 213

[9] Adl-Tabatabai, A.-R., Kozyrakis, C., and Saha, B. Unlocking concurrency. *ACM Queue 4* (Dec. 2006), 24–33. 2

[10] Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2006), pp. 26–37. 39

[11] Adl-Tabatabai, A.-R., Luchangco, V., Marathe, V. J., Moir, M., Narayanaswamy, R., Ni, Y., Nussbaum, D., Tian, X., Welc, A., and Wu, P. Exceptions and transactions in C++. In *HotPar '09: Proceedings of the First USENIX conference on Hot Topics in Parallelism* (Mar. 2009), pp. 14–18. 36, 38, 79

[12] Adl-Tabatabai, A.-R., and Shpeisman, T. Draft specification of transactional language constructs for C++, version: 1.0. Tech. rep., Transactional Memory Specification Drafting Group (Intel, IBM and Sun), Aug. 2009. 7, 52, 116, 130, 132, 231

[13] Adve, S. V., and Hill, M. D. Weak ordering – a new definition. In *ISCA '90: Proceedings of the 17th annual International Symposium on Computer Architecture* (June 1990), pp. 2–14. 30

[14] Afek, Y., Drepper, U., Felber, P., Fetzer, C., Gramoli, V., Hohmuth, M., Riviere, E., Stenstrom, P., Unsal, O. S., Moreira, W. M., Harmanci, D., Marlier, P., Diestelhorst, S., Pohlack, M., Cristal, A., Hur, I., Dragojevic, A., Guerraoui, R., Kapalka, M., Tomic, S., Korland, G., Shavit, N., Nowack, M., and Riegel, T. The Velox transactional memory stack. *IEEE Micro Special Issue - European Multicore Computing 30*, 5 (Sept. 2010), 76–87. 7

[15] Afek, Y., Morrison, A., and Tzafrir, M. Brief announcement: View transactions: Transactional model with relaxed consistency checks. In *PODC '10: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing* (July 2010), pp. 65–66. 28

[16] Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., and Watson, I. Lee-TM: A non-trivial benchmark for transactional memory. In *ICA3PP '08: Proceedings of the International Conference on Algorithms and Architectures 2008* (June 2008), pp. 196–207. 86

[17] Aslot, V., Domeika, M. J., Eigenmann, R., Gaertner, G., Jones, W. B., and Parady, B. SPEComp: A new benchmark suite for measuring parallel com-

puter performance. In *WOMPAT '01: International Workshop on OpenMP Applications and Tools* (July 2001), pp. 1–10. 85

[18] Aydonat, U., and Abdelrahman, T. S. Serializability of transactions in software transactional memory. In *TRANSACT '08: 3rd ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2008). 60, 70, 74, 75, 76, 77

[19] Baugh, L., and Zilles, C. Analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing* (Aug. 2007). 42, 44

[20] Beder, D., Randell, B., Romanovsky, A., and Rubira, C. On applying coordinated atomic actions and dependable software architectures for developing complex systems. In *ISORC '01: Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (May 2001), pp. 103–112. 152

[21] Ben-Asher, Y., Farchi, E., and Eytani, Y. Heuristics for finding concurrent bugs. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Apr. 2003), p. 288.1. 84

[22] Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13* (June 1981), 185–221. 28

[23] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. 70, 75, 230

[24] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques* (Oct. 2008), pp. 72–81. 85

[25] Binder, W., Hulaas, J., and Moret, P. Advanced Java bytecode instrumentation. In *PPPJ '07: Proceedings of the International Symposium on Principles and Practice of Programming in Java* (Sept. 2007), pp. 135–144. 136

[26] Blundell, C., Lewis, E. C., and Martin, M. M. K. Deconstructing transactions: The subtleties of atomicity. In *WDDD '05: 4th Workshop on Duplicating, Deconstructing, and Debunking* (June 2005). 3, 17, 86

[27] Blundell, C., Lewis, E. C., and Martin, M. M. K. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr. 2006. 47, 214, 216, 217

[28] Bruening, D., and Chapin, J. Systematic testing of multithreaded Java programs. Tech. Rep. LCS-TM-607, MIT/LCS, 2000. 84

[29] Cabral, B., and Marques, P. Exception handling: A field study in Java and .NET. In *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming* (Aug. 2007), vol. 4609 of *LNCS*, pp. 151–175. 148

[30] Cabral, B., and Marques, P. Implementing retry – featuring AOP. In *LADC '09: Proceedings of the 2009 Fourth Latin-American Symposium on Dependable Computing* (Sept. 2009), pp. 73–80. 37, 153

[31] Capozucca, A., Guelfi, N., Pelliccione, P., Romanovsky, A., and Zorzo, A. F. Frameworks for designing and implementing dependable systems using coordinated atomic actions: A comparative study. *J. Syst. Softw. 82* (February 2009), 207–228. 152

[32] Carlstrom, B. D., Chung, J., Chafi, H., McDonald, A., Cao Minh, C., Hammond, L., Kozyrakis, C., and and Olukotun, K. Transactional execution of Java programs. In *SCOOL '05: Workshop on Synchronization and Concurrency in Object-Oriented Languages* (Oct. 2005). 40

[33] Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2006), pp. 1–13. 39, 40, 43, 177

[34] Chen, J., and MacDonald, S. Testing concurrent programs using value schedules. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering* (Nov. 2007), pp. 313–322. 85

[35] Christie, D., Chung, J.-W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., and

Rivière, E. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems* (Apr. 2010), pp. 27–40. 132

[36] Chung, J., Chafi, H., Minh, C., McDonald, A., Carlstrom, B., and Kozyrakis, C. Olukotun, K. The common case transactional behavior of multithreaded programs. In *HPCA '06: Proceedings of the International Symposium on High Performance Computer Architecture* (Feb. 2006), pp. 266–277. 86

[37] Crowl, L., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Integrating transactional memory into C++. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing* (Aug. 2007). 34, 36, 37, 38

[38] Dalessandro, L., Marathe, V. J., Spear, M. F., and Scott, M. L. Capabilities and limitations of library-based software transactional memory in C++. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing* (Aug. 2007). 59, 109

[39] Dalessandro, L., and Scott, M. L. Strong isolation is a weak idea. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2009). 17, 29, 30

[40] Dalessandro, L., Spear, M. F., and Scott, M. L. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Jan. 2010), pp. 67–78. 20, 187, 191, 204, 209, 210

[41] Demsky, B., and Dash, A. Evaluating contention management using discrete event simulation. In *5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'10)* (Apr. 2010). 87

[42] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *DISC'06: Proc. of the 20th International Symposium on Distributed Computing* (Sept. 2006), pp. 194–208. 26, 58, 66, 70, 72, 83, 86, 136, 187, 193, 197, 203, 206, 211

[43] Dice, D., and Shavit, N. What really makes transactions faster? In *TRANSACT '06: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006). 19, 211

[44] Doherty, S., Groves, L., Luchangco, V., and Moir, M. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci. 259* (Dec. 2009), 245–261. 17, 26, 27

[45] Dolev, S., Hendler, D., and Suissa, A. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of Distributed Computing* (Aug. 2008), pp. 125–134. 223

[46] Dragojević, A., Felber, P., Gramoli, V., and Guerraoui, R. Why STM can be more than a research toy. *Commun. ACM 54* (Apr. 2011), 70–77. 55

[47] Dragojević, A., Guerraoui, R., and Kapalka, M. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2009), pp. 155–165. 60, 83, 187, 193, 204, 206, 224

[48] Drepper, U. Parallel programming with transactional memory. *ACM Queue 6* (Sept. 2008), 38–45. 1

[49] Dudnik, P., and Swift, M. Condition variables and transactional memory: Problem or opportunity? In *TRANSACT'09: 4th ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2009). 40

[50] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., and Ur, S. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience 15*, 3-5 (2003), 485–499. 82, 84

[51] Ekman, T., and Hedin, G. The Jastadd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2007), pp. 1–18. 138

[52] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: Efficiently computing the happens-before relation using locksets. *Formal Approaches to Software Testing and Runtime Verification* (2006), 193 – 208. 84

[53] Ennals, R. Efficient software transactional memory. Tech. Rep. IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan. 2005. 187, 188, 191

[54] Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurr. Comput. : Pract. Exper. 19*, 3 (2007), 267–279. 83

[55] Felber, P., Fetzer, C., Guerraoui, R., and Harris, T. Transactions are back—but are they the same? *SIGACT News 39* (Mar. 2008), 48–58. 3, 15, 16

[56] Felber, P., Fetzer, C., Müller, U., Riegel, T., Süsskraut, M., and Sturzrehm, H. Transactifying applications using an open compiler framework. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing* (Aug. 2007). 43

[57] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *PPoPP 08: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb. 2008), pp. 237–246. 59, 66, 70, 71, 83, 86, 107, 187, 193, 196, 197, 206

[58] Felber, P., Gramoli, V., and Guerraoui, R. Elastic transactions. In *DISC '09: Proceedings of the 23rd International Symposium on Distributed Computing* (Sept. 2009), vol. 5805 of *LNCS*, pp. 93–107. 27, 61, 83, 86

[59] Fetzer, C., and Felber, P. Improving program correctness with atomic exception handling. *Journal of Universal Computer Science 13*, 8 (2007), 1047–1072. 5, 37, 153

[60] Filho, F., and Rubira, C. F. Implementing coordinated error recovery for distributed object-oriented systems with AspectJ. *Journal of Universal Computer Science 10*, 7 (2004), 843–858. 152

[61] Flanagan, C., and Freund, S. N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL'04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Jan. 2004), pp. 256–267. 84

[62] Fraser, K. Pratical lock-freedom, 2004. 35

[63] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), article 5. 187, 188, 200, 224

[64] Fu, C., Wu, Z., Wang, X., and Yang, X. A review of software transactional memory in multicore processors. *Information Technology Journal 8*, 8 (2009), 1269–1274. 31

[65] Gajinov, V., Zyulkyarov, F., Unsal, O. S., Cristal, A., Ayguade, E., Harris, T., and Valero, M. QuakeTM: Parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing* (June 2009), pp. 126–135. 86

[66] Godefroid, P. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Jan. 1997), pp. 174–186. 85

[67] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java(TM) Language Specification, (Third Edition)*. Addison-Wesley, 2005. 52

[68] Gottschlich, J. E., Siek, J. G., Vachharajani, M., Winkler, D. Y., and Connors, D. A. An efficient lock-aware transactional memory implementation. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (July 2009), pp. 10–17. 46

[69] Gramoli, V., Harmanci, D., and Felber, P. Toward a theory of input acceptance for transactional memories. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems* (Dec. 2008), vol. 5401 of *LNCS*, pp. 527–533. 6, 79, 81, 89, 94, 100

[70] Gramoli, V., Harmanci, D., and Felber, P. On the input acceptance of transactional memory. *Parallel Processing Letters (PPL) 20*, 1 (mar 2010), 31–50. 6, 54, 55

[71] Gray, J. The transaction concept, virtues and limitations. In *VLDB '81: Proceedings of the 7th International Conference on Very Large Data Bases* (Sept. 1981), pp. 144–154. 14

[72] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992. 14, 42, 59, 98

[73] Grossman, D. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2007), pp. 695–706. 5

[74] Grossman, D., Manson, J., and Pugh, W. What do high-level memory models mean for transactions? In *MSPC '06: Proceedings of the 2006 workshop on Memory System Performance and Correctness* (Oct. 2006), pp. 62–69. 17, 53, 81

[75] Guerraoui, R., Henzinger, T. A., Jobstmann, B., and Singh, V. Model checking transactional memories. In *PLDI '08: Proceedings of the 2008 Conference on Programming Language Design and Implementation* (June 2008), pp. 372–382. 85

[76] Guerraoui, R., Henzinger, T. A., and Singh, V. Nondeterminism and completeness in transactional memories. In *CONCUR '08: Proceedings of the 19th International Conference on Concurrency Theory* (Aug. 2008), pp. 21–35. 85

[77] Guerraoui, R., Henzinger, T. A., and Singh, V. Software transactional memory on relaxed memory models. In *CAV '09: Proceedings of the 21st International Conference On Computer Aided Verification* (June 2009), pp. 321–336. 85

[78] Guerraoui, R., Herlihy, M., and Pochon, B. Polymorphic contention management. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing* (Sept. 2005), vol. 3724 of *LNCS*, pp. 303–323. 58, 66, 70, 191, 204, 220, 224

[79] Guerraoui, R., and Kapałka, M. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb. 2008), pp. 175–184. 16, 24, 26, 58, 61, 75, 81, 89, 94, 230

[80] Guerraoui, R., and Kapałka, M. *Principles of Transactional Memory*. Morgan & Claypool Publishers, California, 2010. 219

[81] Guerraoui, R., Kapałka, M., and Vitek, J. STMBench7: A benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (Mar. 2007), pp. 315–324. 86

[82] Haines, N., Kindred, D., Morrisett, J. G., Nettles, S. M., and Wing, J. M. Composing first-class transactions. *ACM Trans. Program. Lang. Syst. 16* (Nov. 1994), 1719–1736. 152

[83] Hammer, C., Dolby, J., Vaziri, M., and Tip, F. Dynamic detection of atomic-set-serializability violations. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (May 2008), pp. 231–240. 83

[84] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer architecture* (June 2004), pp. 102–113. 47, 216

[85] Harmanci, D., Felber, P., Gramoli, V., and Fetzer, C. TMunit: Testing software transactional memories. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2009). 7

[86] Harmanci, D., Gramoli, V., and Felber, P. Atomic boxes: Coordinated exception handling with transactional memory. In *ECOOP '11: Proceedings of the 25th European Conference on Object Oriented Programming* (July 2011), vol. 6813 of *LNCS*, pp. 634–657. 8

[87] Harmanci, D., Gramoli, V., Felber, P., and Fetzer, C. Extensible transactional memory testbed. *Journal of Parallel and Distributed Computing - Special Issue on Transactional Memory (JPDC) 70*, 10 (2010), 1053–1067. 7

[88] Harris, T. Exceptions and side-effects in atomic blocks. In *CSJP '04: Proceedings of the Workshop on Concurrency and Synchronization in Java programs* (July 2004), pp. 46–53. paper also published in Sci. of Comp. Prog. in December 2005. 36, 38, 43, 45, 153

[89] Harris, T., and Fraser, K. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications* (Oct. 2003), pp. 388–402. 39, 57, 66, 70, 72, 83, 187, 191, 200

[90] Harris, T., Larus, J. R., and Rajwar, R. *Transactional Memory*, 2nd edition ed. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher, 2010. 1, 2, 3, 11, 14, 15, 16, 17, 18, 22, 28, 31, 33, 35, 39, 46, 53, 81, 82, 191, 193, 196, 197, 199, 211, 218, 219, 224

[91] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (June 2005), pp. 48–60. 12, 13, 37, 39, 153

[92] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 Conference on Programming Language Design and Implementation* (June 2006), pp. 14–25. 191, 196, 197

[93] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (Jan 1991), 124–149. 31

[94] Herlihy, M., and Lev, Y. tm_db: A generic debugging library for transactional programs. In *PACT '09: Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques* (Sept. 2009), pp. 136–145. 85

[95] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (May 2003), pp. 522–529. 31, 32, 218

[96] Herlihy, M., Luchangco, V., and Moir, M. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2006), pp. 253–262. 135

[97] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of Distributed Computing* (July 2003), pp. 92–101. 32, 56, 66, 70, 71, 86, 187, 188, 189, 191, 199, 202, 204, 218

[98] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual International Symposium on Computer architecture* (May 1993), pp. 289–300. 11

[99] Herlihy, M., and Shavit, N. *The art of multiprocessor programming*. Morgan Kaufmann, 2008. 31

[100] Herlihy, M., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems 12*, 3 (1990), 463–492. 25, 75, 230

[101] Hindman, B., and Grossman, D. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory System Performance and Correctness* (Oct. 2006), pp. 82–91. 132, 212

[102] Ho, A., Smith, S., and Hand, S. On deadlock, livelock, and forward progress. Tech. Rep. UCAM-CL-TR-633, University of Cambridge Computer Laboratory, 2005. 30

[103] Hudson, R. L., Saha, B., Adl-Tabatabai, A.-R., and Hertzberg, B. C. McRT-Malloc: A scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management* (June 2006), pp. 74–83. 211

[104] Imbs, D., González de Mendvil, J. R., and Raynal, M. Virtual world consistency: A new condition for STM systems. In *PODC '09: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Aug. 2009), pp. 280–281. 26

[105] Issarny, V. An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software. *J. of Object-Oriented Programming 6*, 6 (1993), 29–40. 152

[106] Jacobs, B., and Piessens, F. Failboxes: Provably safe exception handling. In *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming* (July 2009), vol. 5653 of *LNCS*, pp. 470–494. 149, 153, 169, 172, 177, 180

[107] Jaleel, A., Mattina, M., and Jacob, B. Last level cache (LLC) performance of data mining workloads on a CMP – a case study of parallel bioinformatics workloads. In *HPCA '06: Proceedings of the International Symposium on High Performance Computer Architecture* (Feb. 2006), pp. 88–98. 86

[108] Jimenez-Peris, R., Patino-Martinez, M., Arevalo, S., Peris, R. J., Ballesteros, F., and Carlos, J. TransLib: an Ada 95 object oriented framework for building transactional applications. *Computer Systems: Science & Engineering Journal 15* (2000), 113–125. 152

[109] Kestor, G., Stipic, S., Unsal, O. S., Cristal, A., and Valero, M. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2009). 55, 86

[110] Kienzle, J., and Romanovsky, A. Combining tasking and transactions, part II: Open multithreaded transactions. In *IRTAW '00: Proceedings of the 10th International Real-Time Ada Workshop* (Sept. 2000), pp. 67–74. 152

[111] Korland, G., Shavit, N., and Felber, P. Deuce: Noninvasive software transactional memory in Java. *Transactions on HiPEAC 5*, 2 (2010). 43, 132, 133, 135, 217

[112] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28* (Sept. 1979), 690–691. 29

[113] Larus, J., and Kozyrakis, C. Transactional memory. *Communications of the ACM 51* (July 2008), 80–88. 1, 2, 15, 19, 21

[114] Lea, D. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1999. 9, 13

[115] Lei, Y., and Carver, R. H. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering 32*, 6 (June 2006), 382–403. 84

[116] Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., and Olszewski, M. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing* (Feb. 2009). 19, 22, 59, 202, 210

[117] Lev, Y., and Maessen, J.-W. Toward a safer interaction with transactional memory by tracking object visibility. In *SCOOL '05: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages* (Oct. 2005). 212

[118] Liskov, B. Distributed programming in Argus. *Commun. ACM 31* (March 1988), 300–312. 152

[119] Long, B., Hoffman, D., and Strooper, P. Tool support for testing concurrent Java components. *IEEE Trans. Softw. Eng. 29*, 6 (2003), 555–566. 85

[120] Lourenço, J., and Cunha, G. Testing patterns for software transactional memory engines. In *PADTAD '07: Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging* (July 2007), pp. 36–42. 17, 81, 89

[121] Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th*

International Conference on Architectural Support for Programming Languages and Operating Systems (Oct. 2006), pp. 37–48. 84

[122] Luchangco, V., and Marathe, V. J. Transaction synchronizers. In *SCOOL '05: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages* (Oct. 2005). 40, 41

[123] Lucia, B., Devietti, J., Strauss, K., and Ceze, L. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 277–288. 84

[124] Man-Lap Li Sasanka, R., Adve, S., Chen, Y.-K., and Debes, E. The ALPBench benchmark suite for complex multimedia applications. In *IISWC '05: IEEE International Symposium on Workload Characterization* (Oct. 2005), pp. 34–45. 86

[125] Manovit, C., Hangal, S., Chafi, H., McDonald, A., Kozyrakis, C., and Olukotun, K. Testing implementations of transactional memory. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques* (Sept. 2006), pp. 134–143. 84

[126] Marathe, V. J., and Moir, M. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb. 2008), pp. 227–236. 196, 224

[127] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. Lowering the overhead of software transactional memory. In *TRANSACT '06: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006). Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006. 32, 59, 83, 187, 188, 191, 199, 202, 204

[128] Marathe, V. J., Spear, M. F., and Scott, M. L. Scalable techniques for transparent privatization in software transactional memory. In *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing* (Sept. 2008), pp. 67–74. 18, 22, 210

[129] McDonald, A., Chung, J., Carlstrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural semantics for practical transac-

tional memory. In *ISCA '06: Proceedings of the 33rd annual International Symposium on Computer Architecture* (June 2006), pp. 53–65. 39, 40, 43, 44

[130] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R. L., Saha, B., and Welc, A. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures* (June 2008), pp. 314–325. xxi, 17, 18, 19, 20, 21, 79, 81, 89, 94, 96, 97, 113, 209, 210

[131] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R. L., Saha, B., and Welc, A. Single global lock semantics in a weakly atomic STM. In *TRANSACT'08: 3rd ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2008). 28, 29

[132] Miletić, N., Smiljković, V., Perfumo, C., Harris, T., Cristal, A., Hur, I., Unsal, O., and Valero, M. Transactification of a real-world system library. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing* (Apr. 2010). 43, 44, 45, 49, 50

[133] Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: IEEE International Symposium on Workload Characterization* (Sept. 2008), pp. 35–46. 55, 86

[134] Moir, M. Transparent support for wait-free transactions. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms* (Sept. 1997), pp. 305–319. 224

[135] Moore, K. F., and Grossman, D. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Jan. 2008), pp. 51–62. 29, 53

[136] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logTM. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2006), pp. 359–370. 42, 43, 45

[137] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*

'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (Dec. 2008), pp. 267–280. 82, 84

[138] Napper, J., and Alvisi, L. Lock-free serializable transactions. Tech. Rep. TR-05-04, Department of Computer Sciences, University of Texas at Austin, 2005. 75

[139] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Memik, G., and Choudhary, A. MineBench: A benchmark suite for data mining workloads. In *IISWC '06: IEEE International Symposium on Workload Characterization* (Oct. 2006), pp. 182–188. 86

[140] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2008), pp. 195–212. 36, 47, 132, 216, 218

[141] O'Callahan, R., and Choi, J.-D. Hybrid dynamic data race detection. In *PPoPP'03: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (June 2003), pp. 167–178. 84

[142] O'Leary, J., Saha, B., and Tuttle, M. R. Model checking transactional memory with Spin. In *ICDCS '09: Proceedings of the 29th International Conference on Distributed Computing Systems* (June 2009), pp. 335–342. 85

[143] Olszewski, M., Cutler, J., and Steffan, J. G. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Sept. 2007), pp. 365–375. 47, 133, 191, 204, 210, 216

[144] Olszewski, M., Mierle, K., Czajkowski, A., and Brown, A. D. JIT instrumentation: A novel approach to dynamically instrument operating systems. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (Mar. 2007), pp. 3–16. 43

[145] Olukotun, K., and Hammond, L. The future of microprocessors. *ACM Queue 3* (Sept. 2005), 26–29. 1

[146] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653. 25

[147] Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., and Holmes, D. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005. 55

[148] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *CF '08: Conference on Computing Frontiers* (May 2008), pp. 67–78. 86

[149] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Oct. 2009), pp. 161–176. 49

[150] Pozniansky, E., and Schuster, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (June 2003), pp. 179–190. 84

[151] Pugh, W., and Ayewah, N. Unit testing concurrent software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering* (Nov. 2007), pp. 513–516. 85

[152] Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R., and Pattabiraman, K. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Feb. 2009), pp. 173–184. 18

[153] Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing* (Sept. 2006), pp. 284–298. 43, 132, 136, 193, 206

[154] Riegel, T., Fetzer, C., and Felber, P. Time-based transactional memory with scalable time bases. In *SPAA '07: Proceedings of the nineteenth annual Symposium on Parallel Algorithms and Architectures* (June 2007), pp. 221–228. 205

[155] Riegel, T., Fetzer, C., Sturzrehm, H., and Felber, P. From causal to z-linearizable transactional memory. Tech. Rep. RR-I-07-02.1, Université de Neuchâtel, Switzerland, 2007. 81, 89, 94

[156] Ringenburg, M. F., and Grossman, D. AtomCaml: First-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming* (Sept. 2005), pp. 92–104. 36, 39, 40, 43, 45

[157] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Aditya, B., and Witchel, E. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Oct. 2007), pp. 87–102. 47, 49

[158] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Mar. 2006), pp. 187–197. 187, 188, 191, 196

[159] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997), 391–411. 83

[160] Schindewolf, M., Cohen, A., Karl, W., Marongiu, A., and Benini, L. Towards transactional memory support for GCC. In *GROW-2009: International Workshop on GCC Research Opportunities* (Jan. 2009). 132

[161] Schneider, F. T., Menon, V., Shpeisman, T., and Adl-Tabatabai, A.-R. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2008), pp. 181–194. 213

[162] Scott, M. L. Sequential specification of transactional memory semantics. In *TRANSACT '06: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006). Held in conjunction with PLDI 2006. 17, 204

[163] Scott, M. L., Spear, M. F., Dalessandro, L., and Marathe, V. J. Transactions and privatization in delaunay triangulation. In *PODC '07: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing* (Aug. 2007), pp. 336–337. Brief announcement. 19

[164] Shavit, N., and Touitou, D. Software transactional memory. In *PODC '95: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing* (Aug. 1995), pp. 204–213. 183, 224

[165] Shinnar, A., Tarditi, D., Plesko, M., and Steensgaard, B. Integrating support for undo with exception handling. Tech. Rep. MSR-TR-2004-140, Microsoft Research, Dec. 2004. 37, 153

[166] Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K. F., and Saha, B. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2007), pp. 78–88. 17, 81, 89, 98, 212

[167] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. Transactions with isolation and cooperation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2007), pp. 191–210. 5, 40, 41, 45, 48, 79

[168] Spear, M., Silverman, M., Dalessandro, L., Michael, M., and Scott, M. Implementing and exploiting inevitability in software transactional memory. In *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing* (Sept. 2008), pp. 59–66. 47, 214, 215, 216

[169] Spear, M. F., Dalessandro, L., Marathe, V. J., and Scott, M. L. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems* (Dec. 2008), pp. 275–294. 18, 22

[170] Spear, M. F., Dalessandro, L., Marathe, V. J., and Scott, M. L. A comprehensive strategy for contention management in software transactional memory. In *PPoPP'09: Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Feb. 2009), pp. 141–150. 59, 110, 113, 197, 223

[171] Spear, M. F., Marathe, V. J., Dalessandro, L., and Scott, M. L. Privatization techniques for software transactional memory. Tech. Rep. TR 915, Department of Computer Science, University of Rochester, Feb. 2007. 19, 26, 208, 210, 211

[172] Spear, M. F., Marathe, V. J., Scherer III, W. N., and Scott, M. L. Conflict detection and validation strategies for software transactional memory. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing* (Sept. 2006), pp. 179–193. 192, 202, 203

[173] Spear, M. F., Michael, M., and Scott, M. L. Inevitability mechanisms for software transactional memory. In *TRANSACT '08: 3rd ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2008). 47, 216

[174] Spear, M. F., Michael, M. M., and von Praun, C. RingSTM: Scalable transactions with a single atomic instruction. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures* (June 2008), pp. 275–284. 210

[175] Spector, A. Z., Daniels, D. S., Duchamp, D., Eppinger, J. L., and Pausch, R. F. Distributed transactions for reliable systems. In *SOSP'85: Proceedings of the Tenth ACM Symposium on Operating System Principles* (Dec. 1985), pp. 127–146. 151

[176] Stelting, S. *Robust Java: Exception Handling, Testing and Debugging*. Prentice Hall, 2005. 148

[177] Stoller, S. D. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)* (July 2002), vol. 70(4) of *Electronic Notes in Theoretical Computer Science*, Elsevier. 84

[178] Stroustrup, B. Exception safety: Concepts and techniques. In *Advances in Exception Handling Techniques* (London, UK, 2001), Springer-Verlag, pp. 60–76. the book grow out of a ECOOP 2000 workshop. 37

[179] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30* (2005). 1

[180] Sutter, H., and Larus, J. Software and the concurrency revolution. *ACM Queue 3* (Sept. 2005), 54–62. 1, 2

[181] Tabba, F., Moir, M., Goodman, J. R., Hay, A., and Wang, C. NZTM: Nonblocking zero-indirection transactional memory. In *SPAA '09: Proc. 21st Symposium on Parallelism in Algorithms and Architectures* (Aug. 2009), pp. 204–213. 188, 199

[182] Tai, K.-C., Carver, R. H., and Obaid, E. E. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. Softw. Eng. 17*, 1 (1991), 45–63. 82, 84

[183] Volos, H., Goyal, N., and Swift, M. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd ACM SIGPLAN Workshop on Transactional Computing* (Feb. 2008). 46

[184] Volos, H., Tack, A. J., Goyal, N., Swift, M. M., and Welc, A. xCalls: Safe I/O in memory transactions. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems* (Apr. 2009), pp. 247–260. 42, 43, 45, 49, 79

[185] von Praun, C., and Gross, T. R. Object race detection. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2001), pp. 70–82. 83

[186] Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (Mar. 2007), pp. 34–48. 211

[187] Wang, L., and Stoller, S. D. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng. 32*, 2 (2006), 93–110. 84

[188] Welc, A., Jagannathan, S., and Hosking, A. Safe futures for Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Oct. 2005), pp. 439–453. 151

[189] Welc, A., Jagannathan, S., and Hosking, A. L. Transactional monitors for concurrent objects. In *ECOOP '04: 18th European Conference for Object-Oriented Programming* (June 2004), pp. 519–542. 40

[190] Welc, A., Saha, B., and Adl-Tabatabai, A.-R. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures* (June 2008), pp. 285–296. 5, 47, 214, 215, 216

[191] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*

'95: Proceedings of the 22nd annual International Symposium on Computer architecture (June 1995), pp. 24–36. 85

[192] Xu, J., Randell, B., Romanovsky, A., Rubira, C. M. F., Stroud, R. J., and Wu, Z. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95: Proceedings of th 25th Internationl Symposium on Fault Tolerant Computing* (June 1995), pp. 499–508. 152

[193] Xu, M., Bodík, R., and Hill, M. D. A serializability violation detector for shared-memory server programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2005), pp. 1–14. 84

[194] Yoo, R. M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., and Lee, H.-H. S. Kicking the tires of software transactional memory: Why the going gets tough? In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures* (June 2008), pp. 265–274. 210

[195] Yu, Y., Rodeheffer, T., and Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP'05: Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Oct. 2005), pp. 221–234. 84

[196] Ziarek, L., Schatz, P., and Jagannathan, S. Stabilizers: A modular checkpointing abstraction for concurrent functional programs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming* (Sept. 2006), pp. 136–147. 152

[197] Ziarek, L., Welc, A., Adl-Tabatabai, A.-R., Menon, V., Shpeisman, T., and Jagannathan, S. A uniform transactional execution environment for Java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming* (July 2008), pp. 129–154. 47, 48, 132

[198] Zilles, C., and Baugh, L. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT '06: Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006). 39, 45

[199] Zorzo, A. F., and Stroud, R. J. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th annual*

ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (Nov. 1999), pp. 435–446. 152

[200] Zyulkyarov, F., Cvijic, S., Unsal, O., Cristal, A., Ayguade, E., Harris, T., and Valero, M. WormBench - A configurable workload for evaluating transactional memory systems. In *MEDEA Wokshop* (Oct. 2008), pp. 61–68. 86

[201] Zyulkyarov, F., Gajinov, V., Unsal, O. S., Cristal, A., Ayguadé, E., Harris, T., and Valero, M. Atomic Quake: Using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Feb. 2009), pp. 25–34. 55, 86

[202] Zyulkyarov, F., Harris, T., Unsal, O. S., Cristal, A., and Valero, M. Debugging programs that use atomic blocks and transactional memory. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (Jan. 2010), pp. 57–66. 85

Index

INDEX