Formalizing and Verifying Transactional Memories

THÈSE N^{O} 4541 (2009)

PRÉSENTÉE LE 7 AVRIL 2010

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE MODÈLES ET THÉORIE DE CALCULS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Vasu Singh

acceptée sur proposition du jury:

Prof. B. Faltings, président du jury
Prof. T. Henzinger, directeur de thèse
Prof. R. Alur, rapporteur
Prof. H. Attiya, rapporteur
Prof. R. Guerraoui, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

Abstract

Transactional memory (TM) has shown potential to simplify the task of writing concurrent programs. TM shifts the burden of managing concurrency from the programmer to the TM algorithm. The correctness of TM algorithms is generally proved manually. The goal of this thesis is to provide the mathematical and software tools to automatically verify TM algorithms under realistic memory models.

Our first contribution is to develop a mathematical framework to capture the behavior of TM algorithms and the required correctness properties. We consider the safety property of opacity and the liveness properties of obstruction freedom and livelock freedom. We build a specification language of opacity. We build a framework to express hardware relaxed memory models. We develop a new high-level language, Relaxed Memory Language (RML), for expressing concurrent algorithms with a hardware-level atomicity of instructions, whose semantics is parametrized by various relaxed memory models. We express TM algorithms like TL2, DSTM, and McRT STM in our framework.

The verification of TM algorithms is difficult because of the unbounded number, length, and delay of concurrent transactions and the unbounded size of the memory. The second contribution of the thesis is to identify structural properties of TM algorithms which allow us to reduce the unbounded verification problem to a language-inclusion check between two finite state systems. We show that common TM algorithms satisfy these structural properties.

The third contribution of the thesis is our tool FOIL for model checking TM algorithms. FOIL takes as input the RML description of a TM algorithm and the description of a memory model. FOIL uses the operational semantics of RML to compute the language of the TM algorithm for two threads and two variables. FOIL then checks whether the language of the TM algorithm is included in the specification language of opacity. FOIL automatically determines the locations of fences, which if inserted, ensure the correctness of the TM algorithm under the given memory model. We use FOIL to verify DSTM, TL2, and McRT STM under the memory models of sequential consistency, total store order, partial store order, and relaxed memory order.

Keywords: Concurrency, Transactional memories, Semantics, Specification, Software verification, Model checking, Relaxed memory models.

Résumé

Les Mémoires Transactionnelles (MT) ont démontré un grand potentiel pour simplifier l'écriture de programmes concurrents. Les MT évitent aux programmeurs de se préoccuper de la concurrence, qui est gérée au niveau des algorithmes de MT elles-mêmes. La correction des algorithmes de MT est généralement prouvée manuellement. Le but de cette thèse est de donner des outils à la fois mathématiques et logiciels pour vérifier par *model checking* les algorithmes de MT supposant un modèle de mémoire faible.

Notre première contribution est de développer des outils mathématiques pour exprimer le comportement des algorithmes de MT et les propriétés de correction requises. Nous nous intéressons à la fois à des propriétés de sûreté, comme l'opacité, et de vivacité, comme la non-obstruction et l'abscence de *livelock*. De plus, nous construisons un outil logiciel pour exprimer les modèles mémoires matériels dits faibles. Nous développons un nouveau langage de haut niveau, RML, pour exprimer les algorithmes concurrents avec un niveau matériel d'atomicité des instructions, et dont la sémantique est paramétrée par de nombreux modèles mémoires faibles. Nous exprimons les algorithmes MT comme TL2, DSTM et McRT SRT dans notre nouveau langage, RML.

La vérification des algorithmes de MT est difficile à cause du nombre illimité de transactions concurrentes, de la durée et des délais non-bornés d'une transactions et de la taille non-bornée de la mémoire. La deuxième contribution de cette thèse est d'identifier les propriétés des algorithmes MT qui permettent de réduire le problème de vérification d'un modèle infini au problème d'inclusion de langage de deux machines d'états finies. Nous montrons que les algorithmes de MT usuels satisfont ces propriétés.

La troisième contribution de cette thèse est notre outil FOIL pour la vérification par *model checking* des algorithmes de MT. FOIL prend comme entrée la description RML d'un algorithme de MT et le description du modèle mémoire. FOIL utilise la sémantique opérationnelle de RML pour calculer le langage de l'algorithme de MT pour deux fils d'exécution et deux variables. FOIL contrôle ensuite si le langage de l'algorithme de MT est inclus ou non dans le langage de la spécification. FOIL détermine automatiquement les emplacements des barrières, qui, si elles sont insérées, assure la correction de l'algorithme de MT dans le modèle mémoire donné. Nous utilisons FOIL pour

vérifier les algorithmes DSTM, TL2 et McRT STM dans des modèles mémoires à cohérence séquentielle, ordre total des écritures, ordre partiel des écritures et ordre mémoire affaibli.
Mots clés : Concurrence, mémoire transactionnelle, sémantique, spécification, vérification de logiciels, model checking, modèles mémoires faibles.

Acknowledgments

Today, the achievement that I am most proud of is that I had Tom as the supervisor for my PhD. I was completely new to the field of verification and theoretical computer science when I started working with Tom. It was Tom's simple way of doing profound research and his devotion to his students that made me comfortable in my doctoral studies over time. Tom taught me the importance of being precise and formal in my ideas. Tom improved my writing skills beyond recognition. Moreover, getting to know Tom as the kind and wonderful person he is, was a great pleasure.

I have been fortunate to have Rachid as a co-author for my publications. He always encouraged me to put in the best effort and target the top conferences in the field. Rachid taught me how to present new ideas in research to others. I also thank the thesis reviewers, Hagit and Rajeev, for taking the time to read my thesis and being a part of the jury.

I take this opportunity to thank my family for the many things that I, in childhood, assumed came with a family; but after growing up, realized that I was fortunate to have those things. The love and affection of my mother made my childhood so memorable. She taught me how to make the best effort and expect the least returns. My father taught me many important things, of which I find two things hard not to mention. One is adjustment to any circumstances in life, and another is the value of time. These things made my life simpler and my work more efficient. I wonder if any father devotes so much time and energy to his children. I have also been lucky to have an elder brother and an elder sister. My brother, eight years elder to me, enjoyed teaching me engineering mechanics while I was in secondary school. Although I understood just a part of what he taught, I thank him for developing in me an ability to grasp new knowledge faster. I am also thankful to my brother's family (my sister-in-law, my niece, and my nephew) for their patience during the time I could not visit them for the last 2 years. My elder sister helped me with my school homework, and taught me new and interesting things throughout my childhood. It took me years to learn to live without her. I am grateful to her family (my brother-in-law, and my two nieces) for keeping me so close to their hearts even when I have been so distant. Without my wife, I would not have been able to finish my doctorate with this peace and balance of mind. She

always stood by me and adjusted herself with my deadlines and travel plans. I also express gratitude to my parents-in-law and my sister-in-law for their support, love, and affection.

I extend my gratitude to Dirk, Grégory, Maria, Nir and all other colleagues for keeping EPFL a place I loved to work at. I thank Dirk for boosting my confidence in my early days of working in the field of verification. I cannot thank Sylvie enough for all her administrative help, including the phone calls on my behalf when my modest French vocabulary was insufficient. I thank Fabien for the flawless and powerful IT infrastructure setup.

Last but not the least, I thank the Swiss National Science Foundation and EPFL for supporting my doctoral studies. I spent five great years at EPFL. The resources and the work environment provided at EPFL are certainly among the best in the world. I enjoyed living in the beautiful city of Lausanne. The serene views of Lac Léman from the shores in St. Sulpice are unforgettable. I felt proud to be a part of Switzerland during my stay. The Swiss impressed me with the importance they give to safety, quality, and preserving the environment. I loved the efficiency of the unmatched Swiss public transport, which I thoroughly used to visit the farthest and the prettiest corners of the country.

To
Amma, Appa, and Tina

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Over the last fifty years, the transistor density on a chip has doubled every two years. This phenomenon is widely known as the Moore's law [Moo65]. The exponential increase in transistor density has also resulted in an exponential increase in microprocessor clock rates, memory capacity, and similar capabilities of digital electronic devices. Moore's law is expected to continue to hold in the near future. However, higher clock rates have been accompanied by a much higher power consumption and heat dissipation, as these are cubic functions of the clock rate [FH05]. Around five years ago, the microprocessor clock rate hit the heat wall: the heat dissipation would damage the processor on increasing the clock rate further. Since then, the clock rates of a microprocessor have not scaled proportionately with the transistor density. This, in turn, led to the saturation of the performance of a uniprocessor. An alternative to increase the computation power proportionately with the transistor density is to build multiple processor cores in a single chip, where the cores run at lower clock rates. Lower clock rates result in lower power consumption and lower heat dissipation. Moreover, lower clock rates reduce the design complexity of a microprocessor.

However, to exponentially scale up the speed of a program using a multicore, it becomes crucial to exploit the computation power of all cores simultaneously. This requires the program to be *parallel*. A parallel program is one written to exploit the potential of a parallel computing resource like a multiprocessor or a cluster of processors. The research in parallel programs has been driven by high performance computing for the last few decades. However, the use of parallel programming has been limited to expert programmers. To achieve the full potential offered by multicore processors, we require suitable programming infrastructures and tools that make parallel programming accessible to mainstream programmers.

1.1 Parallel Programming

Parallel programming poses several challenges to the programmer. First of all, a programmer needs to identify computations that can be executed simultaneously. Secondly, a parallel program often requires concurrent access to underlying shared data. To prevent inconsistencies due to concurrent accesses, a parallel program requires precise synchronization. The performance of a parallel program depends on how the program handles concurrency. Concurrency makes parallel programs hard to write, and even harder to verify. Conventionally, concurrency relies on blocking synchronization primitives like locks and semaphores. These synchronization primitives often pose a trade-off between performance and programming effort. While coarse-grained locking is easy to reason about and prove correct, it does not often harness the computation power of a multicore efficiently. On the other hand, fine-grained locking yields an efficient program, but is a challenge to use correctly. Errors in using these synchronization techniques can result in unexpected data races, deadlocks, and livelocks. Moreover, concurrency leads to behaviors that are exponential in the number of threads, which makes errors hard to find or reproduce.

Other means of managing concurrency in parallel programs include non-blocking synchronization. Non-blocking algorithms use hardware provided atomic read-modify-write operations, and avoid the problems associated with locks or semaphores. However, non-blocking algorithms are extremely hard to design and to prove correct.

These problems with the current mechanisms to handle synchronization lead us to the question: can we provide a programming paradigm that ensures correctness of a parallel program, and at the same time, yields good performance. Transactional memory (TM) is one proposal that holds high potential to serve this requirement.

1.2 Transactional Memories

The idea of TM is inspired by the notion of database transactions. Transactional memory was first introduced by Herlihy and Moss [HM93] in multiprocessor design. Later Shavit and Touitou [ST95] introduced software transactional memory (STM), a software-based variant of the concept.

The basic notion of the execution of a TM is a transaction: a sequence of memory instructions that satisfies the following three properties: atomicity, isolation, and consistency. Atomicity ensures that either all or none of the effects of a transaction are visible. Isolation ensures that the intermediate state of a transaction is never visible to other transactions. Consistency ensures that the memory remains in a consistent state before a transaction starts, and after a transaction is over. TM are based on optimistic concurrency. That is, they allow speculative execution of transactions. If a transaction commits, all its effects are visible to other transactions. If a transaction aborts, none of the

effects are visible, and the transaction may be retried. An extensive overview of TM can be found in Larus et al. [LR07].

A decade after TM was invented, Harris and Fraser [HF03] proposed that the notion of a transaction can be used as a language level construct for synchronization in parallel programs. The runtime uses a TM implementation to execute a block of code, marked as atomic, as a transaction. This gave TM the potential to serve as an important paradigm to handle concurrency in parallel programs. Examples of programs with atomic blocks are shown in Figure 1.1. TM alleviates the difficulties of parallel programming. Although a programmer still needs to identify the parallelizable components in a program, a TM guarantees that these components are executed in parallel correctly, and with good performance.

1.3 TM Implementations

The success of transactional memories as a programming paradigm is evident from the plethora of TM implementations available today. TM have been implemented in hardware, software, and as a hybrid of hardware and software.

Hardware TM

TM was initially proposed by Herlihy and Moss [HM93] as a hardware mechanism. Most hardware TM (HTM) [MBM+06, HWC+04] rely on a modification to the cache coherence protocol to achieve atomicity of transactions. Proposals to exploit HTM [RG02] to implement conventional lock based critical sections also exist. HTM offer good performance, but suffer from the following drawbacks. Firstly, HTM require architectural changes to hardware and require the TM algorithm to be embedded in hardware. This makes HTM an expensive and inflexible option. Secondly, most HTM only support bounded transactions that can fit the on-chip resources. Although proposals [AAK+05, RHL05] exist in the literature to avoid this limitation, the design complexity of these proposals is significant.

Software TM

The limitations of HTM make STM a compelling choice to implement TM. Although the first STM was proposed by Shavit et al. [ST95] in 1995, STM gained popularity in 2003, when Herlihy et al. [HLMS03] proposed DSTM, and Harris et al. [HF03] proposed to use TM to provide transactions as a language level construct for synchronization in parallel programs. Since then, many STM [HLMS03, DSS06, SATH+06, FH07, MM07, RFF06] have been proposed. Software provides the flexibility to design and test new sophisticated TM algorithms. So, unlike HTM, STM widely differ in their protocols, and make an interesting verification problem. However, STM do not perform as well as HTM due to runtime overhead and bookkeeping in software.

Hybrid TM

Research has also focused on hybrid approaches [DFL+06, KCH+06], where transactions are first tried in the hardware component. If the transaction exhausts the resources available, it is retried in software. Thus, hybrid TM aims to provide the benefits of both HTM and STM. Sun's Rock processor [DLMN09], the first commercial processor with hardware support for transactions, is designed to be a hybrid TM.

Apart from the infrastructure of implementation, TM can also be classified on the basis of conflict detection and version management in their algorithms. Two transactions are in conflict if they access the same data, and at least one of the transactions writes. As TM rely on speculative execution, transactions are prone to conflict with each other. The conflict detection protocol varies in different TM algorithms. While some TM algorithms detect conflicts as soon as they happen, other TM algorithms detect conflicts when transactions commit.

As a running transaction has a possibility to abort, different versions of the data have to be managed at the same time: one version for recording the changes made to the data by the transaction, another to hold the original value of the data. Different TM algorithms differ in the way they manage the versions. A *deferred update* TM algorithm creates a local copy of all data written in a transaction, and uses this local copy to update the global data at the time of commit. On the other hand, a *direct update* TM algorithm updates the global data during the execution of a transaction and maintains a local copy of the original data to restore the global data upon an abort. A deferred update TM algorithm is also known as lazy update or redo log TM algorithm. A direct update TM algorithm is also known as eager update or undo log TM algorithm.

1.4 Correctness in TM

As a TM algorithm promises proper synchronization in a parallel program to the programmer, correctness is an important issue in TM algorithms. A TM algorithm is correct if it ensures correctness for all programs. Precisely because a TM algorithm encapsulates the difficulty of handling concurrency, the potential of subtle errors is enormous. This makes verification of TM algorithms an important candidate for formal methods. We now look at the safety and liveness properties expected in TM algorithms.

1.4.1 Safety

The correctness properties in TM build upon the correctness notions of database transactions. The most common correctness requirement in databases is serializability [Pap79] which requires that committed transactions appear to

initially $x = y = z = 0$ in every case

Thread 1	Thread 2
atomic {	
$x := 1$	
$x := 2$	*atomic* {
}	$r := x$
	}

(a) Can $r = 1$?

Thread 1	Thread 2
atomic {	
$x := 1$	
$x := 2$	*atomic* {
}	$z := x - y$
atomic {	}
$y := 2$	
}	

(b) Can $z < 0$?

Thread 1	Thread 2
atomic {	
$x := 1$	*atomic* {
$y := 1$	$r_1 := x$
}	$r_2 := y$
	$r_3 := 1/(r_1 - r_2 - 1)$
	}

(c) Can thread 2 perform a division by 0?

Figure 1.1: Examples of programs with transactions

be sequential. Strict serializability further requires that the order of non-overlapping transactions is preserved. However, it was observed [DSS06, HLMS03] that strict serializability is not sufficient for memory transactions. It is important that even aborted transactions do not observe an inconsistent state of the memory, as this could lead to unexpected side effects like infinite loops or array bound violations. This has led a strong notion of correctness in TM, referred to as *opacity* [GK08]. Opacity requires that all transactions appear to execute sequentially. This prevents aborting transactions from reading inconsistent values. Most of the TM implementations [HLMS03, DSS06, SATH$^+$06, FH07, MM07] indeed claim to satisfy opacity.

Figure 1.1 illustrates the three safety properties: serializability, strict serializability, and opacity. In Figure 1.1(a), thread 2 cannot observe the value of x as 1 under the correctness properties of serializability, strict serializability, and opacity. This is because all three correctness properties require the basic principle of isolation and atomicity of transactions. Consider Figure 1.1(b). Strict serializability and opacity require that if thread 2 observes the value of y as 2, then thread 2 observes the value of x as 2. That is, strict serializability and opacity require that the effect of transactions is visible in the order the transactions appear. Consider Figure 1.1(c). As serializability and strict serializability do not pose any restrictions on aborted transactions, it is possible that an aborted transaction views an inconsistent state by reading x as 1 and y as 0. Note that this might result in a division by zero by an aborting transaction which might crash the whole program. Opacity requires that even aborting transactions view consistent state. Thus, a division by zero error is not possible if the TM satisfies opacity.

1.4.2 Liveness

Early TM algorithms provided nonblocking progress guarantees. Below, we discuss liveness properties that provide nonblocking progress guarantees:

- The weakest nonblocking progress guarantee is known as obstruction freedom. A TM implementation is *obstruction free* if every transaction is guaranteed to commit as long as no other transaction makes a step. Note that obstruction freedom does not guarantee whether any transaction will commit if more than one transaction take steps. In other words, a TM implementation may livelock even if it is obstruction free.

- A TM implementation is *livelock free* if some transaction is guaranteed to commit in a finite number of steps. Livelock freedom implies obstruction freedom. But, a TM implementation that is livelock free does not guarantee that every transaction will commit.

- A TM implementation is *wait free* if every transaction is guaranteed to commit in a finite number of steps. Wait freedom implies livelock freedom.

The most efficient TM implementations today rely on locks. Thus, they fail to guarantee even obstruction freedom, the weakest non-blocking progress guarantee. Weaker progress guarantees [GK09] have also been defined in the framework of TM implementations.

1.5 Relaxed Memory Models

The correctness of a program depends on the infrastructure on which it runs. The same holds for TM algorithms. To ensure correctness, it is important to understand the behavior of TM algorithms on commercial processors. Processors, for reasons of performance, do not guarantee that the instructions in a program are executed in program order. Rather, processors specify a memory model [AG96] that specifies the set of allowed behaviors of memory accesses.

For example, the memory model *sequential consistency* specifies that a multiprocessor executes the instructions of a thread in program order. On the other hand, the memory model *total store order* specifies that a multiprocessor may relax the order of a store followed by a load to a different address, by delaying stores using a store buffer. In principle, a memory model offers a tradeoff between transparency to the programmer and flexibility to the hardware to optimize performance. Sequential consistency is the most stringent memory model, and thus the most intuitive to the programmer. But, most of the available multiprocessors do not support sequential consistency for reasons of performance.

The problem of synchronization in parallel programs becomes harder under relaxed memory models: apart from the interleavings of different threads, one needs to consider the possible reorderings of instructions within each thread. To illustrate the point, we give some examples of memory models. We give a parallel program and describe the outcomes under different relaxed memory models.

Examples of memory models

Initially : $x_1 = y_1 = x_2 = y_2 = 0$	
Thread 1	Thread 2
$x_1 := 1$	$x_2 := 1$
$y_1 := 1$	$y_2 := 1$
$r_1 := y_2$	$r_2 := y_1$
$r_3 := x_2$	$r_4 := x_1$
$x_1 := 2$	$x_2 := 2$

$O_1 : r_1 = 1, r_2 = 1, r_3 = 1, r_4 = 1$
$O_2 : r_1 = 0, r_2 = 0, r_3 = 0, r_4 = 0$
$O_3 : r_1 = 1, r_2 = 1, r_3 = 0, r_4 = 0$
$O_4 : r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 2$

Figure 1.2: The outcomes of a parallel program under different memory models

- *Sequential consistency (SC)* does not allow any pair of instructions to be reordered.
- *Total store order (TSO)* relaxes the order of a store followed by a load to a different address. But, the order of stores cannot be changed. TSO allows a load that follows a store to the same address to be eliminated.
- *Partial store order (PSO)* is similar to TSO, but further relaxes the order of stores.
- *Relaxed memory order (RMO)* relaxes the order of instructions even more than PSO, by allowing to reorder all memory instructions to different addresses.

Figure 1.2 illustrates a parallel program with two threads that distinguishes between the different memory models in terms of the possible outcomes. Outcome O_1 is allowed by SC, while other outcomes are not. Outcomes O_1 and O_2 are allowed by TSO. Outcomes O_1, O_2, and O_3 are allowed by PSO. All outcomes O_1, O_2, O_3, and O_4 are allowed by RMO.

Although most commercial multiprocessors use a relaxed memory model, TM algorithms published in the literature assume sequentially consistent behavior. The relaxations introduce a scope of errors when TM algorithms are executed on relaxed memory models. Typically, TM designers use *fences* to ensure a strict ordering of memory operations. As fences hurt performance, TM designers want to use fences only when necessary for correctness.

To illustrate the point, we consider the code fragments of the commit and the read procedures of a typical timestamp-based TM like TL2 [DSS06] in Figure 1.3. Assume that at the start of a transaction, t_1 and t_2 are set to the global timestamp *ts*. The commit procedure updates the timestamp *ts* before it updates the variables to be written. The read procedure first reads the timestamp, followed by the read of the variable, followed by a second read of the timestamp. The read is successful only if the two timestamps are equal. A

```
txCommit : ...
           update global timestamp ts
           for each variable v in write set
               update value of v
           ...
```

```
txRead : ...
         t1 := ts
         if (t1 ≠ t2) then abort
         read value of v
         t2 := ts
         if (t1 ≠ t2) then abort
```

Figure 1.3: Sample code fragments of commit and read procedures of a TM

crucial question is, given the memory model, which fences are required to keep the STM correct. On a memory model like sequential consistency [Lam79] or total store order [WG94], the code fragment in Figure 1.3 is correct without fences. On the other hand, on memory models that relax store order, like partial store order [WG94], we need to add a *store fence* after the timestamp update in the commit procedure. For even more relaxed memory models that may swap independent loads, like relaxed memory order [WG94], as well as the Java memory model [MPA05], we need more fences, namely, *load fences* in the read procedure. But the question is how many? Do we need to ensure that the read of v is between the two reads of ts, and thus put two fences? The answer is no. To ensure correctness, we just need one fence and guarantee that the second read of ts comes after the read of v.

1.6 Problem Statement

This thesis develops a formalism and a verification tool to answer the question: does a TM algorithm ensure a given safety or liveness property on a given memory model? If not, can the TM algorithm use fences to ensure the property under the memory model?

1.6.1 Challenges

This problem translates into three main challenges as described below.

Formalization

Although TM have been built and implemented since almost two decades, there is little work in the direction of formalizing the properties of TM. The challenge lies in mapping a given transactional program, as written by a programmer, to the set of instruction sequences (called histories) as issued by a TM on the processors with a given relaxed memory model. This requires to take into account, the state of each transaction in the TM, the interaction between different transactions, and the behavior of the scheduler. Intuitively, a TM algorithm is correct if for all transactional programs, all histories produced by the TM are correct.

Correctness specification

We consider opacity as our safety correctness criterion. While opacity has been formally defined, verification requires to check whether a history satisfies opacity or not. Thus, we need to build a specification that captures the set of all histories that satisfy opacity. We find it non-trivial to build a specification that can be used to efficiently verify TM algorithms.

Verification

The formal framework and the correctness specification lay the groundwork for verification. However, there remain two challenges in verification. First of all, with an unbounded number of threads and variables, the state space of a TM algorithm is unbounded. So, we need to either reduce the problem to a finite number of threads and variables, or use infinite state verification. Secondly, even with a small number of threads and variables, there is a high level of nondeterminism due to multiple threads and the relaxed memory model. Building an automated tool to capture state spaces of such magnitude is challenging.

1.6.2 Our approach

We tackle the above challenges in two steps. In the first step, we ignore the challenges posed by hardware level atomicity and relaxed memory models. We assume atomicity of a coarse grained alphabet of instructions. This step aims to provide an intuition of the verification technique without delving into the complications introduced by real hardware. We develop a formalism for TM algorithms and correctness properties of TM. We obtain a transition system corresponding to a TM algorithm. The set of histories produced by the TM algorithm corresponds to the language of the transition system. At first, we restrict ourselves to checking the correctness of a TM algorithm with a finite number of threads and variables. To do this, we first describe the correctness property as a finite state specification, and then check that the language of the TM algorithm is included in the specification. We build nondeterministic and deterministic transition systems (specifications) corresponding to the safety property of opacity. It is easy to show that the nondeterministic specification is indeed the set of all correct histories. But, to check that the language of the TM algorithm is included in the nondeterministic specification, we would need to determinize the specification. The large size of the nondeterministic specification makes determinization infeasible. Instead, we manually create a deterministic specification for the correctness property and use automated tools to establish the equivalence of nondeterministic and deterministic specifications. The deterministic specification allows us to easily check the correctness of a TM algorithm. We reduce the problem of verification of TM algorithms with an unbounded number of threads and variables, to verifying TM algorithms with two threads and two variables. This reduction relies on structural proper-

ties of TM algorithms. We express two phase locking, TL2, and DSTM in our framework, and show that these TM algorithms indeed satisfy the structural properties. Then, we prove opacity of the TM algorithms by checking that the language of the TM algorithms is contained in that of the deterministic TM specification. We also verify the liveness properties of TM algorithms. The first step indeed solves the verification problem at an abstraction.

In the second step, we address the complications in verification introduced by modeling TM algorithms at the level of atomicity provided by hardware, under relaxed memory models. We build a formalism for relaxed memory models. We express memory models as a function of hardware memory instructions, that is, loads and stores to 32 bit words. We describe various relaxed memory models, such as total store order (TSO), partial store order (PSO), and relaxed memory order (RMO) in our formalism. The reason for choosing these memory models is to capture different levels of relaxations allowed by different multiprocessors. Then, we build a new language, Relaxed Memory Language (RML), with a hardware-level of atomicity, whose semantics is parametrized by a relaxed memory model. We describe different TM algorithms in RML. We develop a new tool FOIL[1] to verify the safety and liveness of three different TM algorithms under different memory models. While we choose opacity as the safety criterion, using FOIL we can also verify other safety properties such as strict serializability that can be specified in our formalism. FOIL proves the opacity of the considered TM algorithms under sequential consistency and TSO. As the original TM algorithms have no fences, FOIL generates counterexamples to opacity for the TM algorithms under further relaxed memory models (PSO and RMO), and automatically inserts fences within the RML description of the TM algorithms that are required (depending upon the memory model) to ensure opacity. We observe that FOIL inserts fences in a pragmatic manner, as all fences it inserts match those in the manually optimized official implementations of the considered TM algorithms. Our verification leads to an interesting observation that many TM algorithms are sensitive to the order of loads and stores, but neither to the order of a store followed by a load, nor to store buffering. In other words, all TM algorithms we consider are opaque under TSO without any fences.

1.7 Generalizing the Notion of Opacity

The verification technique built in this thesis checks correctness of pure transactional programs. This technique would suffice if a program could execute all operations on shared data within transactions, and apply non-transactional operations only to thread-local data. In practice, however, not all operations on shared data can be wrapped in transactions. For example, a programmer may wish to make shared data local to a thread, operate non-transactionally

[1]Foil is a fencing weapon. Our tool inserts fences in the TM algorithm to make the TM algorithm correct.

Thread 1	Thread 2
atomic {	
$x := 1$	
$y := 1$	$r_1 := x$
}	$r_2 := y$

Figure 1.4: Can $r_1 = 1$ and $r_2 = 0$? It depends on the memory model (initially $x = y = 0$).

upon it for a while, and make it shared again [MBS+08, SMDS07]. It is thus not surprising to see a large body of research dedicated to exploring the various models of interaction between transactions and non-transactional code [ABHI08, DS09, GMP06, SMDS07, MBL06, MG08], and building TM implementations based on those models [SMAT+07, MBS+08].

The interaction between transactions and non-transactional operations has been defined using a notion of *strong atomicity* [MBL06, LR07] in the literature. The intuition behind strong atomicity is that transactions execute atomically with respect to other transactions and non-transactional operations. Unfortunately, strong atomicity has not been formally defined. This has led to multiple interpretations [SDMS08]. Consider, for example, the execution depicted in Figure 1.4 (adapted from Grossman et al. [GMP06]). The transaction executed by thread 1 updates variables x and y. Thread 2 reads the variables x and y non-transactionally. Is it possible that thread 2 reads x as 0 and y as 1? According to the definition by Martin et al. [MBL06], strong atomicity allows this result. But, according to the definition of strong atomicity by Larus et al. [LR07], this result is not allowed. The ambiguity in this definition can be attributed to an implicit assumption on the interaction between non-transactional operations, which in turn, depends upon the underlying memory model [AG96]. While Martin et al. [MBL06] assume a relaxed memory model that allows to reorder independent reads (for example, RMO [WG94]), Larus et al. [LR07] assume a sequentially consistent memory model.

We claim that while a TM can be implemented in a way to ensure opacity for transactions, there is little one can do (on a given platform or run-time environment) to change the underlying memory model. Hence, it is desirable to define opacity *parametrized* by a memory model. This thesis provides a general formal framework for describing the interactions between transactions and non-transactional operations. We consider opacity as a correctness condition for transactions, and parametrize it by a memory model.

1.8 Related Work

This thesis builds upon and improves the existing formalisms for TM and relaxed memory models. Moreover, this thesis presents a new verification tool for TM algorithms. We describe some existing related work in these directions.

1.8.1 Formalisms for transactional memories

While a lot of work has been carried out in the direction of developing fast and efficient transactional memories, there is limited research in the direction of formalizing correctness properties of TM. Scott [Sco06] was the first to provide a formal semantics for STM. However, his correctness criterion requires the order of commits to be preserved. Most of the popular STM algorithms, for example TL2 [DSS06], do not preserve the order of conflicts. Guerraoui and Kapalka [GK08] define opacity to precisely capture the safety aspect of STM and highlighted the subtle differences with database transactions.

The interaction of transactions with non-transactional operations has been widely studied. The study was pioneered by Grossman et al. [GMP06], where the authors raised several issues that need to be tackled in order to create TM implementations that handle non-transactional operations properly. The authors express the correctness property using sample executions, and thus the property lacks a formal specification. Work by Scott et al. [SDMS08] focuses on providing a set of rules that should hold irrespective of the memory model. This work is restricted to memory models that do not allow out-of-thin-air values. Menon et al. [MBS+08] define correctness by mapping transactions to critical sections, thus providing an intuitive definition of single global lock atomicity. Type systems and operational semantics for transactional programs with non-transactional accesses have been proposed by Abadi et al. [ABHI08] and Grossman et al. [MG08]. We provide a formal specification of the correctness property in the presence of relaxed memory models.

1.8.2 Formalisms for relaxed memory models

Adve et al. [AG96] provide a detailed description of hardware relaxed memory models. Language level memory models have been developed for Java [MPA05] and C++ [BA08]. Various formalisms for memory models have been proposed in the literature [BP09, BMS08, GHS09, SJMvP07, SSN+09]. Most of these formalisms provide an axiomatic definition of memory models. Architectural manuals [WG94, Sit02] also describe memory models in an axiomatic style. Operational semantics of relaxed memory models were developed by Petri et al. [BP09]. Their formalism captures write-to-read/write reordering used in memory models like TSO and PSO. Moreover, the formalism allows thread creation. However, it cannot express read-to-read/write relaxations found in memory models like RMO.

Verification based on axiomatic memory model specifications relies on constraint solving (like SAT solvers) to validate execution traces. As our tool is based on explicit state model checking, we find it more intuitive to define an operational semantics of relaxed memory models. Our operational semantics handles write-to-read/write and read-to-read/write relaxations, but cannot handle thread creation.

For the definition of parametrized opacity, we use an axiomatic definition of memory models. Most of the existing formalisms for memory models are tailored to capture the intricacies of specific memory models. We build a general formalism, with the focus on classification of memory models on the basis of reordering of instructions they allow.

1.8.3 Verification tools

Many tools for verification have been built in the last several years. Model checkers like BLAST [BHJM07] and SLAM [BR02] verify properties of sequential programs. Model checkers like Zing [AQR+04], SPIN [Hol97], KISS [QW04], and CHESS [QR05, MQB+08] are developed for verification of concurrent programs. These tools are built to detect races in concurrent programs. However, TM algorithms, by design, often consist of benign races. Moreover, these tools assume a sequentially consistent memory model, and miss out a whole range of interleavings that arise due to the reorderings of the instructions of a thread allowed by a relaxed memory model. Verification of concurrent data types has been attempted with theorem proving [CGLM06, VHHS06]. These methods require a manual tedious proof construction, and assume sequential consistency.

Dynamic tools [FF09, EQT07, FFY08, FF04] for verifying atomicity and race freedom in concurrent programs have also been built. Manovit et al. [MHC+06] used testing to find errors in TM implementations. Elmas et al. [ETQ05] have built tools for runtime verification of concurrent data types. The motivation for dynamic tools is to check the correctness of a computation at runtime, and throw an exception in case of error. Dynamic tools can find errors in a TM algorithm only when the TM algorithm is used to execute a transactional program. Dynamic tools cannot be used to establish the correctness of a TM algorithm, that is, to check whether the TM algorithm is correct for all programs. However, as static analysis is tricky for real TM implementations, we believe that dynamic tools can be useful to check correctness properties of TM implementations at runtime. Instead of the data race specification, the specification of opacity constructed in this thesis could be used.

Burckhardt et al. [BAM06, BAM07] developed CheckFence, a static verification tool for concurrent C programs under relaxed memory models. The tool requires as input a bounded test program (a finite sequence of operations) for a concurrent data type and uses a SAT solver to check the consistency and report if any fences are required. However, CheckFence cannot automatically introduce fences. We use the structural properties of TM, which allow us to

consider a maximal program on two threads and two variables in order to generalize the result to all programs with any number of threads and variables. We model the correctness problem as a relation between transition systems. Moreover, our tool, FOIL, automatically inserts fences. Padua et al. [LP01, FLM03] developed mechanisms to ensure sequential consistency under relaxed memory models. However, conservatively putting fences into TM implementations to guarantee sequential consistency would badly hurt TM performance. TM programmers put fences only where necessary. Gopalakrishnan et al. [GYS04] developed a verification tool for checking memory orderings for small programs.

Recent work has addressed verification in the context of transactional memories. Tasiran [Tas08] verified the correctness of the Bartok STM. The author manually proves the correctness of the Bartok STM algorithm, and uses assertions in the Bartok STM implementation to ensure that the implementation refines the algorithm. This work is orthogonal to ours, as we focus on automated techniques to prove the correctness of TM algorithms. Cohen et al. [CPZ08] model checked STM applied to programs with a small number of threads and variables, against the strong correctness property of Scott [Sco06]. Further, they studied safety properties in situations where transactional code has to interact with non-transactional accesses.

1.9 Organization of the Thesis

Chapter 2 presents a preliminary formalism of TM algorithms, and specifications for opacity. The chapter presents a language Simple, followed by examples of TM algorithms in the language. Chapter 3 presents results of model checking safety and liveness of TM algorithms for a finite number of threads and variables. Then, the chapter presents structural properties of TM algorithms that extend the verification results to an arbitrary number of threads and variables. These two chapters simplify and extend the formalism and the verification technique we presented at PLDI 2008 [GHJS08] and CONCUR 2008 [GHS08].

Chapter 4 builds upon the preliminary formalism. It presents a formalism and examples of memory models. The chapter presents the syntax and operational semantics of our language RML, followed by examples of different TM algorithms expressed in RML. Chapter 5 presents our tool FOIL to verify opacity of different TM algorithms under different relaxed memory models. Then, we show that the structural properties of TM algorithms hold at this low level of atomicity. These two chapters are based on our work presented at CAV 2009 [GHS09]. Chapter 6 extends the notion of opacity to mixed transactional code by introducing parametrized opacity. Chapter 7 concludes the thesis and discusses directions for future work.

2 A Preliminary Formalism

The motivation of the preliminary formalism is to explain our verification technique for TM algorithms without the complications introduced by hardware level atomicity and relaxed memory models. Our preliminary approach tackles the problem of verifying TM algorithms at a level of abstraction by making certain assumptions. We assume a level of atomicity higher than that provided by the hardware. We assume that the instructions of a thread execute in program order. We restrict our verification to deferred update TM algorithms, that is, to TM algorithms which update global memory on a commit. Moreover, we assume that all writes are committed atomically.

2.1 Transactional Programs

Let $V = \{1, \ldots, k\}$ be a set of *transactional variables*. Let $T = \{1, \ldots, n\}$ be a set of *threads*. Let the set C of *transactional commands* be $(\{\mathit{txrd}, \mathit{txwr}\} \times V) \cup \{\mathit{txend}\}$. These commands correspond to a read or write of a transactional variable, and to the end of a transaction.

Depending upon the underlying TM, the execution of these commands may correspond to a sequence of instructions in our abstraction. For example, a read of a transactional variable may require to check the consistency of the variable by first reading a version number. Similarly, a transaction end may require to validate the variables read, followed by copying many variables from a thread-local buffer to global memory. Moreover, the semantics of the *txwr* and the *txend* commands depend on the underlying TM. For example, a (txwr, v) command does not alter the value of v in a deferred update TM, whereas it does in a direct update TM. On the other hand, an *txend* command may alter transactional variables in a deferred update TM, and not in a direct update TM. In this chapter, we assume that a (txwr, v) command does not alter

the value of v globally, which restricts the verification technique to deferred update TM algorithms.

We consider transactional programs as our basic sequential unit of computation. We express transactional programs as infinite binary trees on transactional commands, which makes the representation independent of specific control flow statements, such as exceptions for handling aborts of transactions. Our definition restricts us to purely transactional code, as every command is part of some transaction. For every command of a thread, we define two successor commands, one if the command is successfully executed, and another if the command fails due to an abort of the transaction. Note that this definition allows us to capture different retry mechanisms of TM, e.g., retry the same transaction until it succeeds, or try another transaction after an abort. We use a set of transactional programs to define a multithreaded transactional program. A *transactional program* θ is an infinite binary tree $\theta : \mathbb{B}^* \to C$. Let Θ be the set of all transactional programs. A *(multithreaded) transactional program* $prog : T \to \Theta$ is a function from the set of threads to the set of transactional programs. Let *Progs* be the set of all multithreaded transactional programs.

2.2 Transactional Memories

We characterize a TM by the sequences of memory instructions the TM can produce. We call a sequence of instructions a *history*.

2.2.1 Histories

In order to reason about the correctness of TM, the history must contain, apart from the sequence of memory instructions that capture the reads and writes of transactional variables in the program, the information of when transactions finish. This is because the correctness of a TM depends on the sequence of reads and writes issued to transactional variables and the boundaries where the transactions finish. Let the set *In* of *instructions* be $(\{\mathsf{read}, \mathsf{write}\} \times V)$. Let $\hat{In}$ be $In \cup \{\mathsf{commit}, \mathsf{abort}\}$.

Let $\hat{Op} = \hat{In} \times T$ be the set of *operations*. A *history* $h \in \hat{Op}^*$ is a finite sequence of operations. We characterize a TM by the set of finite and infinite histories it produces. Formally, a *transactional memory (TM)* Γ is a prefix-closed subset of $\hat{Op}^* \cup \hat{Op}^\omega$.

2.2.2 Transactions

Given a history $h \in \hat{Op}^*$, we define the *thread projection* $h|_t$ of h on thread $t \in T$ as the subsequence of h consisting of all operations op in h such that $op \in \hat{In} \times \{t\}$. Given a thread projection $h|_t = op_0 \dots op_m$ of a history h on thread t, an operation op_i is *finishing in* $h|_t$ if op_i is a commit or an abort.

An operation op_i is *initiating in* $h|_t$ if op_i is the first operation in $h|_t$, or the previous operation op_{i-1} is a finishing operation.

Given a thread projection $h|_t$ of a history h on thread t, a consecutive subsequence $x = op_0 \dots op_m$ of $h|_t$ is a *transaction* of thread t in h if (i) op_0 is initiating in $h|_t$, and (ii) op_m is either finishing in $h|_t$, or op_m is the last operation in $h|_t$, and (iii) no other operation in x is finishing in $h|_t$. The transaction x is *committing* in h if op_m is a commit. The transaction x is *aborting* in h if op_m is an abort. Otherwise, the transaction x is *unfinished* in h. We say that a transaction x is *finished* in h if x is committing or aborting in h. Given a history h and two transactions x and y in h (possibly of different threads), we say that x *precedes* y in h, written as $x <_h y$, if the last operation of x occurs before the first operation of y in h. A history h is *sequential* if for every pair x, y of transactions in h, either $x <_h y$ or $y <_h x$.

2.2.3 Safety in TM

We consider the safety property, opacity, for transactional memories. Opacity builds upon the safety property of strict serializability. Strict serializability [Pap79] requires that the order of conflicting statements from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity [GK08], in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in TM, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations. Although we restrict our attention to read and write operations, the notion of opacity has been extended to arbitrary operations.

An operation op_1 of transaction x and an operation op_2 of transaction y (where x is different from y) *conflict* in a history h if (i) op_1 is a read of some variable v, and op_2 is a commit of a transaction that writes to v, or (ii) op_1 and op_2 are both commits of transactions that write to some variable v. Note that this notion of conflict corresponds to the deferred update semantics [LR07] in transactional memories, where the writes of a transaction are made global upon the commit. A history $h = op_0 \dots op_m$ is *strictly equivalent* to a history h' if (i) for every thread $t \in T$, we have $h|_t = h'|_t$, and (ii) for every pair op_i, op_j of operations in h, if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h', and (iii) for every pair x, y of transactions in h, where x is a committing or an aborting transaction, if $x <_h y$, then it is not the case that $y <_{h'} x$.

We define the safety property *opacity* $\pi \subseteq \hat{Op}^*$ as the set of histories h such that there exists a sequential history h', where h' is strictly equivalent to h. Although we do not use the property of strict serializability, we define it to illustrate the formalism. We define a function $com : \hat{Op}^* \to \hat{Op}^*$ such that for all histories $h \in \hat{Op}^*$, the history $com(h)$ is the subsequence of h which consists of every operation in h that is a part of a committing transaction. We define the safety property *strict serializability* $\pi_{ss} \subseteq \hat{Op}^*$ as the set of histories

h such that there exists a sequential history h', where h' is strictly equivalent to $com(h)$. We note that $\pi \subseteq \pi_{ss}$, that is, if a history is opaque, then it is strictly serializable.

2.2.4 Liveness in TM

We formalize two different notions of liveness, obstruction freedom and livelock freedom, as discussed in the TM literature. A third notion, wait freedom [Her91], implies livelock freedom. Since we will show that none of our TM examples satisfy livelock freedom, they do not satisfy wait freedom either.

We consider infinite histories in $\hat{Op}^{\omega}$. An infinite history $\bar{h} \in \hat{Op}^{\omega}$ is *obstruction free* [HLM03] if for all threads t, if the history $\bar{h}$ has an infinite number of aborts of t, then either $\bar{h}$ has an infinite number of commits of t, or there are infinitely many operations of some thread $u \neq t$. Formally, $\hat{h}$ is *obstruction free* if $\bigwedge_{t \in T}(\Box\Diamond(\mathsf{abort}, t) \rightarrow \Box\Diamond((\mathsf{commit}, t) \vee \bigvee_{in \in \hat{In}, u \in T \setminus \{t\}}(in, u)))$, where the temporal operation $\Box$ denotes 'always' and the temporal operation $\Diamond$ denotes 'eventually'. Obstruction freedom is a Streett condition [Str82].

An infinite history $\bar{h} \in \hat{Op}^{\omega}$ is *livelock free* (often referred to as lock-free [HLM03, FH07]) if the history has an infinite number of commits, or there is a thread t such that t has infinitely many operations and finitely many aborts in $\hat{h}$. Formally, $\hat{h}$ is *livelock free* if $\Box\Diamond(\bigvee_{t \in T}(\mathsf{commit}, t)) \vee \bigvee_{t \in T}(\Box\Diamond(\bigvee_{in \in In}(c, t)) \wedge \Diamond\Box\neg(\mathsf{abort}, t))$. Note that livelock freedom implies obstruction freedom.

We say that Γ *ensures* (n, k) *opacity* if for every finite history h in Γ such that h has at most n threads and at most k variables, we have $h \in \pi$. Moreover, Γ *ensures opacity* if Γ ensures (n, k) opacity for all n and k. A TM Γ *ensures* (n, k) *obstruction freedom (resp.* (n, k) *livelock freedom)* if every infinite history $\bar{h} \in \Gamma$ such that $\bar{h}$ has at most n threads and at most k variables is obstruction free (resp. livelock free). Moreover, Γ *ensures obstruction freedom (resp. livelock freedom)* if Γ ensures (n, k) obstruction freedom (resp. (n, k) livelock freedom) for all n and k.

2.3 TM Specifications

We capture safety properties of TM using TM specifications. We use the set of instructions as the alphabet of the TM specification. We also allow a special ε instruction in the alphabet, which we use in nondeterministic TM specifications. As we see later, the ε instruction marks the serialization point of a transaction in a nondeterministic specification.

A *TM specification Spec* on an alphabet $\hat{In}$ is a 3-tuple $\langle Q, q_{init}, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta \subseteq Q \times ((\hat{In} \cup \{\varepsilon\}) \times T) \times Q$ is a transition relation. A sequence $r_0 \dots r_m$ in $((\hat{In} \cup \{\varepsilon\}) \times T)^*$ is a *run* of the TM specification if there exist states $q_0 \dots q_{m+1}$ in Q such that $q_0 = q_{init}$, and for all i such that $0 \leq i \leq m$, we have $(q_i, r_i, q_{i+1}) \in \delta$. A history $h = op_0 \dots op_k$

corresponds to a run $r_0 \dots r_m$ if h is the longest subsequence of operations in $\hat{Op}$ in $r_0 \dots r_m$.

The *language* L of a TM specification is the set of all histories h such that h corresponds to a run of the TM specification. A *TM specification Spec* defines opacity π if $L(Spec) = \pi$. A TM specification is *deterministic* if for every state $q \in Q$, the following hold: (i) for every operation $op \in \hat{Op}$, there is at most one state $q' \in Q$ such that $(q, op, q') \in \delta$, and (ii) for every thread $t \in T$, there is no state $q' \in Q$ such that $(q, (\varepsilon, t), q') \in \delta$. Note that for a deterministic TM specification, the history corresponding to a run is the run itself.

2.3.1 Construction of TM specifications

We now provide both nondeterministic and deterministic TM specifications for opacity for the alphabet $\hat{In}$. First, we give a nondeterministic TM specification, and manually prove its correctness. Later, we give a deterministic TM specification, which we shall use in our verification tool. As our verification technique requires the specification of opacity for two threads and two variables, it is sufficient to build a deterministic TM specification for two threads and two variables. We use an antichain-based tool [WDHR06] to prove that the language of the deterministic TM specification for two threads and two variables is indeed equivalent to that of the nondeterministic counterpart.

Our construction of the TM specification for opacity for a finite number of threads and variables uses a finite number of states. The construction is non-trivial, as threads may be delayed arbitrarily, and execute an arbitrary number of transactions, where each transaction may contain arbitrarily many operations and may be aborted arbitrarily often. The classical approach to checking whether a history is opaque is to construct a directed graph $G = (V, E)$, called the conflict graph [Pap79], of the transactions in the history. The conflict graph captures the precedence of the transactions based on the conflicts. Given a history $h = op_0 \dots op_n$, the transactions in h form the set V of vertices in the conflict graph. There exists an edge from a vertex v_1 to a vertex v_2 if v_2 commits or aborts before v_1 starts, or an operation op_i of v_1 conflicts with an operation op_j of v_2 and $i > j$. The conflict graph G is acyclic if and only if the history h is opaque. We note that the size of this construction is unbounded. The following parametrized history illustrates the point: $h_m = ((\mathsf{read}, v_1), t_1), (((\mathsf{write}, v_1), t_2), (\mathsf{commit}, t_2))^m, (\mathsf{commit}, t_1)$. The number of vertices in the conflict graph of h_m is $m + 1$. Thus, we cannot aim to create a finite-state TM specification for opacity using conflict graphs. Note that it can be similarly shown that conflict graphs for serializability and strict serializability are unbounded.

The key idea to get around the problem of infinite states is to maintain sets called *prohibited read and write sets* for every unfinished transaction. These sets allow to handle unbounded delay between transactions, as finished transactions store the required information in the state of unfinished transactions. Once a transaction commits or aborts, we remove the transaction from our

conflict graph (unlike classical conflict graphs). Thus, we need to store information of at most one transaction per thread.

2.3.2 A nondeterministic TM specification for opacity

Nondeterminism allows a natural construction of the TM specification, where a transaction nondeterministically guesses a serialization point during its lifetime. A branch of the nondeterministic TM specification corresponds to a specific serialization choice of the transactions, which makes the construction simple and intuitive, though redundant.

The nondeterministic TM specification *Spec* for opacity is based on the observation that *every finished transaction should serialize at some point during its execution.* The TM specification *Spec* makes a nondeterministic guess of when a transaction serializes. Upon every read and every commit of a transaction, *Spec* checks whether the command can be executed or the transaction needs to be aborted.

Formally, we define the *nondeterministic TM specification for opacity* as the tuple $Spec = \langle Q, q_{init}, \delta \rangle$. A state $q \in Q$ is a 7-tuple $\langle Status, SerStatus, rs, ws, prs, pws, serp \rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{invalid}\}$ is the status, $SerStatus : T \to \{true, false\}$ is the serialization status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, and $serp : T \to 2^T$ is the serialization predecessor set for the threads. Note that although the state refers to a thread, we often say "state of a transaction x" for brevity. We mean the state of the thread t which executes the transaction x. In other words, the state can be attributed to the unfinished transaction of the thread. Note that when a transaction of a thread finishes, we say that the next transaction of the thread is unfinished.

The initial state q_{init} is $\langle Status_0, SerStatus_0, rs_0, ws_0, prs_0, pws_0, serp_0 \rangle$, where $Status_0(t) = \mathsf{finished}$, and $SerStatus_0(t) = false$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = serp_0(t) = \emptyset$ for all threads $t \in T$. We express the transition function δ using the procedure *nondetSpec* shown in Algorithm 2.1. For all states $q \in Q$ and all operations $op \in \hat{Op}$, the following hold: (i) if $nondetSpec(q, op) = \bot$, then there is no state $q' \in Q$ such that $(q, op, q') \in \delta$, and (ii) if $nondetSpec(q, op) = q'$ for some state $q' \in Q$, then $(q, op, q') \in \delta$. We use this notation of the transition relation of a TM specification in the form of an algorithm multiple times in this thesis.

Given a state q and a thread $t \in T$, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $SerStatus(t)$ to $false$, (iii) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, and (iv) for all threads $u \neq t$, removes t from $serp(u)$.

$nondetSpec(\langle Status, SerStatus, rs, ws, prs, pws, serp\rangle, op)$

if $op = ((\mathsf{read}, v), t)$ **then**
  $rs(t) := rs(t) \cup \{v\}$
  if $v \in prs(t)$ **then** return $\bot$
  for all threads $u \neq t$ **do**
    if $SerStatus(u) = true$ and $t \notin serp(u)$ **then**
      if $v \in ws(u)$ **then**
        $Status(u) := \mathsf{invalid}$
      else
        $pws(u) := pws(u) \cup \{v\}$

if $op = ((\mathsf{write}, v), t)$ **then**
  if $v \in pws(t)$ **then**
    $Status(t) := \mathsf{invalid}$
  $ws(t) := ws(t) \cup \{v\}$

Algorithm 2.1: The nondeterministic TM specification for opacity

Construction

We describe the set of histories which are produced by the TM specification. Let r be a run of the TM specification. Let x be an unfinished transaction in r. The construction of the nondeterministic TM specification ensures the following:

Rule 1. The serialization status of x is *true* in a run $r' = r \cdot op$ if (i) the serialization status of x in r is *true*, and op is neither a commit nor an abort instruction of x, or (ii) op is a serialize of transaction x

Rule 2. A variable v is in the prohibited write set of x if

- there is a committed transaction y in r such that y serializes after x and y writes or reads v
- there is a transaction y such that x is serialized and y does not serialize before x and y reads v

Rule 3. A variable v is in the prohibited read set of x if there is a committed transaction y in r such that y serializes after x and y writes v

Rule 4. The status of a transaction x is invalid in a run $r' = r \cdot op$ if one of the following holds:

a. the status of x is invalid in r, and op is not an abort of x, or

b. op is a read of some variable v by another transaction y and x writes to v and x is serialized and y is not serialized before x, or

nondetSpec($\langle Status, SerStatus, rs, ws, prs, pws, serp\rangle, op$)

if $op = (\mathsf{commit}, t)$ **then**
  if $Status(t) = \mathsf{invalid}$ **then** return $\bot$
  if $SerStatus(t) = false$ **then** return $\bot$
  for all threads $u \neq t$ **do**
    if $u \in serp(t)$ **then**
      $prs(u) := prs(u) \cup ws(t)$
      $pws(u) := pws(u) \cup rs(t) \cup ws(t)$
      if $(ws(u) \cap (ws(t) \cup rs(t)) \neq \emptyset$ **then**
        $Status(u) := \mathsf{invalid}$
  ResetState(q, t)

if $op = (\varepsilon, t)$ **then**
  if $SerStatus(t) = true$ **then** return $\bot$
  else $SerStatus(t) := true$
  $serp(t) := \{u \in T \mid SerStatus(u) = true\}$
  for all threads $u \neq t$ **do**
    if $SerStatus(u) = false$ **then**
      if $rs(u) \cap ws(t) \neq \emptyset$ **then** $Status(t) := \mathsf{invalid}$
      $pws(t) := pws(t) \cup rs(u)$

if $op = (\mathsf{abort}, t)$ **then**
  if $SerStatus(t) = false$ **then** return $\bot$
  ResetState(q, t)
return $\langle Status, SerStatus, rs, ws, prs, pws, serp\rangle$

continued **Algorithm 2.1**

c. *op* is a write of some variable v by x and v belongs to the prohibited write set of x, or

d. *op* is the commit of a transaction y and x serializes before y and y reads or writes v and commits and x writes v, or

e. *op* is the serialize of x and there is a transaction y which is not serialized and there exists a variable v such that x writes to v and y reads v

Rule 5. Given an operation *op* of x, the run $r \cdot op$ is a run of the TM specification *Spec* if

a. *op* is a write, or

b. *op* is a commit and the serialization status of x is *true* and x is in status finished, or

c. *op* is an abort and the serialization status of x is *true*, or

d. *op* is a serialize and the serialization status of x is *false*, or

e. *op* is a read of variable v and v is not in the prohibited read set of x.

Note that a variable is added to the prohibited read or write set of a transaction x only due to transactions that serialize after x. Similarly, the status of a transaction x is set to invalid only due to transactions that serialize after x.

Correctness

We now prove that Algorithm 2.1 indeed provides the language of opacity for n threads and k variables.

Theorem 2.1. *Given a history h on n threads and k variables, h is opaque if and only if $h \in L(Spec)$.*

Proof. First, we note that the TM specification *Spec* for opacity gives the largest set R of runs such that for every run $r \in R$, for every transaction x in r, the following conditions are satisfied (C1-C3 are graphically shown in Figure 2.1):

C1. x does not commit if there exists a transaction y such that x serializes before y and x writes to v and y reads v

C2. x does not commit if there exists a transaction y such that x serializes before y and both x and y write to a variable v, and y commits before x does

C3. x does not read a variable v if there exists a transaction y such that x serializes before y and y writes to v and commits

C4. x serializes at most once

C5. if x is a finished transaction, then x serializes exactly once

The condition C1 follows from the rules 4.b, 4.d, 4.e, and 5.b. The condition C2 follows from the rules 2, 4.c, 4.d, and 5.b. The condition C3 follows from the rules 3 and 5.d. The condition C4 follows from the rules 1 and 5.c.

Let h be opaque. As h is opaque, there is a sequential history h_s such that h_s is strictly equivalent to h. Let the transactions in the sequential history h_s be given by the sequence $x_1 \dots x_n$ of transactions. We claim that there exists a run r of the TM specification *Spec* such that h is the corresponding history of the run r. Consider a run r such that the transactions serialize in the order $x_1 \dots x_n$, and h is the corresponding history of r. As h_s is strictly equivalent to h, we know that for every pair x_i, x_j of transactions, such that $i < j$, the following are not true:

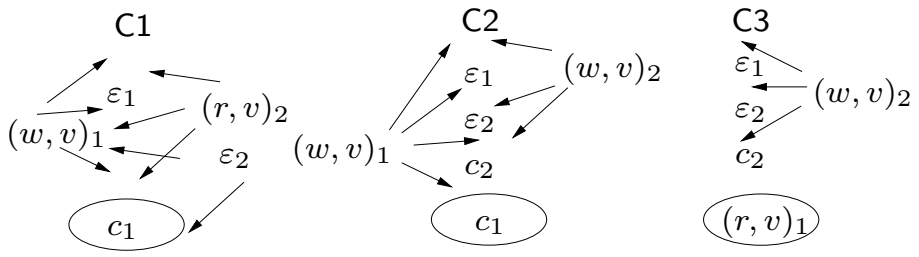

Figure 2.1: The statements inside ovals are disallowed by the TM specification for opacity. Each condition shows various cases. The arrows represent different possible positions for a command to occur in a given condition. We write w for write, r for read, and c for commit. We write the statement $((w,v),t_k)$ as $(w,v)_k$. Thread t_1 executes transaction x and thread t_2 executes transaction y.

- x_i reads a variable after x_j writes and after x_j commits, and
- x_i is a committing transaction, and x_i writes a variable and x_j reads it before x_i commits, and
- x_i is a committing transaction, and there exists a variable v such that x_i and x_j both write to v, and x_j commits before x_i commits.

This implies that for all transactions x in the run r, the conditions C1 - C3 hold. Thus, r is a run of the TM specification *Spec*.

Conversely, let r be a run produced by the nondeterministic TM specification *Spec*. Let h be the corresponding history to the run r. We know from conditions C4 and C5 that every transaction serializes at most once in the run, and every finished transaction serializes exactly once in the run. Let h_s be a sequential history such that (i) for all serialized transactions in r (which includes all finished transactions), the order of transactions in h_s is same as the order of serialization in r, and (ii) all unfinished transactions which have not serialized in r occur in an arbitrary order after all serialized transactions in h_s, and (iii) for all threads, the thread projection of h_s is equivalent to the thread projection of h. The conditions C1 - C3 guarantee that for every pair op_i, op_j of operations in h, if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h_s. Note that if a transaction x finishes before a transaction y starts in h, then x serializes before y in r. Thus, for every pair x, y of transactions in h, if x is a finished transaction and $x <_h y$, then we have $x <_{h_s} y$. Thus, h_s is strictly equivalent to h. Hence, h is opaque.

□

2.3.3 A deterministic TM specification for opacity

In the nondeterministic TM specification, we consider a particular order of the serialization of the transactions in a given branch. This allows us to argue individually for different serialization orders, which in turn, allows us to locally

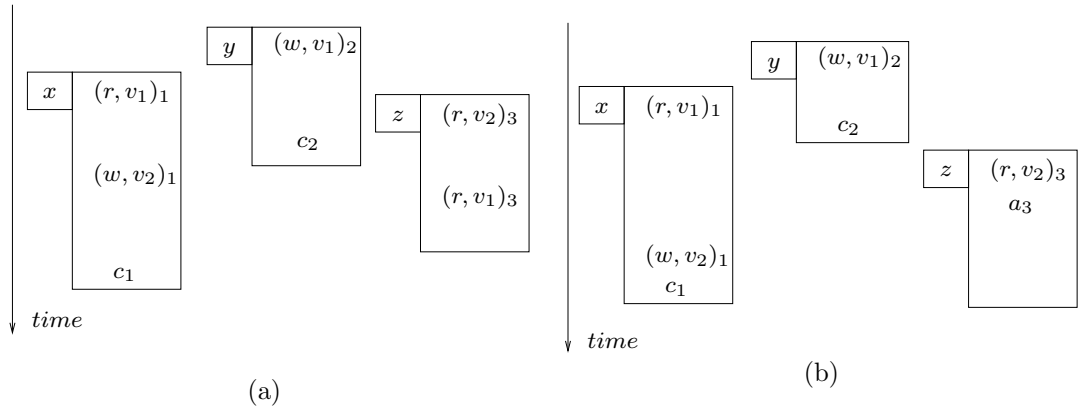

Figure 2.2: Analysis for creating a deterministic TM specification for opacity. The histories are fragmented into transactions of different threads.

reason for every pair of transactions. On the other hand, in a deterministic TM specification, we have to consider all possible serialization orders at the same time, which complicates the reasoning and the specification. We first look at the issues we face in creating a deterministic TM specification for opacity.

Analysis of opacity

Consider the history $h = ((\mathsf{write},\ v_1),\ t_2),\ ((\mathsf{read},\ v_1),\ t_1),\ ((\mathsf{read},\ v_2),\ t_3),\ (\mathsf{commit},\ t_2),\ ((\mathsf{write},\ v_2),\ t_1),\ ((\mathsf{read},\ v_1),\ t_3),\ (\mathsf{commit},\ t_1)$. The history is illustrated in Figure 2.2(a). Transaction x has to serialize before y due to a conflict on v_1. Also, z has to serialize after y due to a conflict on v_1, and before x due to a conflict on v_2. Note that although z does not commit, opacity requires that transaction x does not commit. So, h is not opaque.

Consider the history $h = ((\mathsf{write},\ v_1),\ t_2),\ ((\mathsf{read},\ v_1),\ t_1),\ (\mathsf{commit},\ t_2),\ ((\mathsf{read},\ v_2),\ t_3),\ (\mathsf{abort},\ t_3),\ ((\mathsf{write},\ v_2),\ t_1),\ (\mathsf{commit},\ t_1)$. The history is illustrated in Figure 2.2(b). Transaction x has to serialize before y due to a conflict on v_1. Transaction z has to serialize after y as they do not overlap in w. Also, z has to serialize before x due to the conflict on v_2. This makes h not opaque. This shows how a read of an aborting transaction may disallow a commit of another transaction, for the sake of opacity.

As our verification technique requires the TM specification for two threads and two variables, we build the deterministic TM specification for only two threads. This makes the construction simple to express and understand. The *deterministic TM specification for opacity* $Spec^d$ is given by the tuple $\langle Q, q_{init}, \delta^d \rangle$. A state $q \in Q$ is a 7-tuple $\langle Status,\ rs,\ ws,\ prs,\ pws,\ wp,\ sp \rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{invalid}, \mathsf{pending}\}$ is the status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $wp : T \to \{true, false\}$ tells whether the other thread is a weak predecessor, and $sp : T \to \{true, false\}$ tells whether the other thread is a strong predecessor. The initial state q_{init}

detSpec$(\langle Status, rs, ws, prs, pws, wp, sp\rangle, op)$

if $op = ((\mathsf{read}, v), t)$ **then**
  if $v \in prs(t)$ **then return** $\bot$
  if $v \in prs(u)$ **and** $sp(u)$ **then return** $\bot$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then**
      if $sp(u)$ **then return** $\bot$
      $wp(t) := true$; $sp(t) := true$
    $Status(t) := \mathsf{started}$
  $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then** $wp(u) := true$
  if $v \in prs(u)$ **then** $wp(t) := true$
  if $sp(t)$ **then**
    $pws(u) := pws(u) \cup \{v\}$
    if $v \in ws(u)$ **then** $Status(u) := \mathsf{invalid}$
  if $sp(u)$ **and** $sp(t)$ **then return** $\bot$

if $op = ((\mathsf{write}, v), t)$ **then**
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then**
      if $sp(u)$ **then return** $\bot$
      $wp(t) := true$; $sp(t) := true$
    $Status(t) := \mathsf{started}$
  $ws(t) := ws(t) \cup \{v\}$
  if $v \in pws(t)$ **then** $Status(t) := \mathsf{invalid}$
  if $v \in rs(u)$ **then**
    $wp(t) := true$
    if $sp(u)$ **then** $Status(t) := \mathsf{invalid}$
  if $v \in pws(u)$ **then** $wp(t) := true$
  if $sp(u)$ **and** $sp(t)$ **then return** $\bot$

Algorithm 2.2: The deterministic TM specification for opacity

is $\langle Status_0,\ rs_0,\ ws_0,\ prs_0,\ pws_0,\ wp_0,\ sp_0\rangle$, where $Status_0(t) = \mathsf{finished}$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = \emptyset$, and $wp_0(t) = sp_0(t) = false$ for both threads. The transition relation δ^d is obtained using the procedure *detSpec* shown in algorithm 2.2. The notation of *detSpec* is similar to that of the procedure *nondetSpec*. The thread t refers to the thread taking the step, and the thread u refers to the other thread. Given a state q, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, and (iii) sets $wp(t)$, $wp(u)$, and $sp(u)$ to false.

detSpec($\langle Status, rs, ws, prs, pws, wp, sp\rangle$, op)

if $op = (\mathsf{commit}, t)$ **then**
  if $Status(t) = \mathsf{invalid}$ **then** return $\bot$
  if $wp(t)$ **or** $sp(t)$ **then**
    if $wp(u)$ **then** $Status(u) := \mathsf{invalid}$
    $sp(t) := true$
    if $ws(t) \cap ws(u) \neq \emptyset$ **then** $Status(u) := \mathsf{invalid}$
    if $Status(u) \neq \mathsf{invalid}$ **then** $Status(u) = \mathsf{pending}$
    $prs(u) := prs(u) \cup prs(t) \cup ws(t)$
    $pws(u) := pws(u) \cup pws(t) \cup rs(t) \cup ws(t)$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$
  ResetState(q, t)

if $op = (\mathsf{abort}, t)$ **then** *ResetState*(q, t)
return $\langle Status, rs, ws, prs, pws, wp, sp\rangle$

continued **Algorithm 2.2**

Correctness

We implement the algorithms to build nondeterministic and deterministic TM specifications for two threads and two variables. We observe that the nondeterministic TM specification presented is too large to be automatically determinized. However, the deterministic TM specification we present turns out to be much smaller in size. Using an antichain-based tool [WDHR06], we establish that for two threads and two variables, the language of the deterministic TM specification for opacity is equivalent to the language of the nondeterministic TM specification for opacity.

The deterministic TM specification $Spec^d$ has 2272 states, while the nondeterministic TM specification *Spec* consists of 9202 states. The antichain-based tool can prove the equivalences within ten seconds. This leads us to the following theorem.

Theorem 2.2. $L(Spec) = L(Spec^d)$.

This theorem allows us to now use the deterministic TM specification for model checking TM algorithms.

2.4 TM Algorithms

We now present a formalism to express various TM using TM algorithms.

$$
\begin{array}{lcl}
l & \in & L \\
g & \in & G \\
in & \in & In_A \\
e & ::= & f(l, \ldots, l, g, \ldots, g) \\
s & ::= & g := e \;\mid\; l := e \;\mid\; \textbf{if } e \textbf{ then } s \textbf{ else } s \\
 & & \mid\; \textbf{for all } e \; in \; E \textbf{ do } s \;\mid\; s \,;s \;\mid\; \mathsf{skip} \\
p & ::= & (in : s) \;\mid\; p \,;p \;\mid\; \textbf{for all } e \; in \; E \textbf{ do } p \;\mid\; \textbf{if } e \textbf{ then } p \textbf{ else } p
\end{array}
$$

Figure 2.3: The syntax of the language Simple

2.4.1 The Simple language

As this chapter assumes an abstraction that provides a high level of atomicity, we model TM algorithms at that level in our language Simple. We describe a TM algorithm as sets of global and local variables, an initial valuation, and programs in our language Simple corresponding to a transactional read or write of a variable, and the transactional end.

Syntax

A statement in the language Simple can have one or multiple assignments to global or local variables. These assignments could also depend on the global and local variables. Every statement in the language Simple executes atomically. A statement has an associated instruction, which represents the execution of the statement.

Let G be the set of global variables and L be the set of local variables in the TM algorithm. Let In_A be the set of instructions of the TM algorithm. The syntax of the language Simple is shown in Figure 2.3.

Operational semantics

The semantics of the Simple language are intuitive. Nevertheless, we give the semantics here for the sake of completeness. Let σ_G be a valuation of the global variables, and let σ_L be a valuation of the local variables for each thread. Note that $dom(\sigma_G) = G$ and $dom(\sigma_L) = L \times T$. We define a *complete valuation* $\hat{\sigma} = \sigma_G \cup \sigma_L$ as a valuation of the global variables, and the local variables of each thread. Let $\hat{\Sigma}$ be the set of all complete valuations.

Given a thread t, we have the *t-valuation* as $\sigma = \sigma_G \cup \sigma_L^t$, where σ_L^t is the valuation of the local variables of thread t. A t-valuation consists of a valuation of the global variables, and a valuation of the local variables of thread t. Let Σ be the set of all t-valuations.

The operational semantics at the level of statements is given as:

$$\begin{array}{rcl}
\langle \mathsf{skip}, \sigma\rangle & \to & \sigma \\
\langle g := e, \sigma\rangle & \to & \sigma[g/\sigma[e]] \\
\langle l := e, \sigma\rangle & \to & \sigma[l/\sigma[e]] \\
\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma\rangle & \to & \langle s_1, \sigma\rangle \text{ if } \sigma[e] \neq 0 \\
\langle \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \sigma\rangle & \to & \langle s_2, \sigma\rangle \text{ if } \sigma[e] = 0 \\
\langle \textbf{for all } e \textit{ in } E \textbf{ do } s, \sigma\rangle & \to & \langle e := e_1; s; \textbf{for all } e \textit{ in } E \setminus \{e_1\} \textbf{ do } s, \sigma\rangle \\
 & & \text{where } e_1 \in E \\
\langle \textbf{for all } e \textit{ in } \emptyset \textbf{ do } s, \sigma\rangle & \to & \sigma \text{ where } E = \emptyset \\
\langle s; s', \sigma\rangle & \to & \langle s', \sigma'\rangle \text{ where } \langle s, \sigma\rangle \to \sigma'
\end{array}$$

The operational semantics at the level of programs is given as follows:

$$\frac{\sigma[e] \neq 0}{\langle \textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2; p,\ \sigma\rangle \xrightarrow{\varepsilon} \langle p_1; p,\ \sigma\rangle} \quad \text{(IF TRUE)}$$

$$\frac{\sigma[e] = 0}{\langle \textbf{if } e \textbf{ then } p_1 \textbf{ else } p_2; p,\ \sigma\rangle \xrightarrow{\varepsilon} \langle p_2; p,\ \sigma\rangle} \quad \text{(IF FALSE)}$$

$$\frac{e_1 \in E}{\langle \textbf{for all } e \textit{ in } E \textbf{ do } p; p_1,\ \sigma\rangle \xrightarrow{\varepsilon} \langle e := e_1; p; \textbf{for all } e \textit{ in } E \setminus \{e_1\} \textbf{ do } p; p_1,\ \sigma\rangle} \quad \text{(FOR ALL)}$$

$$\frac{E = \emptyset}{\langle \textbf{for all } e \textit{ in } E \textbf{ do } p; p_1,\ \sigma\rangle \xrightarrow{\varepsilon} \langle p_1,\ \sigma\rangle} \quad \text{(FOR ALL)}$$

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle in : s; p_1,\ \sigma\rangle \xrightarrow{in} \langle p_1,\ \sigma'\rangle} \quad \text{(INSTRUCTION)}$$

Given valuations σ, σ', programs p, p', and an instruction in, we say that $\langle p, \sigma\rangle$ produces $\langle p', \sigma'\rangle$ on instruction in if there exist valuations $\sigma_1 \ldots \sigma_n$ and programs $p_1 \ldots p_n$ such that

$$\langle p, \sigma\rangle \xrightarrow{\varepsilon} \langle p_1, \sigma_1\rangle \xrightarrow{\varepsilon} \ldots \langle p_n, \sigma_n\rangle \xrightarrow{in} \langle p', \sigma'\rangle$$

2.4.2 Language of a TM algorithm

A state of the TM algorithm consists of a valuation of the global variables, a valuation of the local variables for each thread, a Simple program to be executed by each thread, and a location in the transactional program of each thread. A *state* z of an TM algorithm is given by $\langle \sigma_G, \sigma_L, pc, txpc\rangle$, where σ_G is the valuation of the global variables of the TM algorithm, σ_L is the valuation of the local variables for each thread, $pc : T \to P$ is a function which represents the Simple program for each thread, and $txpc : T \to \mathbb{B}^*$ represents the location

in the transactional program for each thread. A TM algorithm A is a 5-tuple $\langle p_r, p_w, p_e, \sigma_G^{init}, \sigma_L^{init} \rangle$, where p_r, p_w, and p_e are Simple programs, and σ_G^{init} is the initial valuation of the global variables, and σ_L^{init} is the initial valuation of the local variables. For the transactional command $(txrd, k)$, the corresponding Simple program is p_r with $v = k$. For the transactional command $(txwr, k)$, the corresponding Simple program is p_w with $v = k$. For the transactional command xend, the corresponding Simple program is p_e.

Let a *scheduler* α on T be a function $\alpha : \mathbb{N} \to T$. Given a scheduler α, a transactional program *prog*, a *run* of a TM algorithm A is a sequence $\langle z_0, op_0 \rangle, \ldots \langle z_n, op_n \rangle$ such that $z_0 = \langle \sigma_G^{init}, \sigma_L^{init}, pc^{init}, txpc^{init} \rangle$ where $txpc^{init}(t) = \varepsilon$ for all threads $t \in T$, and $pc^{init}(t) = prog(t)(\varepsilon)$ for all threads $t \in T$, and for all j such that $0 \leq j < n$, if $z_j = \langle \sigma_G, \sigma_L, pc, txpc \rangle$ and $z_{j+1} = \langle \sigma'_G, \sigma'_L, pc', txpc' \rangle$, then the following hold:

1. for the thread $t = \alpha(j)$, we have that $\langle pc(t), \sigma \rangle$ produces $\langle p', \sigma' \rangle$ on instruction in_j for some Simple program p', where (a) $\sigma = \sigma_G \cup \sigma_L^t$ and $\sigma' = \sigma'_G \cup \sigma_L^{'t}$, (b) $txpc'(t) = txpc(t) \cdot 1$ if in_j is a read, write, or a commit instruction, and $txpc'(t) = txpc(t) \cdot 0$ if in_j is an abort instruction, and $txpc'(t) = txpc(t)$ otherwise, and (c) $pc'(t) = p'$ if $txpc'(t) = txpc(t)$, and $pc'(t)$ is the corresponding program of $prog(t)(txpc'(t))$ otherwise, and (c) $op_j = (in_j, \alpha(j))$, and

2. for all threads $t \neq \alpha(j)$ and all local variables $l \in L$, we have $\sigma_L^{'t} = \sigma_L^t$, and $pc(t) = pc'(t)$, and $txpc(t) = txpc'(t)$.

A history h corresponds to a run $\langle z_0, op_0 \rangle, \ldots, \langle z_n, op_n \rangle$ of a TM algorithm A if h is the longest subsequence of operations of $op_0 \ldots op_n$ in $\hat{Op}^*$. In other words, a history is obtained by removing the TM specific operations from a run. The *language* $L(A)$ of a TM algorithm A is the set of all histories h, such that h is the corresponding history of a run of the TM algorithm A. A TM algorithm A defines a TM Γ if the language $L(A)$ of the TM algorithm is equivalent to the set of all finite histories in the TM Γ.

2.4.3 The transition system of a TM algorithm

TM algorithms often use timestamps or version numbers to manage concurrency. These result in global and local variables with an unbounded number of valuations. To characterize TM algorithms using a finite number of states, we use boolean predicates over the set of global and local variables. A *boolean predicate pred* is defined by the function $pred : \hat{\Sigma} \to \mathbb{B}$.

A state q of a transition system is given by the pair $q = \langle pc, \sigma_{Preds} \rangle$, where $pc : T \to P$ is the program counter for the threads, and $\sigma_{Preds} : Preds \to \mathbb{B}$ is a valuation of the set *Preds* of boolean predicates.

Given a TM algorithm A on global variables G and local variables L, we define a *transition system* A^{ts} on a set *Preds* of boolean predicates by the tuple

$\langle Q, q_{init}, \delta \rangle$, where Q is the set of states of the transition system, q_{init} is the initial state, and $\delta \subseteq Q \times (In_A \times T) \times Q$ is the transition relation, such that

- $q_{init} = \langle pc^{init}, \sigma^{init}_{Preds} \rangle$ where $\sigma^{init}_{Preds}(pred) = pred(\sigma^{init}_G \cup \sigma^{init}_L)$ for all predicates $pred \in Preds$, and

- for every two distinct states $q_1 = \langle pc, \sigma_{Preds} \rangle$ and $q_2 = \langle pc', \sigma'_{Preds} \rangle$ in Q, either there exists a thread $t \in T$ such that $pc(t) \neq pc'(t)$, or there exists a predicate $pred \in Preds$ such that $q_1(pred) \neq q_2(pred)$, and

- if there exists a run $r = \langle z_0, op_0 \rangle \ldots \langle \langle \sigma_G, \sigma_L, pc, txpc \rangle, op \rangle \langle \langle \sigma'_G, \sigma'_L, pc', txpc' \rangle, op' \rangle \ldots \langle z_n, op_n \rangle$ of the TM algorithm A, then $(\langle pc, \sigma_{Preds} \rangle, op, \langle pc', \sigma'_{Preds} \rangle) \in \delta$ where $\sigma_{Preds}(pred) = pred(\sigma_G \cup \sigma_L)$ and $\sigma'_{Preds}(pred) = pred(\sigma'_G \cup \sigma'_L)$ for all predicates $pred \in Preds$.

A sequence $op_0 \ldots op_m$ in $(In_A \times T)^*$ is a *run* of the transition system of the TM algorithm if there exist states $q_0 \ldots q_{m+1}$ in Q such that $q_0 = q_{init}$, and for all i such that $0 \leq i \leq m$, we have $(q_i, op_i, q_{i+1}) \in \delta$. A history $h \in \hat{Op}^*$ corresponding to a run r is the longest subsequence of operations in $\hat{Op}^*$ in r. The language $L(A^{ts})$ of the transition system of a TM algorithm A is the set of all histories h such that h corresponds to a run of A^{ts}. In fact, the transition system captures the set of all runs of a TM algorithm, that is, $L(A^{ts}) \subseteq L(A)$. The motivation behind the set of boolean predicates is to capture the behavior of a TM algorithm in a finite number of states. We describe how we build transition systems of TM algorithms in more detail in Chapter 3.

2.5 Examples of TM Algorithms

We describe four TM algorithms. We prove the safety and liveness properties of these TM algorithms. For ease of understanding, we start with a basic TM which executes transactions serially, and then move to realistic TM algorithms. For all TM algorithms, we present the p_r, p_w, and p_e programs in our Simple language. We also give the initial valuation of the global and local variables. In all TM algorithms, the value `self` denotes the number of the thread executing the algorithm.

2.5.1 Sequential TM

The sequential TM executes the transactions sequentially. There is one global variable `glock` such that `glock` $\in T \cup \{0\}$. The initial value of `glock` is 0. The set In_{seq} of instructions is $\hat{In}$. The sequential TM algorithm is shown in Algorithm 2.3.

```
program p_r :
if glock ≠ 0 and glock ≠ self then p_a
else (read, v) : glock := self

program p_w :
if glock ≠ 0 and glock ≠ self then p_a
else (write, v) : glock := self

program p_e :
if glock ≠ 0 and glock ≠ self then p_a
else commit : glock := 0

program p_a :
abort : skip
```

Algorithm 2.3: The sequential TM algorithm in Simple

2.5.2 Two phase locking TM

Two phase locking TM (2PL TM) defines a locking protocol per variable. A thread acquires a shared read lock for a variable when the thread wants to read the variable. A thread acquires an exclusive write lock for a variable when a thread wants to write the variable.

The 2PL TM algorithm consists of the global variables `wlock` and `rlock`, where $\texttt{wlock} : V \to T \cup \{0\}$ represents the write lock, and $\texttt{rlock} : V \times T \to \{\mathit{true}, \mathit{false}\}$ represents the read lock for the variables. The initial value of `wlock` is $\texttt{wlock}_0$ where $\texttt{wlock}_0(v) = 0$ for all variables $v \in V$. The initial value of `rlock` is $\texttt{rlock}_0$ where $\texttt{rlock}_0(v, t) = \mathit{false}$ for all variables $v \in V$ and all threads $t \in T$. The set In_{2PL} of instructions is $\hat{In}$. The 2PL TM algorithm is shown in Algorithm 2.4.

2.5.3 Transactional locking II

Two phase locking requires that a thread locks a variable when it reads or writes to the variable. This means that a thread may lock a variable for the whole duration of a transaction. Thus, a transaction wishing to read a variable may have to wait for the whole duration of a transaction writing to the variable. Transactional locking II (TL2) [DSS06] improves efficiency by not requiring threads to lock variables at encounter-time, rather lock them at commit time. When a transaction writes to a variable, the variable is simply added to the write set. The locks are obtained just before the commit (lazy acquire). Then, it is checked, for every variable read, that the current version is not newer than the version read by the transaction (read set validation). Also, it is checked that every variable in the read set is not locked by any other thread. The use of version numbers in TL2 allows efficient read set validation.

```
program p_r :
if wlock(v) ≠ 0 and wlock(v) ≠ self then p_a
else (read, v) : rlock(v, self) := true

program p_w :
if wlock(v) ≠ 0 and wlock(v) ≠ self then p_a
if rlock(v, u) = true for some thread u ≠ self then p_a
(write, v) : wlock(v) := self

program p_e :
commit :
forall v in V do
  if wlock(v) = self then wlock(v) := 0
  if rlock(v, self) = true then rlock(v, self) := false

program p_a :
abort :
forall v in V do
  if wlock(v) = self then wlock(v) := 0
  if rlock(v, self) = true then rlock(v, self) := false
```

Algorithm 2.4: The two-phase locking TM algorithm in Simple

As version numbers are integer valued and increase monotonically, TL2 has an infinite state space even for a finite number of threads and variables. In Chapter 3, we describe the set of boolean predicates we consider in order to get a finite state representation of TL2.

The TL2 TM algorithm consists of the global variables `lock`, `clk`, and `version`, and the local variables `wflag`, `rflag`, lver, l, and `lclock`. The lock variable is a function $\texttt{lock} : V \to T \cup \{0\}$. The clock variable $\texttt{clk} \in \mathbb{N}$ is a natural number. The version variable is a function $\texttt{version} : V \to \mathbb{N}$. The write flag and the read flag variables are functions $\texttt{wflag} : V \to \{true, false\}$ and $\texttt{rflag} : V \to \{true, false\}$. The local version variable is a function lver : $V \to \mathbb{N}$. The variable l is either 0 or 1. The local clock variable $\texttt{lclock} \in \mathbb{N}$ is a natural number. The initial valuation of the variables is as follows: `lock` is initially $\texttt{lock}_0$ such that $\texttt{lock}_0(v) = 0$ for all variables v. `version` is initially $\texttt{version}_0$ such that $\texttt{version}_0(v) = 0$ for all variables v. `clk` is initially 0. `wflag` is initially $\texttt{wflag}_0$ such that $\texttt{wflag}_0(v) = false$ for all variables v. `rflag` is initially $\texttt{rflag}_0$ such that $\texttt{rflag}_0(v) = false$ for all variables v. lver is initially lver_0 such that $\text{lver}_0(v) = 0$ for all variables v. l is initially 0. `lclock` is initially 0. The set In_{TL2} of instructions is $\hat{In} \cup (\{\text{lock}, \text{chklock}, \text{validate}\} \times V) \cup \{\text{start}\}$. The TL2 TM algorithm is shown in Algorithm 2.5.

```
program p_r :
if lclock = 0 then
  start : lclock := clk
if lock(v) ≠ 0 then p_a
if clk ≠ lclock then p_a
(read, v) :
  rflag(v) := true
  lver(v) := version(v)

program p_w :
if lclock = 0 then
  start : lclock := clk
(write, v) : wflag(v) := true

program p_a :
abort :
forall v in V do
  if lock(v) = self then
    lock(v) := 0
    wflag(v) := false
  if rflag(v) = true then
    rflag(v) := false
lclock := 0
```

```
program p_e :
forall v in V do
  if wflag(v) = true then
    if lock(v) = 0 then
      (lock, v) : lock(v) = self
    else p_a
increment :
  clk := clk + 1
  lclock := clk
l := 0
forall v in V do
  if rflag(v) = true then
    (chklock, v) :
    if lock(v) ≠ 0 then l := 1
    (validate, v) :
    if version(v) > lver(v) then l := 1
    if l = 1 then p_a
commit :
forall v in V do
  if wflag(v) = true then
    lock(v) := 0
    wflag(v) := false
    version(v) := lclock
  if rflag(v) = true then
    rflag(v) := false
lclock := 0
```

Algorithm 2.5: The TL2 TM algorithm in Simple

2.5.4 DSTM

All the above three TM algorithms are based on locks. If a thread has locked a variable, another thread cannot acquire the lock. Thus, none of the three algorithms can guarantee nonblocking progress. We shall prove this using our automated verification tool later. Dynamic software transactional memory (DSTM) [HLMS03] is a TM algorithm designed such that it guarantees the nonblocking progress property of obstruction freedom. DSTM is based on the notion of ownership. Even if a thread owns a variable, another thread may aggressively acquire ownership from the thread.

The DSTM algorithm consists of the global variables `status`, `owner`, and `readset`. The status variable is a function $\texttt{status} : T \to \{\mathsf{aborted}, \mathsf{finished}, \mathsf{invalid}\}$, the owner variable is a function $\texttt{owner} : V \to T \cup \{0\}$, and the readset variable is a function $\texttt{readset} : V \times T \to \{\mathit{true}, \mathit{false}\}$. The initial valuation of these variables are $\texttt{status}_0$, $\texttt{owner}_0$, and $\texttt{readset}_0$ respectively, where $\texttt{status}_0(t) = \mathsf{finished}$ for all threads $t \in T$ and $\texttt{owner}_0(v) = 0$ for all variables

```
program p_r :
if status(t) = finished then
  (read, v) : readset(v, t) := true
else p_a

program p_w :
if status(t) = aborted then p_a
(own, v) :
  u := owner(v)
  if u ∉ {0, t} then
    status(u) := aborted
    forall v' in V do
      if owner(v') = u then
        owner(v') := 0
    owner(v) := t

program p_a :
abort :
forall v in V do
  readset(v, t) := false

program p_e :
if status(t) = finished then
validate :
  forall v in V do
    if readset(v, t) = true then
      u := owner(v)
      if u ∉ {0, t} then
        status(u) := aborted
        forall v' in V do
          if owner(v') = u then
            owner(v') := 0
else p_a
if status(t) = finished then
commit :
  forall v in V do
    readset(v, t) := false
    if owner(v) = t then
      owner(v) := 0
      forall u in T do
        if readset(v, u) = true then
          status(u) := invalid
else p_a
```

Algorithm 2.6: The DSTM algorithm in Simple

$v \in V$, and $\texttt{readset}_0(v, t) = \mathit{false}$ for all variables $v \in V$ and all threads $t \in T$.

3 Preliminary Verification

As TM provide a programmer with a flexible programming paradigm, a TM can involve an arbitrary number of concurrent threads and variables. Thus, a TM algorithm may have an unbounded number of states (corresponding to state of every variable for every thread), where every state has an unbounded number of transitions (corresponding to read or write for every variable). Moreover, as we saw in the TL2 algorithm, a TM algorithm which uses integer valued variables, like version numbers or timestamps, for managing concurrency may have an infinite number of states even for a finite number of threads and variables.

The TM specifications developed in Chapter 2 show that checking whether a history is opaque or not requires $O(n \cdot |h|)$ time, where n is the number of threads, and $|h|$ is the length of the history. This check can be expensive for TM algorithms, which are highly tuned for performance, rather than preciseness. In other words, efficient TM algorithms rely on coarse means to detect conflicts, and thus often abort even when the history is opaque. These coarse means of conflict detection turn out to be helpful in our verification technique. TM algorithms detect conflicts on each variable independently, based on the interaction of a thread with all other threads. A common technique in checking correctness of arbitrarily sized systems lies in exploiting the inherent symmetry of the system [ES96, HQR99]. We develop similar techniques to prove the correctness of TM for any number of threads and variables.

We prove the correctness of TM algorithms in two parts. First, we establish the correctness of a TM algorithm for a finite number of threads and variables, and second, we describe and use the structural properties of TM to prove a reduction theorem which states that if a TM is correct for the finite number of threads and variables as established in the first part, then the TM is correct for an arbitrary number of threads and variables.

3.1 Model Checking Safety

We model check TM with two threads and two variables for opacity. Later, in Theorem 3.2 we show that if a TM Γ guarantees opacity for two threads and two variables, then Γ guarantees opacity for an arbitrary number of threads and variables.

3.1.1 Obtaining a suitable transition system

We now verify the opacity of the TM algorithms presented in the previous chapter. For each TM algorithm, we give a set of suitable boolean predicates such that the transition system of the TM algorithm has a finite number of states. We check whether the language of the transition system of the TM algorithm is included in the language of the deterministic TM specification for opacity. We then check whether the transition system obtained in this manner is deterministic. If so, we know that our verification is sound and complete. If the transition system obtained is nondeterministic, our verification technique would be sound but incomplete: if we find that $L(A^{ts}) \subseteq L(Spec)$, then the TM algorithm A guarantees opacity, but if $L(A^{ts}) \not\subseteq L(Spec)$, we cannot say that the TM algorithm A does not guarantee opacity. In this case, we need to check that the counterexample to opacity on the transition system is indeed a counterexample on the TM algorithm. We manually refine the transition system by changing the set of boolean predicates until we prove that the TM algorithm is opaque, or obtain a counterexample to opacity of the TM algorithm.

The TM algorithms, sequential, 2PL, and DSTM, consist of finite-valued global and local variables. For these algorithms, the set of boolean predicates is simply the set of valuations for these variables. For the sequential TM algorithm, the set of boolean predicates is $\{\texttt{glock} = t \mid t \in T \cup \{0\}\}$. For the two phase locking TM algorithm, the set of boolean predicates is $\{\texttt{wlock}(v) = t \mid v \in V, t \in T \cup \{0\}\} \cup \{\texttt{rlock}(v,t) \mid v \in V, t \in T\}$. For the DSTM algorithm, the set of boolean predicates is $\{\texttt{status}(t) = s \mid t \in T, s \in \{\mathsf{aborted}, \mathsf{finished}, \mathsf{invalid}\}\} \cup \{\texttt{owner}(v) = t \mid v \in V, t \in T\} \cup \{\texttt{readset}(v,t) \mid v \in V, t \in T\}$. The TM algorithm TL2 contains version numbers and clock variables which are integer valued. So, for TL2 we consider the boolean predicates as follows: $\{\mathsf{lock}(v) = t \mid v \in V, t \in T \cup \{0\}\} \cup \{(\texttt{wflag}(v), t) \mid v \in V, t \in T\} \cup \{(\texttt{rflag}(v), t) \mid v \in V, t \in T\} \cup \{(\texttt{clk} = \texttt{lclock}(t)) \mid t \in T\} \cup \{(\texttt{version}(v) = (\mathsf{lver}(v), t) \mid v \in V, t \in T\}$ With the given boolean predicates for each TM algorithm, we construct the transition system A^{ts} for each TM algorithm.

3.1.2 Results

We now check the opacity of different TM by checking whether the language $L(A^{ts})$ of the transition system A^{ts} is included in the language $L(Spec^d)$ of the deterministic TM specification $Spec^d$ of opacity. We find that the transition

Table 3.1: Time for verifying opacity of TM algorithms using language inclusion. The experiments are performed on a dual core 2.8 GHz PC with 2 GB RAM. In case the language inclusion holds, we write YES followed by the time required for finding it. Otherwise, we write NO followed by the counterexample produced, followed by the time required to find the counterexample. $(\mathsf{write}, v_2)_1$ denotes a write to variable v_2 by thread 1.

TM	Size	$L(A^{ts}) \subseteq L(Spec)$
seq	3	YES, $0.01s$
2PL	99	YES, $0.01s$
dstm	1846	YES, $0.13s$
TL2	21568	YES, $2.4s$
mod *TL2*	17520	NO, h_1, $8s$
	Counterexamples	
h_1	$(\mathsf{write}, v_2)_1, (\mathsf{write}, v_1)_2, (\mathsf{read}, v_2)_2, (\mathsf{read}, v_1)_1, \mathsf{commit}_2, \mathsf{commit}_1$	

system of each TM algorithm is deterministic. Table 3.1 shows our results and leads to the following theorem.

Theorem 3.1. *The sequential TM, two-phase locking TM, TL2 and DSTM ensure opacity for two threads and two variables.*

3.1.3 Ordering in TL2

Our tool discovered a subtle point in TL2. In the description of the published TL2 algorithm, we found the order of two operations, validating the read set (validate), and checking whether a variable in the read set is locked (chklock), ambiguous. We reordered the statements corresponding to the instructions chklock and validate. We call this new TM algorithm as the modified TL2 TM algorithm. We found that the language of the transition system of the TL2 algorithm is not included in the language of the TM specification for opacity. We obtain a counterexample which is also a counterexample of the TL2 TM algorithm. In the published TL2 algorithm, the authors maintain the version number and the lock bit of every variable in the same memory word. This ensures that the two operations chklock and validate execute atomically, and thus they can be executed in any order. Our experiments discover that the correctness of TL2 is based on the subtle fact that either the version number and the lock bit have to be accessed atomically, or validate has to occur after chklock.

Note that this observation motivates the importance of verifying the correctness of TM algorithms under relaxed memory models. Although the programmer first performs a chklock followed by a validate in the TM algorithm, it is possible that a relaxed memory model reorders the two checks.

3.2 Model Checking Liveness

We use the formalism of TM algorithms to verify liveness properties of TM. We define a *loop* l in a TM algorithm A as a finite history $op_m \dots op_n$ such that there exists a run $\langle z_0, op_0\rangle \dots \langle z_m, op_m\rangle \dots \langle z_n, op_n\rangle$ of A such that $z_m = z_n$. Similarly, we define a loop in the transition system of a TM algorithm.

Note that we defined obstruction freedom using a Streett condition in Chapter 2 as $\bigwedge_{t\in T}(\Box\Diamond(\mathsf{abort}, t) \rightarrow \Box\Diamond((\mathsf{commit}, t) \vee \bigvee_{in\in\hat{I}n, u\in T\setminus\{t\}}(in, u)))$.

Every history h that is not obstruction free violates at least one of the conjuncts of the Streett condition stated above. Each conjunct (Streett pair) corresponds to one thread. A history h can violate the condition for thread t, only if h has from some point on only statements of t. Note that in this case h trivially satisfies the Streett pairs for other threads. This fact allows us to use a simple model checking procedure, even though obstruction freedom is formally a Streett condition.

In particular, a TM defined by a TM algorithm A ensures obstruction freedom iff there is no loop l in A such that all statements in l are from the same thread, and l contains no commit, and l contains an abort. Similarly, a TM ensures livelock freedom iff there is no loop l in A such l contains no commit, and every thread that has a statement in l, has an abort in l.

Liveness verification results

We built a verification tool to check obstruction freedom and livelock freedom properties of TM algorithms. Our tool provides a platform for TM designers to check which liveness properties are ensured. If the liveness property fails, then the tool provides feedback in the form of a history that represents a counterexample. Our results are shown in Table 3.2.

We give the description of the TM algorithms and the boolean predicates for each TM algorithm as input to our model checking tool. To check obstruction freedom, our tool tries to find a history $h = op_0 \dots op_n$ of the transition system of the TM algorithm such that there exists a suffix h' of h such that all operations in h' are from the same thread, and h' has no commit, and h' has an abort, and h' is a loop in the transition system. If the tool finds such a history, the tool checks that h' is also a loop of the TM algorithm. If so, the loop is a counterexample to obstruction freedom. If the tool does not find a loop, we know that the TM ensures obstruction freedom. Similarly, to check livelock freedom, our tool tries to find a loop l in the TM transition system such that there is no commit in l, and every thread that has a statement in l, has an abort in l. If our tool finds a loop l in the transition system of the TM algorithm, such that l is not a loop in the TM algorithm, then we need to manually refine the transition system, until we prove that the TM algorithm satisfies the required liveness property, or find a loop in the transition system which is also a loop in the TM algorithm.

Table 3.2: Time for verifying liveness of TM algorithms. The experiments are performed on a dual core 2.66GHz desktop PC with 2 GB RAM. The notation is similar to Table 3.1. The time denotes the time required to prove a liveness property or find a counterexample. The counterexamples obtained are of the form $a \cdot b^{\omega}$. We write the loop b here.

TM	Obstruction freedom	Livelock freedom
seq	NO, h_1, $0.1s$	NO, h_1, $0.1s$
2PL	NO, h_1, $0.1s$	NO, h_1, $0.1s$
dstm	YES, $2s$	NO, h_2, $0.2s$
TL2	NO, h_1, $0.4s$	NO, h_1, $0.4s$
	Counterexamples	
h_1	abort_1	
h_2	$(\mathsf{own}, v_1)_1, (\mathsf{own}, v_1)_2, \mathsf{abort}_1, \mathsf{abort}_2$	

3.3 Extending the Verification Results

We now reduce the problem of verifying opacity of a TM algorithm with an arbitrary number of threads and variables to verifying opacity of the TM algorithm with two threads and two variables.

3.3.1 Properties of TM algorithms

To reduce the verification problem of TM algorithms to a finite number of threads and variables, we reason about TM algorithms in terms of the following properties.

- A TM algorithm A is *abort isolated* if for every history $h \in L(A)$, for every aborted transaction x in h, if an instruction *in* of x changes the value of a global variable g and a transaction y observes the value of g before x aborts, then y aborts in the step of observing g.

- A TM algorithm is *pending isolated* if for every history h, for every pending transaction x in h, if an instruction *in* of x changes the value of a global variable g and a transaction y observes the value of g before x finishes, then y aborts.

  These two properties restrict the aborting and pending transactions in a TM. These properties require that if an aborting or a pending transaction change the global state, then that change is not visible to committing transactions. We shall later use these properties to remove the aborting and pending transactions from a history.

- A TM algorithm is *conflict commutative* if for every history h where the execution of a transactional command c_1 by thread t overlaps with the

execution of a transactional command c_2 by thread u, if c_1 consists of an instructions in_1 and in_3, and c_2 consists of instructions in_2 and in_4 such that in_1 occurs before in_2 in h and they conflict, and $h = h_1 in_4 in_3 h_2$, then $h' = h_1 in_3 in_4 h_2$ is also a history in the language of the TM algorithm.

3.3.2 Checking properties of example TM algorithms

We shall now informally explain why the TM algorithms we considered are abort isolated, pending isolated, and conflict commutative.

Sequential TM

Let us consider the sequential TM algorithm. It is definitely one of the most basic TM algorithms that allows multiple threads to commit transactions. The sequential TM algorithm consists of $n + 1$ states, where n is the number of threads. We now provide the intuition why the sequential TM algorithm satisfies the above definitions.
Abort isolated. The thread t aborts in a transition if some other thread $u \neq t$ holds the global lock, that is, $\texttt{glock} = u$. An aborting transaction does not modify the global variable `glock`, and thus the sequential TM algorithm is abort isolated.
Pending isolated. The thread t is pending if t holds the global lock, that is, $\texttt{glock} = t$. Every thread $u \neq t$ aborts if it observes $\texttt{glock} = t$. Thus, the sequential TM algorithm is pending isolated.
Conflict commutative. Every command in the sequential TM executes within a single instruction. Thus, the sequential TM algorithm is conflict commutative.

Two phase locking TM

The sequential TM algorithm is extremely coarse-grained. No two threads can execute transactions at the same time. The two phase locking TM is a relatively fine-grained locking scheme which allows multiple transactions executing on different variables to run concurrently. We now show that the two phase locking TM satisfies the required definitions.
Abort and pending isolated. An aborted transaction can change the state of the variables in intermediate steps in two phase locking TM. The states which differ from the initial state are when the aborted transaction x of a thread t sets $\texttt{wlock}(v) = t$ and when it sets $\texttt{rlock}(v, t) = \mathit{true}$. Note that a thread u observes the state if u wants to read or write v. In this case, u aborts. Thus, the two phase locking TM is abort isolated. By a similar argument as for abort isolation, two phase locking TM is pending isolated.
Conflict commutative. A read command consists of obtaining a read lock followed by performing the actual read. A commit command consists of releasing all locks, followed by the commit. Note that a read command cannot be con-

current with a conflicting commit. Similarly, two conflicting commits cannot be concurrent.

Transactional locking II

TL2 differs from two phase locking protocol in two ways: write locks are acquired at commit time instead of encounter time, and reads use version numbers to validate, rather than acquiring read locks. We explain why TL2 satisfies the required properties.
Abort and pending isolated. The TL2 algorithm changes the global state on a lock instruction, and the commit instruction. An intermediate state change by an aborting transaction can be observed by another transaction y only if y attempts to access the lock held by x. However, in that case, y aborts. Similarly, TL2 is pending isolated.
Conflict commutative. As the read step is considered atomic, it cannot overlap with a conflicting commit. Two conflicting commits cannot overlap as they both require to lock the variable. Thus, the TL2 algorithm is conflict commutative.

DSTM

We show that the required properties hold for DSTM.
Abort and pending isolated. An aborted or a pending transaction does not change the state in DSTM. Thus, DSTM is abort and pending isolated.
Conflict commutative. As the read consists of a single instruction, it cannot be concurrent with a commit instruction. Moreover, two commits cannot be concurrent as they both own the variable they write to. Hence, DSTM is conflict commutative.

3.3.3 Structural properties of TM

We now present four structural properties of TM algorithms. We then use these properties to prove the reduction theorem for opacity. We use the above definitions to show that the TM algorithms, sequential TM, two-phase locking TM, DSTM, and TL2 TM, satisfy these structural properties. Note that the properties are sufficient (and not necessary) conditions for the reduction theorem to hold. Let Γ be a transactional memory and let A be the corresponding TM algorithm. Let h be a history in Γ.

P1 Transaction projection

We define the *transaction projection* of h on $X' \subseteq X$ as the subsequence of h that contains every statement of all transactions in X'. The property P1 states that the transaction projection of h on X' is in Γ, where X' contains all committed transactions and no aborted transactions.

We should note that, however, we cannot project away a subset of the aborted transactions. This is because removing an aborted transaction may allow another aborted transaction to commit.

Lemma 3.1. *If a TM algorithm A is abort isolated and pending isolated, then the TM Γ satisfies transaction projection.*

Proof. Consider an arbitrary history $h \in L(A)$. We can divide the history h into subsequences $h_1 \dots h_n$, where for all i, all statements in h_i are committing, aborting, or pending. As the TM algorithm is abort isolated, we can remove the subsequences from $h_1 \dots h_n$ which are aborting. Moreover, as the TM algorithm is pending isolated, we can remove a subset of the subsequences which are pending. Hence, we get a new history h' such that all statements in h' belong to committed or pending transactions, and $h' \in L(A)$. □

P2 Thread renaming

For non-overlapping transactions, the TM is oblivious to the identity of the thread executing the transaction. The property P2 states that if (i) h has no aborting transactions, and (ii) there exist two threads u and t such that for all committing transactions x of u and y of v in the history h, either $x <_w y$ or $y <_w x$, then the history h' obtained by renaming all transactions of thread u to be from thread t is in Γ. We note that for all TM algorithms discussed, the programs p_r, p_w, and p_e and the initial state z^{init} do not distinguish between the threads. Thus, the TM algorithms we consider satisfy the thread renaming property.

P3 Variable projection

If a transaction can commit, then removing all statements that involve some particular variables does not cause the transaction to abort. We define the *variable projection* of h on $V' \subseteq V$ as the subsequence of h that contains all commit and abort statements, and all read and write statements to variables in V'. The property P3 states that if h has no aborting transactions, then for all $V' \subseteq V$, the variable projection of h on V' is in Γ. A TM satisfies variable projection if reading or writing a variable does not remove a conflict on other variables. All TM we discussed (sequential TM, two phase locking TM, TL2, and DSTM) satisfy P3 as they track every variable accessed by every thread independently.

P4 Monotonicity

The most important property which allows to reduce the verification property is the monotonicity in TM. Monotonicity states that if a history is allowed by the TM, then more sequential forms of the history are also allowed. Formally, let $F \subseteq \hat{Op}^*$ be the set of opaque histories with all committed and exactly one

unfinished transaction. We define a function $seq : F \to 2^F$ such that if $h_2 \in seq(h_1)$, then h_2 is sequential and strictly equivalent to h_1. The monotonicity property for opacity states that if $h = h' \cdot op$, where $h' \in F$, and op is not an abort, and op is an operation of the unfinished transaction in h', then for every history $h_2 \in seq(h')$, the history $h_2 \cdot op$ is a finite prefix of a history in Γ.

Lemma 3.2. *If the TM algorithm A is conflict commutative, then the TM Γ is monotonic.*

Proof. Consider a history $h = h' \cdot op$ produced by the TM algorithm. We want to prove that if h' is opaque and h' consists of all committed and exactly one unfinished transaction, then h' can be sequentialized. We consider the history h'. For every pair of conflicting operations, we use conflict commutativity to sequentialize the corresponding commands. As h' is opaque, sequentializing the commands gives a sequential history such that the state of the TM algorithm after the sequential version of h' is equivalent to the state of the TM algorithm after h'. Thus, $h' \cdot op$ is produced by the TM algorithm. Thus, Γ is monotonic. □

3.3.4 The reduction theorem

Based on the above four structural properties, we now prove that the problem of verifying TM algorithms with an arbitrary number of threads and variables can be reduced to verifying TM algorithms with two threads and two variables.

Theorem 3.2. *If a TM Γ ensures $(2, 2)$ opacity and satisfies the properties P1, P2, P3, and P4, then Γ ensures opacity.*

Proof. The proof is by contradiction. Let $h \in \Gamma$ be not opaque. Let h_p be the longest finite prefix of h such that h_p is opaque and let $h_1 = h_p \cdot op$, where $op = (in, t)$ is an operation of transaction x. Let X be the set of committed transactions in h_p. By property P1, there exists a history h_2 generated by projecting h_1 to $X \cup \{x\}$ such that $h_2 \in \Gamma$. We note that $h_2 = h'_p \cdot op$ and h'_p is opaque and h_2 is not opaque. So, using property P4, there exists a history $h''_p \in seq(h'_p)$ such that the history $h_3 = h''_p \cdot op$ is in Γ. In h_3 only one transaction, x, does not execute sequentially. Using property P2, we rename the threads for the transactions in h_3. We let all transactions except x to be executed by thread u. Let this renaming give history h_4. We note that the last statement of x is a commitor a read of a variable. As h_4 is not opaque, we know (by the definition of conflict) that one of the following holds: (i) $op_1 = ((\mathsf{read}, v_1), t)$ and $op_2 = ((\mathsf{read}, v_2), t)$ are global reads of transaction x such that some transaction y of thread u writes to v_1 and some transaction y' of u with $y' = y$ or $y <_{h_4} y'$ writes to v_2 and both commit between op_1 and op_2, (note that y and y' cannot overlap due to the structure of h_4,) or (ii) $op_1 = ((\mathsf{read}, v_1), t)$ is a global read of transaction x such that some transaction y of thread u writes to v_1 and commits after op_1, and there is a committing

transaction y' with $y' = y$ or $y <_{h_4} y'$ which has a command (read, v_2) or (write, v_2), and x also writes to v_2. (Note that v_1 may be same as v_2). Let h_5 be a variable projection of h_4 on $\{v_1, v_2\}$. We know that h_5 is in Γ, by property P3. Also, we note that h_5 is not opaque. As we know that all histories $h \in \Gamma$ on two threads and two variables are opacity, we get a contradiction. □

3.4 Reducing the Liveness Verification Problem

We now present a reduction theorem that proves that it is sufficient to verify obstruction freedom of an opaque TM on histories with two threads and one variable to generalize the result to all histories. As we saw that none of our TM algorithms is livelock free, we do not build a reduction theorem for livelock freedom.

Theorem 3.3. *If an opaque TM Γ ensures $(2, 1)$ obstruction freedom and satisfies the properties P1-P4, then Γ ensures obstruction freedom.*

Proof. Let $\bar{h} \in \Gamma$ be a history such that $\bar{h}$ is not obstruction free. As $\bar{h}$ is not obstruction free, we note that $\bar{h}$ can be written in the form $h_1 \cdot \bar{h_2}$, such that (i) no unfinished transaction in h_1 has a statement in $\bar{h_2}$, and (ii) all statements in $\bar{h_2}$ are from the same thread, and (iii) there is no commit instruction in $\bar{h_2}$. First, we use the property P3 (variable projection) and claim that there exists a history $h_1 \cdot \bar{h_3} \in \Gamma$, such that $\bar{h_3}$ accesses at most one variable. We now use the property of transactional projection and claim that there exists a history $h_4 \cdot \bar{h_3} \in \Gamma$ such that h_4 has no aborting transactions and at most one pending transaction. Using the fact that the TM Γ satisfies opacity, we use the monotonicity property to claim that $h_5 \cdot \bar{h_3} \in \Gamma$ such that h_5 is sequential and h_5 is strictly equivalent to h_4. Using the property of variable projection again, we claim that there exists a history $h_6 \cdot \bar{h_3} \in \Gamma$ such that h_6 is the variable projection of h_5 on v. Using thread symmetry, we can rename all transactions in h_6 to be from the same thread u, and thus obtain a history $h_7 \cdot \bar{h_3} \in \Gamma$. Note that $h_7 \cdot \bar{h_3}$ is a history on 2 threads and 1 variable and is not obstruction-free, which leads to a contradiction. □

Theorem 3.4. *DSTM ensures obstruction freedom but does not ensure livelock freedom. The sequential TM, two-phase locking TM, and TL2 do not ensure obstruction freedom.*

3.4.1 Discussion

We believe that the structural properties are easier to prove than manually proving the opacity of the TM algorithms. This is because the structural properties can be established by arguing about the protocol of a single thread for checking if the TM algorithm is abort isolated or pending isolated, or for checking if the TM algorithm satisfies the thread symmetry or the variable

projection property. Moreover, to check conflict commutative property, we need to locally reason about the interaction of two atomic instructions. On the other hand, reasoning about the correctness of a TM algorithm requires us to reason for the whole TM algorithm. We believe that the proofs in this chapter can largely be automated.

At this point, we hope the reader has an understanding of our verification technique: our formalism to express TM algorithms, our TM specifications, and the structural properties of TM. Now, we delve into intricate issues. We now consider the set of instructions to be hardware instructions. We use a language RML instead of the language Simple, to capture the effect of memory models. The semantics of RML allow to reorder and eliminate memory instructions. Based on RML, we develop a tool FOIL, which allows us to verify the results obtained in this chapter at hardware-level atomicity under various relaxed memory models.

4 The Formalism

The preliminary formalism and the verification approach assumed that the high-level transactional commands like read, write, commit, and abort execute atomically and in a sequentially consistent manner. Verification of a TM at this level of abstraction leaves much room for errors in a realistic setting. For our verification to be useful, we need to prove correctness of TM on real hardware with relaxed memory models. Moreover, the assumption that the write command to a variable does not update the variable globally (the writes are flushed at commit time) cannot model direct update TM.

We revise the framework developed in Chapter 2 for verifying TM algorithms on relaxed memory models at hardware level atomicity. This requires us to revisit the formalism and describe the instructions which are guaranteed to be atomic by the hardware. Then, we present our language, Relaxed Memory Language (RML), which allows to describe TM algorithms at hardware level atomicity and to capture their behavior under relaxed memory models.

4.1 Framework

We first present a general formalism to express hardware memory instructions. We formalize memory models which describe the interaction between memory instructions. Then, we define opacity at the fine-grained level of atomicity, and represent it using a TM specification. We retain the model of transactional programs developed in Chapter 2.

4.1.1 Memory instructions

Let *Addr* be a set of memory addresses. Let *In* be the set of *memory instructions* that are executed atomically by the hardware. We define the set *In* as

follows, where $a \in Addr$:

$$In ::= \langle \mathsf{load}\ a \rangle \mid \langle \mathsf{store}\ a \rangle \mid \langle \mathsf{cas}\ a \rangle$$

We use the $\langle \mathsf{cas}\ a \rangle$ instruction as a generic read-modify-write instruction.

4.1.2 Memory models

A *memory model* is a function $M : In \times In \to \{N, E, Y\}$. For all instructions $in_1, in_2 \in In$, when in_1 is immediately followed by in_2, we have: (i) if $M(in_1, in_2) = N$, then M imposes a strict order between in_1 and in_2, (ii) if $M(in_1, in_2) = E$, then M allows to eliminate the instruction in_2, and (iii) if $M(in_1, in_2) = Y$, then M allows to reorder in_1 and in_2. The case (ii) allows us to model store load forwarding using store buffers, and the case (iii) allows us to model reordering of instructions. Our formalism can capture many hardware memory models. But, our formalism cannot capture some common compiler optimizations like irrelevant read elimination, and thus disallows many software memory models (like the Java memory model [MPA05]). We specify different memory models in our framework. These memory models are chosen to illustrate different levels of relaxations generally provided by the hardware. Let $\mathbf{M}$ be the set of all memory models.

Sequential consistency

Sequential consistency does not allow any pair of instructions to be reordered. *Sequential consistency* [Lam79] is specified by the memory model M_{sc}. We have $M_{sc}(in_1, in_2) = N$ for all instructions $in_1, in_2 \in In$.

Total store order

Total store order (TSO) relaxes the order of a store followed by a load to a different address. But, TSO enforces a strict order on the stores (and hence the name). TSO allows a load which follows a store to the same address to be eliminated. TSO [WG94] is given by the memory model M_{tso} such that for all memory instructions $in_1, in_2 \in In$, (i) if $in_1 = \langle \mathsf{store}\ a \rangle$ and $in_2 = \langle \mathsf{load}\ a' \rangle$ such that $a \neq a'$, then $M_{tso}(in_1, in_2) = Y$, (ii) if $in_1 \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $in_2 = \langle \mathsf{load}\ a \rangle$, then $M_{tso}(in_1, in_2) = E$, (iii) else $M_{tso}(in_1, in_2) = N$.

Partial store order

Partial store order (PSO) is similar to TSO, but further relaxes the order of stores. PSO [WG94] is specified by M_{pso}, such that for all memory instructions $in_1, in_2 \in In$, (i) if $in_1 = \langle \mathsf{store}\ a \rangle$ and $in_2 \in \{\langle \mathsf{load}\ a' \rangle, \langle \mathsf{store}\ a' \rangle, \langle \mathsf{cas}\ a' \rangle\}$ such that $a \neq a'$, then $M_{pso}(in_1, in_2) = Y$, (ii) if $in_1 \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $in_2 = \langle \mathsf{load}\ a \rangle$, then $M_{pso}(in_1, in_2) = E$, (iii) else $M_{pso}(in_1, in_2) = N$.

Relaxed memory order

Relaxed memory order (RMO) relaxes the order of instructions even more than PSO. RMO allows to reorder a load with a following load or store to a different address. RMO [WG94] is specified by M_{rmo}, such that for all memory instructions $in_1, in_2 \in In$, (i) if $in_1 \in \{\langle \mathsf{load}\ a\rangle, \langle \mathsf{store}\ a\rangle, \langle \mathsf{cas}\ a\rangle\}$ and $in_2 \in \{\langle \mathsf{load}\ a'\rangle, \langle \mathsf{store}\ a'\rangle, \langle \mathsf{cas}\ a'\rangle\}$ such that $a \neq a'$, then $M_{rmo}(in_1, in_2) = Y$, (ii) if $in_1 \in \{\langle \mathsf{store}\ a\rangle, \langle \mathsf{cas}\ a\rangle\}$ and $in_2 = \langle \mathsf{load}\ a\rangle$, then $M_{rmo}(in_1, in_2) = E$, (iii) else $M_{rmo}(in_1, in_2) = N$. Note that at the level of instruction streams, we do not capture control/data dependence. Rather, we allow RMO to reorder any pair of instructions.

4.1.3 Histories

In order to reason about opacity in TM at hardware level atomicity, the history must contain, apart from the sequence of memory instructions that capture the loads and stores to transactional variables in the program, the following information: (i) when transactions finish (captured with commit and abort instructions), (ii) when a read command finishes (captured with rfin instruction), and (iii) rollback of stores to transactional variables in V (captured with rollback a). The commit and abort instructions are needed to reason about the serialization of transactions. The rfin instruction is needed in formalizing opacity. For example, rfin allows to distinguish the point in time where a variable is loaded from the point where the value loaded is used. The rollback instruction is used in direct update TM to undo a store to a variable. We define $\hat{In} = In_V \cup (\mathsf{rollback} \times V) \cup \{\mathsf{rfin}, \mathsf{commit}, \mathsf{abort}\}$, where $In_V \subseteq In$ is the set of memory instructions to the transactional variables V.

Based on the set $\hat{In}$ of instructions defined above, we can use the framework developed in Chapter 2 to define operations, histories, and transactions.

We characterize a TM by the set of histories (sequences of memory instructions) the TM produces for a given memory model. Formally, a *transactional memory* is a function $\Gamma : \mathbf{M} \rightarrow 2^{\hat{Op}^* \cup \hat{Op}^\omega}$.

4.1.4 Correctness in TM

We present the formalism of safety properties of a TM again, as it has subtle differences from the formalism developed in Chapter 2. A *correctness property* π is a subset of $\hat{Op}^*$. It is natural to require that a TM is correct for *all* programs on a *specific* memory model. This is because a TM may be optimized for performance for a specific memory model, while it could be incorrect on weaker models. That is, different implementation versions may be designed for different memory models. A TM Γ is *correct* for a property π under a memory model M if we have $\Gamma(M) \subseteq \pi$.

We now define opacity for the new fine grained alphabet. At the hardware level of atomicity, we distinguish the point where a variable is loaded from the

point where the read is determined as finished (using the rfin instruction). We thus need to define opacity carefully. Basically, we do not want to say that a history violates opacity if it loads an inconsistent value but does not use it.

4.1.5 Discussion on opacity

For the feasibility of the verification problem, we restrict the notion of opacity with two assumptions. We describe the assumptions and justify them below. Both assumptions restrict the scope of direct update TM allowed by our formalism. However, our assumptions do not restrict the deferred update TM.

Firstly, we assume that if a store of a transactional variable rolls back some time later, then the store should not be observed by a read and should not be overwritten by another store. In other words, we assume that a direct update TM algorithm uses exclusive locks for the variables being written. If a TM does not satisfy this assumption, we cannot verify whether the TM is correct. Moreover, a rollback instruction does not precede a store instruction, as a rollback instruction undoes the effect of a store instruction. Also, by the property of isolation and atomicity, aborted transactions do not change the value of transactional variables. We formalize our assumption as a notion of well-formedness of histories. Given a transactional variable v, we define the *variable projection* $h|_v$ of a history h on v as the longest subsequence of loads, stores, rollbacks, and compare-and-swaps to the variable v. We say that a load of a transaction variable by thread t is *used* in a history h if the load is immediately succeeded by an rfin statement in $h|_t$. Given a history h, we define $\mathit{usedloads}(h)$ as the longest subsequence of h such that all loads of transaction variables in $\mathit{usedloads}(h)$ are used. Given a history h and two transactions x and y in h (possibly of different threads), we say that x *precedes* y in h, written as $x <_h y$, if the last statement of x occurs before the first statement of y in h. A history h is *sequential* if for every pair x, y of transactions in h, either $x <_h y$ or $y <_h x$. We say that a store or a compare-and-swap instruction *in* to variable v in transaction x is *final* in a history h if there does not exist a rollback instruction to v after *in* in x.

We say that a history h is *well-formed* if for all transactions x, (i) if x consists of a rollback instruction *in* to a variable v, then x consists of a store instruction to v before *in*, (ii) if x is an aborted transaction, then x does not consist of any final stores, and (iii) a non-final store to some variable v in x in $h|_v$ is not immediately followed by a compare-and-swap, a used load, or a store to v. This assumption is valid as all direct update TM algorithms we know of rely on locking protocols.

Secondly, we consider a prefix closed subset of opacity as our correctness notion. The justification is that all prefixes of a history produced by a TM should be correct. For example, the history $h = ((\mathsf{read}, v_1), 1), (\mathsf{rfin}, 1), ((\mathsf{read}, v_2), 2), (\mathsf{rfin}, 2), ((\mathsf{write}, v_2), 1), ((\mathsf{write}, v_1), 1)$ is not opaque by the standard definition of opacity. Now let h be suffixed by $((\mathsf{rollback}, v_2), 1)$ $((\mathsf{rollback}, v_1), 2)$ to get history h'. We note that h' is opaque, as all operations of thread 1 may pre-

cede all operations of thread 2. Note that the reason that opacity is not prefix closed are the rollback instructions. Intuitively, a rollback operation may remove conflicts from a conflict graph, and thus remove a cycle which might exist in a prefix of the history. We do not know of a TM algorithm which produces histories like h. All direct update TM we know rely on a locking protocol and hence satisfy the two assumptions. If a TM algorithm does not satisfy one of the two assumptions, we say that the TM algorithm is incorrect.

An operation $op_1 = (in_1, t)$ of transaction x and an operation $op_2 = (in_2, u)$ of transaction y (where x is different from y) *conflict* in a history h if

- in_1 is a load, a final compare-and-swap, or a final store instruction to some transactional variable v and in_2 is a final store instruction to v in h
- in_1 and in_2 are final store instructions to some transactional variable v

A history $h = op_0 \dots op_m$ is *strictly equivalent* to a history h' if (i) for every thread $t \in T$, we have $h|_t = h'|_t$, and (ii) for every pair op_i, op_j of operations in h, if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h', and (iii) for every pair x, y of transactions in h, where x is a finished transaction, if $x <_h y$, then it is not the case that $y <_{h'} x$.

We define *opacity* as the set of all well-formed histories h such that there exists a sequential history h', where h' is strictly equivalent to $\mathit{usedloads}(h)$.

A TM specification for opacity at a coarse-grained alphabet of read, write, commit, and abort instructions was developed in Chapter 2. To verify the TM algorithms at the hardware low-level atomicity, we build TM specifications for opacity with the new alphabet $\hat{In} = In_V \cup (\mathsf{rollback} \times V) \cup \{\mathsf{rfin}, \mathsf{commit}, \mathsf{abort}\}$.

4.2 TM Specifications for Opacity

Developing a TM specification at the hardware level atomicity is challenging due to the following reasons:

- A commit is not atomic in our revised formalism. A commit may consist of multiple store instructions. As soon as a transaction stores, and some other transaction loads the value or overwrites the value, the first transaction cannot abort anymore.
- There is a distinction between the point when a variable is read and when the read is declared as finished. Although a transaction reads an inconsistent value, the history may still be opaque. But, if a transaction finishes the read of an inconsistent value, the history is not opaque.
- Stores may roll back in direct update systems. For example, if a transaction has rolled back its store, then it could appear as if the store was never performed.

As we use the specification for two threads and two variables (owing to the reduction theorem), we develop the TM specification for opacity for two threads. This allows us to keep the specification simple. We first develop a nondeterministic TM specification for opacity. As in Chapter 2, we manually prove the correctness of the nondeterministic TM specification.

4.2.1 A nondeterministic TM specification

We define the *nondeterministic TM specification for opacity Spec* for two threads as the tuple $\langle Q, q_{init}, \delta \rangle$. A state $q \in Q$ is a 10-tuple $\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp \rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{abortsure}, \mathsf{commitsure}\}$ is the status, $SerStatus : T \to \{true, false\}$ is the serialization status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $urs : T \to \{\bot\} \cup V$ is the unfinished read variable, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $wp : T \to \{true, false\}$ is the write predecessor flag, $rp : T \to \{true, false\}$ is the read predecessor flag, and $serp : T \to \{true, false\}$ is the serialization predecessor flag for the threads. The initial state $q_{init} = \langle Status_0, SerStatus_0, rs_0, ws_0, urs_0, prs_0, pws_0, wp_0, rp_0, serp_0 \rangle$, where $Status_0(t) = \mathsf{finished}$, $SerStatus_0(t) = false$, $urs_0(t) = \bot$, $wp_0(t) = rp_0(t) = serp_0(t) = false$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = \emptyset$ for both threads. The transition relation is obtained using Algorithm 4.1. The thread t refers to the thread taking the step, and the thread u refers to the other thread. Given a state q, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $SerStatus(t)$ to $false$, (iii) sets $urs(t)$ to $\bot$, (iv) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, and (v) sets $rp(t)$, $wp(t)$, $rp(u)$, $wp(u)$, and $serp(u)$ to $false$.

Construction

We describe the rules that govern the set of runs that are produced by the nondeterministic TM specification. Let r be a run of the TM specification *Spec*. Let x be the unfinished transaction of a thread, and let y be the unfinished transaction of the other thread in the run r. The nondeterministic TM specification ensures the following:

Rule 1. A variable v is in the prohibited write set of x if there is a committed transaction z in r such that z serializes after x and z has a final store or a finished read of v

Rule 2. A variable v is in the prohibited read set of x if there is a committed transaction z in r such that z serializes after x and z has a final store of v

Rule 3. The serialization status of x is $true$ in a run $r' = r \cdot op$ if

a. the serialization status of x in r is $true$, and op is not a commit or an abort of x, or

$nondetTMSpec(\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op)$

if $op = ((\mathsf{store}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $v \in pws(t)$ **then** return $\bot$
  $ws(t) := ws(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
    if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$
    if $wp(u)$ **then** return $\bot$
    if $Status(u) = \mathsf{finished}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then**
        $rs(u) := rs(u) \cup \{v\}$
  if $v \in rs(u)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
  if $v \in urs(u)$ **then**
    if $serp(u)$ **then**
      $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
    $rp(t) := true$

Algorithm 4.1: The nondeterministic TM specification for fine grained opacity

b. op is a serialize of transaction x

Rule 4. The status of x is commitsure in a run $r' = r \cdot op$ if

a. the status of x is commitsure in r and op is not a commit

b. the status of x is finished in r and op is a store to v by y and x stores to v in r

c. the status of x is finished in r and the status of y is commitsure and x stores to v in r and op is a load of v by y

d. the status of x is finished in r and x stores to v and later y loads v in r and op is a finish of the load of v by y

Rule 5. The status of x is abortsure in a run $r' = r \cdot op$ if the status of x is not commitsure in r and one of the following holds:

a. op is a store of a variable v by y and y serializes before x and x has an unused load of v in r

b. op is a load of a variable v by x such that y stores to v in the run r and y serializes after x

c. op is a rollback of v by y and x loads v after y stores to v in r

d. op is a rollback of v by x and x stores to v in r

nondetTMSpec($\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$, op)

```
if op = ((load, v), t) then
  if Status(t) = abortsure then return ⊥
  if urs(t) ≠⊥ then return ⊥
  if v ∈ prs(t) then
    if Status(t) = commitsure then return ⊥
    Status(t) := abortsure
  urs(t) := v
  if Status(t) = commitsure then rs(t) := rs(t) ∪ {v}
  if v ∈ ws(u) then
    if Status(t) = commitsure then
      if serp(u) then return ⊥ else serp(t) := true
      if Status(u) = abortsure then return ⊥
      if wp(u) then return ⊥
      if Status(u) = finished then
        Status(u) := commitsure
        if urs(u) = v for some variable v ∈ V then
          rs(u) := rs(u) ∪ {v}
    else
      wp(t) := true
      if serp(u) then
        Status(t) := abortsure;    urs(t) :=⊥
```

Continued **Algorithm 4.1**

e. op is a serialize of x and y is unserialized and there exists a variable v such that y stores to v and x loads v after y stores to v

f. op is a load of a variable v by x and v is in the prohibited read set of x

Rule 6. The serialization predecessor *serp* of x is *true* in run $r' = r \cdot op$ if:

a. the serialization predecessor of x is true in r and op is not a commit or an abort of transaction y

b. op is a store of v by transaction x and y stores to v in r

c. op is a store of v by x and y has a used load of v in r

d. op is a store of v by x and the status of y is commitsure and y loads v

e. op is a load of v by x and the status of x is commitsure in r and y stores to v in r

f. op is a serialize of transaction y and the serialization status of x is false

nondetTMSpec$(\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op)$

if $op = (\mathsf{rollback}, v), t)$ **then**
  if $Status(t) = \mathsf{commitsure}$ **then** return $\bot$
  if $v \notin ws(t)$ **then** return $\bot$
  if $wp(u)$ **then**
    $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  $ws(t) := ws(t) \setminus \{v\}$
  $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$

if $op = (\mathsf{rfin}, t)$ **then**
  if $urs(t) = \bot$ **then** return $\bot$ **else** $v := urs(t)$
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) := \bot$
  if $wp(t)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
    if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$
    if $wp(u)$ **then** return $\bot$
    if $Status(u) = \mathsf{finished}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then**
        $rs(u) := rs(u) \cup \{v\}$
  if $rp(u)$ **then**
    if $serp(t)$ **then** return $\bot$ **else** $serp(u) := true$

Continued **Algorithm 4.1**

g. *op* is a finish of a read by transaction x and y stores to v in r, and later x loads v in r

h. *op* is a finish of a read by transaction y and y loads v in r, and later x stores to v in r

Rule 7. The serialization predecessor of the transaction following x in the thread of x is *true* in a run $r' = r \cdot op$ if *op* is a commit or abort of x and the serialization status of y is *true*

Rule 8. Given a run r produced by *Spec* and an operation *op* of transaction x, the run $r' = r \cdot op$ is produced by *Spec* if the following hold:

a. if the status of x is abortsure, then *op* is an abort, a rollback, or a serialize

b. if *op* is a store of v, then x has no unused load in r and v is not in the prohibited write set of x

nondetTMSpec($\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op$)

if $op = (\varepsilon, t)$ **then**
  if $SerStatus(t) = true$ **then** return $\perp$
  $SerStatus(t) := true$
  if $SerStatus(u) = false$ **then**
    if $serp(t)$ **then** return $\perp$
    $serp(u) := true$
    if $wp(t) = true$ **then**
      $Status(t) :=$ abortsure; $urs(t) := \perp$

if $op = (\mathsf{commit}, t)$ **then**
  if $SerStatus(t) \neq true$ **then** return $\perp$
  if $Status(t) =$ abortsure **then** return $\perp$
  if $urs(t) \neq \perp$ **then** return $\perp$
  if $rp(t)$ **then**
    if $serp(u) = true$ **then**
      $Status(u) :=$ abortsure; $urs(u) := \perp$
  if $serp(t)$ **then**
    $prs(u) := prs(u) \cup ws(t) \cup prs(t)$
    $pws(u) := pws(u) \cup ws(t) \cup rs(t) \cup pws(t)$
  $ResetState(t)$
  if $SerStatus(u)$ **then** $serp(t) := true$

if $op = (\mathsf{abort}, t)$ **then**
  if $SerStatus(t) = false$ **then** return $\perp$
  if $ws(t) \neq \emptyset$ **then** return $\perp$
  $ResetState(t)$
  if $SerStatus(u) = true$ **then** $serp(t) := true$
return $\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$

Continued **Algorithm 4.1**

c. if op is a rollback of v, then the status of x is not commitsure and x stores to v in r

d. if op is a load of v, then x has no unused load in r

e. if op is a load of v and v is in the prohibited read set of x, then status of x is not commitsure

f. if op is a finish of a read, then there is an unused load by x and the status of x is not abortsure in r

g. if op is a commit, then the serialization status of x is true and all loads by x in r are used

h. if op is an abort, then there does not exist a variable v such that x stores to v and x does not rollback v in r

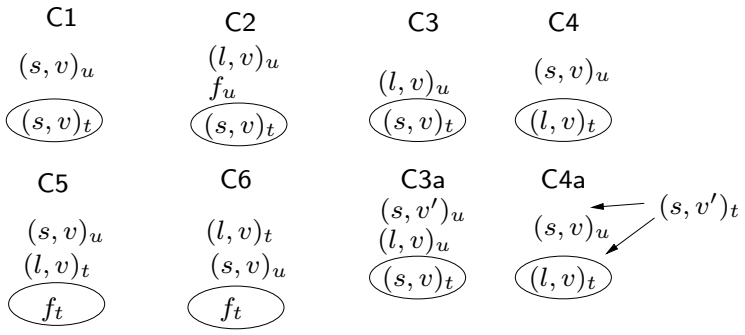

Figure 4.1: The operations inside ovals are disallowed by the TM specification for opacity. An arrow represents different possible positions for a command to occur in a given condition. We write s for store, l for load, c for commit, b for rollback, and f for rfin. We write the operation $((s,v),t)$ as $(s,v)_t$. Thread t executes transaction x and thread u executes transaction y. For conditions C1-C5, the transaction y must serialize before x. For condition C6, the transaction x must serialize before y. For condition C3, y must commit in every extension of the run. For condition C4, x must commit in every extension of the run. Conditions C3a and C4a pertain to the nondeterministic TM specification for opacity without rollbacks, as described in the appendix.

i. if *op* is an abort, then the serialization status of x is true

j. if *op* is a serialize, then the serialization status of x is false

k. if the serialization predecessor of x is true, then the serialization predecessor of y is false in r'

Using the above rules of construction, we now prove the correctness of the nondeterministic TM specification for opacity.

Correctness

Theorem 4.1. *Given a history h on 2 threads and k variables, h is opaque if and only if $h \in L(Spec)$.*

Proof. We say that a transaction x must serialize before a transaction y in a run r if one of the following holds:

- x and y have final stores to a variable v and y stores to v after x stores to v
- the serialize of x occurs before the serialize of y in r
- x stores to v and y has a used load of v, where y loads v after x stores to v
- x has a used load of v and y stores v, where x loads v before y stores to v

Note that from rule 4, 8.c, and 8.h, a transaction x that stores to a variable v must commit in every extension of r if one of the following holds:

- there exists a transaction y such that y stores to v after x stores to v and before x rolls back
- there exists a transaction y such that y loads v after x stores to v, and the read of v is finished by y
- there exists a transaction y such that y must commit, and y loads v after x stores to v

Note that for two unfinished transactions x, y in a run r, if y must serialize before x, then the serialization predecessor of x is true in r.

Now, we note that the TM specification *Spec* for opacity gives the largest set R of runs such that for every run r produced by the TM specification, for every transaction x in r, the following conditions hold (conditions C1-C6 are graphically shown in Figure 4.1):

C1. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y stores to v and y does not rollback its store to v (from rules 1, 6.b, and 8.k)

C2. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a used load of v (from rules 1, 6.c, and 8.k)

C3. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a load of v and y must commit in every extension of r (from rules 1, 6.d, and 8.k)

C4. x does not load a variable v if there exists a transaction y such that y must serialize after x and y stores to v and x must serialize in every extension of r (from rules 2, 6.e, 8.e, and 8.k)

C5. x does not finish the read of a variable v if there exists a transaction y such that y must serialize after x and y stores to v before x loads v (from rules 2, 5.b, 5.f, 6.g, 8.f, and 8.k)

C6. x does not finish the read of a variable v if there exists a transaction y such that y must serialize before x and y stores to v after x loads v (from rules 5.a, 6.h, and 8.f)

C7. x serializes at most once (from rules 3 and 8.j)

C8. if x is a finished transaction, then x serializes exactly once (from rules 3, 8.g, 8.i, and 8.j)

C9. x contains a rollback of v only if the transaction consists of a store to v before the rollback (from rule 8.c)

C10. after a rollback of a variable v in x, the only possible instruction in x is a rollback of another variable or a serialize or an abort (from rules 5.d and 8.a)

C11. if x is an aborting transaction, then every aborted transaction rollbacks all the stores before aborting (from rule 8.h)

C12. x does not serialize if there exists a transaction y such that y is unserialized and y must serialize before x (from rules 6.f and 8.k)

Let h be an opaque history on 2 threads and k variables. As h is opaque, there is a sequential history h_s such that h_s is strictly equivalent to h. Let the transactions in the sequential history h_s be given by the sequence $x_1 \dots x_n$ of transactions. We claim that there exists a run r of the TM specification *Spec* such that h is the corresponding history of r. As h_s is strictly equivalent to h, we know that for every pair x_i, x_j of transactions in h such that $i < j$, the following are not true:

- x_i loads v after a final store to v by x_j, and x_i finishes the read
- x_i is a committing transaction and x_i loads v after a final store by x_j to v
- x_i and x_j have a final store to v and x_i stores to v after x_j stores to v
- x_i stores to v and x_j loads v before the store to v, and the read of v is finished by x_j

These conditions are equivalent to the conditions C1-C6. Moreover, h is well-formed. This is equivalent to the conditions C9-C11. Thus, we know that there exists a run r of the TM specification *Spec*, where the order of serialization of transactions is the same as $x_1 \dots x_n$.

Conversely, let r be a run produced by the nondeterministic TM specification *Spec*. Let h be the corresponding history to the run r. We know from conditions C7 and C8 that every transaction serializes at most once in the run, and every finished transaction serializes exactly once in the run. Let h_s be a sequential history such that (i) a transaction x appears before a transaction y in h_s if x must serialize before y in r, (ii) all other transactions appear in an arbitrary order later in r, and (iii) for all threads, the thread projection of h_s is equivalent to the thread projection of h. The conditions C1 - C6 guarantee that for every pair op_i, op_j of operations in h if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h_s. Note that the order of serialization in r respects the real time order of the transactions in h, that is, if a transaction x finishes before a transaction y starts, then x serializes before y in r. Thus, h_s is strictly equivalent to h. Hence, every history in $L(Spec)$ is opaque.

□

$detTMSpec(\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op)$

if $op = ((\mathsf{store}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $v \in pws(t)$ **then return** $\bot$
  $ws(t) := ws(t) \cup \{v\}$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then** $hp(t) := true$
    if $Status(u) = \mathsf{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \mathsf{started}$
  if $v \in ws(u)$ **then**
    if $wp(u)$ **or** $hp(u)$ **then** return $\bot$
    if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$
    if $Status(u) = \mathsf{commitimpossible}$ **then** return $\bot$
    if $Status(u) = \mathsf{pending}$ **then**
      $Status(u) := \mathsf{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    if $Status(u) = \mathsf{started}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    $sp(t) := true$
  if $v \in rs(u)$ **then**
    $sp(t) := true$
    if $wp(u)$ **then** $Status(u) := \mathsf{abortsure}$; $urs(u) := \emptyset$
  if $v \in urs(u)$ **then**
    $rp(t) := true$
    if $sp(u)$ **or** $hp(u)$ **or** $wp(u)$ **then**
      $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

Algorithm 4.2: The deterministic TM specification for fine grained opacity

4.2.2 A deterministic TM specification

We define the *deterministic TM specification for opacity* $Spec^d$ for two threads as the tuple $\langle Q, q_{init}, \delta\rangle$. A state $q \in Q$ is a 10-tuple $\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{pending}, \mathsf{commitsurepending}, \mathsf{abortsure}, \mathsf{commitsure}\}$ is the status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $urs : T \to \{\bot\} \cup V$ is the unfinished read set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $hp : T \to \{true, false\}$ is the hidden predecessor

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$, op)

if $op = (\mathsf{rollback}, v), t)$ **then**
  if $Status(t) \in \{\mathsf{commitsure}, \mathsf{commitsurepending}\}$ **then** return $\bot$
  if $v \notin ws(t)$ **then** return $\bot$
  if $wp(u)$ **then**
    $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  $ws(t) := ws(t) \setminus \{v\}$
  $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$

if $op = (\mathsf{rfin}, t)$ **then**
  if $urs(t) = \bot$ **then** return $\bot$
  $v := urs(t)$
  if $v \in prs(t)$ **and** $Status(t) = \mathsf{commitimpossible}$ **then** return $\bot$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) = \bot$
  if $rp(u)$ **then** $sp(u) := true$
  if $Status(t) = \mathsf{pending}$ **or** $Status(t) = \mathsf{commitsurepending}$ **then**
    $sp(u) := true$
  if $wp(t)$ **then**
    if $Status(u) \in \{\mathsf{abortsure}, \mathsf{commitimpossible}\}$ **then** return $\bot$
    if $Status(u) = \mathsf{pending}$ **then**
      $Status(u) := \mathsf{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    if $Status(u) = \mathsf{started}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    $sp(t) := true$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

continued **Algorithm 4.2**

(to due a transaction already committed) flag, $wp : T \to \{true, false\}$ is the write predecessor flag, $rp : T \to \{true, false\}$ is the read predecessor flag, and $sp : T \to \{true, false\}$ is the strong predecessor flag for the threads. The initial state $q_{init} = \langle Status_0, rs_0, ws_0, urs_0, prs_0, pws_0, hp_0, wp_0, rp_0, sp_0\rangle$, where $Status_0(t) = \mathsf{finished}$, $urs_0(t) = \bot$, $hp_0(t) = wp_0(t) = rp_0(t) = sp_0(t) = false$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = \emptyset$ for both threads. The transition relation of the deterministic TM specification is obtained using Algorithm 4.2. Given a state q, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $urs(t)$ to $\bot$, (iii) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, (iv) sets $hp(t)$, $rp(t)$, $wp(t)$, $hp(u)$, $rp(u)$, $wp(u)$, and $sp(u)$

$detTMSpec(\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op)$

if $op = ((\mathsf{load}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then** $hp(t) := true$
    if $Status(u) = \mathsf{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \mathsf{started}$
  if $v \in prs(t)$ **then**
    if $Status(t) \in \{\mathsf{commitsure}, \mathsf{commitsurepending}\}$ **then** return $\bot$
    $Status(t) := \mathsf{commitimpossible}$
  $urs(t) := v$
  if $Status(t) \in \{\mathsf{commitsure}, \mathsf{commitsurepending}\}$ **then** $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $Status(t) \in \{\mathsf{commitsure}, \mathsf{commitsurepending}\}$ **then**
      $rs(t) := rs(t) \cup \{v\}$
      if $Status(u) \in \{\mathsf{abortsure}, \mathsf{commitimpossible}\}$ **then** return $\bot$
    if $Status(u) = \mathsf{pending}$ **then**
      $Status(u) := \mathsf{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    if $Status(u) = \mathsf{started}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    $sp(t) := true$
    else
      $wp(t) := true$
      if $sp(u)$ **then**
        $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

continued **Algorithm 4.2**

to *false*. As in the nondeterministic TM specification, t refers to the thread taking the step, and u refers to the other thread.

Correctness

We use an antichain based tool [WDHR06] to prove that for two threads and two variables, the language of the deterministic TM specification is equivalent to the language of the nondeterministic TM specification. The nondeterministic TM specification has 155'000 states, while the deterministic TM specifi-

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$, op)

if $op = (\mathsf{commit}, t)$ **then**
  if $Status(t) \in \{\mathsf{abortsure}, \mathsf{commitimpossible}\}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $hp(t)$ **then**
    if $sp(u)$ **then**
      if $urs(u) \neq \bot$ **then**
        $Status(u) := \mathsf{abortsure}$; $urs(u) := \emptyset$
      else $Status(u) = \mathsf{commitimpossible}$
  if $hp(t)$ **or** $rp(t)$ **or** $sp(t)$ **then**
    if $Status(u) = \mathsf{started}$ **then** $Status(u) = \mathsf{pending}$
    if $Status(u) = \mathsf{commitsure}$ **then** $Status(u) = \mathsf{commitsurepending}$
    $prs(u) := prs(u) \cup ws(t) \cup prs(t)$
    $pws(u) := pws(u) \cup ws(t) \cup rs(t) \cup pws(t)$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$
  ResetState(t)

if $op = (\mathsf{abort}, t)$ **then**
  if $ws(t) \neq \emptyset$ **then** return $\bot$
  ResetState(t)
return $\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$

continued **Algorithm 4.2**

cation has 46'000 states. The execution time for checking equivalence of the two specifications using the antichain based tool [WDHR06] on an Opteron machine with 2.66 GHz processors and 16 GB RAM is around two hours. This high execution time is mostly due to the high memory consumption. The check consumes around 15 GB RAM, and thus most of the time is spent in swapping memory. The process execution time is around ten minutes.

The manual proof for the intuitive nondeterministic TM specification and the automated language equivalence check with the deterministic TM specification allow us to claim that the deterministic TM specification accepts exactly the set of opaque histories.

4.3 Relaxed Memory Language

We introduce a high-level language, RML, to express TM algorithms with hardware-level atomicity on relaxed memory models. The key idea behind the design of RML is to have a semantics parametrized by the underlying memory model. To capture a relaxed memory model, RML defers a statement until the statement is forced to execute due to a fence, and RML reorders or eliminates

$$
\begin{array}{lll}
l & ::= & lv \mid la[idx] \\
g & ::= & gv \mid ga[idx] \\
e & ::= & f(l, \ldots, l, idx, \ldots, idx) \\
c & ::= & f(idx, \ldots, idx) \\
tm\_stmt & ::= & \mathsf{rfin} \mid \mathsf{commit} \mid \mathsf{abort} \\
mem\_stmt & ::= & g := e \mid l := g \mid l := e \mid idx := c \mid l := \mathsf{cas}(g, e, e) \\
 & & \mid \mathsf{rollback}\ g := e \\
fence & ::= & \mathsf{stfence} \mid \mathsf{ldfence} \\
p & ::= & mem\_stmt \mid tm\_stmt \mid fence \mid p\,;p \\
 & & \mid \textbf{if}\ e\ \textbf{then}\ p\ \textbf{else}\ p \mid \textbf{while}\ e\ \textbf{do}\ p
\end{array}
$$

Figure 4.2: The syntax of the language RML

deferred statements according to the memory model. We describe below the syntax and semantics of RML.

4.3.1 Syntax

To describe TM algorithms in RML, we use local and global integer-valued locations, which are either variables or arrays. We also have a set of array index variables. The syntax of RML is given in Figure 4.2. A memory statement (denoted by *mem_stmt*) in RML models an instruction that executes atomically on the hardware. It can, for instance, be a store or a load of a global variable. Moreover, the TM specific statements are denoted by *tm_stmt*, and fence statements are denoted by *fence*. Let S_M be the set of memory statements, S_{tm} be the set of TM specific statements, and S_F be the set of fence statements in RML. Let P be the set of RML programs.

4.3.2 Semantics

Intuitively, capturing a relaxed memory model requires us to defer statements across following statements, unless the memory model guarantees an ordering. So, RML maintains as part of the state, a queue of statements whose execution has been deferred. When a statement with a memory instruction is encountered, RML inserts the statement in the queue of deferred statements. However, the relaxations allowed by the memory model allow to insert the statement at multiple places in the queue. Thus, we obtain multiple transitions from the original state on a statement with a memory instruction, where each destination state differs only in the queue of deferred statements. When RML encounters a store (resp. load) fence, RML dequeues statements for execution, until the queue has no store (resp. load) instructions. We now formalize the semantics of RML.

Let G and L be the set of global and local addresses respectively. Let $Idx \subseteq L$ be the set of local index addresses. Consider a particular thread t. Let $\sigma : G \cup L \to \mathbb{N}$ be a *valuation* of the global addresses, and the local

Table 4.1: The formal definitions of the functions γ, lw, and lr for a statement s in a valuation σ

Statement s	$\gamma(s,\sigma)$	$lw(s,\sigma)$	$lr(s,\sigma)$
$g := e$	$\langle$store $[\![g]\!]_\sigma\rangle$	$\emptyset$	$lvars(e,\sigma)$
$l := g$	$\langle$load $[\![g]\!]_\sigma\rangle$	$\{[\![l]\!]_\sigma\}$	$\emptyset$
$l := e$	skip	$\{[\![l]\!]_\sigma\}$	$lvars(e,\sigma)$
$l :=$ cas(g, e_1, e_2)	$\langle$cas $[\![g]\!]_\sigma\rangle$	$\{[\![l]\!]_\sigma\}$	$lvars(e_1,\sigma) \cup lvars(e_2,\sigma)$
rollback $g := e$	$\langle$store $[\![g]\!]_\sigma\rangle$	$\emptyset$	$lvars(e,\sigma)$
$idx := c$	skip	$\{[\![idx]\!]_\sigma\}$	$lvars(c,\sigma)$

addresses of thread t. Let Σ be the set of all valuations. Note that the syntax of RML is defined in a way that the value of an index variable idx may not depend on the queue of deferred statements. Given a global location g and a valuation σ, we write $[\![g]\!]_\sigma \in G$ to denote the global address represented by g in valuation σ. Similarly, we write $[\![l]\!]_\sigma \in L \setminus Idx$ (resp. $[\![idx]\!]_\sigma \in Idx$) to denote the local address (resp. local index address) represented by a local location l (resp. index variable idx) in valuation σ.

Let $\gamma : S_M \times \Sigma \to In \cup \{\text{skip}\}$ be a mapping function for memory statements, which for a given memory statement and a valuation, gives the generated hardware instruction or the skip instruction if no hardware instruction is generated. For example, we have $\gamma(g := e, \sigma) = \langle$store $[\![g]\!]_\sigma\rangle$ in valuation σ, as the statement $g := e$ causes a store to the global address represented by g in valuation σ. The statement rollback $g := e$ is physically a store instruction, as a rollback undoes the effect of a previous store instruction. We define a *local-variables* function $lvars$ such that given an expression e and a valuation σ, we have $lvars(e,\sigma)$ as the smallest set of local addresses in L such that if the location l (resp. index variable idx) appears in e, then the address $[\![l]\!]_\sigma$ is in $lvars(e,\sigma)$ (resp. $[\![idx]\!]_\sigma$ is in $lvars(e,\sigma)$). We define a *write-locals* function $lw : S_M \times \Sigma \to 2^L$ and a *read-locals* function $lr : S_M \times \Sigma \to 2^L$ to obtain the written and read local addresses in a statement respectively. Table 4.1 gives the formal definitions of the functions γ, lw, and lr.

We now describe when two memory statements can be reordered in a given valuation under a given memory model. Let $\mathbf{M}$ be the set of all memory models. Let $R : S_M \times S_M \times \Sigma \times \mathbf{M} \to \{true, false\}$ be a *reordering* function such that $R(s_1, s_2, \sigma, M) = true$ if the following conditions hold: (i) either $\gamma(s_1,\sigma) =$ skip or $\gamma(s_2,\sigma) =$ skip, or $M(\gamma(s_1,\sigma), \gamma(s_2,\sigma)) = Y$, (ii) $lw(s_1,\sigma) \cap lr(s_2,\sigma) = \emptyset$, (iii) $lw(s_1,\sigma) \cap lw(s_2,\sigma) = \emptyset$, and (iv) $lr(s_1,\sigma) \cap lw(s_2,\sigma) = \emptyset$. Here, the first condition restricts reorderings to those allowed by the memory model, and the remaining conditions check for data dependence between the statements. To defer memory statements and execute them in as many ways as possible, we define a model-dependent enqueue function. This function takes as input the current valuation, the current sequence of deferred statements, a statement to defer, and a memory model, and produces the set of new possible sequences of deferred statements. We define the *enqueue* function $Enq : S_M^* \times S_M \times \Sigma \times \mathbf{M} \to$

$2^{S_M^*}$ such that given a sequence $d = s_1 \dots s_n$ of memory statements, a statement s, a valuation σ, and a memory model M, the function $Enq(d, s, \sigma, M)$ is the largest set such that (i) $s_1 \dots s_k \cdot s \cdot s_{k+1} \dots s_n \in Enq(d, s, \sigma, M)$ if for all i such that $k < i \leq n$, we have $R(s_i, s, \sigma, M) = true$, and (ii) if s is of the form $l := g$, then $s_1 \dots s_k \cdot (l := e) \cdot s_{k+1} \dots s_n \in Enq(d, s, \sigma, M)$ if for all i with $k < i \leq n$, we have $R(s_i, s, \sigma, M) = true$, and $M(\gamma(s_k, \sigma), \gamma(s, \sigma)) = E$ where (a) if s_k is $g := f$, then $e = f$, (b) if s_k is $m := g$ or $m := \mathsf{cas}(g, e_1, e_2)$, then $e = m$. Note that the definition of the reordering function restricts the reordering of control and data-dependent statements. Thus, our model of RMO slightly differs from the definition of the RMO model in the sense that we impose an order on control dependent load instructions. Similarly, the enqueue function restricts the elimination of only load instructions. While this is sufficient to model many hardware memory models, we cannot capture coalesced stores or redundant store elimination.

Given a valuation σ, a program p, and a sequence d of deferred statements, we define a predicate $allowDequeue(\sigma, d, p)$ to be $true$ if (i) p is of the form ($\mathbf{while}\ \ e\ \mathbf{do}\ \ p_1; p'$) or ($\mathbf{if}\ \ e\ \mathbf{then}\ \ p_1\ \mathbf{else}\ \ p_2; p'$) for some programs $p_1, p_2, p' \in P$, and there exists a memory statement s in d such that $lw(s, \sigma) \cap lvars(e, \sigma) \neq \emptyset$, or (ii) p is a store fence and there exists a statement s of the form $g := l$ or $l := \mathsf{cas}(g, e, e)$ in d, or (iii) p is a load fence and there exists a statement s of the form $l := g$ or $l := \mathsf{cas}(g, e, e)$ in d.

Conditionals and loops

When an RML program reaches a condition or a loop, it requires the condition to be evaluated. RML first checks whether the local variables appearing in the condition are modified in any deferred statement in the queue. If this is not the case, the execution is governed by the following rules.

$$\frac{\sigma[e] \neq 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \mathbf{if}\ \ e\ \mathbf{then}\ \ p_1\ \mathbf{else}\ \ p_2;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p',\ \sigma,\ d\rangle} \quad \text{(IF TRUE)}$$

$$\frac{\sigma[e] = 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \mathbf{if}\ \ e\ \mathbf{then}\ \ p_1\ \mathbf{else}\ \ p_2;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_2; p,\ \sigma,\ d\rangle} \quad \text{(IF FALSE)}$$

$$\frac{\sigma[e] \neq 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \mathbf{while}\ \ e\ \mathbf{do}\ \ p_1;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p,\ \sigma,\ d\rangle} \quad \text{(WHILE TRUE)}$$

$$\frac{\sigma[e] = 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \textbf{while}\ \ e\ \textbf{do}\ \ p_1;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p',\ \sigma,\ d\rangle} \qquad \text{(WHILE FALSE)}$$

Index variable update

As the value of an index variable does not depend on the deferred statements, it is straightforward to modify the valuation according to the variable update.

$$\frac{}{\langle idx := c; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma[idx/c],\ d\rangle} \qquad \text{(INDEX)}$$

Fences

When an RML program encounters a store (resp. load) fence, we ensure that there is no store or cas (resp. load or cas) instruction in the queue of deferred statements.

$$\frac{allowDequeue(\sigma, d, \mathsf{stfence}) = false}{\langle \mathsf{stfence}; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d\rangle} \qquad \text{(STORE FENCE)}$$

$$\frac{allowDequeue(\sigma, d, \mathsf{ldfence}) = false}{\langle \mathsf{ldfence}; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d\rangle} \qquad \text{(LOAD FENCE)}$$

TM specific

A commit and an abort instruction behave like store fences. This is to avoid instructions of two transactions from the same thread to interleave with each other. Moreover, an rfin instruction behaves like a load fence. This ensures that during the transactional read of a variable, the variable is loaded before the read is declared as finished.

$$\frac{s \in \{\mathsf{commit}, \mathsf{abort}\} \quad allowDequeue(\sigma, d, \mathsf{stfence}) = false}{\langle s; p,\ \sigma,\ d\rangle \xrightarrow{s} \langle p,\ \sigma,\ d\rangle} \qquad \text{(TRANSACTION END)}$$

$$\frac{allowDequeue(\sigma, d, \mathsf{ldfence}) = false}{\langle \mathsf{rfin}; p,\ \sigma,\ d\rangle \xrightarrow{\mathsf{rfin}} \langle p,\ \sigma,\ d\rangle} \qquad \text{(READ FINISH)}$$

Enqueue

When RML encounters a memory instruction in the form of a load, store, compare-and-swap, or rollback instruction, RML enqueues the statement into the queue of deferred statements. Then, RML nondeterministically shuffles the statement in the queue according to the relaxations allowed by the underlying memory model M.

$$\frac{d' \in Enq(d, s, \sigma, M) \quad s \in \{g := e, l := g, l := e, l := \mathsf{cas}(g, e_1, e_2), \mathsf{rollback}\ g := e\}}{\langle s; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d'\rangle} \quad \text{(ENQUEUE)}$$

Dequeue

A statement is executed by RML only under the circumstances that the value of the conditional variable depends on the contents on the queue, or RML reaches a store (resp. load) fence, and there is a store (resp. load) instruction in the queue. In this case, the first statement in the queue is dequeued, and the effect of the statement is made to the global and local variables as required.

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[g] = c \quad d = (l := g) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{load}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[l/c],\ d'\rangle} \quad \text{(DEQUEUE LOAD)}$$

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[e] = c \quad d = (g := e) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{store}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[g/c],\ d'\rangle} \quad \text{(DEQUEUE STORE)}$$

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[g] = c \quad \sigma[e_1] \neq c \quad d = (l := \mathsf{cas}(g, e_1, e_2)) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{load}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[l/c],\ d'\rangle}$$

(DEQUEUE CAS FAILURE)

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[g] = \sigma[e_1] = c \quad \sigma[e_2] = c' \quad d = (l := \mathsf{cas}(g, e_1, e_2) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{cas}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[g/c'][l/c],\ d'\rangle}$$

(DEQUEUE CAS SUCCESS)

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[e] = c \quad d = (l := e) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p,\ \sigma[l/c],\ d'\rangle} \quad \text{(DEQUEUE LOCAL)}$$

```
initially, X1 := 0, X2 := 0, Y1 := 0, Y2 := 0
          r1 := 0, r2 := 0, r3 := 0, r4 := 0
if id = 0                          if id = 1
    X1 := 0                            X2 := 0
    Y1 := 0                            Y2 := 0
    r1 := Y2          ||               r2 := Y1
    r3 := X2                           r4 := X1
    X1 := 2                            X2 := 2
```

Figure 4.3: An example of an RML program

$$\frac{\begin{array}{c} allowDequeue(\sigma, d, p_1) = true \\ \sigma[e] = c \qquad d = (\mathsf{rollback}\ g := e) \cdot d' \end{array}}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\mathsf{rollback}[\![g]\!]_\sigma} \langle p_1; p,\ \sigma[g/c],\ d'\rangle} \qquad \text{(DEQUEUE ROLLBACK)}$$

4.3.3 Example execution in RML

Consider the RML program shown in Figure 4.3. This example was presented in the introduction to relaxed memory models in Chapter 1. We discuss the operation of RML under different memory models.

- *Sequential consistency* does not allow any reorderings. The example generates 247 states[1].

  There are seven possible valuations for the tuple ⟨ r1, r2, r3, r4 ⟩ of variables: $\langle 1,0,2,0\rangle$, $\langle 1,0,1,0\rangle$, $\langle 1,1,2,1\rangle$, $\langle 1,1,1,1\rangle$, $\langle 1,1,1,2\rangle$, $\langle 0,1,0,1\rangle$, $\langle 0,1,0,2\rangle$.

- *Total store order* allows to reorder a store followed by a read of a different variable, and also allows to eliminate a read of a variable following a store to the same variable. The example generates 943 states. Apart from the valuations allowed in sequential consistency, TSO allows one more valuation for ⟨ r1, r2, r3, r4 ⟩: $\langle 0,0,0,0\rangle$.

- *Partial store order* further allows to reorder stores of different variables. The example generates 3382 states. Apart from the valuations allowed in TSO, PSO allows seven more valuation for ⟨ r1, r2, r3, r4 ⟩: $\langle 1,0,0,0\rangle$, $\langle 1,1,0,1\rangle$, $\langle 1,1,0,2\rangle$, $\langle 1,1,2,0\rangle$, $\langle 1,1,1,0\rangle$, $\langle 1,1,0,0\rangle$, $\langle 0,1,0,0\rangle$.

- *Relaxed memory order* further allows to reorder loads and a load followed by a store to a different variable. The example generates 26596 states. Apart from the valuations allowed in PSO, RMO allows one more valuation: $\langle 1,1,2,2\rangle$.

[1] In this example, we define a state as a valuation of the global and local variables, the program counters, and the deferred statements for each thread.

We indeed produce these outcomes using our automated tool FOIL which we present in the next chapter. Some of the outcomes in the above examples are hard to reason about manually. That shows the importance of automated tools to reason about outcomes under relaxed memory models.

4.4 TM Algorithms in RML

Similar to the way we expressed TM algorithms in our language Simple, we now express TM algorithms in RML. A state of a TM algorithm now captures, apart from the global and local valuations, the program counters, and the transactional program counters, the set of deferred statements per thread. So, a state z of the TM algorithm now becomes a 5-tuple $\langle \sigma_G, \sigma_L, pc, txpc, D \rangle$, where $D(t)$ is the set of deferred statements of thread t in the state z. We can now extend the formalism developed in Section 2.4.2 to describe a run and a corresponding history. However, as the semantics of RML are parametrized by a memory model, the set of runs produced by a TM algorithm depends on the underlying memory model. Hence, the language of the TM algorithm depends on the memory model. The *language* $L(A, M)$ of a TM algorithm A under a memory model M is the set of histories h such that h is the corresponding history of a run of A under the memory model M. A TM algorithm A is safe for property π under a memory model M if every history in the language of A under M is included in π.

We now describe three TM algorithms, DSTM, TL2, and McRT-STM in RML. We present the RML programs p_r, p_w, and p_e for each TM algorithm. We also give the initial valuation of the variables of the TM algorithm. We use the notation `owner`[V] to denote that `owner` is an array of size V. All TM algorithms also consist of a program p_a which corresponds to the abort of the transaction.

TL2

Algorithm 4.3 shows four RML programs: p_r (read), p_w (write), p_e (end), and p_a (abort). The program p_a can be called from within p_r, p_w, and p_e. The global variables are `lock`[V], `version`[V], `g`[V], and `clk`. The local variables are `rs`[V], `ws`[V], `lver`[V], `lclock`, `c`, and `l`. The index variables are `u` and `v`. `self` denotes the thread number of the executing thread. The initial valuation of all variables is 0. The array `g`[V] of global variables corresponds to the addresses of the transactional variables V.

Comparing against the description of TL2 in the language Simple, we find that the program p_e requires many atomic steps to complete, which increases the possible interleavings, and thus makes verification more challenging.

```
01  program p_a :
02  u := 0
03  while u < V do
04    u := u + 1;
05    if owner[u] = self then
06      owner[u] := 0;
07    rs[u] := 0; ws[u] := 0
08  lclock := 0
09  abort
```

```
01  program p_w :
02  if lclock = 0 then
03    lclock := clk;
04  ws[v] := 1;
```

```
01  program p_r :
02  if lclock = 0 then
03    lclock := clk;
04  if ws[v] = 0 then
05    l := owner[v];
06    if l ≠ 0 then p_a
07    l := g[v];
08    lver[v] := version[v];
09    if lclock ≠ lver[v] then p_a
10    rs[v] := 1;
11    rfin
```

```
01  program p_e :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    if (ws[u] = 1) then
06      l := cas(owner[u], 0, self);
07      if l ≠ self then p_a
08  l := 0;
09  while l ≠ lclock + 1 do
10    lclock := clk
11    l := cas(clk, lclock, lclock + 1);
12  u := 0;
13  while u < V do
14    u := u + 1;
15    if rs[u] = 1 then
16      rs[u] := 0;
17      l := owner[u];
18      c := version[u];
19      if c ≠ lver[u] then p_a
20      if l ≠ 0 then p_a
21  u := 0;
22  while u < V do
23    u := u + 1;
24    if ws[u] = 1 then
25      version[u] := lclock;
26      g[u] := l;
27  u := 0;
28  while u < V do
29    u := u + 1;
30    if ws[u] = 1 then
31      owner[u] := 0;
32      ws[u] := 0;
33  lclock := 0;
34  commit
```

Algorithm 4.3: The TL2 algorithm in RML

DSTM

The global variables are `txr[V]`. The local variables are `rs[V]`, `v`, `k`, `l`, `m`, `t`, and `u`. All variables are initialized to 0. DSTM consists of structures which can be implemented using a sequence of addresses. We use these structures and slightly modify the DSTM algorithm to avoid dynamic memory allocation. We express the programs p_r, p_w, and p_e of DSTM in RML in Algorithm 4.4.

```
01  program p_a :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := cas(txr[v].tid, self, aborTx)
06  abort
```

```
01  program p_r :
02  p_w
03  rfin
```

```
01  program p_e :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := cas(txr.tid[v], self, commTx)
06  commit
```

```
01  program p_w :
02  k := 0
03  l := txr[v];   t := l.tid
04  if t ≠ self then
05    while k ≠ l do
06      k := 0
07      m.tid := self
08      if t.status = 1 then
09        m.oldv := l.newv
10        m.newv := l.newv
11      if t.status = 2 then
12        m.oldv := l.oldv
13        m.newv := l.oldv
14      if t.status = 0 then
15        k := cas(t.status, 0, 2)
16        if k = 0 then
17          m.oldv := l.oldv
18          m.newv := l.oldv
19          k := cas(txr[v], l, m)
```

Algorithm 4.4: The DSTM algorithm in RML

McRT STM

McRT STM [SATH+06] is an STM proposal from Intel. It significantly differs from the previous two TM we discussed, due to the fact that McRT STM is a direct update TM. In fact, McRT STM is similar to the two phase locking TM presented in Chapter 2. The difference lies in the time of when the stores are made. While the two phase locking TM updated the global memory on a commit, McRT STM updates the global memory during the write command. The global variables are `owner`[V] and `g`[V]. The local variables are `rs`[V], `l`, `m`, `u`, and `v`. All variables are initialized to 0. The array `g`[V] of global variables corresponds to the addresses of the transactional variables V. McRT STM is presented in RML in Algorithm 4.5.

```
01  program p_a :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := owner[u];
06    if l = self then
07      rollback[v];
08      owner[v] := 0;
09    rs[v] := 0
10  abort
```

```
01  program p_w :
02  l := owner[v];
03  if l ≠ self then
04    if l ≠ 0 then p_a
05    l := cas(owner[v], 0, self);
06    if l ≠ self then p_a
07    g[v] := l
```

```
01  program p_r :
02  if (rs[v] ≠ 1) then
03    l := owner[v];
04    if l ≠ self and l ≠ 0 then
05      if l < R then p_a
06      else
07        m := cas(owner[v], l, l + 1);
08        if m ≠ l + 1 then p_a
09        rs[v] := 1
10        l := g[v]
11        rfin
```

```
01  procedure p_e :
02  v := 0;
03  while v < V do
04    v := v + 1;
05    if rs[v] = 1 then
06      l := 0; m := 0
07      while (l − 1 ≠ m)
08        l := owner[v];
09        m := cas(owner[v], l, l − 1);
10      rs[v] := 0
11    else
12      if owner[v] = self then
13        owner[v] := 0
14  commit
```

Algorithm 4.5: The McRT-STM algorithm in RML

5 The Verification

We shall first verify the correctness of TM algorithms for a finite number of threads and variables. We check whether the TM algorithms are opaque for two threads and two variables. Also, we verify that DSTM is obstruction free for two threads and one variable. Later in this chapter, we extend the structural properties discussed in Chapter 3 to the fine-grained formalism.

5.1 The FOIL Tool

We developed a stateful explicit-state model checker, FOIL, that takes as input the RML description of a TM algorithm A, a memory model M, and a correctness property π, and checks whether A is correct with two threads and two variables for π under the memory model M. FOIL uses the RML semantics with respect to the memory model M to compute the state space of the TM algorithm A, and checks inclusion within the correctness property π. FOIL builds on the fly, the product of the transition system for A and the TM specification for π. In our case, we let the correctness criterion be opacity. If a TM algorithm A is not opaque for a memory model M, FOIL automatically

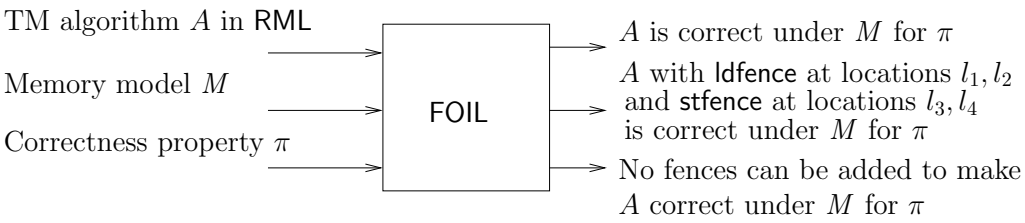

Figure 5.1: A schematic of FOIL

Table 5.1: Time for checking the opacity of TM algorithms under sequential consistency on a 2.8 GHz PC with 2 GB RAM. The time is divided into time t_g needed to generate the language of the TM algorithm from the RML description, and time t_i needed to check inclusion within the property of opacity.

TM algorithm A	Number of states	A is opaque?	t_g	t_i
TL2	1888674	Yes	$581s$	$1.1s$
DSTM	3060158	Yes	$1327s$	$2.3s$
McRT STM	479234	Yes	$265s$	$0.9s$

inserts fences within the RML representation of A in order to make A opaque. FOIL succeeds if it is indeed possible to make A opaque solely with the use of fences. In this case, FOIL reports a possible set of missing fences. FOIL fails if inserting fences cannot make A opaque. In this case, FOIL produces a shortest counterexample to opacity under sequential consistency.[1] We used FOIL to check the opacity of DSTM, TL2, and McRT STM under different memory models.

5.2 Results

We first present the results we obtained from FOIL. Then, we provide some implementation details which make FOIL work.

5.2.1 Sequential consistency

We first model check the TM algorithms for opacity on a sequentially consistent memory model. We find that all of DSTM, TL2, and McRT STM are opaque. The state space obtained for these TM algorithms is large as it covers every possible interleaving, where the level of atomicity is that of the hardware. Table 5.1 lists the number of states of different TM algorithms with the verification results under sequential consistency. The usefulness of FOIL is demonstrated by the size of the state spaces it can handle.

5.2.2 Relaxed memory models

Next, we model check the TM algorithms on the following relaxed memory models: TSO, PSO, and RMO. We find that TL2 and McRT STM are not opaque for PSO and RMO. FOIL gives counterexamples to opacity. We let FOIL insert fences automatically until the TM algorithms are opaque under different memory models. Table 5.2 lists the number and location of fences inserted

[1] Note that if a TM algorithm A cannot be made opaque with fences under some memory model M, then A is not opaque even under sequential consistency.

Table 5.2: Counterexamples generated for opacity, and the type and location of fences required to remove all counterexamples on different relaxed memory models. We list the statement number after which the fence has to be inserted in the RML program.

STM	TSO	PSO	RMO
TL2	No fences	w_1, stfence: $p_e, 26$	w_1, stfence: $p_e, 26$ w_3, ldfence: $p_e, 17$ w_4, ldfence: $p_r, 07$
DSTM	No fences	No fences	No fences
McRT STM	No fences	w_2, stfence: $p_a, 07$	w_2, stfence: $p_a, 07$
Counterexamples			
$w_1 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_1\rangle, t_1)$			
$w_2 : (\langle \text{store } v_1\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{load } v_1\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{rollback } v_1\rangle, t_1)$			
$w_3 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_2\rangle, t_1)$			
$w_4 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1)$			

by FOIL to make the various TM algorithms opaque under various memory models. Note that the counterexamples shown in the table are projected to the loads, stores, and rollbacks of the transactional variables, and rfin instructions. We omit the original long counterexamples (containing for example, a sequence of loads and stores of locks and version numbers) for brevity.

Currently, TM designers use intuition to place fences, as lack of fences risks correctness, and too many fences hamper performance. As FOIL takes as input a memory model, it makes it easy to customize a TM implementation according to the relaxations allowed by the memory model. Although FOIL is not guaranteed to put the minimal number of fences, we found that FOIL indeed inserts the same fences as those in the official TM implementations.

5.2.3 Analysis

We note that reordering a store followed by a load, and reading own write early (due to store buffers) does not create a problem in the TM we have studied. This is evident from the fact that all TM are correct under the TSO memory model without any fences. On the other hand, relaxing the order of stores or loads can be disastrous for the correctness of a TM. This is because most TM use version numbers or locks to control access. For example, a reading thread first checks that the variable is unlocked and then reads the variable. A writing thread first updates the variable and then unlocks it. Reversing the order of writes or reads renders the TM incorrect.

5.2.4 Model checking liveness

We saw in Chapter 3 that the TL2 algorithm does not ensure obstruction freedom with two threads and one variable. Intuitively, proving that a TM algorithm does not satisfy a liveness property at a coarse-grained alphabet is sufficient to prove that the TM algorithm does not satisfy the liveness property at a finer-grained alphabet. This is because we can create a schedule in such a way that the TM algorithm takes steps at the coarse-grained alphabet. However, to prove the result formally, we model check TL2 at the finer-grained level of atomicity.

We model check DSTM, TL2, and McRT STM for obstruction freedom using our tool FOIL under relaxed memory models. For each TM algorithm, we put in the required fences (as obtained in the previous section) in order to make the TM algorithm opaque under the given memory model. We observe that McRT STM and TL2 do not satisfy obstruction freedom. We also observe that for two threads and one variable, DSTM satisfies obstruction freedom at hardware level atomicity under the memory models, sequential consistency, TSO, PSO, and RMO. With the result that DSTM satisfies opacity, we can extend our reduction theorem for liveness discussed in Chapter 3 to prove that DSTM ensures obstruction freedom.

5.3 Implementation Details

We implemented FOIL in OCaml. FOIL supports two modes: generating the state space of a TM algorithm and finding a counterexample history on-the-fly. We found it important to allow both modes in FOIL due to the following reason. The state spaces of the TM algorithms are very large. Checking for a counterexample on the fly requires to build the product automaton. The state space of the product automaton could be as large as the product of the state spaces of the TM algorithm and the TM specification. The size of the transition system of the TM algorithms is, on average, a million states under sequential consistency. The size is much bigger under highly relaxed memory models. The size of the deterministic TM specification is around 46'000 states. We found it impossible, with modest computing resources, to construct the whole product automaton or the transition system of a TM algorithm under a highly relaxed memory model using FOIL. However, FOIL could construct the transition system of the TM algorithms under sequential consistency. We then use a lightweight language inclusion tool to check whether the language of the TM algorithm is included in the language of the TM specification.

5.3.1 Generating the state space

We obtain the transition relation of a TM algorithm using Algorithm 5.1. Intuitively, the algorithm finds all states reachable from the initial state, and outputs the transition system. How FOIL handles relaxed memory models, is

generateTransitionSystem(A)

$frontier := \{q_{init}\}$
$Q := frontier$
while $frontier \neq \emptyset$
  pick and remove a state q from $frontier$
  $T := findnext(A, q)$
  $\delta := \delta \cup T$
  let Q' be the set of destination states in T
  add the set $Q \setminus Q'$ of states to $frontier$
  $Q := Q \cup Q'$
output δ

Algorithm 5.1: Obtaining the transition system of a TM algorithm

findCounterexample$(A, Spec)$

$frontier := \{(q_{init}, p_{init})\}$
$path(q_{init}, p_{init}) := \varepsilon$
$Q := frontier$
while $frontier \neq \emptyset$
  pick and remove a state (q, p) from $frontier$
  $T := findnext(A, q)$
  for each transition $(q, op, q') \in T$ **do**
    if $op \in \hat{O}p$ **then**
      if there exists a state $p_1 \in P$ such that $(p, op, p_1) \in \delta_p$ **then**
        $p' := p_1$ such that $(p, op, p_1) \in \delta_p$
      else
        report counterexample $path(q, p) \cdot op$
    else $p' := p$
    if $(q', p') \notin Q$ **then**
      add (q', p') to Q
      add (q', p') to $frontier$
      $path(q', p') = path(q, p) \cdot op$
report no counterexample found

Algorithm 5.2: Obtaining a counterexample to opacity on-the-fly

hidden beneath the *findnext* procedure. The *findnext* procedure takes as input a TM algorithm and a state q of the TM algorithm, and gives all transitions in the TM algorithm from the state q.

5.3.2 Finding a counterexample

Although the state space generation mode of FOIL gives the transition system for the TM algorithms under sequential consistency, the mode cannot produce the transition system for the TM algorithms under relaxed memory models. This is because the state space of a TM algorithm under a relaxed memory model can be too large to explore with modest computation speed and memory. However, many histories produced under these memory models may not even satisfy opacity. To handle this situation, we use on-the-fly verification. FOIL maintains the product automaton of the transition system of the TM algorithm and the TM specification, and tries to find counterexamples early. FOIL uses Algorithm 5.2 to construct the product automaton and find a counterexample to opacity. We represent the deterministic TM specification for opacity as $Spec = \langle P, p_{init}, \delta_p \rangle$.

We find this procedure highly successful, because counterexamples of opacity tend to be short. On observing a counterexample, FOIL suggests the location of a fence which might make the TM algorithm correct under the given relaxed memory model. On inserting the fence, the number of interleavings decreases, and thus the size of the state space decreases. We run FOIL in this mode until it takes a few minutes to find a counterexample. After that, we run FOIL in the state space generation mode. Once we obtain the state space of the TM algorithm, we can use our lightweight language inclusion tool to check whether the language of the TM algorithm is included in the specification. The interesting part is how the two modes help each other. It is not possible to reach our goal with either of the modes. The state space generation mode fails to generate the state space of a TM algorithm under a highly relaxed memory model due to the large number of interleavings. The counterexample finder mode fails to finish due to the large size of the product automaton even under sequential consistency.

5.3.3 Counterexample analysis

Our tool FOIL automatically inserts fences. However, FOIL does not ensure that the number of fences it inserts is minimal. We describe how FOIL chooses the place where the fence needs to be inserted.

Recall that when FOIL encounters a statement s with a memory instruction in of thread t, FOIL adds the instruction in the queue $d = s_0 \dots s_n$ of deferred statements of thread t according to the given memory model. If FOIL inserts the memory instruction in the middle of the queue to obtain $d' = s_0 \dots s_i \cdot s \cdot s_{i+1} \dots s_n$, FOIL tags the statement s in the queue with the string $s_n \cdot s_{n-1} \dots s_{i+1}$. When FOIL reports a counterexample, we search for the last statement with a reordering tag in the counterexample. Let the tag be $s_1 \dots s_k$. We attribute the error to the reordering allowed by s_k. So, we insert a fence after the statement s_k. The inserted fence is a store (resp. load) fence if s_k is a store (resp. load) instruction.

5.4 Structural Properties

We extend the structural properties discussed in Chapter 3. Our analysis that the TM algorithms, TL2 and DSTM, at coarse-grained atomicity are abort isolated and pending isolated can be extended to the fine-grained atomicity framework as described below. These definitions depend only on the instructions of a single thread, and thus, relaxing the memory model does not break these properties. However, we cannot prove that the TM algorithm McRT STM is abort isolated. This is because McRT STM is a direct update TM, which writes to memory during the transaction. Thus, a state change by a thread in the TM algorithm can be observed by other threads. The structural properties P2 (thread symmetry) and P3 (variable projection) do not depend on the level of atomicity, and thus can be directly extended to the hardware level atomicity.

DSTM

DSTM relies on a notion of ownership. If a transaction wants to read or write a variable, it first atomically sets the transaction record of the variable to itself. If another transaction wants to access the same variable, it first sets the status of the owner transaction to aborted, and reads the old value of the variable.
Abort and pending isolation. An aborting or pending transaction in DSTM changes the global valuation by changing the status of other transactions to aborted. All other changes are not observable to committed transactions.
Conflict commutative. DSTM uses encounter-time ownership for both reads and writes. Thus, if a transaction loads a variable before another transaction stores to the same variable, then all instructions of the *txrd* command can appear before all instructions of the *txwr* command.

TL2

Abort and pending isolation. An aborting transaction x in TL2 can hold a lock and change the value of the global timestamps. If another transaction y observes that x holds the lock for some variable, y aborts. Similarly, if y observes an increment of the global timestamp by x during a read, y aborts. A similar argument holds for pending isolation.
Conflict commutative. To prove that TL2 is conflict commutative at fine-grained atomicity, we need to consider the relaxed memory model. It turns out that the TL2 algorithm (shown in Algorithm 4.3) is not conflict-commutative under some relaxed memory models. This is because if a *txend* command overlaps with a *txrd* command, such that the store instruction in the *txend* command appears before the load instruction in the *txrd* command, it may not be possible to move all instructions corresponding to the *txrd* command after all instructions of the *txend* command. This problem goes away if we insert some fences in the RML program of the TL2 algorithm. We observe that

once FOIL inserts the required fences, TL2 is indeed conflict commutative. Consider a *txrd* command overlapping with a conflicting *txend* command. If the transactional variable is loaded in *txrd* before the variable is stored in *txend*, then all instructions of the *txrd* command can be moved above all instructions of the *txend* command (otherwise the reading transaction must abort). Similarly, if the *txend* commands of two transactions x and y overlap, if x stores to the transactional variable before y, then all instructions in the *txend* command of x can be moved above all instructions of the *txend* command of y.

6 Parametrized Opacity

The formal definition of opacity has allowed for a clear understanding of the behavior of pure transactional programs. Also, it allowed the verification techniques as described in the previous chapters. Unfortunately, the semantics of the interaction of transactions with non-transactional code has not been formally defined. This chapter formalizes this interaction.

We provide a general formal framework for describing the interactions between transactions and non-transactional operations. We consider opacity as a correctness condition for transactions, and parametrize it by a memory model. We claim that while a TM can be implemented in a way to ensure opacity for transactions, there is little one can do (on a given platform or run-time environment) to change the underlying memory model. Hence, it is desirable to define opacity *parametrized* by a memory model. Moreover, we want the definition of parametrized opacity to be implementation-agnostic (like opacity), so that it allows for transactional objects with semantics richer than that of simple read-write variables. This could help describing, for example, TM that implement transactional boosting [HK08] or similar techniques.

We present a definition of a memory model which is general enough to capture a variety of memory models. Intuitively, we formalize a memory model as a function which, depending upon the sequence of operations, gives the set of possible orders of operations. Our formalism can capture common memory models like TSO, PSO, RMO [WG94], and Alpha [Sit02]. Moreover, we allow different processes to observe different order of operations, which allows us to capture memory models with non-atomic stores, like IA-32 [Int06]. We classify memory models on the basis of the possible reorderings of operations they allow.

Our formalization of parametrized opacity is guided by the following intuition:

initially $x = y = z = 0$ in every case

Thread 1	Thread 2
atomic {	
$x := 1$	
$x := 2$	*atomic* {
}	$z := x - y$
atomic {	}
$y := 2$	
}	

(a) Can $z < 0$?

Thread 1	Thread 2
$x := 1$	$r_1 := y$
$y := 1$	$r_2 := x$

(b) Can $r_1 = 1$ and $r_2 = 0$?

Thread 1	Thread 2
atomic {	$z := x$
$x := 1$	
$x := 2$	
}	
atomic {	
$r_1 := z$	
$r_2 := z$	
}	

(c) Can $z = 1$ or $r_1 \neq r_2$?

Figure 6.1: Motivating examples for the definition of parametrized opacity

1. *Opacity for transactions.* Whatever the memory model is, executions that are purely transactional must ensure opacity. Indeed, the semantics of transactions should be intuitive and strong—in the end, we want TM to be as easy to use as coarse-grained locking. For example, consider Figure 6.1(a). Thread 2 should observe x as 0 or 2, because the intermediate state of a transaction ($x = 1$) is not visible to other transactions. Also, y can be observed as 0 or 2. Moreover, y can be observed as 2 only if x is observed as 2, because the effect of transactions is visible in real-time order. Thus, the possible values of z are 0 and 2. Note that even if thread 2 aborts, opacity requires that z is 0 or 2.

2. *Efficiency of non-transactional operations.* Executions that are purely non-transactional have to adhere to the given memory model. In particular, parametrized opacity should not strengthen the semantics of non-transactional operations. The motivation here is to avoid a framework that would inherently require non-transactional operations to be instrumented with additional memory fences or software barriers, even for very weak memory models. For example, in Figure 6.1(b), a memory model may relax the order of write operations in Thread 1 or the read operations in Thread 2, resulting in $r_1 = 1$ and $r_2 = 0$.

3. *Isolation of transactions from non-transactional operations.* Transactions should appear, both to other transactions and non-transactional operations, as if they were executed instantaneously. In particular, isolation of transactions should be respected, regardless of the memory model. That is, first, the intermediate computations of transactions, or updates by aborted transactions, should never be visible to non-transactional operations, and, second, the non-transactional operations concurrent to a transaction should appear as if they happened before or after this transaction. For example, in Figure 6.1(c), Thread 2 cannot observe an intermediate state of a transaction, and thus $z \neq 1$. Moreover, the ef-

fect of a non-transactional operation cannot show up in the middle of a transaction. Thus, $r_1 = r_2$.

Note that opacity parametrized by sequential consistency gives the notion of strong atomicity as proposed by Larus et al. [LR07]. On the other hand, parametrized opacity for a relaxed memory model like RMO matches the notion of strong atomicity given by Martin et al. [MBL06].

In practice, TM implementations that guarantee strong atomicity require that non-transactional operations, instead of accessing memory directly, adhere to an access protocol as defined by the TM implementation. This modification of the semantics of non-transactional operations is known as *instrumentation*. For example, Tabatabai et al. [SMAT$^+$07] propose a TM implementation, where the non-transactional read and write operations follow the locking discipline as done by the transactions. The formal definition of opacity parametrized by a memory model allows us to theoretically analyze the cost of creating TM implementations that guarantee parametrized opacity. While parametrized opacity is the intuitive correctness property for transactional programs with non-transactional operations, we show that, without instrumentation, it cannot be achieved on most memory models. Even for the small class of idealized memory models, where parametrized opacity can be achieved without instrumentation, we show that a TM implementation *must* use expensive read-modify-write operations for each object modified by a transaction.

Next, we focus on TM implementations that instrument non-transactional write operations, without any instrumentation for non-transactional read operations. We start with a basic result which shows that for a class of memory models which allows to reorder independent reads, it is possible to achieve parametrized opacity without instrumenting the read operations, and treating every non-transactional write as a transaction in itself. Note that this might not provide a practical solution, as we do not want a non-transactional operation to carry the overhead associated with a transaction. Moreover, we want non-transactional operations to finish in bounded time, while a transaction, in general, may take arbitrarily long to finish. The next question we ask is whether we can obtain parametrized opacity for a class of memory models with constant-time instrumentation on the writes. We show that for a class of memory models which allows to reorder a read of a variable following a read or write of another variable (like Alpha [Sit02]), it is indeed possible to achieve parametrized opacity with constant-time instrumentation for non-transactional write operations. We also discuss how to adapt the constant-time write instrumentation solution for memory models which do not allow to reorder data-dependent reads (like RMO [WG94] and Java [MPA05]).

Using our theoretical framework, we examine existing TM implementations that guarantee parametrized opacity. TM implementations in the literature that satisfy strong atomicity [SMAT$^+$07], in fact, satisfy parametrized opacity with respect to sequential consistency. We observe that a TM implementation can be designed to be more efficient, if it is to satisfy opacity parametrized

by a weaker memory model. We extract the key practical ideas from our proofs which shall help to design more efficient TM implementations which guarantee opacity parametrized by relaxed memory models. We also show that our theoretical framework can be used to specify weaker notions of correctness. As an example, we formalize single global lock atomicity (SGLA) [GMP06, LR07, MBS+08]. Moreover, we show that the impossibility results we obtain for parametrized opacity do not hold for SGLA.

6.1 Preliminaries

We first describe a framework of a shared memory system consisting of shared objects and operations on those objects. The previous chapters consider single-version opacity, which allowed us to ignore the values of the variables read and written. In general, opacity allows multiple versions for all variables, which requires us to track the values read and written in each operation.

Operations. Let Obj denote a set of shared objects. We consider a shared-memory system consisting of a set P of processes, which communicate by executing commands on (shared) objects. Let C be a set of commands on shared objects, where arguments and return values are treated as part of a command. For example, in a system that supports only reading and writing of shared (natural number) variables, we have $C = \{\mathsf{rd}, \mathsf{wr}\} \times \mathbb{N}$. We define *operations* $Op \subseteq C \times Obj$ as the set of all allowed command-object pairs.

Besides operations that issue commands on shared objects, every process in P can execute the following special operations: start to start a new transaction, operation commit to commit a transaction, and abort to abort a transaction. Let $\hat{Op} = Op \cup \{\mathsf{start}, \mathsf{commit}, \mathsf{abort}\}$.

Histories. We define an *operation instance* as (op, p, k), where $op \in \hat{Op}$ is an operation, $p \in P$ is a process which issues the operation, and $k \in \mathbb{N}$ is a natural number representing the identifier of the operation instance.

A *history* $h \in (\hat{Op} \times P \times \mathbb{N})^*$ is a sequence of operation instances, such that for every pair $(op, p, i), (op', p', j)$ of operation instances in h, we have $i \neq j$. Intuitively, we want each operation instance in a history to have a unique identifier. For a natural number k, when we say "operation k", we mean "operation instance with identifier k", i.e., the element of h of the form (op, p, k), where $o \in \hat{Op}$ and $p \in P$.

A *transaction* of a process p is a subsequence $(op_1, p, i_1) \ldots (op_n, p, i_n)$ of a history h such that (i) op_1 is a start operation, (ii) either operation i_n is the last operation instance of p in h, or we have $op_n \in \{\mathsf{commit}, \mathsf{abort}\}$, and (iii) all operations $op_2, \ldots, op_{n-1}$ belong to set Op.

A transaction T is *committed* (resp. *aborted*) in a history h if the last operation instance of T has a commit operation (resp. an abort operation). A transaction T is *completed* if T is committed or aborted.

Given a history h and a natural number k, we say that operation k is *transactional* in h if operation k is part of a transaction in h. Otherwise, operation k is said to be *non-transactional*. We assume that every history h is *well-formed*: that is, every non-transactional operation in h belongs to set Op. Intuitively, well-formedness of a history requires that every commit and abort of a transaction matches with a corresponding start, and there are no nested transactions. We denote by H the set of all histories.

Given a history h, we define the *real-time* partial order relation $\prec_h \subset \mathbb{N} \times \mathbb{N}$ of the operation identifiers in h, such that for two natural numbers i and j, we have $i \prec_h j$ if:

1. operations i and j belong to transactions T and T', respectively, where T is completed in h and the last operation instance of T precedes the first operation instance of T' in h, or

2. operation i precedes operation j in h, both operations are executed by the same process, and at least one of those operation instances is transactional.

For example, consider the history h illustrated in Figure 6.2(a). The transaction of process p_1 finishes before the transaction of process p_3 starts. The precedence relation $\prec_h$ consists of elements $(1, 2)$, $(5, 7)$, and $(1, 9)$. On the other hand, $(1, 6)$ and $(6, 9)$ are *not* in $\prec_h$.

Object semantics. We use the concept of a *sequential specification* [HW90, Wei89] to describe the semantics of objects. Given an object $x \in Obj$, we define the semantics $[\![x]\!] \subseteq C^*$ as the set of all sequences of commands on x that could be generated by a single process accessing x.

For example, let x be a shared variable that supports only the commands to read and write its value (with initial value 0). Then, $[\![x]\!]$ is a subset of $(\{\mathsf{rd}, \mathsf{wr}\} \times \mathbb{N})^*$, such that, for every sequence $c_1 \dots c_n$ in $[\![x]\!]$, and for all $i, v \in \mathbb{N}$, if $c_i = (\mathsf{rd}, v)$ then either (a) the latest write operation preceding c_i in $c_1 \dots c_n$ is (wr, v), or (b) $v = 0$ and no write operation precedes c_i in $c_1 \dots c_n$.

Sequential histories. We say that a history h is *sequential* if, for every transaction T in h, every operation instance between the start operation instance of T and the last operation instance of T in h is a part of T. That is, intuitively, no transaction T overlaps with another transaction or with any non-transactional operation in h.

We say that a sequential history s *respects* a partial or a total order $\prec \subseteq \mathbb{N} \times \mathbb{N}$ if, for every pair (i, j), if $i \prec j$ then operation i precedes operation j in s.

Let s be a sequential history. We denote by $s|_x$ the longest subsequence of all commands invoked on object x in s. We say that s is *legal* if for every object x, we have $s|_x \in [\![x]\!]$.

p_1	p_2	p_3
$((\mathsf{wr}, 1, x), 1)$		
$((\mathsf{start}), 2)$		
	$((\mathsf{rd}, 1, y), 3)$	
$((\mathsf{wr}, 1, y), 4)$		
$((\mathsf{commit}), 5)$		
	$((\mathsf{rd}, v, x), 6)$	
		$((\mathsf{start}), 7)$
		$((\mathsf{commit}), 8)$
		$((\mathsf{rd}, v', x), 9)$

Every history is read top to bottom. Notation: $(\mathsf{wr}, 1, x), 4)$ under a column marked with process p stands for the operation instance $(((\mathsf{wr}, 1), x), p, 4)$.

(a) History h

p_1	p_2	p_3
$((\mathsf{wr}, 1, x), 1)$		
$((\mathsf{start}), 2)$		
$((\mathsf{wr}, 1, y), 4)$		
$((\mathsf{commit}), 5)$		
	$((\mathsf{rd}, 1, y), 3)$	
	$((\mathsf{rd}, v, x), 6)$	
		$((\mathsf{start}), 7)$
		$((\mathsf{commit}), 8)$
		$((\mathsf{rd}, v', x), 9)$

(b) Sequential history s_1

p_1	p_2	p_3
	$((\mathsf{rd}, v, x), 6)$	
$((\mathsf{wr}, 1, x), 1)$		
$((\mathsf{start}), 2)$		
$((\mathsf{wr}, 1, y), 4)$		
$((\mathsf{commit}), 5)$		
	$((\mathsf{rd}, 1, y), 3)$	
		$((\mathsf{start}), 7)$
		$((\mathsf{commit}), 8)$
		$((\mathsf{rd}, v', x), 9)$

(c) Sequential history s_2

Figure 6.2: Examples of histories and sequential histories

We denote by $visible(s)$ the longest subsequence of s that does not contain any operation instance of a non-committed transaction T, except if the T is not followed in s by any other transaction or non-transactional operation instance. We say that an operation k in s is *legal* in s if history $visible(s')$ is legal, where s' is the prefix of s that ends with operation k.

Two examples of sequential histories are given in Figure 6.2(b) and Figure 6.2(c). Note that s_1 and s_2 respect $\prec_h$ (where h is the history shown in Figure 6.2(a)). History s_1 is legal if $v = v' = 1$. Similarly, s_2 is legal if $v = 0$ and $v' = 1$.

6.2 Parametrized Opacity

We first formalize a memory model in our framework. Then, we define opacity parametrized by a memory model.

6.2.1 Memory models

The operational semantics of memory models defined in the previous chapter do not capture memory models which allow non-atomic stores and those which allow out-of-thin-air values. This chapter defines memory models using an axiomatic approach in order to capture a vast range of memory models that

occur in practice. We formalize a memory model as a transformation followed by per-process reordering of the history. The reordering function is defined in such a way that it allows different processes to have different views of the history, similar to the formalism by Sarkar et al. [SSN+09]. The transformation function allows us to have complex operations at the level of the history, which may need a sequence of operations at the level of the implementation. Moreover, the transformation function allows us to capture memory models which do not provide out-of-thin-air guarantees (e.g., in languages like C/C++ for unsynchronized code).

We define a *process view* $v : P \to 2^{\mathbb{N}\times\mathbb{N}}$ as a function such that $v(p)$ is a partial order on the set of natural numbers for every process $p \in P$. Let V be the set of all process views. A *memory model* is a pair $M = (\tau, R)$, where

- $\tau : Op \to Op^*$ is a *transformation* function that maps an operation to a sequence of operations, and
- $R : H \to 2^V$ is a *reordering* function that maps a history to a set of process views.

Intuitively, τ maps every operation to its internal representation (e.g., in hardware or a run-time environment). For instance, a write to a 64-bit memory word might be, on some systems, executed as two writes to its 32-bit parts. Moreover, if a memory model does not give out-of-thin-air guarantees, τ can map every write to a special *havoc* operation followed by the write. A read which follows the havoc operation and precedes the write operation can return any value. A process view allows different processes to observe different orders of operations in a given history. This can capture memory models, like IA-32 [Int06] which allow non-atomic stores.

If h is a well-formed history, then $\tau(h)$ denotes a well-formed history obtained from h by replacing every operation instance (op, p_j, k) of h with a sequence (op_1, p_j, k_1), ..., (op_m, p_j, k_m), where $\tau(op) = op_1, \ldots, op_m$ and $k_1, \ldots, k_m \in \mathbb{N}$. (Note that identifiers of operation instances of $\tau(h)$ can be arbitrary, as long as they are unique.)

Well-formed memory models. A transformation function τ is *well-formed* if for every well-formed, sequential and legal history s, sequence $\tau(s)$ is also a well-formed, sequential, and legal history. Intuitively, a transformation is well-formed if it does not change the semantics of transactional operations.

A reordering function R is *well-formed* if for every history $h \in H$, for every view $v \in R(h)$, for every process $p \in P$: (i) if $(i, j) \in v(p)$, then operations i and j are non-transactional in h, (ii) if $(i, j) \in v(p)$, then i precedes j in h, and (iii) if $(i, j) \in v(p)$, then $(j, i) \notin v(p')$ for all processes $p' \in P$. Intuitively, condition (i) requires that a reordering function can impose an order only on non-transactional operations, condition (ii) requires that a view does not force a process to observe operations of some process in out of program order, and condition (iii) requires that a view should not force two processes to observe

operations to occur in different orders. A memory model $M = (\tau, R)$ is well-formed if τ and R are well-formed. All memory models we know of are indeed well-formed.

Capturing dependence of operations. Often, a memory model [MPA05, WG94] allows to reorder operations unless the latter operation is control-dependent or data-dependent on the former operation. We need to distinguish dependent reads from independent reads in our framework to capture these memory models. We can capture these dependencies in our framework using additional commands: $\{\mathsf{cdrd}, \mathsf{ddrd}, \mathsf{cdwr}, \mathsf{ddwr}\} \times \mathbb{N} \times 2^{\mathbb{N}}$. For example, an operation instance (op, p, k) in h with $op = ((\mathsf{cdrd}, v, \{k_1, \ldots, k_n\}), x)$ denotes a read operation which is control-dependent on operations $k_1 \ldots k_n$ in h. Well-formedness of a history with control and data dependent operations requires that if an operation k is dependent on $k_1 \ldots k_n$ in h, then all operations $k_1 \ldots k_n$ precede k in h.

6.2.2 Examples of memory models

We now give examples of memory models for histories on read and write operations on shared variables. We first define an *identity transformation* function τ_I such that $\tau_I(op) = op$ for every operation $op \in Op$. We say that a view v is identical across processes, if $v(p) = v(p')$ for all processes $p, p' \in P$.

- *Sequential consistency* requires that the order of operations of a process in a history is preserved in every view, and all processes view an identical order of operations of different processes. Formally, $M_{sc} = (\tau_I, R_{sc})$ such that for all histories h, we have $v \in R_{sc}(h)$ if (a) v is identical across processes, and (b) for every process $p \in P$, for every pair $(op_1, p, i), (op_2, p, j)$ of operations such that operation i precedes operation j in h, we have $(i, j) \in v(p)$.

- *Total store order* allows a write operation to forward the value of a variable to a following read operation, and allows to reorder a write operation followed by a read operation to a different variable. Formally, the memory model $M_{tso} = (\tau_I, R_{tso})$ such that for all histories h, we have $v \in R_{tso}(h)$ if (a) v is identical across processes, and (b) for every process p, for every pair $(op_1, p, i), (op_2, p, j)$ of operations such that operation i precedes operation j in h, we have $(i, j) \in v(p)$ if one of following conditions holds:

  - op_2 is a write operation[1],
  - op_1 and op_2 are to the same object x, or

[1]We use the term "write operation" (resp. "read operation") as a general term for a simple write or a control/data dependent write (resp. read).

– op_1 is a read operation of the form (rd, v, x)[2] such that (wr, v, x) is the last preceding write operation to x by process p in h.

The intuition for the last case is to allow two read operations to different variables to reorder if the first read obtains the value from a store buffer.

- *Relaxed memory order* allows to reorder read and write operations to different variables, unless the first operation is a read, and the second operation is either a write control/data-dependent on the first operation, or a read data-dependent on the first operation. RMO is specified by the memory model $M_{rmo} = (\tau_I, R_{rmo})$ such that for all histories h, we have $v \in R_{rmo}(h)$ if (a) v is identical across processes, and (b) for every process p, for every pair $(op_1, p, i), (op_2, p, j)$ of operations such that operation i precedes operation j in h, we have $(i, j) \in v(p)$ if one of the following conditions holds:

  – op_1 and op_2 are to the same object x,
  – op_1 is a read of a variable x and $op_2 = ((\mathsf{cdwr}, v, K), y)$ or $op_2 = ((\mathsf{ddwr}, v, K), x)$ for some $v \in V$ and $K \subseteq \mathbb{N}$ such that $i \in K$, or
  – op_1 is a read of a variable x and $op_2 = ((\mathsf{ddrd}, v, K), y)$ for some $v \in V$ and $K \subseteq \mathbb{N}$ such that $i \in K$.

- *Junk-SC* is a memory model we describe here to show how memory models that allow junk (out-of-thin-air) values can be expressed in our formalism. Junk-SC requires sequential consistency, but if there exist a read and a write to a variable, such that they are not real-time ordered with respect to each other, then the read can observe any junk value. We denote Junk-SC as $M_{junk} = (\tau, R_{sc})$, where τ is given as: $\tau(\mathsf{wr}, v, x) = havoc(x) \cdot (\mathsf{wr}, v, x)$ and $\tau(op) = op$ otherwise.

Classes of memory models. We now present a classification of memory models on the basis of reorderings they allow. We build the following four classes depending upon the restrictions posed by the memory model.

1. We first describe the class M_{rr} which represents the *read-read restrictive memory models*. We let $M_{rr} = M^i_{rr} \cup M^c_{rr} \cup M^d_{rr}$, where:

   M^i_{rr} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is a read operation to y such that $x \neq y$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

   M^c_{rr} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation

[2]Throughout the paper, for ease of readability, we write $((\mathsf{rd}, v), x)$ as (rd, v, x) and $((\mathsf{wr}, v), x)$ as (wr, v, x).

j is of the form $(((\mathsf{cdrd}, v, K), y), p, j)$ for some v and $K \subseteq \mathbb{N}$ such that $x \neq y$ and $i \in K$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

M^d_{rr} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is of the form $(((\mathsf{ddrd}, v, K), y), p, j)$ for some v and $K \subseteq \mathbb{N}$ such that $x \neq y$ and $i \in K$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

Intuitively, M^i_{rr} restricts the order of read operations to different variables. M^c_{rr} (resp. M^d_{rr}) restricts the order of read operations to different variables if the second read is control (resp. data) dependent on the first read. Note that if a memory model M is in M^i_{rr}, then $M \in M^c_{rr}$ and $M \in M^d_{rr}$.

2. We define the class of *read-write restrictive memory models* as $M_{rw} = M^i_{rw} \cup M^c_{rw} \cup M^d_{rw}$, where:

   M^i_{rw} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is a write operation to y such that $x \neq y$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

   M^c_{rw} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is of the form $(((\mathsf{cdwr}, v, K), y), p, j)$ for some v and $K \subseteq \mathbb{N}$ such that $x \neq y$ and $i \in K$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

   M^d_{rw} is the set of memory models $M = (\tau_I, R)$ such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is of the form $(((\mathsf{ddwr}, v, K), y), p, j)$ for some v and $K \subseteq \mathbb{N}$ such that $x \neq y$ and $i \in K$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

3. We define the class of *write-read restrictive memory models* as M_{wr}, where $M = (\tau_I, R)$ belongs to M_{wr} if for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a write operation to x and operation j is a read operation to y such that $x \neq y$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

4. We define the class of *write-write restrictive memory models* as M_{ww}, where $M = (\tau_I, R)$ belongs to M_{ww} if for all histories h, for all $i, j \in \mathbb{N}$, if operations i and j are write operations to x and y such that $x \neq y$, and both operations i, j are by process p, then for all view orders $v \in R(h)$, for all $p' \in P$, we have $(i, j) \in v(p')$.

We classify some well-known memory models in these classes.

- SC memory model M_{sc} is in $M^i_{rr} \cap M^i_{rw} \cap M_{wr} \cap M_{ww}$.
- TSO memory model M_{tso} is in $M^i_{rr} \cap M^i_{rw} \cap M_{ww}$ and $M_{tso} \notin M_{wr}$.
- Partial store order (PSO) memory model M_{pso} is in $M^i_{rr} \cap M^i_{rw}$ and $M_{pso} \notin M_{ww} \cup M_{wr}$.
- Relaxed memory order (RMO) M_{rmo} is in $M^d_{rr} \cap M_{rw}$ and $M_{rmo} \notin M_{ww} \cup M_{wr}$. Moreover, note that $M_{rmo} \notin M^i_{rr}$ and $M_{rmo} \notin M^i_{rw}$.
- Alpha memory model M_α is in M_{rw} and $M_\alpha \notin M_{rr} \cup M_{wr} \cup M_{ww}$.

Note that these classes do not impose any restrictions on views of different processes, and thus memory models which allow non-atomic stores (like IA-32 [Int06]) can also be classified under these classes. For example, the IA-32 memory model has a similar classification as TSO.

6.2.3 Parametrized opacity

We now define the notion of parametrized opacity, i.e., opacity parametrized by a given memory model M. Recall that, intuitively, parametrized opacity requires that (1) every transaction appears as if it took place instantaneously between its first and last operation, and (2) non-transactional operations ensure the requirements specified by the given memory model.

We say that a history h ensures *opacity parametrized by a memory model* $M = (\tau, R)$, if there exists a total order $\ll$ on the set of transactional operations in h and a process view $v \in R(\tau(h))$, such that, for every process $p \in P$, there exists a sequential history s that satisfies the following conditions:

1. s is a permutation of $\tau(h)$,
2. s respects relation $\ll \cup \prec_h \cup v(p)$, and
3. every operation is legal in s.

For example, the history h shown in Figure 6.2(a) is parametrized opaque with respect to memory model M_{sc} if $v = 1$. This is because: (a) operation 3 reads the value of y as 1 which is written by the transaction of process p_1, (b) operation 1 writes x as 1 before the transaction, and (c) SC requires that p_2 reads x after y. Moreover, h is parametrized opaque with respect to memory model M_{rmo} if $v = 0$ or $v = 1$. This is because RMO allows to reorder reads of different variables. h is parametrized opaque with respect to M_{junk} if $v = 1$ for the same reasons as for M_{sc}. But note that if operation 3 read y as 0, then opacity parametrized by M_{junk} allows operation 6 to read any value $v \in \mathbb{N}$.

p_1	p_2
$((\triangleright, \mathsf{start}), 1)$	
$(\langle \mathsf{cas}\ g, 0, 1\rangle, 1)$	
	$((\triangleright, (\mathsf{rd}, 1, x)), 2)$
$((\triangleleft, \mathsf{start}), 1)$	
	$((\langle \mathsf{load}\ x, 1\rangle), 2)$
$((\triangleright, (\mathsf{wr}, 1, x)), 3)$	
	$((\triangleleft, (\mathsf{rd}, 1, x)), 2)$
$((\langle \mathsf{store}\ a_x, 1\rangle, 3)$	
$((\triangleleft, (\mathsf{wr}, 1, x)), 3)$	
$((\triangleright, \mathsf{commit}), 4)$	
$(\langle \mathsf{store}\ g, 0\rangle, 4)$	
$((\triangleleft, \mathsf{commit}), 4)$	

(a) Trace r

The trace and the histories are read top to bottom.
Trace notation: (in, k) under column p represents the instruction instance (in, p, k).
History notation: $(\mathsf{wr}, 1, x), 4)$ under a column marked with process p stands for the operation instance $(((\mathsf{wr}, 1), x), p, 4)$.

p_1	p_2
$(\mathsf{start}, 1)$	
	$((\mathsf{rd}, 1, x), 2)$
$((\mathsf{wr}, 1, x), 3)$	
$(\mathsf{commit}, 4)$	

(b) History h_1

p_1	p_2
	$((\mathsf{rd}, 1, x), 2)$
$(\mathsf{start}, 1)$	
$((\mathsf{wr}, 1, x), 3)$	
$(\mathsf{commit}, 4)$	

(c) History h_2

Figure 6.3: An example of a trace and corresponding histories. Notation: (in, k) under column p represents the instruction instance (in, p, k)

6.3 TM Implementations

In this section, we define a TM implementation $\mathcal{I}$, and when a given TM implementation ensures opacity parametrized by a given memory model M. Intuitively, $\mathcal{I}$ ensures opacity parametrized by M if every history *generated* by $\mathcal{I}$ ensures opacity parametrized by M. We thus need to define precisely which histories are generated by a given TM implementation.

Instructions. We start by defining hardware primitives that a TM implementation is allowed to use. Let *Addr* be a set of memory addresses. We define the set *In* of *instructions* as follows, where $a \in Addr$ and $v, v' \in \mathbb{N}$:

$$In ::= \langle \mathsf{load}\ a, v\rangle \mid \langle \mathsf{store}\ a, v\rangle \mid \langle \mathsf{cas}\ a, v, v'\rangle$$

We call the store and CAS instructions as update instructions. An operation of a history corresponds to a sequence of instructions. To know the begin and end points of an operation at the level of hardware, we use two special instructions, $\triangleright$ and $\triangleleft$, for each operation. For every operation $op \in \hat{Op}$, we denote the *invocation* (instruction) of *op* as $(\triangleright, op)$, and the *response* (instruction) of *op* as $(\triangleleft, op)$. Let $\hat{In} = In \cup (\{\triangleright, \triangleleft\} \times \hat{Op})$.

Traces. We define an *instruction instance* as (in, p, k), where $in \in \hat{In}$ is an instruction, $p \in P$ is the process that issues the instance, and $k \in \mathbb{N}$ is an

operation identifier. Every instruction instance (in, p, k) is said to *correspond* to operation k.

A *trace* $r \in (\hat{In} \times P \times \mathbb{N})^*$ is a sequence of instruction instances. Let $k \in \mathbb{N}$ be an operation identifier. A *complete operation trace* of operation k is a sequence of the form $((\triangleright, op), p, k)$ (in_1, p, k) $\ldots$ (in_m, p, k) $((\triangleleft, op), p, k)$, where $p \in P$ is a process, $op \in \hat{Op}$ is an operation, and $in_1 \ldots in_m \in In$ are instructions. An *incomplete operation trace* of operation k is a sequence of the form $((\triangleright, op), p, k)$ (in_1, p, k) $\ldots (in_m, p, k)$, where $p \in P$, $op \in \hat{Op}$, and $in_1 \ldots in_m \in In$.

Let r be a trace. Given a process $p \in P$, we denote by $r|_p$ the longest subsequence of instruction instances in r issued by process p. We assume that every trace r satisfies the following property: for every process $p \in P$, the sequence $r|_p$ is a sequence of complete operation traces, possibly ending with an incomplete operation trace. We say that a history h *corresponds* to a trace r if:

1. Given any natural number k, an operation k is in h if and only if there is an instruction instance that corresponds to operation k in r, and

2. Operation k occurs before operation j if some instruction instance that corresponds to operation k occurs in r before some instruction instance that corresponds to operation j.

Intuitively, a history h that corresponds to a trace r represents the logical order of operations in r. If we assigned a point in time to every instruction in r, and every operation in h, then every operation (op, k) in h must be somewhere in between the corresponding invocation instruction $((\triangleright, op), p, k))$ and response instruction $((\triangleleft, op), p, k)$ in r.

For example, consider the trace r shown in Figure 6.3(a). In r, the start operation issues a CAS instruction to address g to change the value from 0 to 1, and the commit instruction stores value 0 to g. Histories h_1 and h_2 are two examples of histories that correspond to r.

A trace r is *well-formed* if every history h corresponding to r is well-formed. We assume that every trace is well-formed.

Let r be a trace, and p be a process. A *transaction* T of p in r is a sequence in $r|_p$ of the form $((\triangleright, \mathsf{start}), p, k)$ (in_1, p, k_1) $\ldots$ (in_m, p, k_m), where the following conditions are satisfied:

- $in_m \in \{(\triangleleft, \mathsf{commit}), (\triangleleft, \mathsf{abort})\}$, or (in_m, p, k_m) is the last instruction instance of r_p, and

- for all j s.t. $1 \leq j < m$, we have $in_j \notin \{(\triangleright, \mathsf{start}), (\triangleleft, \mathsf{commit}), (\triangleleft, \mathsf{abort})\}$

Moreover, T is committed (resp. aborted) if the last instruction instance in T is a response of a commit (resp. abort) operation.

An instance $((\triangleright, op), p, k)$ of an invocation is said to be *transactional* in a trace r if it belongs to some transaction in r. Otherwise, the instance is said to

be *non-transactional*. For example, consider the trace r shown in Figure 6.3(a). The (single) invocation instance of process p_2 is non-transactional, while all invocation instances of process p_1 are transactional in r.

TM implementations. A *TM implementation* $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ is a pair, where $\mathcal{I}_T : \hat{Op} \to 2^{In^*}$ is the implementation for transactional operations, and $\mathcal{I}_N : Op \to 2^{In^*}$ is the implementation for non-transactional operations.

Let $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ be a TM implementation. We say that a complete operation trace $((\triangleright, op), p, k)\ (in_1, p, k) \ldots (in_m, p, k)\ ((\triangleleft, op), p, k)$ is *transactionally* (resp. *non-transactionally*) *generated* by $\mathcal{I}$, if sequence $in_1 \ldots in_m$ is in $\mathcal{I}_T(op)$ (resp. in $\mathcal{I}_N(op)$). We say that an incomplete operation trace $((\triangleright, op), p, k)$ $(in_1, p, k) \ldots (in_m, p, k)$ is *transactionally* (resp. *non-transactionally*) *generated* by $\mathcal{I}$, if sequence $in_1 \ldots in_m$ is a prefix of some element in $\mathcal{I}_T(op)$ (resp. in $\mathcal{I}_N(op)$).

Given a trace r and a TM implementation $\mathcal{I}$, we say that r is *generated* by $\mathcal{I}$ if for every transactional (resp. non-transactional) operation k in r, the complete or incomplete operation trace of operation k in r is transactionally (resp. non-transactionally) generated by $\mathcal{I}$.

Instrumentation. We say that a TM implementation $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ is *uninstrumented* if for every variable x, we have $\mathcal{I}_N(\mathsf{rd}, v, x) = \{\langle \mathsf{load}\ a_x, v \rangle\}$ and $\mathcal{I}_N(\mathsf{wr}, v, x) = \{\langle \mathsf{store}\ a_x, v \rangle\}$, where a_x is the address of variable x. Otherwise, the TM implementation is *instrumented*. Note that these terms refer only to the implementation of non-transactional operations.

Languages. We define the *language* $L(\mathcal{I})$ of a TM implementation as the set of all traces generated by a TM implementation. We define that a TM implementation $\mathcal{I}$ *guarantees* opacity parametrized by a memory model M if, for every trace $r \in L(\mathcal{I})$, there is a history h that corresponds to r such that h ensures opacity parametrized by M.

Note that our definition of a trace does not require that after an instruction of the form $\langle \mathsf{store}\ a_x, v \rangle$, the subsequent load of a_x is of the form $\langle \mathsf{load}\ a_x, v \rangle$. Indeed, the underlying hardware may execute a relaxed memory model (which, in principle, may be different from the programmer's memory model at the level of operations). For example, a programmer may wish to guarantee opacity parametrized by sequential consistency on a hardware with memory model RMO. But for the sake of simplicity, in the following sections, we assume that the underlying hardware guarantees a strong memory model equivalent to linearizability, that is, every instruction is executed to completion when it is issued. Note that our impossibility results hold even when the underlying hardware executes a weaker memory model. Moreover, we assume that a transaction does not have the information of other concurrent transactions. So, a transaction cannot tell whether it is running in isolation, or concurrently with other transactions.

6.4 Achieving Parametrized Opacity

We use our framework to investigate the inherent cost of achieving opacity parametrized by a memory model.

6.4.1 Uninstrumented TM implementations

We first study uninstrumented TM implementations. We show that for most of the practical memory models, uninstrumented TM implementations cannot achieve parametrized opacity. Moreover, we show that even to achieve opacity parametrized by very relaxed memory models, it is required that transactional write operations are implemented as expensive compare-and-swap instructions.

We start with a lemma which states that if a committed transaction consists of a write operation, then the transaction must consist of a store or a compare-and-swap instruction. We assume that there are no other transactions running in the duration of the committed transaction.

Lemma 6.1. *For every memory model M, an uninstrumented TM implementation $\mathcal{I}$ guarantees parametrized opacity only if for all traces $r \in L(\mathcal{I})$, for every committing transaction T in r, if there is an operation (wr, v, x) in T, then the transaction T in r consists of an update instruction to a_x with value v.*

Proof. Let the value of a_x be initially 0. Consider a trace r with a single process p: $(\triangleright, \mathsf{start})$, ..., $(\triangleleft, \mathsf{start})$, $(\triangleright, (\mathsf{wr}, v, x))$, ..., $(\triangleleft, (\mathsf{wr}, v, x))$, $(\triangleright, \mathsf{commit})$, ..., $(\triangleleft, \mathsf{commit})$, $(\triangleright, (\mathsf{rd}, v', x))$, $\langle \mathsf{load}\ a_x, v' \rangle$, $(\triangleleft, (\mathsf{rd}, v', x))$. Let T be the committed transaction of process p in r. We observe that T consists of an operation $op_1 = (\mathsf{wr}, v, x)$. Let r consist of a non-transactional operation $op_2 = (\mathsf{rd}, v', x)$, such that the invocation of op_2 occurs after the last instruction of T, that is, the response of the commit operation. As the TM implementation $\mathcal{I}$ is uninstrumented, we have $\mathcal{I}_N(\mathsf{rd}, v', x) = \{\langle \mathsf{load}\ a_x, v' \rangle\}$. Note that as r has a single process, there is only one history h corresponding to r. By definition of $\prec_h$, we know that $op_1 \prec_h op_2$. Thus, to guarantee parametrized opacity, we must have $v = v'$. Thus, T must issue an update instruction to a_x with value v.

□

Using the above lemma, we now prove that for memory models which restrict the order of some pair of read or write operations to different variables, parametrized opacity cannot be achieved.

Theorem 6.1. *Given a memory model M such that $M \in (M_{rr} \cup M_{rw} \cup M_{wr} \cup M_{ww})$, there does not exist an uninstrumented TM implementation that guarantees opacity parametrized with respect to M.*

Proof. We prove the theorem in four parts, where each part corresponds to one of the cases of ordering restrictions between read and write operations. In every case, we construct a trace r with two processes p_1 and p_2, where p_1 executes

p_1	p_2
$(\triangleright, \mathsf{start})$	
...	
$(\triangleleft, \mathsf{start})$	
$(\triangleright, (\mathsf{wr}, v_1, x))$	
...	
$(\triangleleft, (\mathsf{wr}, v_1, x))$	
$(\triangleright, (\mathsf{wr}, v_2, y))$	
...	
$(\triangleleft, (\mathsf{wr}, v_2, y))$	
$(\triangleright, \mathsf{commit})$	
...	
$\langle \mathsf{update}\ a_x, v_1 \rangle$	
...	
	$(\triangleright, (\mathsf{rd}, v_3, x))$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_3, x))$
	$(\triangleright, (\mathsf{rd}, v_4, y))$
	$\langle \mathsf{load}\ a_y, v_4 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_4, y))$
$\langle \mathsf{update}\ a_y, v_2 \rangle$	
...	
$(\triangleleft, \mathsf{commit})$	

An $\langle \mathsf{update}\ a, v \rangle$ instruction denotes a store or a successful cas instruction to address a with value v. We omit the instruction identifiers. We use ... as a shorthand for a sequence of instructions.

Figure 6.4: The trace r constructed in Theorem 1, Case 1

a transaction T, and p_2 issues non-transactional (and possibly transactional) operations in such a way that for every history corresponding to r, there is no sequential history s which respects the restriction posed by the memory model, and at the same time, every operation in s is legal. In every case, a_x and a_y are initialized to 0.

Case 1. Let $M \in M_{rr}$. That is, M does not allow to reorder two read operations to different variables. The trace r is shown in Figure 6.4. Let T consist of operations (wr, v_1, x) and (wr, v_2, y). From Lemma 1, we know that T updates addresses a_x and a_y with values v_1 and v_2 respectively. Without loss of generality, we assume that a_x is updated before a_y.[3] Let the trace r consist of two non-transactional operations (rd, v_3, x) and (rd, v_4, y) issued by process p_2 (with identifiers j and k respectively). As $\mathcal{I}$ is uninstrumented, the non-transactional read operations are implemented as load instructions. Let the two load instructions $\langle \mathsf{load}\ a_x, v_3 \rangle$ and $\langle \mathsf{load}\ a_y, v_4 \rangle$ of process p_2 execute between the updates of a_x and a_y. Thus, we have $v_3 = v_1$ and $v_4 = 0$. Note that if T is an aborted transaction in r, then there is no history h corresponding to

[3]Figure 6.4 shows the updates as part of the commit operation. In general, the updates can happen anywhere during the transaction, but always as two separate instructions.

p_1	p_2
$(\triangleright, \mathsf{start})$	
...	
$(\triangleleft, \mathsf{start})$	
$(\triangleright, (\mathsf{rd}, v_1, x))$	
...	
$(\triangleleft, (\mathsf{rd}, v_1, x))$	
$(\triangleright, (\mathsf{wr}, v_2, y))$	
...	
$(\triangleleft, (\mathsf{wr}, v_2, y))$	
$(\triangleright, \mathsf{commit})$	
...	
	$(\triangleright, (\mathsf{wr}, v_3, x))$
	$\langle \mathsf{store}\ a_x, v_3 \rangle$
	$(\triangleleft, (\mathsf{wr}, v_3, x))$
	$(\triangleright, (\mathsf{rd}, 0, y))$
	$\langle \mathsf{load}\ a_y, 0 \rangle$
	$(\triangleleft, (\mathsf{rd}, 0, y))$
$\langle \mathsf{update}\ a_y, v_2 \rangle$	
...	
$(\triangleleft, \mathsf{commit})$	

(a) Case 2

p_1	p_2
$(\triangleright, \mathsf{start})$	
...	
$(\triangleleft, \mathsf{start})$	
$(\triangleright, (\mathsf{wr}, v_1, x))$	
...	
$(\triangleleft, (\mathsf{wr}, v_1, x))$	
$(\triangleright, (\mathsf{wr}, v_2, y))$	
...	
$(\triangleleft, (\mathsf{wr}, v_2, y))$	
$(\triangleright, \mathsf{commit})$	
...	
$\langle \mathsf{update}\ a_x, v \rangle$	
	$(\triangleright, (\mathsf{rd}, v_3, x)$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_3, x)$
	$(\triangleright, (\mathsf{wr}, v_4, y))$
	$\langle \mathsf{store}\ a_y, v_4 \rangle$
	$(\triangleleft, (\mathsf{wr}, v_4, y))$
	$((\triangleright, (\mathsf{wr}, 0, y))$
	$\langle \mathsf{store}\ a_y, 0 \rangle$
	$((\triangleleft, (\mathsf{wr}, 0, y))$
$\langle \mathsf{update}\ a_y, v \rangle$	
...	
$(\triangleleft, \mathsf{commit})$	
	$(\triangleright, \mathsf{start})$
	...
	$(\triangleleft, \mathsf{commit})$
	$(\triangleright, (\mathsf{rd}, v_5, x))$
	$\langle \mathsf{load}\ a_x, v_5 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_5, x))$
	$(\triangleright, (\mathsf{rd}, v_6, y))$
	$\langle \mathsf{load}\ a_y, v_6 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_6, y))$

(b) Case 3

Figure 6.5: Traces r constructed in Theorem 1, Case 2 and 3

r such that h satisfies opacity parametrized by M. This is because operation j observes the update to a_x. Thus, let T be a committed transaction in r. Consider an arbitrary history h corresponding to r. By definition of M_{rr}, every view function $v \in R(h)$ requires that $(j, k) \in v(p)$ for all $p \in P$. On the other hand, legality requires operation j to appear after T, and operation k to appear before T. Thus, there does not exist a sequential history which

satisfies conditions 2 and 3 at the same time.

Case 2. Let $M \in M_{wr}$. That is, M does not allow to reorder a write followed by a read operation to a different variable. The trace r is shown in Figure 6.5(a). Let T consist of operations $op_1 = (\mathsf{rd}, v_1, x)$ and $op_2 = (\mathsf{wr}, v_2, y)$. From Lemma 1, we know that T updates address a_y with value v_2. Let r consist of two non-transactional operations of p_2 in the order (wr, v_3, x) (with identifier j) followed by $(\mathsf{rd}, 0, y)$ (with identifier k), such that $v_3 \neq v_1$. As $\mathcal{I}$ is uninstrumented, p_2 executes $\langle \mathsf{store}\ a_x, v_3 \rangle$ followed by $\langle \mathsf{load}\ a_y, 0 \rangle$. Let these two instructions execute after the response of op_1 and immediately before the update of a_y by process p_1.[4] Note that as p_2 does not change the value of a_y, the use of CAS instruction by p_1 to update a_y will be successful. Moreover, the key point to note here is that once p_1 updates a_y with value v_2, the transaction T cannot abort. This is because r can be extended to trace r', where a non-transactional read by process p_2 executes $\langle \mathsf{load}\ a_y, v_2 \rangle$. If T is an aborted transaction, then no history corresponding to r' is parametrized opaque with respect to M. Thus, T is a committed transaction. Consider an arbitrary history h corresponding to r. Legality requires that operation k occurs before T and operation j occurs after T. By definition of M_{wr}, every view $v \in R(h)$ requires that $(j, k) \in v(p)$ for all $p \in P$. Thus, h does not ensure opacity parametrized by M.

Case 3. Let $M \in M_{rw}$. That is, M does not allow to reorder a read followed by a write operation to a different variable. The trace r is shown in Figure 6.5(b). Let T consist of operations $op_1 = (\mathsf{wr}, v_1, x)$ and $op_2 = (\mathsf{wr}, v_2, y)$. From Lemma 1, we know that T updates addresses a_x and a_y with values v_3 and v_4. Without loss of generality, we assume that a_x is updated before a_y. Process p_2 issues the following operations non-transactionally in the order: (rd, v_3, x), (wr, v_4, y), $(\mathsf{wr}, 0, y)$. As $\mathcal{I}$ is uninstrumented, these three operations are executed as: $\langle \mathsf{load}\ a_x, v_3 \rangle$, $\langle \mathsf{store}\ a_y, v_4 \rangle$, and $\langle \mathsf{store}\ a_y, 0 \rangle$. Let r be such that these three instructions occur immediately before the update of a_y with value v_2. As p_2 changes and restores the value of a_y to 0, process p_1 does not observe the change, and thus, the update of a_y with value v_2 is successful. As in case 2, T cannot be an aborting transaction, once T updates a_y. We now extend r as follows: after the response of the commit operation of T, let p_2 execute a transaction T' followed by two non-transactional operations: (rd, v_5, x), (rd, v_6, y). Note that $v_5 = v_1$ and $v_6 = v_2$. Consider an arbitrary history h corresponding to r. Legality requires that the first non-transactional read by process p_2 occurs after T, and the two non-transactional writes of p_2 occur before T. This contradicts the expected view order for a memory model $M \in M_{rw}$. Thus, the history h does not ensure opacity parametrized by M. Note that T cannot update a_y with value 0 due to the following argument: consider a trace r' which is identical to r, except that in r', process p_2 does not issue the two non-transactional writes. At the point where p_1

[4]For simplicity, we assume that the transaction loads the value of a_x before the response of op_1. The argument holds in general, as the value of a_x is loaded before the update of a_y.

updates a_y, it cannot observe a difference from trace r (as process p_2 uses two store instructions for the non-transactional write operations). Consider an arbitrary history h corresponding to r'. Legality requires that the last two non-transactional operations by p_2 see value v_1 and v_2. Thus, p_1 cannot update a_y to 0 in transaction T.

Case 4. Let $M \in M_{ww}$. That is, M does not allow to reorder two write operations to different variables. The trace r is similar to the one for case 3, shown in Figure 6.5(b), except that p_1 has two read operations, and the first operation of p_2 is a write instead of a read operation. Let T consist of four operations: (rd, v_1, x), (rd, v_2, y), (wr, v_3, x) and (wr, v_4, y). As in case 3, we know that T updates addresses a_x and a_y with values v_3 and v_4, and we assume that a_x is updated before a_y. Let the trace r consist of three non-transactional operations of process p_2 in the order (wr, v_5, x), (wr, v_6, y), $(\mathsf{wr}, 0, y)$. As $\mathcal{I}$ is uninstrumented, the non-transactional write operations are implemented as store instructions. Let the three store instructions: $\langle \mathsf{store}\ a_x, v_5 \rangle$, $\langle \mathsf{store}\ a_y, v_6 \rangle$, and $\langle \mathsf{store}\ a_y, 0 \rangle$ by process p_2 occur immediately before the update of a_y with value v_4 by process p_1 in trace r. As in case 2, T cannot be an aborting transaction, after it updates a_y. We can extend the trace r exactly as in case 3 now, and show a contradiction between the legality and the view order required by a memory model $M \in M_{ww}$. Thus, for every history h corresponding to r, we know that h does not ensure opacity parametrized by M.

□

We saw in the classification of memory models (Section 3.2) that most of the practical memory models do restrict some order of operations. Thus, Theorem 1 gives an intuition that without instrumentation, it is not possible to achieve opacity parametrized by practical memory models. We now show that for an idealized memory model, which allows reordering all operations to different variables, achieving parametrized opacity is still expensive: a TM implementation must use cas instruction within a transaction to update every variable which is read and written by the transaction.

Theorem 6.2. *For a memory model M, an uninstrumented TM implementation $\mathcal{I}$ guarantees parametrized opacity with respect to M only if for all traces $r \in L(\mathcal{I})$, for all variables x, if a committed transaction T consists of operations (rd, v, x) and (wr, v', x), then T consists of a $\langle \mathsf{cas}\ x, v, v' \rangle$ instruction.*

Proof. Consider a trace r with two processes p_1 and p_2 as shown in Figure 6.6. Let T be a transaction of process p_1 in r, such that T consists of operations (rd, v, x) and (wr, v', x). By Lemma 1, we know that T updates a_x with value v' using either a store or a compare-and-swap instruction. Suppose T changes the value of a_x using a store instruction. Let r consist of two non-transactional operations (wr, v_1, x) followed by (rd, v_2, x) of process p_2. As $\mathcal{I}$ is uninstrumented, we know that a read operation is implemented as a load, and a write as a store. Consider the trace r where $\langle \mathsf{store}\ a_x, v_1 \rangle$ instruction of process p_2 occurs immediately before $\langle \mathsf{store}\ a_x, v' \rangle$ instruction of process p_1 and the instruction

p_1	p_2
$(\triangleright, \mathsf{start})$	
...	
$(\triangleleft, \mathsf{start})$	
$(\triangleright, (\mathsf{rd}, v, x))$	
...	
$(\triangleleft, (\mathsf{rd}, v, x))$	
$(\triangleright, (\mathsf{wr}, v', x))$	
...	
$(\triangleleft, (\mathsf{wr}, v', x))$	
$(\triangleright, \mathsf{commit})$	
...	
	$(\triangleright, (\mathsf{wr}, v_1, x)$
	$\langle \mathsf{store}\ a_x, v_1 \rangle$
	$(\triangleleft, (\mathsf{wr}, v_1, x)$
$\langle \mathsf{store}\ a_x, v' \rangle$	
	$(\triangleright, (\mathsf{rd}, v_2, x)$
	$\langle \mathsf{load}\ a_x, v_2 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_2, x)$
...	
$(\triangleleft, \mathsf{commit})$	
	$(\triangleright, \mathsf{start})$
	...
	$(\triangleleft, \mathsf{commit})$
	$(\triangleright, (\mathsf{rd}, v_3, x))$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$(\triangleleft, (\mathsf{rd}, v_3, x))$

Figure 6.6: The trace r constructed in Theorem 2

$\langle \mathsf{load}\ x, v_2 \rangle$ occurs immediately after $\langle \mathsf{store}\ x, v' \rangle$. Thus, we get $v_2 = v'$. For a corresponding history of r to be parametrized opaque with respect to M, the transaction T has to be a committed transaction. After the response of the commit of T, let there be an empty transaction T' of process p_2 in r followed by a non-transactional read of x, with a load instruction $\langle \mathsf{load}\ x, v_3 \rangle$. Note that we have $v_3 = v'$. We argue that irrespective of the memory model M, there does not exist a history h corresponding to r such that h ensures opacity parametrized by M. This is because the operation (wr, v_1, x) of process p_2 can appear neither before T (as the read of v in T returns 0), nor after T (as the read after T' returns v'). Thus, an uninstrumented TM implementation has to use a cas instruction to update a variable in a transaction T, if T consists of both read and write operations to T.

□

Now, we establish a complementary result to Theorem 1. We show that

On transaction start:
$lg := g$
while $(lg \neq p)$ **do**
  $\langle \mathsf{cas}\ g, lg, p \rangle$
  $lg := g$
endwhile
return

On transaction write of x with value v':
issue a transactional read of x
if $\exists v \cdot (x, v) \in writeset(p)$ **then**
  update (x, v) to (x, v') in $writeset(p)$
  return
endif
add (x, v) to $writeset(p)$
On transaction commit:
while $writeset(p)$ is not empty
  pick and remove (x, v') from $writeset(p)$
  pick and remove (x, v) from $readset(p)$
  $\langle \mathsf{cas}\ x, v, v' \rangle$
endwhile
empty $readset(p)$
$\langle \mathsf{store}\ g, 0 \rangle$
return

On transaction read of x:
if $\exists v \cdot (x, v) \in readset(p)$ **then**
  return v
endif
$\langle \mathsf{load}\ a_x, v \rangle$
add (x, v) to $readset(p)$
return v

On transaction abort:
empty $readset(p)$
empty $writeset(p)$

Figure 6.7: A global lock based TM implementation

with an idealized memory model which relaxes the order of all operations to different variables, it is possible to obtain parametrized opacity with an uninstrumented TM implementation.

Theorem 6.3. *Given a memory model $M \notin (M_{rr} \cup M_{rw} \cup M_{wr} \cup M_{ww})$, there exists an uninstrumented TM implementation which guarantees parametrized opacity with respect to M.*

Proof. Consider an uninstrumented global lock based TM implementation $\mathcal{I}$ described in pseudo code in Figure 6.7. $\mathcal{I}$ acquires a global lock g during the start of each transaction, and releases the lock (using $\langle \mathsf{store}\ g, 0 \rangle$) immediately before the response of the commit or abort of the transaction. Moreover, for every variable x, every transaction T loads the value v_x of a_x only at the first read or write operation to x within T (if such an operation exists in T), and if T writes to x, then $\mathcal{I}$ uses a CAS instruction to update a_x in T. Consider an arbitrary trace $r \in L(\mathcal{I})$. Consider a history h corresponding to the trace r obtained by choosing the following logical points of execution of an operation within an operation trace: start operation at the successful

cas instruction, commit and abort operations at $\langle \mathsf{store}\ g, 0\rangle$ instruction, every non-transactional read at the load instruction, every non-transactional write at the store instruction, and every transactional read or write at its invocation. First, we note that the history h obtained is such that for every pair T, T' of transactions in h, either $T \prec_h T'$ or $T' \prec_h T$. Now, consider a variable x and a transaction T in h. Let $k_1 \dots k_n$ be a sequence of non-transactional operation instances to x in h, which occur between the first and last operations of the transaction T. We create a sequential history s from h by repeating the following two steps for all variables:
Step 1. Consider a non-transactional write operation k_i for some $1 \leq i \leq n$. If there is no load or cas of x in T after the store instruction with identifier k_i in r, we place all operations $k_i \dots k_n$ after the transaction T in s. For all other non-transactional write operations k_i, we place all operations $k_1 \dots k_i$ before the transaction T in s. We now have no non-transactional write operation to x between the first and last operations of T.
Step 2. For all remaining non-transactional read operations *op*, we place *op* before T in s if the load instruction corresponding to *op* precedes the update to x by T in r, and after T otherwise.

Note that as the memory model freely allows to reorder instructions to independent variables, we can move operations of different variables freely with respect to each other. Moreover, the two steps do not change the order of non-transactional operations of a process to a variable.

□

6.4.2 Instrumented TM implementations

In practice, TM implementations use instrumentation of non-transactional operations to achieve parametrized opacity. We now investigate instrumented TM implementations. Generally, a history contains more read operations than write operations. So, it is worthwhile to study whether we can achieve parametrized opacity by just instrumenting the non-transactional write operations and leaving the non-transactional read operations uninstrumented.

We first show that it is indeed possible to achieve parametrized opacity with uninstrumented reads for a class of memory models that allow to reorder read operations. The construction implements non-transactional write operations as single operation transactions.

Theorem 6.4. *There exists a TM implementation with uninstrumented reads that guarantees parametrized opacity for memory models $M \notin M_{rr}$.*

Proof. Consider a global lock based TM implementation $\mathcal{I}$ that treats transactional operations as in the proof of Theorem 3, and shown in Figure 6.7. Moreover, $\mathcal{I}$ implements a non-transactional write operation (wr, x, v) as: acquire the global lock g using cas (as in transaction start), followed by the instruction $\langle \mathsf{store}\ x, v\rangle$, followed by $\langle \mathsf{store}\ g, 0\rangle$. Intuitively, $\mathcal{I}$ treats every

non-transactional write operation as a transaction in itself. $\mathcal{I}$ implements non-transactional read operations as load instructions (no instrumentation).

Consider an arbitrary trace $r \in L(\mathcal{I})$. Note that there exists a history h corresponding to r (obtained using the logical points as in Theorem 3) such that no non-transactional write occurs between the first and last operation of a transaction, and for every pair T, T' of transactions, either $T \prec_h T'$ or $T' \prec_h T$. Consider a variable x. Given a transaction T in h, let $k_1 \dots k_n$ be the identifiers of non-transactional read operation instances in h, which occur between the first and last operation of the transaction T. We create a sequential history s from h as in Theorem 3: for all non-transactional read operations *op* to x, we place *op* before T in s if the load instruction corresponding to *op* precedes the cas instruction to x by T in r, and after T otherwise. Note that as the memory model freely allows to reorder read operations to different variables, we can move reads of different variables freely with respect to each other.

□

Note that this construction implements non-transactional write operations using a cas instruction to acquire locks. Thus, a non-transactional write operation is implemented in an expensive manner. Moreover, a non-transactional write may may take an arbitrarily long time to complete in this construction. Formally speaking, we have $(\langle \mathsf{load}\ g, 0\rangle)^* \in \mathcal{I}_N(op)$ if *op* is a write operation. We would like to create a TM implementation with inexpensive instrumentation.

Given a TM implementation $\mathcal{I}$, we say that a non-transactional write operation has *constant-time instrumentation* in $\mathcal{I}$ if there exists a constant c such that for every operation $op \in (\mathsf{wr} \times Obj \times \mathbb{N})$, every instruction sequence in $\mathcal{I}_N(op)$ has length at most c. We now prove an interesting result, where we present a TM implementation, with no instrumentation on reads and constant-time instrumentation on writes, which guarantees opacity parametrized by a memory models that allow reordering read/write followed by a read of a different variable.

Theorem 6.5. *There exists a TM implementation $\mathcal{I}$ with constant-time instrumentation of writes and no instrumentation of reads such that $\mathcal{I}$ guarantees opacity parametrized by a memory model M, where $M \notin M_{rr} \cup M_{wr}$.*

Proof. We build a TM implementation $\mathcal{I}$ which uses a global lock to execute transactions (as in Theorem 3 and 4). Moreover, $\mathcal{I}$ uses a version number per process. When a process p issues a non-transactional write operation, p increments its version number, and writes the value, the process id, and the version number using a store instruction. $\mathcal{I}$ does not use instrumentation for non-transactional read operations. Now, we prove that $\mathcal{I}$ guarantees opacity parametrized by M, where $M \notin M_{rr} \cup M_{wr}$.

Consider an arbitrary trace $r \in L(\mathcal{I})$. Consider a history h corresponding to the trace r obtained by choosing the logical points as in Theorem 3. We know that for every two transactions T, T' in h, we have $T \prec_h T'$ or $T' \prec_h T$. Consider an arbitrary transaction T in h and a variable x. Let $k_1 \dots k_m$ be

the identifiers of the non-transactional operations to x that occur between the first and last operation of T in h. Note that as $\mathcal{I}$ uses cas instruction for variables which are written in a transaction, no non-transactional write to x stores to a_x between a load and an update of a_x in T. We thus can obtain a sequential history from h by repeating the following for all variables x and for all transactions in h: for a non-transactional operation k_i to x such that operation k_i occurs between the start and end of transaction T, if the corresponding store (or load) to a_x occurs after the update of a_x in T, we place operation k_i after T in s, otherwise we place k_i before T in s. We place remaining non-transactional read operations before T. Note that it cannot happen that the corresponding store of a non-transactional write operation k_i to a_x occurs after a load and before an update to a_x.

If a process issues non-transactional reads of x and y, such that their corresponding loads occur between the updates by a transaction, we need to reorder the non-transactional reads for legality. Thus, we want $M \notin M_{rr}$. Similarly, note that if a process issues a non-transactional write of x followed by a read of y, such that the corresponding store to a_x and load of a_y occur between the updates to a_x and a_y, we need to reorder the non-transactional accesses for legality. Thus, we want $M \notin M_{wr}$. But interestingly, we built $\mathcal{I}$ in such a way that it first adds all variables to be updated in the writeset. Only then, $\mathcal{I}$ starts updating the variables using cas instruction. Hence, if a process issues a non-transactional read which loads a value updated by T, we know that every following non-transactional write can also occur after T. Thus, we do not require that $M \notin M_{rw}$. Similarly, if a process issues two non-transactional writes, then we know that either both the writes occur before T or after T. This implies that $\mathcal{I}$ guarantees parametrized opacity even for memory models which restrict the order of a read/write followed by a write to a different variable, but allow reordering read/write followed by a read to a different variable.

□

Relaxed memory models like Alpha [Sit02] are neither in M_{rr}, nor in M_{wr}. So, this construction provides an inexpensive way to guarantee opacity parametrized by memory models like Alpha. Moreover, memory models like RMO and Java are in M^d_{rr}, that is, they do not allow data dependent reads to reorder. But, these memory models do allow independent reads or control-dependent reads to reorder. This implies that if we use special synchronization for data-dependent reads, we can use the result of Theorem 5 for a vast class of memory models.[5]

[5]In practice, memory models like Java enforce strict ordering of operations marked with the *volatile* keyword. Indeed, these volatile accesses have to be treated differently than simple non-transactional accesses. For example, a volatile access may be considered as a single operation transaction.

Thread 1	Thread 2
atomic {	
$r_1 := x$	$x := 1$
$y := r_1$	$y := 2$
}	

Figure 6.8: For memory models that reorder writes to different variables, the transaction of thread 1 does not have to abort even if the update of y fails.

6.5 Discussion

We first discuss the practical implications of our work. Then, we discuss how our framework can be extended to formalize weaker notions of correctness, like single global lock atomicity.

6.5.1 Impact on practical TM implementations

The core contribution of our theoretical framework is to provide TM designers with a correctness property that exploits the inherent relaxations of the underlying memory model. For example, most memory models, like TSO, PSO, RMO [WG94], Alpha [Sit02], and Java [MPA05] allow to reorder a write operation followed by a read/write operation. On the other hand, the only TM implementation [SMAT+07] we know of which guarantees strong atomicity, indeed guarantees opacity parametrized by sequential consistency. Given that a programmer expects behavior of non-transactional operations within the scope of behaviors under the given memory model, one can build a more efficient implementation which guarantees opacity with respect to the programmer's memory model.

We now discuss the construction we used in Theorems 3 and 5. We use global locks for transactions in the construction for the sake of simplicity. But the central idea of the construction, given below, can be extended to practical lazy-versioning TM implementations which often rely on some form of two-phased locking. The idea is that for many memory models, it is not necessary for a transaction to successfully update all variables it writes, if there is a concurrent non-transactional write. For example, in Theorem 5, if a transaction observes that a non-transactional operation has written a new value, then the transaction's cas operation fails, but the transaction can still commit. We illustrate this idea in Figure 6.8. Consider a memory model which allows to reorder write operations. Let the non-transactional write operations be implemented in constant-time as in Theorem 5. We observe that even if the update of y by the transaction fails, the transaction does not need to abort. This is because, it can appear that the non-transactional write to y happens immediately after the transaction finishes. This sounds counterintuitive to the general belief that transactions must be atomic, and thus, appear to perform

their updates completely. We suggest that the non-transactional operations, although isolated from transactions from a programmer's point of view, can be used to mask updates of transactions.

Using our theoretical framework, we also observe that no matter what the given memory model M is, if a TM implementation allows a transaction to update the value of an address a_x more than once, then the TM implementation needs to instrument read operations in order of guarantee opacity parametrized by M. This is because a load of a_x (corresponding to a non-transactional operation), if sandwiched between the two updates of a_x, can observe the intermediate state of a transaction. This is also known as *dirty reads* in the literature. This implies that TM implementations, like McRT STM [SATH+06], which use eager-versioning must instrument non-transactional reads.

6.5.2 Weaker notions of correctness

Apart from the intuitive strong correctness property, parametrized opacity, which requires that transactions be isolated from other transactions and non-transactional operations, weaker notions of correctness have also been considered in the literature. Some claim that transactions should behave as global locks, thus isolating transactions from other transactions, but not from non-transactional operations. This yields the correctness property called single global lock atomicity (SGLA). We now formalize SGLA and show that SGLA is strictly weaker than parametrized opacity for every memory model M. We also show that the impossibility result given in Theorem 1 for parametrized opacity does not hold for SGLA.

SGLA. We slightly modify the framework we developed in Section 6.1 and 6.2. SGLA requires a weaker notion of sequential history, where transactions execute sequentially, but non-transactional operations may be concurrent with transactions. A history h is *transactionally sequential* if for every transaction T in h, every operation instance between the start operation instance of T and the last operation instance of T in h is either an operation instance of T, or a non-transactional operation instance.

The behavior of locks in terms of reorderings with other operations of the process depends upon the memory model. Thus, we cannot enforce a strict ordering between non-transactional operations and following or preceding transactions. Instead of defining a precedence relation between operations in a history using $\prec_h$, we now define the partial order using the memory model. Recall that when we talk of a memory model for parametrized opacity, a view does not enforce an order on start, commit, and abort operations. Whereas, when we formalize SGLA, a start operation has the memory model semantics of a lock operation, whereas commit and abort operations have the memory model semantics of an unlock operation. Thus, a memory model for SGLA can be seen as an extension of a memory model for parametrized opacity.

We say that a history h ensures *SGLA parametrized by a memory model* $M = (\tau, R)$, if there exists an ordering function $v \in R(\tau(h))$, such that, for every process $p \in P$, there exists a transactionally sequential history s such that

1. s is a permutation of $\tau(h)$
2. s respects the total order $v(p)$, and
3. every operation in legal in s.

Note that the general definition of a memory model for SGLA may allow counterintuitive results, when a non-transactional access following a transaction is allowed to precede the transaction in a view, or two processes may view two transactions to execute in different orders. Based on this intuition, in the framework for SGLA, we define that $M' = (\tau, R')$ is a *well-formed extension* of $M = (\tau, R)$ if: (i) for every history h, for every view $v \in R'(h)$, for every pair $(op, p_1, i), (op', p_2, j)$ of operation instances such that op and op' belong to $\{\mathsf{start}, \mathsf{commit}, \mathsf{abort}\}$, for all processes $p, p' \in P$, if $(i, j) \in v(p)$, then $(i, j) \in v(p')$, (ii) for every history h and for every process $p \in P$, if a non-transactional operation j of p precedes a transaction T of p in h, then there exists a $v \in R(h)$, such that for every process $p' \in P$, we have $(k, j) \in v$, where operation k is the start operation of transaction T. (iii) for every history h and for every process $p \in P$, if a non-transactional operation j of p precedes a transaction T of p in h, then there does not exist $v \in R(h)$, such that for some process $p' \in P$, we have $(k, j) \in v$, where operation k is the commit or abort operation of transaction T.

Theorem 6.6. *Given a history h and a memory model M, if h ensures opacity parametrized by M, then h ensures SGLA with respect to M', where M' is a well-formed extension of M.*

Proof idea. Consider a history h. As h ensures opacity parametrized by M, we know that there exists a view function $v \in R(h)$ such that for every process $p \in P$, there exists a sequential history s such that s satisfies the three requirements for parametrized opacity. Now, we note that a sequential history is, by definition, transactionally sequential. Moreover, as s respects the partial order $\ll \cup \prec_h$, we know that s respects the view function $v' \in R'(h)$, where the order of a transactional operation with respect to all other operations of a process is maintained in v'. We know that such a v' exists, as M' is a well-formed extension of M. This implies that h ensures SGLA with respect to M'.

□

Theorem 6.7. *Given a memory model M, there exists an uninstrumented TM implementation that guarantees SGLA parametrized by M.*

Proof idea. Consider an uninstrumented TM implementation $\mathcal{I}$ which executes every transaction using a global lock as shown in Figure 6.7 and explained in Theorem 3. Consider an arbitrary trace $r \in L(\mathcal{I})$. Note that there exists a history h corresponding to r (as explained in Theorem 3) such that for every pair T, T' of transactions, either $T \prec_h T'$ or $T' \prec_h T$. This means that h is transactionally sequential. Moreover, every operation in h is legal. Thus, h ensures SGLA parametrized by M.

□

7 Conclusion

We present a summary of the thesis, followed by an analysis of the contributions and directions for future work.

7.1 Summary

This thesis presented a formalism and a verification methodology to check the correctness of TM algorithms. The cornerstones of the thesis are the following: a specification of opacity, a language whose semantics is parametrized by relaxed memory models, structural properties of TM algorithms, and an automated verification tool. The verification technique presented in this thesis aims to bridge the gap between reasoning about the correctness of TM algorithms as described in the literature, typically in high-level pseudo-code assuming a coarse-grained atomicity and sequential consistency, and the correctness of TM algorithms on actual multiprocessors.

This thesis also extends the notion of opacity to non-transactional operations. We present a definition of opacity parametrized by a memory model. Intuitively, parametrized opacity requires two things. Firstly, transactions are isolated from other transactions and non-transactional operations. Secondly, the behavior of non-transactional operations is governed by the underlying memory model. We used our formalism to prove several results on achieving parametrized opacity with instrumented or uninstrumented TM implementations under different memory models. In particular, we show that for most memory models, parametrized opacity cannot be achieved without instrumenting non-transactional operations.

7.2 In Retrospect: Lessons Learned

This thesis started with the goal of verifying safety and liveness properties of TM algorithms.

The first important lesson is the importance of a precise formulation in approaching the problem. TM algorithms use different techniques for managing concurrency efficiently. Building a framework that captures different TM algorithms is time-consuming, but rewarding. A proper framework is indispensable to pose a precise verification problem.

The second important lesson learned in this thesis is the benefit from combining different techniques. For example, consider the TM specifications for opacity. The deterministic TM specifications are hard to construct, and even harder to prove correct. On the other hand, nondeterministic TM specifications are easy to construct and prove correct. However, to check language-inclusion efficiently, we need a deterministic TM specification. The large size of the nondeterministic TM specification does not leave determinization as a feasible option. Constructing both TM specifications, proving the correctness of one manually, and checking language equivalence to establish the equivalence of the two specifications, are the three steps that allow us to build a useful specification and prove it correct. A second important combination of techniques is the combination of manual proof techniques and model checking. While model checking is highly automated and certainly useful for a small number of threads and variables, it is often handicapped for infinite-state systems. This thesis extends the correctness of TM algorithms to an unbounded number of threads and variables using manual proof techniques.

Another important lesson learned is the importance of generality and simplicity. Formalisms and definitions should be general and simple. Our definition of TM algorithms, parametrized opacity, and memory models in Chapter 6 follow this principle. Our formalism of parametrized opacity, though simple, allows us to draw general observations. For example, our results about achieving opacity using instrumentation hold for all memory models that fall in the reordering based classification.

Another important part of the thesis is the implementation of the tool FOIL. Many optimizations were required to make FOIL work. Some of these optimizations were based on interesting insights. For example, in our scope of TM algorithms, the local variables can be divided into two sets: one set contains the variables which can be influenced, directly or indirectly, by global variables, and another set contains the variables which cannot. This segregation allows us to optimize the execution of statements involving the second set of local variables: they need not be enqueued into the queue of deferred statements, and can be directly executed. It is always helpful to design and implement a verification tool with the problem statement in mind. For example, the motivation behind the two modes of FOIL was that the counterexamples to opacity are always short. We would also like to mention our fence insertion technique. While simple, it is very useful for our purposes. Although we can-

not formally prove that the number of fences inserted is small, the technique indeed gives a practically acceptable number of fences in our experiments. The lesson learned is that simple observations may yield fruitful outcomes.

7.3 Future Directions

The thesis offers many possibilities for future work to make the verification technique more general and more automated. Moreover, while working on this thesis, we came across many interesting formalization and verification problems.

Verification of mixed transactional programs

We can use the correctness property of parametrized opacity to verify mixed transactional programs. However, we do not expect the structural properties to hold in the presence of non-transactional operations. It is an interesting research problem to find alternatives to reduce the infinite-state verification to a finite state space for mixed transactional programs.

Efficient TM implementations for parametrized opacity

TM implementations which satisfy strong atomicity [ATLM+06] satisfy opacity parametrized with respect to sequential consistency. This thesis showed that relaxing the memory model might help in creating more efficient TM implementations. One research direction is to instrument non-transactional writes and modify existing efficient TM implementations to create TM implementations which guarantee opacity parametrized with respect to weaker memory models.

Formalizing complex memory models

The semantics of RML capture different reorderings allowed in different memory models. However, they capture neither control/data dependent reordering, nor non-atomic stores. Extending RML to handle control/data dependent reordering is a straightforward extension of the current semantics: for every deferred queue, we maintain a predicate p which depends on the control structure of the program. If p holds at some point later in program execution, then the queue is considered, otherwise the queue is discarded. An interesting and challenging direction of research is to develop formal operational semantics for memory models like PowerPC [Fre05], IA-32, and IA-64 [Int06], which allow non-atomicity of stores. Also, verifying TM algorithms under software memory models like Java and C++ requires an operational semantics of software memory models. Defining an operational semantics of software memory models remains a daunting challenge due to the variety of optimizations they allow.

Verification of TM implementations

This thesis presents a verification technique for TM algorithms. It abstracts away the complications introduced in implementing the TM algorithms. Verification of TM implementations is an interesting problem. One possibility is to use dynamic tools [FF09, FFY08] to check that the sequence of memory accesses in a TM implementation satisfies the correctness property of opacity. Another alternative [Tas08] for static analysis of TM implementations is to check that a TM implementation refines a TM algorithm.

Automatic proofs for structural properties

We manually prove that the TM algorithms satisfy structural properties. It is an interesting direction to attempt to automate these proofs. Another interesting direction of research is to extend the structural properties to general concurrent systems, and search for techniques to reduce verification problems to a small number of threads and variables.

TM Specifications for Opacity Without Rollbacks

A

We construct the TM specifications for fine grained opacity under the assumption that transactions do not rollback. These TM specifications aid in understanding the more complicated TM specifications presented in Chapter 4, where transactions are allowed to rollback.

A.1 A Nondeterministic TM Specification

The set of states and the initial state of the nondeterministic TM specification for opacity without rollbacks are the same as those of the nondeterministic TM specification for opacity with rollbacks as described in Chapter 4. The transition relation is obtained using Algorithm A.1. The thread t refers to the thread taking the step, and the thread u refers to the other thread. The definition of the procedure $\mathit{ResetState}(q, t)$ is given in Section 4.2.1.

Construction

The state of a thread can be attributed to the unfinished transaction of the thread. We describe the set of runs that are produced by the TM specification. Let r be a run of the TM specification *Spec*. Let x be the unfinished transaction of a thread, and let y be the unfinished transaction of the other thread in the run r. The nondeterministic TM specification ensures the following:

Rule 1. A variable v is in the prohibited write set of x if there is a committed transaction z in r such that z serializes after x and z has a final store or a finished read of v

$nondetTMSpec(\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op)$

if $op = ((\mathsf{store}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **or** $v \in pws(t)$ **then** return $\bot$
  $ws(t) := ws(t) \cup \{v\}$; $Status(t) := \mathsf{commitsure}$
  if $v \in ws(u)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
  if $v \in rs(u)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
  if $v \in urs(u)$ **then**
    if $serp(u)$ **then** $Status(u) := \mathsf{abortsure}$
    $rp(t) := true$

if $op = ((\mathsf{load}, v), t)$ **then**
  if $urs(t) \neq \bot$ **or** $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $v \in prs(t)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **then** return $\bot$ **else** $Status(t) := \mathsf{abortsure}$
  $urs(t) := v$
  if $Status(t) = \mathsf{commitsure}$ **then** $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **then**
      if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
    else
      $wp(t) := true$
      if $serp(u)$ **then** $Status(t) := \mathsf{abortsure}$

Algorithm A.1: The nondeterministic TM specification for fine grained opacity without rollbacks

Rule 2. A variable v is in the prohibited read set of x if there is a committed transaction z in r such that z serializes after x and z has a final store of v

Rule 3. The serialization status of x is *true* in a run $r' = r \cdot op$ if

a. the serialization status of x in r is *true*, and op is not a commit or an abort of x, or

b. op is a serialize command of transaction x

Rule 4. The status of a transaction x is commitsure in a run $r' = r \cdot op$ if

a. the status of x is commitsure in r and op is not a commit

b. the status of x is finished in r and op is a store by x

Rule 5. The status of a transaction x is abortsure in a run $r' = r \cdot op$ if the status of x is not commitsure in r and one of the following holds:

nondetTMSpec($\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$, op)

if $op = (\mathsf{rfin}, t)$ **then**
  if $urs(t) = \perp$ **or** $Status(t) = \mathsf{abortsure}$ **then** return $\perp$
  $v := urs(t)$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) := \perp$
  if $wp(t)$ **then**
    if $serp(u)$ **then** return $\perp$ **else** $serp(t) := true$
  if $rp(u)$ **then**
    if $serp(t)$ **then** return $\perp$ **else** $serp(u) := true$

if $op = (\varepsilon, t)$ **then**
  if $SerStatus(t) = true$ **then** return $\perp$ **else** $SerStatus(t) := true$
  if $SerStatus(u) = false$ **then**
    if $serp(t)$ **then** return $\perp$ **else** $serp(u) := true$
    if $wp(t)$ **then** $Status(t) := \mathsf{abortsure}$

if $op = (\mathsf{commit}, t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\perp$
  if $SerStatus(t) \neq true$ **or** $urs(t) \neq \perp$ **then** return $\perp$
  if $rp(t)$ **and** $serp(u)$ **then** $Status(u) := \mathsf{abortsure}$
  if $serp(t)$ **then**
    $prs(u) := prs(u) \cup ws(t) \cup prs(t)$
    $pws(u) := pws(u) \cup ws(t) \cup rs(t) \cup pws(t)$
  ResetState(t)
  if $SerStatus(u)$ **then** $serp(t) := true$

if $op = (\mathsf{abort}, t)$ **then**
  if $SerStatus(t) = false$ **then** return $\perp$
  if $ws(t) \neq \emptyset$ **then** return $\perp$
  ResetState(t)
  if $SerStatus(u)$ **then** $serp(t) := true$
return $\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$

Continued **Algorithm A.1**

a. op is a store of a variable v by y and y serializes before x and x has an unused load of v in r

b. op is a load of a variable v by x such that y stores to v in the run r and y serializes after x

c. op is a serialize of x and y is unserialized and there exists a variable v such that y stores to v and x loads v after y stores to v

d. op is a load of a variable v by x and v is in the prohibited read set of x

Rule 6. The serialization predecessor *serp* of x is *true* in run $r' = r \cdot op$ if:

a. the serialization predecessor of x is true in r and op is not a commit or an abort of transaction y

b. op is a store of v by transaction x and y stores to v in r

c. op is a store of v by x and y has a used load of v in r

d. op is a store of v by x and the status of y is commitsure and y loads v

e. op is a load of v by x and the status of x is commitsure in r and y stores to v in r

f. op is a serialize of transaction y and the serialization status of x is false

g. op is a finish of a read by transaction x and y stores to v in r, and later x loads v in r

h. op is a finish of a read by transaction y and y loads v in r, and later x stores to v in r

Rule 7. The serialization predecessor of the transaction following x in the thread of x is *true* in a run $r' = r \cdot op$ if op is a commit or abort of x and the serialization status of y is true

Rule 8. Given a run r produced by *Spec* and an operation op of transaction x, the run $r' = r \cdot op$ is produced by *Spec* if the following hold:

a. if the status of x is abortsure, then op is an abort or a serialize

b. if op is a store of v, then x has no unused load in r and v is not in the prohibited write set of x

c. if op is a load of v, then x has no unused load in r

d. if op is a load of v and v is in the prohibited read set of x, then status of x is not commitsure

e. if op is a finish of a read, then there is an unused load of v by x and the status of x is not abortsure in r

f. if op is a commit, then the serialization status of x is true and all loads by x in r are used

g. if op is an abort, then there does not exist a variable v such that x stores to v and x does not rollback v in r

h. if op is an abort, then the serialization status of x is true

i. if op is a serialize, then the serialization status of x is false

j. if the serialization predecessor of x is true, then the serialization predecessor of y is false in r'

Correctness

Theorem A.1. *Given a history h on n threads and k variables such that h does not contain rollback instructions, h is opaque if and only if $h \in L(Spec)$.*

Proof. We say that a transaction x must serialize before a transaction y in a run r if one of the following holds:

- x and y both store to a variable v and x stores before y stores
- the serialize of x occurs before the serialize of y in r
- x stores to v and y has a used load of v, where y loads v after x stores to v
- x has a used load of v and y stores v, where x loads v before y stores to v

Note that for two unfinished transactions x, y in a run r, if y must serialize before x, then the serialization predecessor of x is true in r.

Now, we note that the TM specification *Spec* for opacity gives the largest set R of runs such that for every run r produced by the TM specification, for every transaction x in r, the following points hold (conditions C1-C6 are graphically shown in Figure 4.1):

C1. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y stores to v (from rules 1, 6.b, and 8.j)

C2. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a used load of v (from rules 1, 6.c, and 8.j)

C3a. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a load of v and y stores to a variable v' (from rules 1, 4.b, 6.d, and 8.j)

C4a. x does not load a variable v if x has written to some variable v' and there is a transaction y such that y must serialize after x and y stores to v (from rules 2, 4.b, 6.e, and 8.j)

C5. x does not finish the read of a variable v if there exists a transaction y such that y must serialize after x and y stores to v before x loads v (from rules 2, 5.b, 6.g, and 8.j)

C6. x does not finish the read of a variable v if there exists a transaction y such that y must serialize before x and y stores to v after x loads v (from rules 5.a, 6.h, and 8.j)

C7. x serializes at most once (from rules 3 and 8.i)

C8. if x is a finished transaction, then x serializes exactly once (from rules 3, 8.f, 8.h, and 8.i)

C9. x does not serialize if there exists a transaction y such that y is unserialized and y must serialize before x (from rules 6.f and 8.j)

Let h be an opaque history. As h is opaque, there is a sequential history h_s such that h_s is strictly equivalent to h. Let the transactions in the sequential history h_s be given by the sequence $x_1 \dots x_n$ of transactions. We claim that there exists a run r of the TM specification *Spec* such that h is the corresponding history of r. As h_s is strictly equivalent to h, we know that for every pair x_i, x_j of transactions in h such that $i < j$, the following are not true:

- x_i loads v after a store to v by x_j, and x_i finishes the read
- x_i is a committing transaction and x_i loads v after a store by x_j to v
- x_i and x_j have a store to v and x_i stores to v after x_j stores to v
- x_i stores to v and x_j loads v before the store to v, and the read of v is finished by x_j

As these conditions are equivalent to the conditions C1-C6, we know that there exists a run r of the TM specification *Spec*, where the order of serialization of transactions is the same as $x_1 \dots x_n$.

Conversely, let r be a run produced by the nondeterministic TM specification *Spec*. Let h be the corresponding history to the run r. We know that every transaction serializes at most once in the run. Let h_s be a sequential history such that (i) a transaction x appears before a transaction y in h_s if x must serialize before y in r, (ii) all other transactions appear in an arbitrary order later in r, and (iii) for all threads, the thread projection of h_s is equivalent to the thread projection of h. The conditions C1 - C6 guarantee that for every pair op_i, op_j of operations in h if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h_s. Note that the order of serialization in r respects the real time order of the transactions in h, that is, if a transaction x finishes before a transaction y starts, then x serializes before y in r. Thus, h_s is strictly equivalent to h. Hence, every history in $L(Spec)$ is opaque.

□

A.2 A Deterministic TM specification

The set of states and the initial state of the deterministic TM specification for opacity without rollbacks are the same as those of the deterministic TM specification for opacity with rollbacks as described in Chapter 4. The transition relation is obtained using Algorithm A.2. The thread t refers to the thread taking the step, and the thread u refers to the other thread. The definition of the procedure $ResetState(q, t)$ is given in Section 4.2.2.

$detTMSpec(\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op)$

if $op = ((\mathsf{store}, v), t)$ **then**
  if $urs(t) \neq \bot$ **or** $v \in pws(t)$ **then** return $\bot$
  $ws(t) := ws(t) \cup \{v\}$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then** $hp(t) := true$
    if $Status(u) = \mathsf{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \mathsf{started}$
  if $Status(t) = \mathsf{pending}$ **then** $Status(t) := \mathsf{commitsurepending}$
  if $Status(t) = \mathsf{started}$ **then** $Status(t) := \mathsf{commitsure}$
  if $v \in ws(u)$ **then**
    if $hp(u)$ **or** $wp(u)$ **then** return $\bot$
      if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$ **else** $sp(t) := true$
  if $v \in rs(u)$ **then**
    $sp(t) := true$
    if $wp(u)$ **then** $Status(u) := \mathsf{abortsure}$
  if $v \in urs(u)$ **then**
    $rp(t) := true$
    if $sp(u)$ **or** $hp(u)$ **or** $wp(u)$ **then** $Status(u) := \mathsf{abortsure}$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

Algorithm A.2: The deterministic TM specification for fine grained opacity without rollbacks

Correctness

We use an antichain based tool [WDHR06] to prove that for two threads and two variables, the language of the deterministic TM specification for opacity with no rollbacks is equivalent to the language of the nondeterministic counterpart. The nondeterministic TM specification has around 72'000 states, while the deterministic TM specification has around 15'000 states. The execution time for checking equivalence of the two specifications using the antichain based tool [WDHR06] on an Opteron machine with eight 2.66 GHz processors and 16 GB RAM is around 40 seconds.

Based on the manual proof of correctness for the nondeterministic TM specification and the language equivalence check of the deterministic TM specification with respect to the nondeterministic counterpart, we claim that the deterministic TM specification accepts exactly the set of opaque histories with no rollbacks.

$detTMSpec(\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op)$

if $op = ((\mathsf{load}, v), t)$ **then**
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $v \in prs(t)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **then** return $\bot$
    if $Status(t) = \mathsf{commitsurepending}$ **then** return $\bot$
    $Status(t) := \mathsf{abortsure}$
  $urs(t) := v$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then** $hp(t) := true$
    if $Status(u) = \mathsf{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \mathsf{started}$
  if $Status(t) = \mathsf{commitsure}$ **or** $Status(t) = \mathsf{commitsurepending}$ **then**
    $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **or** $Status(t) = \mathsf{commitsurepending}$ **then**
      $sp(t) := true$
    else
      $wp(t) := true$
      if $sp(u)$ **then** $Status(u) := \mathsf{abortsure}$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

continued **Algorithm A.2**

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op$)

if $op = (\mathsf{rfin}, t)$ **then**
  if $urs(t) = \perp$ **or** $Status(t) = \mathsf{abortsure}$ **then** return $\perp$
  $v := urs(t)$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) = \perp$
  if $Status(t) = \mathsf{pending}$ **then** $sp(u) := true$
  if $Status(t) = \mathsf{commitsurepending}$ **then** $sp(u) := true$
  if $rp(u)$ **then** $sp(u) := true$
  if $wp(t)$ **then**
    if $Status(u) = \mathsf{abortsure}$ **then** return $\perp$
    $sp(u) := true$
  if $sp(u)$ **and** $sp(t)$ **then** return $\perp$

if $op = (\mathsf{commit}, t)$ **then**
  if $urs(t) \neq \perp$ **then** return $\perp$
  if $hp(t)$ **then**
    if $sp(u)$ **then**
      if $urs(u) \neq \perp$ **then** $Status(u) := \mathsf{abortsure}$
    else $Status(u) := \mathsf{abortsure}$
  if $hp(t)$ **or** $rp(t)$ **or** $sp(t)$ **then**
    if $Status(u) = \mathsf{started}$ **then** $Status(u) = \mathsf{pending}$
    if $Status(u) = \mathsf{commitsure}$ **then** $Status(u) = \mathsf{commitsurepending}$
    $prs(u) := prs(u) \cup ws(t) \cup prs(t)$
    $pws(u) := pws(u) \cup ws(t) \cup rs(t) \cup pws(t)$
  if $sp(u)$ **and** $sp(t)$ **then** return $\perp$
  ResetState(t)

if $op = (\mathsf{abort}, t)$ **then**
  if $ws(t) \neq \emptyset$ **then** return $\perp$
  ResetState(t)
return $\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$

continued **Algorithm A.2**

Bibliography

[AAK+05] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[ABHI08] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 63–74, 2008.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, 1996.

[AQR+04] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *International Conference on Computer Aided Verification*, pages 484–487, 2004.

[ATLM+06] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37. ACM, 2006.

[BA08] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78. ACM, 2008.

[BAM06] S. Burckhardt, R. Alur, and M. M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *International Conference on Computer Aided Verification*, pages 489–502. Springer, 2006.

[BAM07] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory

models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–21. ACM, 2007.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, pages 505–525, 2007.

[BMS08] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying compiler transformations for concurrent programs. Technical Report MSR-TR-2008-171, Microsoft Research, 2008.

[BP09] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 392–403, 2009.

[BR02] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[CGLM06] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *International Conference on Computer Aided Verification*, pages 475–488, 2006.

[CPZ08] A. Cohen, A. Pnueli, and L. D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *International Conference on Computer Aided Verification*, pages 121–134. Springer, 2008.

[DFL+06] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006.

[DLMN09] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[DS09] L. Dalessandro and M. Scott. Strong isolation is a weak idea. In *ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[EQT07] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[ES96] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, pages 105–131, 1996.

[ETQ05] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: Verifying concurrent programs by runtime refinement-violation detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–37, 2005.

[FF04] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[FF09] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[FFY08] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[FH05] Michael J. Flynn and Patrick Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, pages 16–31, 2005.

[FH07] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 2007.

[FLM03] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing*, pages 285–294, 2003.

[Fre05] B. Frey. *PowerPC Architecture Book, v2.02*. International Business Machies Corporation, 2005.

[GHJS08] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 372–382. ACM, 2008.

[GHS08] R. Guerraoui, T. A. Henzinger, and V. Singh. Nondeterminism and completeness in model checking transactional memories. In

International Conference on Concurrency Theory, pages 21–35. Springer, 2008.

[GHS09] R. Guerraoui, T. A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In *International Conference on Computer Aided Verification*, pages 321–336, 2009.

[GK08] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184. ACM, 2008.

[GK09] R. Guerraoui and M. Kapałka. The semantics of progress in lock-based transactional memory. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 404–415, 2009.

[GMP06] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *Workshop on Memory System Performance and Correctness*, pages 62–69, 2006.

[GYS04] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or Not QB: An efficient execution verification tool for memory orderings. In *International Conference on Computer Aided Verification*, pages 401–413. Springer, 2004.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 124–149, 1991.

[HF03] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[HK08] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 2008.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *IEEE International Parallel and Distributed Processing Symposium*, pages 522–529. IEEE Computer Society, 2003.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101. ACM, 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300. ACM, 1993.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, pages 279–295, 1997.

[HQR99] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared memory multiprocessor systems. In *International Conference on Computer Aided Verification*, pages 301–315. Springer, 1999.

[HW90] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, 1990.

[HWC+04] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture*, pages 102–113, 2004.

[Int06] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, 2006.

[KCH+06] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 209–220, 2006.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, pages 690–691, 1979.

[LP01] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, pages 824–833, 2001.

[LR07] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[MBL06] M. M. K. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[MBM+06] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[MBS+08] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *ACM Symposium on Parallel Algorithms*, pages 314–325, 2008.

[MG08] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 51–62. ACM, 2008.

[MHC+06] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 134–143, 2006.

[MM07] V. J. Marathe and M. Moir. Efficient nonblocking software transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 136–137, 2007.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.

[MPA05] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 378–391. ACM, 2005.

[MQB+08] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[QW04] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24, 2004.

[RFF06] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298, 2006.

[RG02] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, 2002.

[RHL05] R. Rajwar, M. Herlihy, and K. K. Lai. Virtualizing transactional memory. In *International Symposium on Computer Architecture*, pages 494–505, 2005.

[SATH+06] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM, 2006.

[Sco06] M. L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*, 2006.

[SDMS08] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *International Conference on Principles of Distributed Systems*, pages 275–294, 2008.

[Sit02] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 2002.

[SJMvP07] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, New York, NY, USA, 2007. ACM.

[SMAT+07] T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88. ACM, 2007.

[SMDS07] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 338–339, 2007.

[SSN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 379–391, 2009.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[Str82] Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, pages 121–141, 1982.

[Tas08] S. Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, 2008.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–136, 2006.

[WDHR06] M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.

[Wei89] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, pages 249–283, 1989.

[WG94] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., 1994.

Vasu Singh

Ecole Polytechnique Federale de Lausanne
IC - MTC
Station 14
Lausanne 1015, Switzerland

Phone: (21) 693 1220
Fax: (21) 693 7540
Email: vasu.singh@epfl.ch

Personal

Born on 4 February 1982

Indian Citizen

Education

B.Tech Electrical Engineering, Indian Institute of Technology, 2004

M.Tech Electrical Engineering, Indian Institute of Technology, 2004

Honors and Awards

Among top two students in state level Mathematics Olympiad (1998)

Among top 0.2% students in Joint Entrance Exam to IITs (1999)

Received the IIT Bombay Heritage Scholarship (2001)

Achieved the IIT Bombay Academic Award for excellent academic performance (2002)

Received EPFL Doctoral Fellowship (2004)

Publications

Refereed Conference Papers

Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh, "Preventing versus Curing: Avoiding Conflicts in Transactional Memories", *Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC) 2009.*

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh, "Software Transactional Memory on Relaxed Memory Models", *International Conference on Computer Aided Verification (CAV) 2009.*

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh, "Permissiveness in Transactional Memories", *International Symposium on Distributed Computing (DISC) 2008.*

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh, "Nondeterminism and Completeness in Model Checking Transactional Memories", *International Conference on Concurrency Theory (CONCUR) 2008.*

Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh, "Model Checking Transactional Memories", *ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI) 2008.*

Vasu Singh and Raghunath Shevgaonkar, "Effects of phase errors and shot noise on asynchronous coherent optical CDMA", *IEEE Optical Fiber Communications Conference (OFC) 2004.*

Workshop Papers

Parametrized Opacity, *DISC Workshop on What Theory for Transactional Memories 2009*

Generalizing the Notion of Correctness, *CAV Workshop on Exploiting Concurrency Efficiently and Correctly* – $(EC)^2$ *2009.*

Model Checking Transactional Memories, *CAV Workshop on Exploiting Concurrency Efficiently and Correctly* – $(EC)^2$ *2008.*

Invited Papers

Dirk Beyer, Thomas A. Henzinger, and Vasu Singh, "Algorithms for Interface Synthesis", *International Conference on Computer Aided Verification (CAV) 2007.*

Invited Talks

Parametrized Opacity, *Dagstuhl Seminar on Design and Validation of Concurrent Systems 2009, Dagstuhl.*

Model Checking Transactional Memories, *Alpine Verification Meeting 2008, Semmering.*

Completeness and Nondeterminism in Model Checking Transactional Memories, *Microsoft Research 2008, Redmond.*