Static Methods for Checking Correctness of Programs on Relaxed Memory Systems

by

İsmail KURU

A Thesis Submitted to the

Graduate School of Engineering

in Partial Fulfillment of the Requirements for

the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

January 27 , 2015

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

İsmail KURU

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

[illegible]

Assoc. Prof. Serdar Taşıran (Advisor)

[illegible]

Prof. Atilla Gürsoy

[illegible]

Prof. Alper Demir

Date: ______________________

To me

ABSTRACT

Concurrency is everywhere. Many internet, real-time and embedded applications are most naturally designed using concurrency. However, programming conccurent software is complex, error-prone task. Because of inherent non-determinism, it is difficult for the programmer to understand the effect of his or her program. The Swing documentation states :

"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. In general, they aren't necessary for strictly GUI work, such as component properties.", [97]. Another note in [98] states

"Deadlock between threads competing for the same set of locks is the hardest problem to solve in any threaded program. It is a hard enough problem, in fact, that we will not solve it or even attempt to solve it. .... Nontheless, a close examination of the source code is the only option available to determine if deadlock is a possibility ...".

As a result we can state that concurrency bugs are difficult to detect, reproduce and diagnose using conventional software engineering techniques such as code review and test-based techniques developed developed for sequential programs.

Generally, language and concurrency library implementers are the ones who have

difficulty with concurrency. This is mainly due to hardness of resolving the internals of interaction of hardware and software to build reliable software. S.Adve and H.-J.Boehm state:

”Part of challenge for hardware architects was the lack of clear memory models at the programming language level. It was unclear what programmers expect hardware to do.”, [31].

All these problems make formalising HW/SW interaction crucially important because formal models can express aspects of non-determinism that are not apperent. The main product of this formalisation effort has been made on memory models; e.g sequential consistency (SC) in traditional shared memory, serializability in transactional memory. They are important because they define and formalize the expected behaviours of a program. They define safety guarantees for programs with providing contracts such as data-race-freeness (DRF) in traditional shared memory, conflict free in transactional memory between hardware and software; e.g. data-race-free program under SC model.

Mostly due to performance need, we do not need to enforce sequential consistency or strict serializability in transactional memory and slow down the processors for some applications. To optimize memory access, some memory models may introduce relaxing atomicity of writes, allow instruction reordering or ignore write-after-read conflicts. In this case, thinking in terms of these strong models, e.g. SC or strict serializability, when designing concurrent data structures or algorithms cannot be reliable to produce correct programs. As a result, weaker memory models are proposed.

In this thesis, we propose static verification methods for programs running on two weak memory models such as total-store-order (TSO) and snap-shot-isolation (SI) models respectively from traditional shared memory and transactional memory domains.

In chapter 2, we present a static verification approach for programs running under snapshot isolation (SI) and similar relaxed transactional semantics. Relaxed conflict detection schemes such as snapshot isolation (SI) are used widely. Under SI, transactions are no longer guaranteed to be serializable, and the simplicity of reasoning sequentially within a transaction is lost. In this thesis, we present an approach for statically verifying properties of transactional programs operating under SI. Differently from earlier work [23], we handle transactional programs even when they are designed not to be serializable.

We present a source-to-source transformation which augments the program with an encoding of the SI semantics. Verifying the resulting program with transformed user annotations and specifications is equivalent to verifying the original transactional program running under SI – a fact we prove formally. Our encoding preserves the modularity and scalability of VCC's verification approach. We applied our method successfully to benchmark programs from the transactional memory literature [95].

In chapter 3, we propose a proof method approach for refining programs to make them easy to reason under x86-TSO model. In this approach we introduce store buffers explicitly into source and try to refine program via applying TSO specific sound refinement steps [94]. We put the store buffers into source and abstract the TSO memory flush operation explicitly in the source. In this approach, we try to observe whether specific type of programming patterns result in specific type of refinement pattern. We show applicability of the approach with mechanized refinement of mechanized refinement of non-blocking Spin-Lock and acquire/release pattern Send-Receive in Section 3.4.7 by approach 3.4.

ÖZETÇE

Koşut-zamanlılık heryerde bulunmaktadır. Birçok internet, gerçek zamanlı ve gömülü yazılımlar doğası gereği koşut zamanlılık kullanıllarak tasarlanır. Fakat, koşut zamanlı yazılımları programlamak karmaşık ve hataya meyilli eylemlerdir. Doğasında bulunan tutarsızlık gereği, programcının yazdığı programın nasıl bir davranış göstereceğini anlaması zordur. Bir Swing dokümantasyonu der ki :

"Eğer kullanmaktan sakınabiliyorsanız, akışları kullanmaktan sakının. Akışları kullanması zor olabiliyor ve programınızın hata ayıklanma işlemini zorlaştırabiliyor. Genellikle, grafiksel bileşen özellikleri gibi kullanıcı arayüz çalışmalarında kesinlikle gerekli olmayabilirler.", [97]. Diğer bir dökümanda [98] ise :

"Aynı kümedeki kilitler için yarışan akışların neden olduğu kilitlenme akışlı programlarda çözülmesi en zor problemdir. O kadar zordur ki çözmeye bile teşebbüs etmeyiz. ... Buna rağmen, kilitlenmenin var olup olmadığını belirleyebilmek için tek seçenek kaynak kodun sıkı bir incelenmesidir...".

Sonuç olarak koşut-zamanlılıktan kaynaklananan hataları tespit etmek ve tekrar üretebilmek kaynak kodu gözden geçirme, tek akışlı bir program için geliştirilen sınama teknikleri gibi klasik yazılım mühendisliği teknikleri ile zordur.

Çoğunlukla programlama dili gercekleştiricileri ve koşut-zamanlı kütüphane gerçekleştiricileri

koşut-zamanlılık ile zorluk çekenlerdir. Bu temel olarak yazılım ve donanımın ilişkisini çözümlemenin zorluğundan kaynaklanır. S.Adve ve H-J.Boehm der ki:

”Donanım mimarları için zorluğun bir parçası programlama dilleri seviyesinde açık bir bellek modelinin olmamasıdır. Programcının donanımdan ne bekleyeceğini muallakta bırakmaktadır.”, [31].

Bütün bu problemler, yazılım ve donanım ilişkisini formal bir şekilde belirtmeyi önemli kılmaktadır çünkü formal modeller açıkça belli olmayan tutarsızlıkları ifade edebilir. Bu formal tanımlama çabasının ana ürünü bellek modellerinin üzerinde gerçekleştirildi, örneklerden biri geleneksel paylaşımlı belleklerde ardışık tutarlılık, diğeri işlembilgisel belleklerde serileşebilmedir. Bu modelleme çalışmaları bir programın beklenen davranışlarını formal olarak tanımladığı için önemlidir. Bu modelleme çalışmaları geleneksel paylaşımlı belleklerde yarış-içermeyen (DRF) ve işlembilgisel belleklerde çelişki-içermeyen gibi sözleşmeler sağalyarak güvenilirlik garantileri tanımlarlar.

Çoğunlukla daha çok başarıma ihtiyaç duyulduğundan ardışık tutarlılık veya sıkı serileşebilme gibi sözleşmeleri zorunlu kılarak yazılım uygulamalar için işlemcileri yavaşlatmaya gerek yoktur. Bellek erişimini optimize etmek için, bazı bellek modelleri yazmaların atomikliğini gevşetebilir, makina komutlarının yerlerini değiştirebilir veya okumalardan sonra gerçekleşen yazmaların yarattığı çelişkileri göz ardı edebilir. Bu durumda, algoritma veya koşut-zamanlı veri yapısı tasarlarken bu tip güçlü modeller, örneğin, ardışık tutarlılık (SC) veya sıkı serileşebilme, açısından düşünmek doğru program üretmek için güvenilir olmayabilir. Sonuç olarak daha gevşek memory modelleri sunulmuştur.

Bu tezde sırası ile geleneksel paylaşımlı bellek alanında ve işlembilgisel bellek alanında iki gevşetilmis bellek modeli olan yazmaların-tam-sıralaması (TSO) ve bellek-kopyası-soyutlanma (SI) üzerinde çalışan programlar için durağan doğrulama yöntemi sunuyoruz.

Kısım 2 içersinde, bellek-kopyası-soyutlama ve benzeri gevşek bellek modelleri üzerinde çalıştırılan programların doğrulanması için durağan doğrulama yöntemi sunuyoruz. Bellek-kopyası-soyutlama ve benzeri gevşek bellek modeller oldukça fazla kullanılan modellerdir. SI altında çalışan işlemler için ardışık olarak çalışıyormuş gibi yorum yapabilme imkanı kaybolur, bu işlemler artık serileşebilir işlemler değildir. Bu tezde önceki yapılan işten [23] farklı olarak, serileşemez programları da kotarıyoruz.

Bir kaynak kodu diğer kaynak koda SI semantiğini içerecek şekilde dönüştürmeyi sunuyoruz. Dönüştürülmüş programın kullanıcı tarafından kaynak koda eklenen ek sözleşme dipnotları ile birlikte doğrulanması orjinal programın SI altında doğrulanmasına denktir. Bizim SI semantiği için önerdiğimiz dönüştürme tekniği VCC'nin modülerliğini ve ölçeklenirliğini korumaktadır. Yöntemimizi işlembilgisel bellek literatüründeki deneme programlarına uyguladık [95].

Kısım 3 içersinde, x86-TSO bellek modelleri üzerinde çalışan programlar ile ilgili kolay yorumlamayı sağlamak için program iyileştirmeyi sağlayan kanıt yöntemi sunuyoruz. 3.4 bölümünde kaynak kodun içine TSO semantiğini ek kod olarak yerleştiriyoruz ve programın yorumlanmasını kolaylaştırana kadar TSO'ya özgü iyileştirme adımlarını uyguluyoruz, [94]. TSO'ya özgü işlemci yerel yazma tamponlarını ve belleğe yazma işlemini soyutlanmış halını kaynak kodun içine yerleştiriyoruz. Bu yöntemde belirli proglamlama kalıplarının belirli iyileştirme kalıpları ile örtüşüp örtüşmediğini gözlemliyoruz. Bu yöntemin kullanılabilir olduğunu mekanik iyileştirmelerini engelsiz kilitleme yöntemi Spin-Lock ve acquire/release programlama kalıbı gösteren Send-Receive örneği için 3.4.7 kısımında 3.4 ile gösterdik.

ACKNOWLEDGMENTS

Koç University. I would like to express my gratitude to my graduate study advisor, Serdar Taşıran and collaborators; Süha Orhun Mutluergil, Erdal Mutlu, Burcu Özkan, Hassan Matar, Erkut Demirhan, Maryam Dabaghchian, Ömer Subaşı for their selfless, long time collaboration and providing nice research environment at Koç Universtiy Research Center for Multicore Software Engineering which is my home. I would like to thank to old members of room 230; Ayşenur Kibar, dil Arşık and Can Bekar. I would like to thank to the other members of my thesis committee, Atilla Gürsoy, Alper Demir for critical reading of this thesis.

Sponsors. I am grateful to Microsoft Research for supporting my research with Microsoft Research PhD. scholarship. I would like to thank to The Scientific and Technical Research Council of Turkey (TÜBİTAK) for supporting my research.

Seattle. I would like to thank to Dan Grossman and all Programming Language and Software Engineering (PLSE) group members for weekly programming languages seminars at University of Washington. I would like to thank to Shaz Qadeer for his collaboration during my research visits to Microsoft Research Redmond, Seattle. I would like to thank to Kıvanç Muşlu and his family for their hospitality at Seattle.

Cambridge,UK. I would like to thank to Ali Sezgin from University of Cambridge for his collaboration and contributions to my research. I would like to thank to all members of Programming Principles and Tools group specifically to Ben Hall and Matthew Parkinson at Microsoft Research Cambridge for advising, discussion meetings and collaborations during my intership. I would like to thank to my friends, Klaus Gleissenthall, Nik Sultana, from Cambridge for making life in Cambridge more enjoyable.

Finally. I must give immense gratitute to Matthew Parkinson for accepting me to advise, sharing his experience and showing patience while teaching and collaborating on my new research direction.

TABLE OF CONTENTS

List of Figures xviii

Nomenclature xxi

Chapter 1: Introduction 1

1.1 Weak Memory Models 1

1.1.1 Weakness In Shared Memory 2

1.1.2 Weakness In Transactional World 6

1.2 Organization of the Thesis 10

Chapter 2: Static Verification of Relaxed Transactional Semantics: A Case Study with VCC 11

2.1 Introduction to Relaxed Transactional Semantics and Correctness Conditions on them 12

2.2 Problem with an Labyrinth Example 15

2.3 Introduction to our Approach 19

2.4 Related Work 21

2.5 Our Approach 24

2.5.1 Preliminiaries: Transactional Programs 25

2.5.2 SI and other Relaxed Conflict Detection 29

2.5.3 Concurrency, VCC and Modular Verification: An Overview . . 33

2.5.4 Source-to-source Transformation for Simulating SI 36

2.5.5 The Encoding: Discussion 45

2.6 Experiments 47

Chapter 3: QED for Programs Running on x86-TSO Memory Model 55

3.1 A Reduction/Abstraction Concurrency Verification Method: QED . . 56

3.1.1 Movers in QED 56

3.1.2 Reduction/Abstraction 57

3.2 Total Store Order Memory Model 59

3.3 A Survey on Verification of x86-TSO 59

3.4 Using Atomicity as a Refinement Method for Verifying Programs Running on x86-TSO 64

3.4.1 Formal Framework 64

3.4.2 Preliminaries: ActionPL, Reduction, Abstraction 65

3.4.3 Modeling TSO 68

3.4.4 QED Specifications: Proof Rules Tailored For TSO-Encoded Programs, Transformation 74

3.4.5 Discussion and Comparison with a Specialized Logic 83

3.4.6 Experiments 84

3.4.7 Send - Receive 84

3.4.8 Spin Lock 93

Chapter 4: Conclusion **114**

Bibliography **118**

Vita **130**

LIST OF FIGURES

1.1 Observing SC behaviours on a simple program 2

2.1 Outline for FindRoute code and specification. 50

2.2 StringBuffer pool before code transformation. 51

2.3 String Buffer code after transformation. 52

2.4 The insertion operation of a sorted linked list. 53

2.5 Sorted linked list and a write-after-read conflict. 54

3.1 ActionPL syntax. 65

3.2 SimPL Syntax 71

3.3 Additional variables needed for the TSO encoding 72

3.4 Procedures modeling TSO operational semantics. 73

3.5 The code for send-receive without drain* 85

3.6 The code for send-receive with drainHead*. TSOfying operation. . . . 87

3.7 Merging consecutive drainHead*. 88

3.8 Abstraction on drainHead* with isBufferEmpty. 89

3.9 Abstraction and Reduction to transform drainHead* to if*isAtHeadAndDrain. 90

3.10 Peeling out the memory flush at index ti2 from drain actions earlier than tail index of buffer. 91
3.11 Detaching drainIfHeadLessThan from isAtHeadAndDrain 92
3.12 Final transformed source for send-receive under TSO. 93
3.13 Send method transformed with TSO specific rules. Movers annotated. 94
3.14 Send method after reduction. 95
3.15 Spin Lock implementation without any TSO encoding. 97
3.16 Compare and Swap implementation with QED. 97
3.17 Spin Lock implementation with TSO encoding. 98
3.18 Swap local actions with drainHead*. 99
3.19 Swap non-blocking local left actions with drainHead*. $r1$ is swapped with $r0$ 100
3.20 Merge drainHead*. Pair of drainHead* statements in Figure 3.19, $(r0, r2)$, $(a8, a6)$, $(a5.1, a5.3)$ $(a2, a4)$, can be merged into one drainHead* 101
3.21 Peel away command on body of $a4$ block. 102
3.22 Swap $a3$ with $a5.1$ in Figure 3.21 in 1iter or >=2iter functions. Swap is followed by merging drainHead*. 103
3.23 Swap $a3$ with $a6$ in 0iter methods. This is valid local, non-blocking swap. Swap is followed by merging drainHead*. 104
3.24 $a3$ and $a7$ form an atomic block because $a3$ is a right mover. This atomic block forms [assume false]. 105

3.25 $a5.2$ in 1iter method in Figure 3.22 becomes *skip* because of $a3$ in that method. Insert *assert* $isBufferEmpty()$ as a first statement into block $a6$. 106

3.26 Refined 1*iter* procedure after transformation. 107

3.27 Abstraction on drainHead* with *assert* $isBufferEmpty()$. 108

3.28 Reducing loop body of block $a4$ to an atomic action. 110

3.29 Introducing ownership variable and invariants for lock. 111

3.30 drainHead*; atomic{ *StmtList* } pattern. 112

3.31 Final pattern of Spin Lock under TSO. 113

NOMENCLATURE

TSO	Total Store Order
PL	Programming Language
RMO	Relaxed Memory Model
VCC	Verifying Concurrent C Tool
SC	Sequential Consistency
TRF	Triangular Race Free
DRF	Data Race Free
PC	Personal Computer
$StmtList$	Statement List
TM	Transactional Memory
SI	Snapshot Isolation
HW	Hardware
SW	Software
JVM	Java Virtual Machine
$FIFO$	First In First Out
$PSpace$	Polynomial Space

Chapter 1

INTRODUCTION

In this section, we give a lightweight introduction to concepts that we use throughout the thesis script. Mainly, we mention concepts related with concurrency, relaxed memory models, specifically in Transactional Models and Shared Memory Models.

1.1 Weak Memory Models

Memory Models are crucial for providing reliability of softwares. They are important because building proof methods to verify a software requires understanding the underlying memory model, which basically affects the behaviour of running software.

Througout the thesis, we focus on strange behaviours of programs due to some relaxation implemented inside memory models of architectures. We basically, divide weak properties that memory models show in Transactional and Traditional Shared Memory domains. We know that there is not any strict line in between those two domains, Transactional Memory(TM) can also be designed for shared memory architectures, however we can follow this separation dicipline because TM notions and TM related reliability guarantees come from the world of databse transactions.

INIT	$Program_0$	$Program_1$
`x,y =0;`	`w0:  x := 1;`	`w1:  y := 1;`
	`r0:  tmp0:= y;`	`r1:  tmp1:= x;`

Figure 1.1: Obserrving SC behaviours on a simple program

1.1.1 Weakness In Shared Memory

We follow the way for setting context in traditional shared memory reliability as it is followed in [65].

Sequential Consistency

Sequential Consistency(SC) is basically a behaviour of a program that one expects when she writes a concurrent program. L. Lamport defines it in [26] as:

” ...The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” A simple program in Figure 1.1 seems very easy to reason about at first, even in concurrent setting. Assume that *tmpX* variable are registers per processors or thread local variables. $w1$ and $w0$ are labels where memory store actions are performed. $r0$ and $r1$ are labels where reading from memory to a processor local variable is performed. Is it possible to observe $tmp0 = 0$ and $tmp1 = 0$ under SC ? When we assume SC as an execution model then the answer is no.

Weaker than Sequential Consistency

We are in multi-core era and exploiting the performance of multi-core architecture is crucial. More performace is always a need and architectural designs are so important in that sense. These designs may introduce some features that SC can not exhibit. [26]:

" For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied on upon to produce correctly executing programs".

Our simple program can show a behaviour, or an execution that is not sequentially consistent. A feature, reordering store-load or write-read pairs, in x86 machine can lead your program to have an execution showing a non-sequentially-consistent behaviour. $r0$ may occur before $w1$ and as a result $tmp0$ is assigned with initial value of y,which is 0. Similiarly, $r0$ may occure before $w1$ and as a result $tmp1$ is assigned with 0 and non-sequentially-consistent behaviour shows up.

Relaxed or weak memory model means that a program can show more behaviours on relaxed or weak memory model than SC model. There are many ways of introducing relaxation into memory model. One way of introducing relaxation into a memory model is instruction reordering [29, 27], reordering of reads and writes. Another way of doing that is relaxing atomicity of writes [29, 27]. In this type of relaxation, a write operation done by a processor can be delayed via store-buffer [28]. This asynchrony between a write operation and its commit to memory may lead other processors to see most recent value in the store-buffer later. We can easily conclude that understanding the memory model is crucial to reason about program on a specific memory model.

Contracts between HW and SW

Mostly language implementers and concurrent library implementers need resolving the internals of interaction of HW and SW to build reliable implementations. They need to follow a contract between HW and SW. According to S.Adve and M.Hill reliable interaction relation between SW and HW can be set when SW follows a set of formally specified constraints, and we expect HW to appear SC at least SW follows these constraints [32].

Preserving SC is violated when two or more processors interact through memory operations on common variables [32]. In Figure 1.1, there are two shared variables, x and y, between two processes communicating via these shared variables. For example, $r0$ from processor 0 running $Program_0$ and $w1$ from processor 1 running $Program_1$ exhibits a data race. Data-Race-Free(DRF) [32] guarantee states that program ensures DRF if in each sequential consistent execution, for each conflicting actions there is a way to enforce a relation between those two conflicting pairs. To satisfy the constraints imposed by DRF, processors needs to follow a way to related conflicting actions which is generally done via using some synchronization mechanism.

Well-know synchronization mechanisms can be classified as Lock-Based Synchronization and Lock-Free Synchronization. Locks protects some memory location via enabling and disabling a certain flag. If one processor already set a flag then other processor which is trying to set this flag needs to wait until the flag is reset by the processor who set and keep the right to accessing the flag. This ordering property enables locks to ensure a certain execution order on the accesses they protect which results in ordering conflicting accesses properly. For example, lck can be used to order accesses to x in our simple program at $r0$ and $w1$ by different processors. It is well known phenomenon that acquisition and releasing of a lock is vey expensive in addition to letting other processor stall while waiting lock to be released.

Java Concurrency In Practice states an important drawback of using locking as a synchronisation method:

"The JVM can implement blocking either via spin-waiting (repeatedly trying to acquire the lock until it succeeds) or by suspending the blocked thread through the operating system. Which is more efficient depends on the relationship between context switch overhead and the time until the lock becomes available; spin-waiting is preferable for short waits and suspension is preferable for long waits. Some JVMs choose between the two adaptively based on profiling data of past wait times, but most just suspend threads waiting for a lock".

And it follows with a statment on optimization uncontended synchronisation:

"Don't worry excessively about the cost of uncontended synchronization. The basic mechanism is already quite fast, and JVMs can perform additional optimizations that further reduce or eliminate the cost. Instead, focus optimization efforts on areas where lock contention actually occurs".

Lock-Free-Synchronisation is mechanism of providing ordering of accesses to a memory location without using expensive locks. Protocols for ordering is provided by instructions provided by architecture, such as barriers and synchronization instructions, such as compare-and-swap, test-and-set. Data structures or algorithms using lock-free protocols is a fundemental issue in this thesis. Unlike lock-based synchronization protocols which provide ordering to satisfy DRF constraints, lock-free protocols do not arbitrate data races. Instead, they prevent reordering of write-read or store-load pairs. For example, if we put barrier in between $r0$ and $w0$ they can not be reordered, similiarly for $r1$ and $w1$. This prevention of reordering can be imposed to have more efficient code intentionally by architecture or compiler however barriers prevents these reorderings. Things get complicated when we need to design and reason about programs running on such architectures. We can conclude that understanding the underlying memory models is important in design and correctness of programs running on top of these models. Providing proof methods for weak memory in shared memory world is one of the main motivations of this thesis.

1.1.2 Weakness In Transactional World

Butler Lampson states that "Every good idea in operating systems came from the database community" [99]. This statement can be instantiated with example of the adapting Transactional notion in database world to systems world as Transactional

Memory (TM) [1]. TM would be a powerful tool to enable the development and composable use of data structures. Transactional interface hides the underlying complication of design and implementation of concurrent data structures. Besides, understanding semantics of constructs in TM interface is crucial for reasoning on programs using these constructs. Correctness properties, execution models for TM which are already defined in database world and some new models are defined in the following sections.

Strong Consistency, Linearizability, Serializability and Strict Serializability in Transactions

We take the following definitions in italics from [33].

Serializability " ... Much work on databases and distributed systems uses serializability as the basic correctness condition for concurrent computations.l In this model, a transaction is a thread of control that applies a finite sequence of primitive operations to a set of objects shared with other transactions. A history is serializable if it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving. A (partial) precedence order can be defined on non-overlapping pairs of transactions in the obvious way".

The context of serializability is typically that of databases. In this, a bunch of statements are grouped together and form a transaction. To make distinction concrete between linearasibility and serialisability, community usually characterize serialisabliity as having multiple operations (reads and writes) over multiple items.

If these operations generates a execution that is equal to some serial execution then serialasibility holds.

Strict Serializability "... An execution history is strictly serializable if the transactions order in the sequential history is compatible with their precedence order".

This is the strictest memory model. It is same as sequential consistency, except that it absolutely guarantees that any read sees the most recent write across all processors. One way to imagine it is that there is a global clock for strict consistency. Every operation occurs at some point on this global clock and all writes are immediately accessible to every other processor. In other words, this behaves like sequential consistency on a uniprocessor.

For example, assume that we have two transactions, T_1 and T_2. T_1 starts, writes to variable x and commits. T_2 starts, read x and commits. If we want to enforce strict serializability then we place T_1 before T_2. If we would like to enfore serialisability, not the strict one, then we can reorde T_2 before T_1.

Linearizability "...Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object. Nevertheless, this single-operation restriction has far-reaching practical and formal consequences, giving linearizable computations a different flavor from their serializable counterparts. An immediate practical consequence is that concurrency control mechanisms appropriate for serializability are typically inappropriate for linearizability because they introduce unnecessary overhead and place unnecessary restrictions on concurrency...".

Linearisability is a guarantee about single operation on a single item. It has real time enforcement on set of single operatiosn on a single item. In linearisibility, all writes are seen atomically. Once a write is completed then all reads following this write, all reads having real time later than this write operation, see the value of item commited by this write operation. You can also see linearisability as Sequential Consistency with an added caveat that there is a time stamp associated with each operation and that every thread sees the operations in the order given by time stamps.

Conflict, Eager, Lazy Conflict Detection

Read Set, Write Set All memory references which were read by transaction are called read set and all memory references which were written by it are called a write set.

Transaction Validation Backward Validation is validation of commited transaction against active transaction and restarts itself if a violation is detected. Forward Validation is validation of a active transaction against a commited transaction and restarts if a violation is detected.

Conflict is an overlap between the write set (data written) of one transaction and the write set or read set (data read) of other concurrent transactions. Conflict detection is called eager (Forward Validation) if detects offending loads stores immediately and lazy (Backward validation) if it defers detection until later, on commit time.

Relaxed Transactions: Snapshot Isolation(SI) and Simillar Transactional Models

Snapshot Isolation is gurantee where all read operations in all transactions reads from a consistent state of database, snapsot. Basically, a transaction reads from memory in TM world, or more generally in Transactional jargon commited values, when it starts and does its local updates and can successfully commit updates if no update it did conflict with any other concurrent updates made since that last consistent state read when when transaction started.

1.2 Organization of the Thesis

In this thesis, we present proof methods that we build for checking weakness properties in transactional and traditional shared memory domains.

In Chapter 2, we present a proof method built on top of VCC tool which is a static verification tool for analysing concurrent C programs. This proof method enables proving safety invariants for programs running on SI or similar memory models. We show experimental proofs of sample programs that are already assumed to be working correctly on these kind of memory models. This chapter mainly covers our [95].

In Chapter 3, we present a proof method approach built on top of QED [24] which is a static proof refinement tool using atomicity as a tool for concurrent programs. In Section 3.4, we present the proof method that applies specific proof patterns to a specific programming paradigms, (e.g. acquire/release, non-blocking mutual exclusion) to get a refined version for TSO reasoning. This approach does transformation of programs which include store-buffer operations explicitly in source code [94].

Chapter 2

STATIC VERIFICATION OF RELAXED TRANSACTIONAL SEMANTICS: A CASE STUDY WITH VCC

We present a static verification approach for programs running under snapshot isolation (SI) and similar relaxed transactional semantics. In a common pattern in distributed and concurrent programs, transactions each read a large portion of shared data, perform local computation, and then modify a small portion of the shared data. Requiring conflict serializability in this scenario results in serial execution of transactions or worse, and performance suffers. To avoid such performance problems, relaxed conflict detection schemes such as snapshot isolation (SI) are used widely. Under SI, transactions are no longer guaranteed to be serializable, and the simplicity of reasoning sequentially within a transaction is lost. In this chapter, we present an approach for statically verifying properties of transactional programs operating under SI. Differently from earlier work, we handle transactional programs even when they are designed not to be serializable.

In our approach, the user first verifies his program in the static verification tool VCC pretending that transactions run sequentially. This task requires the user to provide program annotations such as loop invariants and function pre- and post-conditions. We then apply a source-to-source transformation which augments the program with an encoding of the SI semantics. Verifying the resulting program with transformed user annotations and specifications is equivalent to verifying the original transactional program running under SI – a fact we prove formally. Our encoding preserves the modularity and scalability of VCC's verification approach. We applied our method successfully to benchmark programs from the transactional memory literature. In each benchmark, we were able to verify the encoded program without manually providing any extra annotations beyond those required for verifying the program sequentially. The correctness argument of the sequential versions generalized to SI, and verification times were similar.

2.1 Introduction to Relaxed Transactional Semantics and Correctness Conditions on them

Transactions provide a convenient, composable mechanism for writing concurrent and distributed programs. They are used to write shared memory programs using transactional memory (TM), programs that access a single, central database, and, more recently, but increasingly widely, to write programs that access geo-replicated databases. In each of these settings, a transactional execution platform can provide a strong or more relaxed programming semantics. The former simplifies program construction and verification, while the latter provides better performance and availability. This chapter is about a technique for verifying transactional programs that operate under relaxed semantics – a problem that has not yet been addressed.

For TM and database transactions, by strong semantics we mean atomicity and serializability of transactions. For programs operating on geo-replicated objects, we mean strong consistency of transactions. For TM and database transactions, the motivation for relaxed semantics is performance and avoidance of frequent transaction aborts, while, for geo-replicated data types, the motivation is providing availability to the objects or databaes, even when the replicas, sometimes located on mobile devices, are disconnected (partitioned).

The best-known relaxed consistency semantics in the TM and database domains is snapshot isolation (SI), where the entire transaction is not guaranteed to be atomic, but all of the read accesses in the transaction are atomic and all the updates performed by the transaction are atomic. Many popular databases provide SI as the default consistency mode. Relaxed semantics (also known in TM literature as relaxed conflict detection schemes) other than SI, such as programmer-defined conflict detection [2], and early release of read set entries [3] have been investigated in the database, software and hardware transactional memory communities. For the distributed transactional programs, relaxed consistency semantics such as session SI [4] and parallel SI [5] have been investigated (See per-record time line consistency [6] and prefix consistency [7] for examples) Relaxations of strong semantics are widely used for geo-replicated databases are variants of eventual consistency, with additional guarantees relating different objects and the order in which different updates are propagated to different replicas.

When a transactional execution platform provides strong consistency and serializable transactions, the code of a transaction can be treated as sequential code. This significantly simplifies writing and verifying applications. For the increasingly common transactional execution platforms with relaxed semantics, one way to retrieve the simplicity of sequential reasoning is to enforce serializability via additional analyses or instrumentation, e.g. by preventing or avoiding write-skew anomalies. This approach can be useful some of the time, but, for many examples, may result in a loss of performance or availability and defeat the purpose of relaxed semantics. On platforms with relaxed semantics, much of the time, it is the application author's intent to implement a transactional program that is correct, e.g.. satisfies assertions and invariants, without enforcing strong consistency or serializbility. Typically, the way relaxed consistency exhibits itself in transactional code is in the form of "stale reads"' – data read by the transaction may not be the most recent version later during the transaction, or even at the time of the read access, in the case of geo-replicated databases. Our verification technique is targeted at such transactional programs.

Although transactional programs operating under relaxed semantics are becoming increasingly commonplace, there are currently no tools for verifying properties (assertions and invariants). Static tools for code verification targeted at sequential programs [8, 9, 10], and the VCC verification tool [11] for verifying concurrent C programs have been quite successful. These tools are (when applicable) thread, function and object-modular, and scale well to large programs. For transactional platforms providing more relaxed semantics such as SI or eventual consistency, these tools cannot be used as they are not aware of transactions or relaxed consistency semantics. In this chapter, we present a static code verification technique for transactional applications running under weak consistency semantics such as SI or eventual consistency. The goal of our technique is to provide a verification environment exactly like that of VCC but for programs running on relaxed tranasactional platforms. The verification approach provides scalability and modularity, as VCC does, but requires programmer annotations for procedure pre- and post-conditions and loops in the same way all existing modular static code verification tools do.

2.2 Problem with an Labyrinth Example

In this section, we illustrate our approach (Figure 2.1) on the Labyrinth benchmark from the STAMP benchmark suite, one of the four benchmark programs we applied our method to. This example is typical of the design and correctness intuition for programs that satisfy desired assertions and invariants while operating under SI. The Labyrinth program is correct., i.e., satisfies the desired invariants and procedure post-conditions despite its executions not being serializable. Enforcing serializability (as is typically accomplished by enforcing conflict serializability [13]) would be an unnecessary restriction that hurts performance.

Labyrinth follows a common parallel programming pattern. Transactions each read a large portion of the shared data, perform local computation and update only a small portion of the shared data. Under conflict-serializability all concurrent transactions conflict and transactions can only run serially, one at a time.

As shown in 2.1, each concurrent transaction runs an instance of the function FindRoute to route a wire "Manhattan-style" in a three-dimensional grid (grid) from point p1 to point p2. Wires are represented as paths: lists of points with integer x, y, and z coordinates, where consecutive entries in the list must be adjacent in the grid. The grid is represented as a three-dimensional array, where each entry [i][j][k] is the unique ID of the path (wire). A data structure pathList keeps pointers to all paths in an array.

Each execution of FindRoute(p1,p2) first takes a snapshot of the grid (line 1) by traversing it and then performs local computation using this local snapshot to compute a path (onePath, line 4) from p1 to p2. Observe that, during this local computation, other executions of FindRoute may complete and modify the grid. In other words, localGridSnapshot may be stale snapshot of grid. SI guarantees in this example that (i) the read of the entire grid in line 4 is atomic, (ii) that the updates to onePath and grid in line 11 are atomic, but does *not* guarantee that the entire transaction is atomic.

Specification Desired properties for this program are that (i) the grid is filled correctly by the information, and that (ii) no two paths overlap. The latter of these is implicitly ensured because each grid point contains a single wire ID number. The former is formally expressed below

```
isValidPath(int ***grid, path_t* p) =
  (forall int i; 0<= i < path->path_len ==>
     p->ID == grid[p->x[i]][p->y[i]][p->z[i]])
   forall int i; 0<= i < path->path_len-1 ==>
```

```
isAdjacent(p->x[i],   p->y[i],   p->z[i],
             p->x[i+1], p->y[i+1], p->z[i+1])
```

FindRoute must preserve this invariant for all paths on pathList in addition to the post-conditions that onePath is a valid path that connects p1 to p2 and is in pathList.

Static Verification of Sequential FindRoute: When FindRoute is viewed as if it is running sequentially, with no interference from other transactions, it is straightforward to verify using VCC. The following are the key steps taken:

- We verify that the code for shortestPath (not shown) satisfies the post-conditions in lines 6 and 7.

- Using this fact, we verify that gridAddPathIfOK, if and when it terminates, satisfies the program invariant (no two paths overlap and pathsList and grid are consistent), and the desired post-conditions in 14.

To carry out the verification tasks above, static code verification tools, including VCC, require the programmer to write loop invariants as annotations. The rest of the verification of function post-conditions is carried out automatically.

Verifying FindRoute Under SI: The verification of FindRoute under SI rests on the key observation that the conditions listed above for correctness of FindRoute under SI remain correct even when thread interference as described by SI occurs. Our technique allows us to verify that this is the case mechanically using VCC.

In a given instance of FindRoute, if gridAddPathIfOK detects that onePath overlaps an existing wire, it explicitly aborts the transaction. Instances of FindRoute that complete do so because they have computed a path onePath that not only does not overlap any of the wires in the initial snapshot localGridSnapshot, but also does not overlap any of the paths added to the grid since.

The intuition behind FindRoute being correct while running under SI is as follows:

1. SI ensures that the traversal and copying of the grid in line 1 is carried out atomically.

2. SI ensures that the updates to pathList and grid performed by gridAddPathIfOK are carried out atomically.

3. To verify that an atomic, terminating execution of gridAddPathIfOK establishes the desired program invariant and post-condition, it is sufficient to know that the post-conditions established by shortestPath in lines 7 and 8 still hold at the time gridAddPathIfOK starts running.

In our technique, we transform and augment the code for FindRoute to obtain another C program with VCC annotations. Verifying the resulting program in VCC amounts to checking that (iii) continues to hold under thread interleavings constrained by (i) and (ii).

Our technique accomplishes this as follows.

- The encoded program has exactly the set of thread interleavings allowed by SI. The auxiliary variables (e.g., version numbers for each grid element and wire, fictitious locks, etc.) and constraints ("assume" statements) on these variables built into the encoded program only allow executions where all read accesses in a transaction are carried out atomically and all write accesses are carried out atomically. There are no other restrictions on how the threads are interleaved.

- When VCC verifies the object and global invariants and procedure post-conditions (e.g., the FindRoute program invariant or post-condition of shortestPath) in the encoded concurrent program, it checks whether they are preserved under thread interference possible in the encoded program. Since the encoded program (an ordinary concurrent C program) allows exactly the interleavings specified by SI, this amounts to verifying that properties of the original program running under SI hold.

The encoded program preserves the structure of the original program, and does not inline code from other possibly interfering transactions.

2.3 Introduction to our Approach

In our approach, we take a transactional program and the semantics of the transactional platform that provides relaxed consistency. We produce an augmented C program with VCC annotations. The program our approach outputs is the same (has the same structure, etc.) as the input program, but includes an encoding of the relaxed transactional semantics and allows exactly the executions and interleavings specified by the relaxed semantics through the use of auxiliary variables in VCC. Our program transformation can be viewed as augmenting the program with a high-level implementation of the transactional platform. The transformation is designed with special attention towards preserving the thread, function and object modularity of the verification of the sequential version of the program in VCC. The encoding avoids inlining code that other, interfering transactions that might be running concurrently.

To illustrate the applicability of our technique, we verified programs that were written to be correct even under relaxed transactional consistency. These programs were three benchmarks from the STAMP [12] transactional benchmarks suite and a StringBuffer pool example. We verified using our approach and tool that these examples satisfy application-level invariants and assertions despite relaxed transactional semantics. The user annotations required in each case reflected the correcness intuition in the relaxed transactional case, were relaxed versions of the annotations that would have been required had the program been running sequentially, and did not refer to the auxiliary variables involved in our encoding. In other words, the user was able to simply express the correctness intuition without referring to the mechanisms implementing the transactional semantics. Verification times for programs running under relaxed semantics were comparable with verification times for programs running sequentially.

Although we recognize the distinctions between different relaxed transaction semantics, in the rest of this chapter, for brevity's sake, we use SI to refer to relaxed consistency models similar to SI and "serializability" to stand for the class of transactional platforms that provide atomic, strongly consistent, serializable transactions.

2.4 Related Work

Relaxed conflict detection. Relaxed conflict detection has been devised to improve concurrent performance by reducing the number of aborted transactions. Titos et al. [2] introduce and investigate conflict-defined blocks and language construct to realize custom conflict definition. Our work builds on this work, and provides a formal reasoning and verification method for such programs. As we have shown with SI and !WAR, we believe that our method can easily be adapted to support other relaxed conflict detection schemes.

Enforcing (conflict) serializability, detecting write-skew anomalies There is a large body of research on verifying or ensuring conflict or view serializability of transactions even while the transactional platform is carrying out relaxed conflict detection [14, 15, 16, 17, 18, 19]. Since runtime and/or static analyses ensure serializability, programmers can reason about transactional code as if it were sequential. For transactional code that is correct but not necessarily conflict or view serializable, as was the case in the examples we studied, verification approaches will signal potential serializability violations while serializability enforcement approaches will result in actual serial execution of transactions. In this work, we enable programmers to verify properties of transactional code on SI even when executions may not be serializable. This allows the user to prove the correctness of and use transactional code that allows more concurrency.

Linearizability. One way to allow low-level conflicts while preserving application-level guarantees is to use linearizability as the correctness criterion [20]. To prove linearizability of a transactional program P running under SI, one could use the encoded program we construct, $\tilde{P}$ as the starting point in a linearizability or other abstraction/refinement proof. In this work, we have chosen not to do so for two reasons. First, abstract specifications with respect to which an entire program is linearizable may not exist or may be hard to write. Second, programmers would like to verify partial specifications such as assertions into their program in terms of the concrete program variables in scope. Verifying linearizability does not help the programmer with this task.

Encodings, source-to-source transformations. As a mechanism for transforming a problem into one for which there exist efficient verification tools, source-to-source code transformations are widely-used in the programming languages and software verification communities. The work along these lines that is closest to ours in spirit involves verifying properties of programs running under weak memory models. Atig et al. [21] propose a method for simulating programs running under total-store order (TSO) semantics with a program running under sequential consistency (SC) semantics. For this purpose, auxiliary variables are introduced to the new program for simulating the store buffers that are part of the TSO operational semantics. Authors prove that the transformed program running under SC correctly models a subset of behaviors of the original program under TSO. Alglave et al. [22] present a sound transformation from programs running under a variety of weak memory models to programs running on the sequential SC memory model. This allows the use of a variety of dynamic and static verification tools for verifying the transformed program. Our work also makes use of a source-to-source translation in order to transform the problem of verifying a transactional program running under SI to a generic C program that can be verified using VCC. Our transformation results in only a linear increase in code size. The distinguishing features of our encoding via source-to-source transformation are as follows.

- We start from code and annotations on a sequential program verified in VCC. In our approach, both the code and the annotations are transformed. In other words, we transform a correctness proof, not only the code that was verified. In the examples we investigated, the transformed annotations provided VCC with enough hints to carry out the verification task under SI.

- In the transformation, the thread, object and procedure structure of the original program is preserved. As a result, while verifying the transactional program under SI, we have the same level of function, object, and thread-modularity that was present in the original VCC verification of the sequential program. No inlining of extra code modeling interference from other transactions is involved.

- While verification of the transactional program under SI may require the user to relax the annotations on the sequential program, the user does not have to provide extra annotations in terms of the extra auxiliary variables in the encoded program.

On earlier work [23] where a program abstraction presented allows us to verify that the abstracted transactional program running under relaxed conflict detection is serializable. While verifying this latter fact, in earlier work, explicit use of left- and right-mover actions and commutativity are made, and the proof was carried out with tool support only for checking the correctness of abstractions and commutativity. In our work, the entire verification of the transactional program running under SI is carried out within the static verification tool VCC, and the soundness of the verification approach is formally proven (Theorem 1). More importantly, in our work, we provide an approach to use the correctness proof of the sequential version of the program to derive a correctness proof for the program running under SI.

2.5 Our Approach

In this section, we formally present our approach to verifying transactional programs running on SI. For simplicity of presentation, we only present our approach for SI. Our approach is applicable to similar relaxed conflict detection schemes such as `!WAR`. We present the formalization for `!WAR`, which is only slightly different from that for SI in the appendix.

We build upon VCC syntax and semantics and give C code with VCC annotations a particular interpretation in order to model database and in-memory transactions. Our model covers equally well both transactions with relaxed consistency semantics being executed by nodes in a distributed program and threads in a shared-memory concurrent program that uses transactional memory with relaxed conflict detection.

In the rest of the formalization, for uniformity, we will refer only to threads, with this understanding.

2.5.1 Preliminiaries: Transactional Programs

A *Program Text* consists of a set of valid `C` functions $\mathcal{F}$. Each C function has the form

```
func(type_0 arg_0, type_1 arg_1, ..., type_n arg_n)
```

Shared data is represented by aliasing among arguments of functions calls representing different transactions. Unless indicated otherwise in VCC function preconditions, when two such functions have arguments that are pointers to the same type, they may potentially alias to the same address. Accesses to such arguments within the function are interpreted as potential shared data accesses. In order to make explicit any potential sharing between transactions, we do not allow any global variables in our VCC programs. All data sharing is modeled via aliasing among input arguments.

A *Program* is a tuple $(Tid, \mathcal{F}, TtoF)$ such that Tid is a set of transactions, $\mathcal{F}$ is a program text and $TtoF : Tid \rightarrow F_{seq}$ is a map which assigns a sequence of functions to each transaction. While referring to an element of the tuple $Element$ representing a program P, we will write it as $Element_P$ (e.g. $GlVar_P$). Transactional programs use C syntax. The code for a transaction has the following structure:

- A call to `beginTrans(`t`)` marks the beginning of a transaction, and a call to `endTrans`(t) marks the end of a transaction. Transaction are not allowed to be nested.

- We make the committing of a transaction syntactically visible by a call to `commitTrans(`t, inv`)`. This call precedes the call to `endTrans`(t). Between `commitTrans(`t, inv`)` and `endTrans`(t), only manipulation of local variables is allowed. The call to `commitTrans(`t, inv`)` allows us to specify invariants that should hold at commit time. When this transaction commits, if inv is not satisfied, a specification violation occurs.

We define states and the transition relation of a program under SI as follows: A *global state* is a tuple $GS = (GlVar, GlMem, TtoLcSts)$ such that

- $GlVar$ is the set of global variables, i.e., shared objects (structs) that multiple transactions hold references to in GS,
- $GlMem : GlVar \rightarrow Val$ maps global variables to their values in the memory, and
- $TtoLcSts : Tid \rightarrow L$ keeps local states of each transaction.

The set of all global states is denoted by G.

A *local state* of a transaction t is a tuple $LS = (LcVar, Stmt, RSet, Wset, ValTrans, LcMem)$ such that

- $LcVar$ is the set of objects local to t,
- $stmt \in Stmt$ is the next statement to be executed by t,
- $RSet \subseteq GlVar$ ($WSet \subseteq GlVar$) is the set of global variables that have been read (written) by t since the beginning of the transaction, until LS was reached,

- $ValTx : Wset \rightarrow Val$ is a map that stores the value of the latest write to each global variable performed by this transaction, and
- $LcMem : LcVar \rightarrow Val$ maps local variables to their values. The domain of this partial map is the local variables of the transaction, i.e. the set of stack variables of the functions in the transaction.

The set of all possible local states is denoted by L. For convenience, let us represent $LcSt(t) = \langle LcVar_t, Stmt_t, RSet_t, WSet_t, ValTx_t, LcMem_t \rangle$

In the initial global state, all local states have `beginTrans(t)` as $stmt$, the statement to be executed. A global state s is a final state if and only if for all $t \in Tid$, $TtoLcSts(t)$ is a final local state. A local state s is final if and only if $Stmt = \bullet$, $RSet = WSet = \emptyset$. Err is a global state that corresponds to a specification violation. Err has no outgoing transitions. While referring to an element of a global or a local state s, we will use the notation $Element^s$ (e.g. $GlMem^s$).

The state transition relation for SI (M_{SI}) is denoted by $\triangle_{si} : (s, stmt) \rightarrow s'$ for some global states $s = (GlVar, GlMem, TtoLcSts)$, $s' = (GlVar', GlMem',$-$TtoLcSts') \in G$. For some $t \in Tid$ such that $Stmt_t = stmt$, $(s,\ stmt) \rightarrow s'$ is defined as follows:

- if $stmt$ is `beginTrans(`t`)`, then $RSet'_t = WSe't_t = \emptyset$.
- if $stmt$ writes val to a local variable a, then $LcMem'_t(a) = val$.
- if $stmt$ assigns a local variable b to a local variable a, then $LcMem'_t(a) =$

$LcMem_t(b)$.

- if *stmt* creates a fresh global variable a, then $GlVar' = GlVar \cup \{a\}$ and $ValTx'_t(a) = GlMem(a) = \perp$.

- if *stmt* reads a global variable a, then $RSet'_t = RSet_t \cup \{a\}$ and $ValTx'_t(a) = GlMem'(a)$ if $ValTx'_t(a) == \perp$.

- if *stmt* writes a local variable b's value to a global variable a, then $WSet'_t = WSet_t \cup \{a\}$ and $ValTx'_t(a) = LcMem_t(b)$.

- `commitTrans(`t, inv`)` statement commits a transaction t and checks if *inv* is satisfied. If it is, the state is left unchanged, otherwise, execution moves to the global state *Err*. This statement corresponds to an atomic execution of a commit operation and an assertion. We find this construct useful when expressing invariants on global state. For all $w \in WSet_t$, $GlMem'(w) = ValTx_t(w)$ is executed atomically..

- if *stmt* is `endTrans`(t), then $stmt' = \bullet$, $RSet'_t = WSet'_t = \emptyset$ and $ValTx'_t(a) = \perp$ for all variables $a \in WSet_t$.

- if *stmt* is `assert(`p`)` $s' = Err$ if p is not satisfied in s. p is a boolean formula involving any $a \in LcVar_t$.

Elements of the state other than the $Stmt_t$ remain unchanged if a modification is not specified in the cases above.

An *Action* is a unique execution of a statement by a transaction t in a state s. An *Execution Prefix* of a program P_{SI} is a tuple $E_N = (\vec{s}, \vec{\alpha})$ where $\vec{\alpha}$ is a finite sequence of actions $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ and $\vec{s} = s_0, s_1, \ldots, s_N$ is a finite sequence of states such that $(s_i, \alpha_i) \rightarrow s_{i+1}$ for all $i < N$. An execution prefix has the form:

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{N-1}} s_N$$

The formalization so far places no restriction on the interleaving of actions from different transactions. The transaction consistency semantics and conflict detection scheme, such as serial execution of transactions, conflict serializability, and SI specify which interleavings of actions from different transactions are allowed in an execution. We denote the set of all possible prefixes of a program P_{SI} under a semantics M as $Pre(P_{SI}, M)$. A *Program Execution* is an execution prefix $E_F = (\vec{s}, \vec{\alpha})$, such that $\vec{s} = s_0, s_1, \ldots, s_F$ and $\vec{\alpha} = \alpha_0, \ldots, \alpha_{F-1}$, s_0 is an initial state and s_F is a final state of P_{SI} . We denote set of all possible executions of a program P_{SI} under a semantic M as $Execs(P_{SI}, M)$.

2.5.2 SI and other Relaxed Conflict Detection

We write $Idx_E(\alpha_i)$ to refer to the index i of action α_i in the execution, and $Tr_E(\alpha_i)$ to refer to the transaction performing α_i. For integers i and j such that $i \leqslant j$, we write $[i..j]$ to refer to the set of integers $\{i, i+1, ..., j-1, j\}$. When talking about ordering between actions and states of the same execution, we overload the comparison operators over integers to actions and states, i.e., for $\bullet \in \{<, >, \leqslant, \geqslant\}$, $\alpha_i \bullet_E \alpha_j$ (resp. $s_i \bullet_E s_j$) if $i \bullet j$.

To make precise the sets of executions of a program allowed by different relaxed conflict-detection schemes, we find it useful to define the *protected span* of a shared variable x within a transaction t for a given consistency model M. Intuitively, this span is a set of indices of actions with the property that, according to the consistency model, at none of these indices can an update to x in shared memory take place due to the commit action of a transactions other than t.

Consider an execution E, a transaction t in E and a variable x read in t. The *read span* of x in t is the interval $[i, \mathit{Idx}(\mathit{commit}(t)]$, where α_i is the first action within t that reads x. Similarly, the *write span* of x in t is the interval $[i, \mathit{Idx}(\mathit{commit}(t)]$, where α_j is the first action within t that writes to x. We adopt the convention that the read span of x in t is empty if t does not read x. Similarly for write spans.

In the conflict serializability consistency model, the protected span for a shared variable x within a transaction t is $[i, \mathit{Idx}(\mathit{commit}(t))]$, where i is the index of the first read or write action in t that accesses x. Typical conflict serializability implementations enforce a stricter non-interference condition, where the read and write spans of a variable x within a transaction do not overlap the write span of x in any other transaction.

It is well known that that every conflict-serializable execution is equivalent to a serial one. In most TM implementations [**?**] the serializability condition above is ensured by maintaining the sets of addresses accessed (the read and write sets) by each transaction and aborting and rolling back transactions that experience disallowed conflicts.

Snapshot Isolation. An execution E is said to obey snapshot isolation iff for all transactions t, (i) all read accesses performed by t are atomic, (ii) all write accesses performed by t are atomic, and (iii) if t both reads and writes to a variable x, the value of x in shared memory is not changed between the first access to x by t and the commit action of t. Requirement (ii) is by satisfied by default for our model of transactions.

Requirements (i) and (iii) are made more precise in the definition below:

Definition 1 (Snapshot isolation)**.** *An execution E obeys snapshot isolation iff there exists an equivalent execution E' such that the following conditions hold:*

- *For all transactions t, the read accesses performed by t are serial in E'.*
- *For all transactions t, if t accesses a variable x more than once, between the first access of t to x in E' and the commit action of t in E', no other transaction that writes to x commits.*

To specify snapshot isolation in terms of spans within an execution, we first define the snapshot read span of a variable x read by a transaction t. Let α_i be the first read action (of any variable) in a transaction t, and let α_j be the last read of a variable x by t. Then, the *snapshot read span* of x in t is the interval $[i, j]$. If x is never read in t, its snapshot read span is the empty interval. The protected span of a variable x in snapshot isolation is defined as follows:

- If x is only read by the transaction, the protected span of x is the snapshot read span of x.

- If x is both read and written to, then the protected span is the interval $[i, j]$ where i is the index of the first access of the transaction to x, and j is the index of the commit action of t.

- If x is only written to, the protected span is defined to be the write span of x.

- Otherwise the protected span is empty.

Snapshot isolation requires that the protected span of each variable x does not contain any commit actions by other threads that write to x. Definition 1 and the definition of snapshot isolation in terms of the snapshot read span are equivalent.

Relaxing Write-After-Read Conflict Detection. We next define the concurrency control semantics, $\mathbf{Rlx}(P)$. This semantics specifies the executions provided by a transactional memory with relaxed detection of conflicts using the `!WAR` annotation as described in [2]. In this semantics, the programmer annotates certain read actions to be *relaxed reads*. The protected span of a variable x in t is still defined as the interval $[i, Idx(commit(t))]$, where α_i is the first regular (not relaxed) read action or write action accessing x as part of t. A relaxed read of x in t is simply required to return the result of the last write to x in shared memory or the last write to x in t, whichever comes later in the execution but before the relaxed read access. Differently from serializable semantics, in read-relaxed semantics, after a relaxed read of x by t but before t commits (or performs a regular read or write to x) other transactions are allowed to commit and update the value of x. In words, after transaction t reads from x, other transactions may write to x arbitrarily many times if it is the case that

t itself does not access x again. However, conflicting writes are never allowed between a write access and the corresponding commit action. Therefore, in this consistency model, we can no longer reason about the code for a transaction sequentially.

2.5.3 Concurrency, VCC and Modular Verification: An Overview

In this section, we briefly, and, due to space constraints, informally, introduce the VCC mechanisms and conventions we make use of in our approach. For an in-depth treatment of VCC, see the manual at

`http://vcc.codeplex.com/documentation`.

Objects and invariants. While the C language does not have objects, VCC allows programmers to think of structs as objects. Each object has a unique owner at any given time. This owner may be another object or thread. The concept of ownership is one mechanism using which access to objects shared between threads is coordinated, and invariants spanning multiple objects are stated and maintained. Objects can be annotated with any number of two-state transition invariants: first-order formulas in terms of any variables in scope in the object that specify the allowed state transitions the object is allowed to make. These invariants may refer to the states of other objects. Objects can be *open* or *closed* at a given time, indicating whether the object is being modified (possibly temporarily violating its invariants) by the owner or not, respectively.

Ghost variables. VCC allows the introduction of ghost variables of all types, including all C types, and more complex ones such as sets or maps. Ghost variables are (auxiliary) history variables, and they do not affect the execution of the program and values of program variables. Maps can be initialized and updated using lambda expressions. Structs can have fields that are ghost variables.

Modular verification. VCC performs modular verification in the following manner. Each function is annotated with pre- and post-conditions. Each loop is annotated with a loop invariant. Every struct may be annotated with two-state transition invariants. Code may also be annotated with assertions in VCC's first-order specification logic, in terms of the program and ghost variables in scope. VCC then verifies the code for one function at a time, using pre-post condition pairs to model function calls, loop invariants to model executions of loops, and "sequential" or "atomic" access, as described below, to model interference from concurrent threads. It generates a first-order verification condition checked by the Z3 theorem prover. Whether all of the annotations are consistent and all assertions, post-conditions, and invariants are satisfied by the code of a function is checked in the same way as typical SMT-based static verifiers, with the exception of how "sequential" and "atomic" accesses to shared objects are modeled. "Sequential access" and "atomic blocks" are two mechanisms using which programmers can describe to VCC how threads within a program coordinate access to shared data.

"Sequential" access and ownership. In "sequential" access, the thread accessing a variable obtains exclusive access to a variable `aVar` by obtaining ownership of `aVar`. This may be accomplished by, for instance, acquiring a lock whose invariant indicates that the lock controls accesses to `aVar`. When the lock is not held by any thread, the lock owns `aVar`. When the lock is acquired by a thread t, the ownership of the object is transferred from the lock to t. The thread then "unwraps" the variable (opens the object it refers to) and manipulates it. While the object is open, its invariants are allowed to be broken temporarily. When the accesses of the thread t are complete, t "wraps" the object and makes it closed. It then transfers its ownership back to the lock by releasing it.

"Atomic" access. One way to coordinate access to shared variables in VCC is to mark them `volatile` and to require that objects they refer to always remain closed. Then, any state transition of the program must adhere to the transition invariants of these objects. Such objects are not ever exclusively owned by any thread, and VCC reasons about a function containing atomic accesses to volatile variables as follows. After each atomic access in the code, VCC only assume that the following are known: (i) the invariants of the atomically-accessed volatile objects hold, and (ii) the states of objects owned by the thread are retained. Atomic accesses are key to how we model the semantics of read accesses according to relaxed consistency. After an atomic read of a reference to object o into thread-local variable l, the verification condition generated by VCC does *not* assume that the state of the object referred to by l does not change. It only assumes that the object referred to by l continues to satisfy the invariants associated with its type. VCC allows the creation of fictitious objects called *claims*. Claims have no state, but only invariants and can be thread-local. One can use a claim to state, prove and retain a formula in terms of variables in scope. VCC verifies in a modular manner that the invariant of a claim holds at the time of its creation, and that it is preserved by atomic accesses to objects by other threads.

2.5.4 Source-to-source Transformation for Simulating SI

In this section, we present how from a given transactional C program P_{SI} (with VCC annotations) running under SI we produce a program $\tilde{P_{SI}} = Encode(P_{SI})$ running under ordinary `VCC` semantics making use of constructs presented in the previous section. We formally prove that verifying $\tilde{P_{SI}}$ under ordinary VCC semantics is equivalent to verifying P_{SI} under transactional SI semantics.

The encoding is obtained via a high-level modeling of the operational semantics of SI using transaction-local and globally-visible shared copies for each object, and VCC statements of the form `assume(`ϕ`)`. A thread in a program can take a state transition by executing `assume(`ϕ`)` only at a state s that satisfies ϕ, in which case, program control moves on to the next statement. Interleavings disallowed by the consistency model M are expressed as a formula ψ in terms of objects' version numbers, and statements of the form `assume` $\neg\psi$ are used in the encoding.

For simplicity, we present the transformation for programs that only use `int` s as primitive types. In the transformation, each shared variable of type `int` is replaced with a variable of type `PInt` as shown below:

```
PInt{
   int inMem;         int inMemVNo;
   int inTM[Trans];   int inTMVNo[Trans];
   Lock lock;
   _(invariant \unchanged(inMemVNo) ==> \unchanged(inMem))
   _(invariant \forall int t;
          \unchanged(inTMVNo[t]) ==> \unchanged(inTM[t]))
};
```

The “wrapper” type `PInt` holds the following information:

- a field `inMem` value that corresponds to the value of the variable in shared memory,
- a version number `inMemVNo` that gets incremented atomically each time the `inMem` field is written to,
- a (ghost) field `inTM[Trans]` which is a map from Tid to integers. `inTM[t]` holds the value of the transaction-local copy of the integer
- a (ghost) field `inTMVNo[Trans]` which is a map from Tid to integers. `inTMVNo[t]` is incremented atomically with each update of `inTM[t]`
- a (ghost) field `lock` that is used to convey to VCC when a transaction has exclusive access to the `int` variable

This wrapper type has an important invariant that indicates that a field’s value remains unchanged if its version number remains unchanged. This invariant, along with `assume` statements involving version numbers allows us to represent constraints such as the value of a variable remaining unchanged between two accesses within a transaction.

To implement transactional semantics, we use one `Trans` struct per transaction.

```
Trans{
   bool holding[PInt];
```

```
    bool readsLockedSet[PInt];

    bool writesLockedSet[PInt];

};
```

Fields of `Trans` are ghost maps that store the variables that have been read and written by a transaction, and variables to which the transaction has exclusive access.

The encoded program makes use of the following VCC statements:

- An assumption statement `assume(`$\tilde{p}$`)`where $\tilde{p}$ is a boolean formula. After this statement, the execution proceeds considering only the cases that $\tilde{p}$ holds.

- An assertion statement `assert(`$\tilde{p}$`)`where $\tilde{p}$ is a first order proposition on program variables. This statement aims to prove whether $\tilde{p}$ is satisfied.

$\widetilde{P_{SI}}$, the encoded version of a program P_{SI} is constructed as follows. For each global variable of type `int` in P_{SI}, the encoded program $\widetilde{P_{SI}}$ has a global variable of type `PInt`. For each global `int` variable a in P_{SI}, we denote the corresponding `PInt` variable in $\widetilde{P_{SI}}$ by $\tilde{a}$. When transforming the program syntactically, we use lowercase variables `a` to refer to variables of type `int` in the original program, and uppercase versions (`A`) to refer to the corresponding wrapper variable of type `PInt` in the encoded program.

The code transformation described below describes a bijection $h_{stmt} : Stmt_{P_{SI}} \times Stmt_{\tilde{P_{SI}}}$ that maps each statement in P_{SI} to its encoded version in $\widetilde{P_{SI}}$. The transformation is described assuming that the code has been decomposed that each statement accesses a global variable at most once, as is typical in transactional applications.

The code transformation makes use of a number of C functions whose pre- and post-conditions are presented later in this section.

- Statements of the form `beginTrans(`t`)` remain unchanged in the encoded version.

  $h_{stmt}($`beginTrans(`t`)`$)$ =`beginTrans(`t`)`

- Statements that only assign a value *val* to a local variable or a local variable to a local variable remain unchanged in the encoding.

- Statements that create a new global variable `A` are transformed to `newPInt(A)`.

- Each statement `l = v` by transaction `t` that reads a global variable `v` into local variable l is transformed to an atomically-executed statement that performs the equivalent of the following VCC code atomically.

```
assume( \forall PInt P;
         t->readsLockedSet[P] ==>
              P->inTMVNo[t] == P->inMemeVNo[t]);
l = transRead(t, V);
```

- Each statement `v = l` that writes the value of a local variable `l` to global variable `v` is transformed to the an atomically-executed statements that performs the equivalent of the following VCC code atomically. We make the assumption that a transaction writes a global variable at most once.

```
assume(V->\owner == t || V->\owner == NULL);
acquireLock(V,t);
assume(V->inTMVNo[t] == V->inMemVNo);
```

```
//V has not been written to since it was read by t.
transWrite(V, l, t);
```

- Each statement `commitTrans(t, inv)`, is transformed to the following atomically-executed sequence of statements:

```
assume( \forall PInt P;
        t->writesLockedSet[P] ==>
           P->inTMVNo == P->inMemVNo + 1);
commitTrans(t);
assert(INV);
```

- For each statement `endTrans(t)`, we replace the statement with `endAndCleanTrans(t)` in the encoded version.

- Each statement `assert(p)`, where `p` is a boolean expression in terms of local variables, is left as is in the encoded version. Each boolean expression `e` involved in a loop invariant, and function pre- and post-condition is transformed to a boolean expression `E`, where each appearance of a global variable `v` is replaced with a reference to the transaction-local copy `v->inTM[t]`.

The states and transition relation of the encoded program, used in the proof of soundness for our approach, are as follows: A *global state* $(GlMem, TtoLcSts, O)$ extends the global state in the formal model of programs operating under SI by a partial map $O : GlVar \to GlVar \cup Tid$. O keeps the owner of each global variable (denoted by the ghost `owner` field that each variable automatically gets in VCC) which can be either another global variable or a transaction. Each global variable can be owned by at most one entity at a time. $GlVar_{\tilde{P_{SI}}}$ is obtained from $GlVar_{P_{SI}}$ by replacing each integer variable by `PInt` variable. $GlVar(a$`->inMem`$)$ represents the value of the `PInt` variable in the memory and $GlVar(\tilde{a}$ `->inTM(`t`)` represents the transaction-local value of the `PInt` variable in a particular transaction $t \in Tid$. In the initial global state, the global variables are not owned by any transaction $O(\tilde{a}) = \perp$, and their `inTM` field are not defined

The functions used in the encoded program are listed below together with their preconditions and postconditions:

- `beginTrans(t)` creates a `Trans` structure for thread `t`. This function has no pre-condition and has the post-condition that the read and write sets of `t` and the set of variables `t` has exclusive ownership of are empty, i.e.,

  ```
  \forall PInt P; !t->readsLockSet[P] &&
            !t->writesLockSet[P] && !t->holding[P]
  ```

- `acquireLock(V, t)` is used to obtain exclusive access to `V` by transaction `t`. This is accomplished by using the fictitious (ghost) lock `V->lock`. Since we are verifying only succeeding executions of transactions (and assuming that aborted transactions have no visible effect), we call `acquireLock` in the encoded program only at a state where it will successfully complete. Thus, this function has the pre-condition that the global variable `V` has no owner or is owned by `t`, and the post-condition that the owner of `V` is the transaction `t`.

- `transRead(V,t)` reads `V` in a transaction `t`. This function does not require `V` to be owned by `t` and has the post-condition that

  ```
  t->readsLockSet[V] == true &&
  V->inTM[t] == V->inMEM &&
  V->inTMVMo[t] == V->inMEMVNo
  ```

- `newPInt(V)` is used to create a new PInt variable. This function has the post-condition that `V->owner` is `t`. All version numbers associated with `V` are initialized to 0.

- `transWrite(V, l,t)` writes the value of the local variable `l` to the `inMem` field of `V` and atomically increments `v->inTMVNo[t]`. If `V` has been read previously by `t`, then this function requires that `V`'s version number has not changed since. These are expressed by the pre-condition

  ```
  V->\owner == t && V->inMemVNo == V->inTMVNo[t]
  ```

  and the post-condition

  ```
  t->writesLockSet[V] == true &&
  t->inTM[t] == l &&
  t->inTMVNo[t] == old(t->inTMVNo[t]) + 1
  ```

  Recall that a transaction can write to a particular global variable only once.

- `commitTrans(t)` commits a transaction by writing the updates performed by the transaction into the memory. Note that a valid execution can have only local statements (that only effect local state) after `commitTrans(`t`)` statement until it ends the transaction. This function is better explained by the following pseudocode

```
_(atomic t {
      \foreach PInt P;
          if (ptrans->writesLockSet[P]) {
               P->inMEM = P->inTM[t];
               P->verNoInMEM = P->verNoInTM[t];
          }
})
```

Since VCC currently does not support loops inside `atomic` statements, the state update corresponding to the loop above is expressed as the function post-condition for `commitTrans`.

- `endAndCleanTrans(t)` ends a transaction `t` by releasing the locks that the transaction holds, cleaning its read and write sets. It has the post-condition that `t` releases ownership of all objects it owns, and the `readLockSet`, `writesLockSet`, and `holding` are all reset to maps corresponding to empty sets.

An example of the encoding is provided in Figures 2.2 and 2.3 for a StringBuffer pool example.

The semantics of the encoded program which makes use of these functions is as follows. The *state transition relation* for the transformed program on global states $s, s' \in G$ is denoted by $\triangle_{st} : (s, stmt) \rightarrow s'$. If there exists a $t \in Tid$ such that

$TtoStmt(t) = stmt$, then $((GlMem, TtoLcSts, O), stmt) \to (GlMem', TtoLcSts',$-$O')$ where $TtoLcSts'(t) = (stmt', RSet'_t, WSet'_t, ValTx'_t, LcMem'_t)$ in which $stmt'$ is the next statement to be executed in $\mathcal{F}$:

- If *stmt* is a function call defined above, then the transition is enabled if s satisfied the function precondition and s' is a state satisfying the postcondition.
- If *stmt* is a write statement to a local variable, i.e. writing *val* to a local variable $a \in LcVar_{\widetilde{P_{SI}}}$, then $LcMem'_t = LcMem_t[a \to val]$ in s'.
- If *stmt* is `assume(`$\tilde{p}$`)`where $\tilde{p}$ is a boolean formula involving any variables in$GlVar_s$, then the transition is enabled if $\tilde{p}$ holds. The transition only skips to next statement $stmt'$ and no other changes are applied to s'.
- If *stmt* is `assert(`$\tilde{p}$`)` where $\tilde{p}$ is a boolean formula involving any variables in$GlVar_s$, $s' = Err$ if $\tilde{p}$ is not satisfied. Otherwise, s=s' except for that s' has stmt' as the next statement to be executed.

The following theorem, the proof of which is available at

`msrc.ku.edu.tr/projects/vcctm`

states the soundness of our verification approach.

Theorem 1 (Soundness)**.** *Let P_{SI} be a transactional program and $\widetilde{P_{SI}}$ be the augmented program obtained from P_{SI} as described above. Then $\widetilde{P_{SI}}$ satisfies its specifications (assertions, invariants, function pre- and post-conditions) if and only if P_{SI} satisfies its specifications.*

It follows from this theorem that users can start with the program P interpreted sequentially, provide the desired specifications, and additional proof annotations and verify P within VCC. Then, to verify properties of P_{SI}, users can follow the (clearly automatable but not yet automated) source-to-source transformation approach described in this section and obtain $\widetilde{P_{SI}}$. Verifying the transformed specifications with the transformed annotations on $\widetilde{P_{SI}}$ is equivalent to verifying the specifications of P_{SI}, by the soundness theorem.

2.5.5 The Encoding: Discussion

The source-to-source code transformation described in the previous sections preserves the thread, function, and object structure of the original program. The newly-introduced objects representing transactions are local to each thread or transaction. All additional invariants introduced are per-object. There is no inlining of code from other, possibly interfereing transactions, and the size of the transformed code is linear in the size of the original code.

To make modular verification possible in VCC, auxiliary and wrapper objects and their invariants all need to be coordinated carefully using ownership and approval relationships. Since presenting ownership, approval and modular verification in VCC is beyond the scope of this thesis, we discuss the rationale for this approach only briefly below.

VCC is the only existing tool for modular static verification of concurrent C programs. By modularity, we mean that a single verification condition generated for each C function, thread and struct/object, and that the effects of other threads and functions are modelled by (i) object invariants, (ii) ownership and approval relationships, and (iii) function pre/post-conditions. The challenge in ensuring modularity is expressing program and proof using (i)-(iii) and ensuring object invariants are "admissible." In return, one gets modular and scalable verification.

The concept of admissibility is key for modularity in VCC and is discussed briefly next: The invariant I_o of object o may refer to o's fields, fields of other objects (e.g. for linked lists, containers), and invariants of other "approver" objects. A program statement s may violate I_o by modifying fields of o or fields of some other object I_o refers to. As a result, s may have to be checked against (potentially) the invariants of all objects. If all object invariants are admissible, one gets more modularity by only checking s against invariants of objects it modifies.

For object o (invariant I_o) let us imagine that I_o refers to objects $q_1, q_2, ..., q_n$ and that these objects take transitions consistent with their object invariants $I_{q_1}, ..., I_{q_n}$. I_o is admissible iff it continues to hold after these transitions. To ensure admissibility, one needs to establish ownership and/or approval relationships between o and $q_1, ..., q_n$. This allows I_o and/or I_{q_i} to refer to each other. To make invariants admissible, one needs to carefully orchestrate ownership relationships, often dynamically.

The original program and its correctness argument dictate one set of ownership relationships among program (and therefore, wrapper) objects. Encoding relaxed transactional semantics correctly using objects representing transactions and locks dictates another. Reconciling these conflicting ownership requirements was the key challenge in formulating the encoding. This was accomplished by delineating the "sequentially-accessed," transaction-local fields of the wrapper object as a "group" (nested object) and managing their ownership separately from the "atomically accessed" shared fields of the wrapper object.

In the next section, we report on our experience applying our approach to a number of benchmark programs and the SI and `!WAR` relaxed conflict detection schemes.

2.6 Experiments

We applied our technique to the Genome, Labyrinth and Self-Organizing Map benchmarks from STAMP [12], a widely-used collection of concurrent benchmark programs containing pre-annotated transactional code blocks, and a StringBuffer pool example. All four of these examples are correct applications but their executions are not conflict serializable. The STAMP examples had been implemented in a way that is correct under SI and using programmer-defined conflict detection previously [2]. We made precise and formally verified the correctness arguments for these implementations and for the `StringBuffer` example.

For each benchmark, we wrote partial specifications and statically verified that they hold for transactional code running with the regarding relaxed consistency semantics, starting from a VCC verification of the specifications on a sequential interpretation of the benchmark.

- **Genome:** Figure 2.4 shows the pseudocode for a linked list implementation used in the `Genome` benchmark [12]. The code in the figure has been simplified for ease of presentation. In the part of this benchmark where relaxed consistency is used, concurrent transactions insert into a shared linked list. Transactions run under programmer-defined conflict detection, where write-after-read conflicts are ignored (`!WAR`), i.e., do not cause transactions to abort. Figure 2.5 illustrates how concurrent insertions experience write-after-read (`!WAR`) conflicts, and how, intuitively, it would be correct implementation to let an insertion commit even though it experiences a `WAR` conflict. Following []. the body of `list_insert` is marked with the `!WAR` annotation to indicate that write-after-read conflicts should be ignored.

We verify that the linked list maintains two invariants under interference : (i) its nodes are in ascending order and (ii) linked list is not circular. We further verify that the `addNode(newNode)` Function satisfies the post-condition that the node it adds (`newNode`) is reachable from the head of the linked list. The read (traversal) phase of the `addNode` function finds a node `prev` in the list after which `newNode` is to be inserted. The assertion that `prev` is reachable from the head of the list and that the appropriate place for `newNode` to be inserted is right after `prev` is preserved despite interference caused by ignoring write-after-read conflicts.

- **SOM:** In this benchmark, concurrent transactions run the learning phase of the machine learning algorithm SOM. SOM contains a shared grid of which nodes are n-dimensional vectors. The learning function `solve` takes an n-dimensional vector v and the grid as input, calculates the Euclidean distance of v to each grid nodes, picks the closest one v' and moves nodes in a neighbourhood of v' closer to v.

- **StringBuffer** In this example, a pool of `StringBuffer` objects are implemented as a collection. The example is written using programmer-defined conflict detection, in particular, using `!WAR` semantics. We verified that a data structure invariant and post-conditions of the `Allocate` and `Free` functions are satisfied.

- **Labyrinth:** This example and its verification process was described earlier in the chapter.

We have demonstrated the applicability of our verification approach on these examples that were written without assuming serializability and satisfied their specifications despite this. In each of these examples, our encoding facilitates thread- and procedure-modular correctness proofs that hold for an arbitrary numbre of threads. Programmer annotations on encoded program makes no reference to auxiliary encoding variables. Our experience with the SI and `!WAR` relaxed consistency models, which are very similar to other relaxed consistency models described earlier leads us to believe that our static verification technique is a useful tool for a programmer building applications in these settings.

```
// Program invariant:
// forall int i; 0<=i && i< pathlist->num_paths
//      ==> isValidPath(grid, pathsList->paths[i])

FindRoute(p1, p2) {
 transaction {
      localGridSnapshot = makeCopy(grid);
                      // Take snapshot of entire grid

      // Local, possibly long computation
      onePath = shortestPath(p1, p2, localGridSnapshot);

      // Desired post-conditions of shortestPath:
      assert(isValidPath(onePath, localGridSnapshot))
      assert(isConnectingPath(onePath, p1, p2);

      // Register points on onePath as "taken" on grid
      // Add onePath to pathsList
      gridAddPathIfOK(grid, pathsList, onePath);

      // FindRoute must ensure program invariants,
      // and the post-condition
      //   onePath in pathsList &&
      //        IsConnectingPath(onePath, p1, p2)
} }
```

Figure 2.1: Outline for FindRoute code and specification.

```
StringBuffer* Allocate(StringBuffer* pool){
        StringBuffer* ptr;

        for(int i = 0; i<1000; i++){
            ptr = pool[i];
            if(ptr!=NULL){
                    pool[i] = NULL;
            }
        }
   return ptr;
}
```

Figure 2.2: StringBuffer pool before code transformation.

```
//Signature of the transformed function
void* Allocate(SBPointer* pool, PTM tm _(ghost \tmbsl claim c))
//Body of transformed function
{
    int i, index;
    void* ptr;
    PTrans ptrans = (PTrans) malloc(sizeof(Trans));
    trans_begin(ptrans, tm _(ghost c));

    for(i=0;i<SIZE; i++)
    _(invariant 0<=i && i< SIZE)
    _(invariant ptrans->tm == tm)
    _(invariant \wrapped(ptrans))
    {
        ptr = tx_relax_read(ptrans, pool[i], tm _(ghost c));
        if(ptr != NULL) break;
    }

    //assume the SB has not been written by another thread
    _(assume pool[i]->version_num == tm->version_numP[pool[i]])
    _(assume (pool[i]->SBPointerLock->locked==0 &&
            pool[i]->SBPointerLock->owning_trans == (PTrans)0 ||
            pool[i]->SBPointer->locked == 1 &&
            pool[i]->SBPointer->owning_trans == ptrans &&
            ptrans->holding[pool[i]]))

    Acquire(pool[i]->SBPointerLock, ptrans _(ghost c))

    //commit to TM
    _(ghost_atomic tm, ptrans, c{
       _(ghost tm->val = (\lambda PInt i;
          ptrans->lockedWritesInteger[i] ? i->data : tm->val[i]))

    _(ghost tm->valP =(\lambda SBPointer pr;
        ptrans->lockedWritesPtr[pr] ? (pr->ptr) :(tm->valP[pr])))

```

```
struct node_t { int key; node_t* next; ghost Set reach;}

bool list_insert(list_t *listPtr,
                 node_t *node) {
  node_t *prev, *curr = listPtr->head;

   do {
     prev = curr;
     curr = curr->next;
  } while (curr != NULL
          && key > curr->key);
  _(invariant loopInv(prev, curr, head, node))
  // loopInv(prev, curr, head, node) ==
  //          prevKey < key && prevKey < curKey
  //          && prev reachable from head
  //          && curr reachable from head

   // assert(prev->next == curr);
  node->next = curr;
  prev->next = node;
  return true; // key was not present
}
```

Figure 2.4: The insertion operation of a sorted linked list.

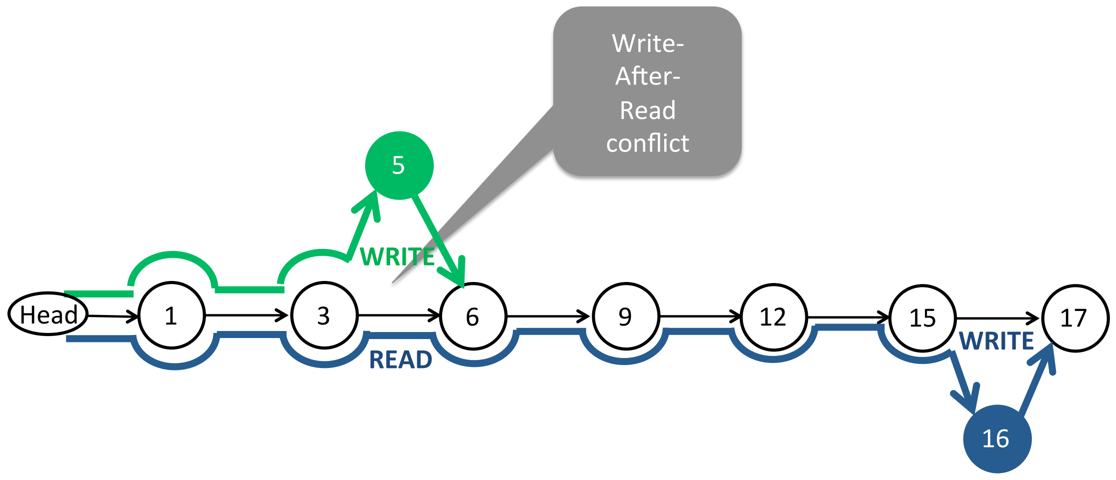

Figure 2.5: Sorted linked list and a write-after-read conflict.

Chapter 3

QED FOR PROGRAMS RUNNING ON X86-TSO MEMORY MODEL

x86-TSO is one of the models that main stream processors' architectures implement. This makes reasoning for programs running on x86-TSO memory model a crucially important effort for providing reliability of those programs. However, this effort is not that much easy to perform. There has been many studies trying to build proof methods for reasoning on programs under x86-TSO model or automated tools to check safety vulnerabilities due to TSO model. Generally, when you attack such important problems you usually come up with a solution that is applicable under some restrictions (e.g. fixed number of threads, bounded buffer size) or you can come up with a complicated proof after a hard verification effort. In this chapter, we propose a proof method approach for refining programs to make them easy to reason under x86-TSO model. We introduce the store-buffers and abstract TSO memory flush operation explicitly into the source. In this approach, we try to observe whether specific type of programming pattern result in specific type of refinement pattern. We show applicability of this the approach with mechanized refinement of non-blocking Spin-Lock and acquire/release pattern Send-Receive in Section 3.4.7 by approach 3.4.

We built our proof method approach for programs running on x86-TSO using ideas and formalizations from QED [24]. We define and explain custom tailiored formalization for the approach under section 3.4.1. This chapter starts with brief introduction to QED idea. Then it mentions related research done for reliability on TSO world. Section 3.4 covers our proof method.

3.1 A Reduction/Abstraction Concurrency Verification Method: QED

QED is a novel proof strategy in which atomicity is used as a proof tool : a program with fine-grained concurrency is transformed iteratively to make it consist of larger atomic actions. These larger blocks enable reduction in reasonsing on thread interleavings which is main factor in complexity of concurrent program verification. When the program reaches to an feasible atomicity level then setting for a tractable, refined proof is established to check correctness condition [66].

3.1.1 Movers in QED

Reduction in QED classifies each action in the program as mover. There are four types of movers : right-mover R, left-mover L, both-mover B, non-mover A. Informally, a right-mover action α is an action that can commute to right of the next action which either leads to the same state or leads to a violation. Similarly, a left-mover action α is an action that can commute to left of the previous action which either leads to the same state or leads to violation. Both-Mover action is an action that shows both left-mover and right-mover properties. Left-Mover and Right-Mover rules inside

QED basically defines the mechanism to label the atomic actions in the program as a left or right-mover. Mover-Check rules perform a pairwise commutativity check with all other atomic actions and performed by QED automatically. An action is defined as $\phi \rhd \tau$ where the store predicate ϕ is the *gate* of the action and represents the set of states from which this action can execute without "going wrong". The transition predicate τ represents the set of state transitions allowed by this gated action. The Left-Mover action $\phi \rhd \tau$ can commute to the left of every action in the program. The Right-Mover action is similar except to show that $\phi \rhd \tau$ is a right mover, QED checks commutativity of true $\phi \rhd \tau$ to the right of actions in the program. This subtlety is very important and stated in [24] in detail. Basically, when an action α that may go wrong cannot commute to hte right of an action β that blocks [66].

3.1.2 Reduction/Abstraction

Each rewrite of the program performs one of two different kinds of transformations; abstraction,reduction. Abstraction replaces an atomic action with a more relaxed atomic action allowing extra behaviours. Abstraction transformations include making shared variable reads or writes non-determinstic, and adding extra assertions to atomic actions. Reduction [25] replaces a compound statement consisting of several atomic actions with a single atomic action if certain non-interference conditions hold. This transformation has the effect of increasing the granularity of the atomic actions in the program.

QED is an iterative proof method which uses reduction and abstraction symbiotically. Reductions creates coarse-grained actions from fine-grained actions and allows a subsequent abstraction step to summarize the entire calculation much as pre-conditions and post-conditions summarize the behaviour of a procedure in a single-threaded program. Conversely, suitably abstracting an atomic action allows us to reason that it does not interfere with other atomic actions, and later applications of reductions are able to merge the action with other actions.

Introducing assertions at any point during the sequence of transformations and deferring the proof until all atomic actions become large enough to discharge all assertions is very useful technique that is offer by QED where we use it for debugging our proof intention. annotating an atomic statement with an assertion is a valid program abstraction. QED has a distinguished approach to assertions. Rather than proving the introduced assertion that it is valid and preserved under interference of other threads, QED uses the introduced assertion without first proving it and take further proof steps that simplify the program. During a following reduction step, the assertion indicates the absence of an apparent interference between the containing action and another action by a different thread, for example, that the actions are not simultaneously enabled, and enables further application of reduction. When a previously inserted assertion is discharged, this indicates that our beliefs about the program state were correct, and thus, the previous reduction steps relying on that assertion were sound [66].

More detailed fundementals such as operational semantics, soundness proofs and proof rules can be found in [24].

3.2 Total Store Order Memory Model

[35] defined x86-TSO memory model in higher order logic and mechanized in the Hol theorem prover. Basically, each processor is connected to main memory by a FIFO store-buffer or write buffer. This queue contains (address,value) pairs. When a write to a memory occurs, one pair from head is taken and processor commits the value of the pair to the address of pair in the memory. A processor put asynchrony to its writes via store-buffer. A read returns the value of the last buffered write to the adress, if any, otherwise the value of the address in the main memory.

3.3 A Survey on Verification of x86-TSO

Program analysis under relaxed memory models has been a field of intense study, e.g. formalization of memory models [84, 85, 88, 83], program logics for relaxed memory models [93, 92, 73, 91] or verification techniques [64, 72, 48, 55, 58]. In this discussion of related work, we focus only on research about TSO program analysis.

Common and distinguished parts of related work can be classified according to the following properties. Axiomatic and operational model of TSO is given by [35]. We mainly follow *abstract machine memory model* of this study as operational model in our approach.

Data Race Freedom. *DRF*, is a property of memory models [30]. This property needs to be satisfied with sequential consistency techniques. The weakened DRF property, *Triangular Race Freedom* [36] needs to be satisfied to guarantee correctness under x86-TSO model. TRF and the notion of TSO-robustness by Bouajjani et al. [72] are closely related. Since our mechanical verification is SMT-based, it is not restricted, as is [72], to finite state programs or a finite number of threads but involves potentially undecidable SMT queries. Algorithm in [37] finds violations of sequential consistency under the TSO and PSO memory models.

Abstracting buffers. Using automata for representing store-buffers is another approach [38]. Since store-buffers have FIFO characteristics [40] the automata needs to behave like Finite State Machine which can perform communication between states.In these communicating FSMs, algorithms perform communication between states using redundant information, queue content which is also used for usage of protocols for communication. This approach is expensive to be applicable with additional non-termination possibility.

Model checking. Explicit enumeration of program states for verification of relaxed memory models [41, 42, 43].

Composing constraint solving and specification generation. [44, 45, 46], *CheckFence* takes an implementation and test program for this implementations. First it creates a specification for the given test by enumerating the set of correct observations. An observation is a combination of argument and return values, and it is correct if it is consistent with some atomic interleaving of the operations. Then it checks all executions of the test on the chosen memory model to see that the observed values are contained in the specification. In backend, it transforms test program and implementation to a pure load-store language by inlining opearations and unrolling loops. Then it encodes possible executions as a propositional formula. To form set of observation, it gives this formula to SAT solver and iteratively add constraints to SAT solver until it reaches unsatisfiability, probably after k iteration. A counter example for a execution is given as a result of existence of the execution in observed set. Incremental SAT solver dependency exists in this approach in addition to have large set of serial execution and it is not clear how expensive to reach observation set. It needs to unroll loops at a preprocessing stage. Thus it cannot verify programs that contain unbounded spinning. *Sober* tool [48] (Chess : stateless model checker + store-buffer monitor) claims to be more scalable with without exploring the additional nondeterminism of memory-model relaxation that can cause state explosion as it is the case in [42].

Fence insertion via model checking. *Fender* [49] does fence insertion under relaxed memory model but not under the sequentially consistent model. Assume that a spin lock includes a loop including store. It will always have infinite state under relaxed memory model unless there is fence after store instruction. Synchronization primitives mainly has implementation following that code pattern. *Fender* can not catch those patterns to insert fence, same limitation that *CheckFence* has. Both *Fender* and *CheckFence* have a bound on the size of buffers or loop iterations which are iteratively increasing. Unbounded or large state space are drawbacks of these approaches for being applicable to all examples. Abstracting store-buffers can overcome these problems. *Synchronization* inference and *fence* [54, 52, 51, 53] insertion are closely related concepts. [54, 52] can only handle sequential consistency models not relaxed memory models. Follow-up approaches on fence insertion are e.g. [69, 68, 45, 59, 57, 71].

Restoring sequential consistency from weaker one and violation of sequential consistency. *Sober* runs algorithm checking violation of sequential consistency under relaxed memory models. [37] like *Sober* has monitoring algorithm checks sequential consistency violations. Beside monitoring algorithms, forcing program to preserve sequential consistency with fence insertion [59, 60, 61, 62] is another way of sequential consistency preservation of executions under relaxed memory models.

Reachability analysis. [68, 67, 72] are studies focusing relaxed memory model reachability analysis of programs. [67] implements a linear source-to-source transformation to perform TSO reachability analysis of a concurrent program as SC reachability analysis, showing that SC reachability set and TSO reachability set are same for shared variables at most k context-switch for each thread. [68] allows automatic reachability analysis to perform automatic inference of memory barriers that are necessary in a program to satisfy given safety properties. [72] introduces algorithms to check robustness semantic of concurrent programs under TSO. Robustness is simply defined as all TSO computations of a program correspond to computations under the SC semantics. Non-robustness is harmful out-of-program order executions. Detecting attacks can be implemented via adding queries for reachability analysis under SC semantics to source done with instrumenting the source, source-to-source transformation. This is a close study to [61] which focuses cheap *trace-robustness* and can not catch cyclic traces. [61] does not introduce algorithm to catch cyclic traces but [72] introduces equality of complexities of SC reachability analysis and trace analysis with gain of analysis of state that cyclic trace has.

Proof system. [73] introduces a rely/guarantee proof system for reasoning about x86 assembly programs running against the weak x86-TSO memory model. This Rely-Guarantee proof system enables the system such that one processor can refer to other's local state. Mechanical proofs are handled by Hol in the backend. [74] presents a proof methodology to verify the correctness of compiler translations from a Java-like intermediate representation to a low-level structured register transfer language (RTL) representation under the TSO model. The refinement method used makes use of some notions (e.g., atomicity, reduction) common with our work. This technique assumes that non-interference between code segments has been verified separately and verifies code transformations under this assumption. Our work [94] explained in Section 3.4 centers on verifying the desired non-interference under TSO.

Code-to-Code Translation. A mean for converting a verification task on a program running on a relaxed memory model to another verification task expressed on a sequentially consistent program is common in the literature (e.g., [67, 72, 55, 64]). We do not consider code-to-code translation to be a verification technique in and of itself, rather, a mechanism for realizing a certain reduction from one problem to another. In terms of the crux of the reduction, techniques differ widely.

3.4 Using Atomicity as a Refinement Method for Verifying Programs Running on x86-TSO

As we discussed throughout the thesis, many architectures of processors have relaxed memory models (RMO). Verification of programs running on these models is hard. One of common relaxations is store-buffers. This relaxation allows a write followed by a read to execute out of program order, buffering writes and allowing a read to bypass the writes in a store-buffer. In this section, we propose a new proof system for reasoning about programs running on TSO using atomicity [24] as a refinement tool. We encode store-buffers explicitly into source and transform the program with *transformation algorithm* to be verified for sequential consistency [26]. We constructed new correctness proofs for several examples showing different characteristics on TSO such as:

- Non-blockign synchronisation example: Spin Lock
- Mutual exclusion example: Dekker's algorithm
- Producer/Consumer relationship example: Send-Receive

We provide semi automated proof support built on top of QED theorem prover.

3.4.1 Formal Framework

In Figure 3.1, we present the syntax of ActionPL. ActionPL was introduced in [24] to facilitate the formalization of a static proof system for shared memory concurrent

$$\phi, \tau \in \mathit{BExpr}$$

$$\begin{array}{rrcl} \mathit{GatedAtomic}: & \alpha & ::= & \phi \rhd \tau \\ \mathit{Statement}: & s & ::= & \alpha \mid \overrightarrow{\ell} := \mathbf{p}(\overrightarrow{E_\ell}) \mid s^{\circlearrowleft} \mid s\,;\,s \mid s \,\square\, s \mid s \parallel s \end{array}$$

Figure 3.1: ActionPL syntax.

programs that run under the sequential consistency memory model. ActionPL is an abstract language which facilitates the statement and proof of soundness for program transformations.

3.4.2 Preliminaries: ActionPL*, Reduction, Abstraction*

In ActionPL, a program P is represented by a tuple $P = \langle Vars, Main, Body\rangle$. $Vars$is the set of uniquely-named global variables. The program heap is modeled by using a map similarly to ESC/Java [9] and Boogie [96]. The statement *Main* is the body of the program. *Body* is a map from procedure names to statements to be executed when a procedure is called. A gated atomic action α of the form $\phi \rhd \tau$ is the simplest statement in ActionPL and represents an atomic action taken by one thread. The store predicate ϕ is the *gate* of the action and represents the set of states from which this action can execute without ”going wrong”. The transition predicate τ represents the set of state transitions allowed by this gated action. This logic-based representation facilitates simple statements for QED proof rules. In practice, the store and transition predicates in a program are written using Boogie statements. In this section, we describe gate and transition predicates using the SimPL syntax shown in Figure 3.2.

The gate and the transition predicate of a gated action may refer to the current thread id through the special variable $tid \in Vars$. The domain of tid is the set Tid of all thread identifiers. When the gated action $\phi \rhd \tau$ is being executed by a thread $t \in Tid$, t is substituted for tid in both ϕ and τ. $Atoms(s)$ is the set of all gated atomic actions in s. The program state evolves over time by threads with id's in Tid executing statements.

We call transitions obtained by executing a gated action *atomic transitions* of the program. If the current store satisfies $\phi[t/tid]$, the store is modified atomically consistent with $\tau[t/tid]$. Otherwise, the execution is said to *go wrong*, where the program goes to an error state and execution terminates.

We define $Good(t, s, \phi)$ as the set of pre- and post-store pairs associated with succeeding executions of s executed by thread t from stores satisfying ϕ. Similarly, $Bad(t, s, \phi)$ is the set of pre-stores associated with failing executions.

We will present code examples and explain most program transformations using the more intuitive, pseudo-code like language SimPL whose syntax is shown in Figure 3.2. SimPL can be translated in a straightforward manned to ActionPL. We will only present specifics of the ActionPL to SimPL translation in cases where it is central to the arguments made in this approach.

In the syntax apresented in Figure 3.2, shared variables are denoted by $[x]$ where x is a local variable containing an address and $[x]$ denotes the value stored in the memory at address x. `assume` statements are a modeling construct. They represent the Boolean predicate under which control flow moves to the code following the `assume` statement.

A program written in Figure 3.2 can be run under sequential consistency semantics (SC) or under total-store order (TSO) semantics. In the former case, actions from threads are interleaved to obtain an execution, and the shared variable read and write actions (`tmp` $:=$ $[x]$ and $[x]$ $:=$ `tmp`) directly access memory. Under TSO semantics, write actions ($[x] :=$ `tmp`) write to the store-buffer of the thread executing the statement, and read actions (`tmp` $:= [x]$) return the value of the most recent write to x in the thread's store-buffer, if one exists, and the contents of the memory at address x otherwise. The `flush` statement has the same semantics as `skip` under SC.

We will provide TSO semantics for a SimPL program by providing a transformed SimPL program $xform(P)$. In addition to the program variables of P, $xform(P)$ has the additional variables show in Figure 3.3 that are required to encode TSO semantics. These variables consist of a store-buffer per thread, and the shared memory modeled as a map from addresses to a record type called `vrbl`. The ghost variables shown in Figure 3.3 are not required for the encoding but are used to facilitate writing assertions and invariants regarding TSO program state. The store-buffer for each thread is modeled as a map from integers to assignments. The head and the tail of the store-buffers for each thread, denoted by `wb_hd` and `wb_tl`, respectively, are incremented each time a `flush` (resp. `write`) occurs. This way of modeling ensures that each item placed in the store-buffer of a thread receives a unique integer identifier.

We use this fact when writing program annotations for commutativity proofs.

3.4.3 Modeling TSO

Figure 3.4 shows the Boogie code for the procedures modeling TSO semantics. To obtain *xform*(P), shared variable reads and writes in atomic statements of P are transformed, and statements that model the committing of entries from a thread's store-buffer to main memory are inserted into the program. We refer to committing the entry at the head of the store-buffer for a thread as *drain*ing the head of the store-buffer.

The `drainHead()` procedure, made more precise below, commits the entry at the head of the store-buffer for a thread if the buffer is non-empty. This way of allowing a non-deterministic number of entries being drained from the store-buffer between every pair of shared variable read and write accesses is a way of composing the behavior of the program executing in a thread and the corresponding store-buffer. This modeling scheme, also employed in [57], is particularly suitable for our purposes, as it allows us to state proof rules syntactically and to correlate the behavior of the store-buffer and the thread syntactically. In Section 3.4.4 we will present the program transformations for our atomicity-based proof system. Many of these rules are aimed at simplifying statements that manipulate and/or refer to the store-buffer state.

`drainHead()` is modeled using the `isAtHeadAndDrain(hdIdx)` procedure, shown in Figure 3.4. `isAtHeadAndDrain(hdIdx)` models the removal and commiting to memory of the item at the head of a thread's store-buffer. This procedure requires the head of the store-buffer to refer to index `hdIdx`. This allows us to use the index returned when this item was placed in the store-buffer using `write` as a mechanism to keep track of what entry is at the head of the store-buffer and what memory address will get updated if this entry is committed to memory.

To obtain *xform*(P), shared variable reads and writes in atomic statements of P are transformed as follows:

- Each statement that performs a shared variable write $[x]$:= `tmp` is transformed to

  ```
  var tlIndex : int;
  while (*) drainHead();
  tlIndex = write(x, tmp);
  while (*) drainHead();
  ```

  where `write` writes the address-value pair $(x,$ `tmp`$)$ to the tail of the store-buffer for the thread executing this statement.

- Eeach statement that performs a shared variable read `tmp` := $[x]$ is transformed to

  ```
  var isFromMem:bool;
  while (*) drainHead();
  tmp, isFromMem = read(x);
  while (*) drainHead();
  ```

where `read` returns the most recently written value to address x in the store-buffer of the reading thread, if it exists, and the value of x in memory otherwise. The variable `isFromMem` gets the value `true` iff a write to x does not exist in the store-buffer of the reading thread, and the returned value `t` is retrieved from the main memory.

The following macros simplify writing assume and assume statements that refer to store-buffer contents. Their encodings are not shown as they are straightforward:

- `isBufferEmpty(tid)`
- `bufferContainsAddress(addr)`
- `noBufferContainsAddress(addr)`

In the following sections we explore refinement tactицts applied on specific programming patterns. We discuss whether those patterns show refinement pattern for a specific kind of programming paradigm under TSO. We focus on the following points when trying to resolve patterns:

- $while(*) drainHead()$ transformed to

```
while (*)
 drainHeadLessThan(writeIndex);
 if (*)
  isAtHeadAndDrain();
```

Program Syntax

$$t \in \mathit{Type}$$

$\mathtt{tmp}, \mathtt{tmp}_1, \mathtt{tmp}_2, ..., \mathtt{tmp}_n,$

$$x, y, z, r_1, r_2, ..., r_n \in \mathit{LocalVar}$$

$$v \in \mathit{Value}$$

$$e \in \mathit{Expr}$$

$$x, y, z \in \mathit{Addr}$$

$$a \in \mathit{Atom} ::= \mathsf{skip} \mid \mathsf{flush} \mid \mathit{asserte} \mid \mathtt{assume}e \mid \mathtt{tmp} := [x] \mid$$

$$\mathtt{tmp} := e \mid [x] := \mathtt{tmp} \mid \mathtt{havoc}x \mid$$

$$r_1, r_2, ..., r_m := p(\mathtt{tmp}_1, \mathtt{tmp}_2, ..., \mathtt{tmp}_n)$$

$$b \in \mathit{Str8ln} ::= a \mid b; b \mid \mathsf{if}(*)\ b; \mathsf{else}\ b \mid \mathsf{atomic}\ b$$

$$c \in \mathit{Atomic} ::= \mathsf{atomic}\ b$$

$$d \in \mathit{Stmt} ::= b \mid \mathsf{while}(*)\ d; \mid \mathsf{if}(*)\ d;$$

Figure 3.2: SimPL Syntax

- Compexity of pre-conditions of a method.

- Complexity of post-coniditions of a method.

```
var vrbl{
  val: int;
  v_no: int;               // Ghost
  lastWrtnValBy: [TID]int; // Ghost
  lastWriter: TID;         // Ghost
  mostRecentVal: int;      // Ghost
}

record assgt {
  addr: int;
  val: int;
}

record thread{
  wb_hd, wb_tl: int;
  wb: [int]assgt;
}

var Thrd: [TID]thread;
var Mem: [int]vrbl;
```

Figure 3.3: Additional variables needed for the TSO encoding

```
procedure :isatomic read(addr:int)
               returns(result : int ){
  var HD, TL: int;
  HD := Thrd[tid].wb_hd;
  TL := Thr[tid].wb_tl;

  if (*) {
      assume (forall i:int :: HD<=i  && i< TL
         ==> Thrd[tid].wb[i].addr != addr);
      result := toRead.value;
   }
   else {
      havoc result;
      assume (exists i:int::
        (HD<=i && i<TL &&
        result == Thrd[tid].wb[i].value &&
        addr == Thrd[tid].wb[i].addr &&
        (forall j:int:: (i<j&&j<TL) ==>
         addr != Thrd[tid].wb[j].addr)
       );
     }
}
procedure :isatomic drainHead()
                returns (hdIdx:int) {
  var hdIdx: int ;
  hdIdx == Thrd[tid].wb_hd ;
  assume (Thrd[tid].wb_hd < Thrd[tid].wb_tl);
  isAtHeadAndDrain(hdIdx);
}
```

```
procedure :isatomic write(taddr:vrbl, sval : int)
                returns(tlIndex :int){

  var as:assgt;
  as.value := sval;
  as.addr := taddr.addr;

  tlIndex := ThreadPool[tid].wb_tl;
  Thrd[tid].wb[tlIndex] := as;
  taddr.lastWrittenValue[tid] := as.value;
  taddr.lastWriter :=tid;
  taddr.mostRecent := as.value;
  Thrd[tid].wb_tl := Thrd[tid].wb_tl + 1;
}

procedure :isatomic isAtHeadAndDrain(hdIdx:int) {
  var hdAddr, hdIdx: int ;
  assume hdIdx == Thrd[tid].wb_hd ;

  assert Thrd[tid].wb_hd < Thrd[tid].wb_tl;
  hdAddr := Thrd[tid].wb[hdIdx].addr;

  Mem[hdAddr].val := Thrd[tid].wb[hdIdx].val;
  Mem[hdAddr].v_no := Mem[hdAddr].v_no + 1;
  Thrd[tid].wb_hd := Thrd[tid].wb_hd + 1;
}
```

Figure 3.4: Procedures modeling TSO operational semantics.

3.4.4 QED Specifications: Proof Rules Tailored For TSO-Encoded Programs, Transformation

Preliminaries on patterns

QED [24] shows that atomicity can be used as a refinement method on various of programing patterns under SC. In the following sections, we do experiments to show whether QED can show its SC applicability also in TSO world. We start exploring some specific refinement tactic patterns that generate some specific refined program patterns when applied to TSO programs. We introduce TSO tailored proof rules and discuss the rules from the following aspects that we point out:

- Pre/Post Conditions: How does state of buffer affect pre/post-conditions?

- Procedure: Procedures may/not be atomic. How does this affect the pre and post conditions?

- Bag or Sequence: When can we think store-buffer as a bag structure? When can we think store-buffer as a sequence structure?

- Coarse Grain Atomic Block: Are larger atomic blocks useful for simplifying invariants?

- Write Index: How does a write index of a write operation into a store-buffer affect post-condition of a method?

Proof Rules

- PeelEmpty: The following sequence of sequentially-composed statements

  ```
  while (*){assert isBufferEmpty();}  assert isBufferEmpty();
  ```

  can be replaced with

  ```
  assert isBufferEmpty();
  ```

- SeparateIndividualDrain: The following sequence of sequentially-composed statements

  ```
  tlIndex = write(x, tmp);
  while (*) drainHead();
  ```

  can be replaced with

  ```
  tlIndex = write(x, tmp);
  while (*) drainIfHeadLessThan(tlIndex);
  if (*) drainIfAtHead(tlIndex);
  ```

  This alternative transformation separates drain actions that operate on store-buffer entries at indices earlier than `tlIndex` from the drain action that commits `tlIndex` to memory.

- SwapWriteAndUnrelatedDrains: The following sequence of sequentially-composed statements

```
tlIndex = write(x, tmp);
while (*) drainIfHeadLessThan(tlIndex);
```

can be replaced with

```
while (*) drainIfHeadLessThan(tlIndex);
tlIndex = write(x, tmp);
```

Important point with this rule is that write action can not refer to any global memory or thread pool.

- NoDrainEmptyBuffer1: The following sequence of sequentially-composed statements

  ```
  assert BufferIsEmpty(tid);
  while (*) drainIfHeadLessThan(tlIndex);
  ```

  can be replaced with

  ```
  assert BufferIsEmpty(tid);
  ```

- NoDrainEmptyBuffer2: The following sequence of sequentially-composed statements

  ```
  assert BufferIsEmpty(tid);
  while (*) drainHead();
  ```

can be replaced with

```
assert BufferIsEmpty(tid);
```

- NoDrainEmptyBuffer3: The following sequence of sequentially-composed statements

  ```
  assume BufferIsEmpty(tid);
  while (*) drainHead();
  ```

  can be replaced with

  ```
  assume BufferIsEmpty(tid);
  ```

- NoDrainEmptyBuffer4: The following sequence of sequentially-composed statements

  ```
  assume BufferIsEmpty(tid);
  while (*) drainIfHeadLessThan(tlIndex);
  ```

  can be replaced with

  ```
  assume BufferIsEmpty(tid);
  ```

- straightLineCodeEndingWithEmptyBuffer: Straightline code that fits the following pattern

```
while (*) drainHead();

i0 = write(x0, tmp0);

while (*) drainHead();

i1 = write(x1, tmp1);

while (*) drainHead();

...

i_n = write(x_n, tmp_n);

while (*) drainHead();

assume BufferIsEmpty(tid);
```

can be replaced with

```
while (*) drainHead();

i0 = write(x0, tmp0);

if (*) drainIfAtHead(i0);

i1 = write(x1, tmp1);

if (*) drainIfAtHead(i1);

...

i_n = write(x_n, tmp_n);

if (*) drainIfAtHead(i_n);

assume BufferIsEmpty(tid);
```

- MergeWhileStar: The following pair of sequentially-composed statements

```
while (*) drainHead();

while (*) drainHead();
```

can be replaced by

```
while (*) drainHead();
```

This rule is useful because the syntactic transformation for obtaining *xform*(P) from P generates many consecutive pairs of `while` statements as in this pattern.

- PeelAway: The following sequence of sequentially-composed statements

```
procedure foo{
 StmtListPre
 while (*) {
   StmtListBody
 }
 StmtListPost
}
can be split into
procedure foo0Iter{
 StmtListPre
 StmtListPost
}

procedure foo1Iter{
 StmtListPre
 StmtListBody
 StmtListPost
}

procedure foo2OrMoreIter{
 StmtListPre
 StmtListBody
 StmtListBody
 while(*){$StmtListBody$}
 StmtListPost
}
```

This rule is useful because many non-blocking algorithms where refined pattern is :
`while` `drainHead`(); `atomic` {StmtList} . Example usage is shown in Spin Lock in 3.4.8.

- IsAssumeFalse: The following sequence of sequentially-composed statements

```
atomic{
  isSuccessful := false;
  assume isSuccessful;
 }
can be replaced by
```

```
assume false;
```

This rule is important because when we reduce a procedure body into a refined one which includes *assume false* then we can remove this procedure, ignore it, for simplifying the following refinement steps.

- DestroyEmptyDrains: The following sequence of sequentially-composed statements

  ```
  ...
  while(*){
   assert isBufferEmpty
   call drainHead();


  }
  ...
  can be replaced with
  ...
  while(*){
   assert isBufferEmpty();
  }
  ...
  ```

- Unzip: The following sequence of sequentially-composed statements

  ```
       tmp1 := x;
  ```

```
          tmp2 := x;

     a:  x := tmp1 + tmp2;

          if (tmp1 > tmp2)

          {

           tmp3 := 6;

           result := x;

          }

          else {

           result := x;

          }

          result := tmp1;

     can be replaced with

         if (*) { // Begin a branch

          tmp1 := x;

          tmp2 := x;

          x := tmp1 + tmp2;

          assume (tmp1 > tmp2);

          tmp3 := 6;

          result := x;

       } // End a branch

       else { // Begin b branch

          tmp1 := x;

          tmp2 := x;
```

```
        x := tmp1 + tmp2;
        assume !(tmp1 > tmp2);
        result := x;
    }
    result := tmp1;
```

All proof rules explained and additional utility functions for transformation are implemented inside QED. Example usage and experiments can be found at qed.codeplex.com link.

3.4.5 Discussion and Comparison with a Specialized Logic

For x86-TSO, a processor's write buffer state is writable by that processor, but inaccessible to other processors. Write buffers affect the semantics of assertions. Traditionally, the validity of a process assertion should not be affected by the behaviour of other processes. We require that the validity is unaffected by the behaviour of the write buffer processes. Writing syntactic assertions that satisfy this constraint is difficult. Then question arises :

What does QED make easier?

- **Store-buffers are thread local:** QED automatically makes use of this when labeling write actions as right movers. We do not need specialized logic to make this argument.

- **Entries in the buffer are totally ordered:** Arithmetic and array/map logic takes care of this. It is very easy to say ”part of the queue is drained.”

- **Expressivity of the system:** We lift both these restrictions, thereby dramatically increasing the expressivity of the system. We present uses processor assertions that refer to the private state of other processors.

These are key properties of QED that we use to make x86-TSO verification more tractable.

3.4.6 Experiments

In this section we show two examples that are implemented with our TSO encodings. We write proof scripts that include implementation of rules in 3.4.4 to be used in these two experiments.

We intentionally chose these two examples because they show different patterns against the points that we stress in Section 3.4.5 and preliminaries part of 3.4.4.

3.4.7 Send - Receive

We start experiments with analysing a simple send-receive example. This example is important because it does not include any memory barrier or fence which flushes the memory. We try to observe when buffer is empty, how acquire/release pattern effects invariants and what sort of procedure summary forms due to this programming pattern. In Figure 3.5, we illustrate a simple send-receive program. tmp and readVal are local variables. toSend, toRecv and val are constant unique values. flag and x are shared variables. [stmtlist] indicates that there is a sequence of statements which are in the same atomic block. [variable] is denotation of how we differentiate the memory content of memory location whose placeholder is variable. In Figure 3.6 illustrates

```
Send                              Receive

s0: tmp := [flag];                r0: tmp := [flag];

s1: assume tmp == toSend;         r1: assume tmp == toRecv;

s2: [x] := val;                   r2: readVal := val;

s3: [flag] := toRecv;             r3: [flag] := toSend;
```

Figure 3.5: The code for send-receive without drain*

the send-receive code implemented with TSO encodings shown in section 3.4.3. The flush operation under x86-TSO non-deterministic. We model this non-determinisim operationally with drainHead* which means that there can be any number of memory commit from store-buffer.

Acquire/Release pattern provides dependency between actions of send and receive. Each atomic actions at $s1$ and $r1$ are blocking actions that can be satisfied only with execution of other $s3$ in send and $r3$ in receive methods. Otherwise, execution will stop at $s1$ or $r1$. As we go through the refinements steps for this example, we observe that two main important points that affect the invariants and pre/post conditions for TSO reasoning are programming pattern (e.g. acquire/release, non-blocking with memory fences) and having memory fence or barrier inside program. The general tactic that we follow to check atomicity of send-receive is doing abstraction via asserting the implications of state of buffer at each interleaving of actions. Abstraction provides making all actions right-mover except the last two non-mover actions in send and receive methods which update the memory value of flag. For example, in Figure 3.8, if an execution reaches to a point after $s2$ then we can conclude that any thread running receive method already flushed its buffer completely. Before any write operation done in send method, between $s2$ and $s4$, we can assert that buffer of thread running send method is empty, $s2.1$. This abstraction with drainHead*, $s3$, can reduce to a single assertion of state of empty buffer.

Another abstraction that we use is in $s4.1$ in Figure 3.9 which shows how drainHead* is reduced to if*isAtHeadAndDrain. Assertion claiming that write index ti1 is equal to head of thread's store-buffer simply provides the guarantee that there exists a single element which is at ti1. This element can either be flushed into memory or not. This non-determinisim is preserved via if*isAtHeadAndDrain.

In Figure 3.10, we discriminate the memory flush that can be done from last index, tail index of store-buffer, from earlier drain actions by reducing drainHead* to $s6.1; s6.2; s63$. Earlier drain actions are abstracted via drainIfLessThan*(tail), $s6.2$, where in send method tail is equal to ti2. The last memory drain which at index ti2 is abstracted if*isAtHeadAndDrain, $s6.3$.

In Figure 3.13, we perform abstraction with $assert\ [flag] == toSend$ in all blocks after $s2$. If execution satisfies $s2$ then all these assertions added to blocks after $s2$ discharge and make all blocks after $s2$ to be right mover, R. Being right mover enables blocks to compose the with following blocks, $s2$ - $s10$ and this composition forms coarse grain atomic block, [s2 ... s11]. In addition to $assert\ [flag] == toSend$

```
Send                                  Receive

s0 :while (*) drainHead();            r0: while (*) drainHead();

s1: tmp = read(flag);                 r1: tmp = read(flag);

s3: assume tmp == toSend;             r2: assume tmp == toRecv;

s4: while (*) drainHead();            r3: while (*) drainHead();


s5: while (*) drainHead();            r4: while (*) drainHead();

s6: ti1 := write(x,val);              r5: readVal := read(X);

s7: while (*) drainHead();            r6: while (*) drainHead();


s9: while (*) drainHead();            r7: while (*) drainHead();

s10: ti2 := write(flag,toRecv);       r8: ti2 := write(flag,toSend);

s11: while (*) drainHead();           r9: while (*) drainHead();
```

Figure 3.6: The code for send-receive with drainHead*. TSOfying operation.

abstraction, we also introduce pre-condition of the send method which states that :

- If flag variable is in thread's local store-buffer then sender thread has not performed memory drain operation for flag variable. This implies that value of flag variable in the store-buffer is toRecv

In Figure 3.14, $assert\ IsBufferEmpty()$ injected at the end of block $s2$ in Figure 3.13 which makes $assert\ [flag] == toSend$ in $s3$ discharged and this forms atomic block $s1$ in Figure 3.14. All blocks after $s3$ in Figure 3.13 include local writes thus reduction of actions in these blocks composes and forms atomic $s2$ in Figure 3.14. After reduction of mover actions, theorem prover can deduce that:

$Mem[flag] == toRecv \implies Mem[x] == val$.

```
Send

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: while (*) drainHead();

s4: ti1 := write(x,val);

s5: while (*) drainHead();

s6: ti2 := write(flag,toRecv);

s7: while (*) drainHead();
```

```
Receive

r0: while (*) drainHead();

r1: tmp = read(flag);

r2: assume tmp == toRecv;

r3: while (*) drainHead();

r4: readVal := read(X);

r5: while (*) drainHead();

r6: ti2 := write(flag,toSend);

r7: while (*) drainHead();
```

Figure 3.7: Merging consecutive drainHead*.

Conclusion and Challenges. Our refinement tactic is mainly making all actions right mover except from last writes to communication variable, flag, which are non-mover actions. Our tactic resulted in a refinement pattern that one can easily get when she thinks in terms of SC model. Assume that we try to reason about send method under SC model. We could be assumnig that buffer is empty at the end of the send method, assume isBufferEmpty(). If we could have such pattern that we can apply STRAIGHTLINECODEENDINGWITHEMPTYBUFFER rule. You can apply this rule with star2single atomicLabel where atomicLabel is the label of assume isBufferEmpty().

Assume that we want the following invariant to hold:

$invariant\ Mem[flag] == toRecv ==> Mem[x] == val;$

This invariant basically shows the acquire and release property of receive and send methods. Assume that we define the following actions:

$a : havoc\ tmp; assume\ tmp == toSend ==> Mem[flag] == toSend;$

```
Send                                   Send'

s0 :while (*) drainHead();             s0 :while (*) drainHead();

s1: tmp = read(flag);                  s1: tmp = read(flag);

s2: assume tmp == toSend;              s2: assume tmp == toSend;

s2.1: [assert isBufferEmpty();]        s3: assert isBufferEmpty();

s3: while (*) drainHead()];            s4: ti1 := write(x,val);

s4: ti1 := write(x,val);               s5: while (*) drainHead();

s5: while (*) drainHead();             s6: ti2 := write(flag,toRecv);

s6: ti2 := write(flag,toRecv);         s7: while (*) drainHead();

s7: while (*) drainHead();
```

Figure 3.8: Abstraction on drainHead* with isBufferEmpty.

$b : Mem[flag] := toSend; lastWriter := receiver;$

a is valid abstraction of read because if it is not empty, it only has toRecv in it.

o denotes sequential composition operation and $\preceq$ denotes simulation relation between actions [24]. Simulation relation, $\preceq$, when we can replace a action with another one. Simulation can be achieved via doing abstraction, basically adding extra failing behaviours by adding assertions. o is sequential composition operator and can be applied to $a\ o\ b$ when either a is right mover or b is left mover. So we can conclude that

$a\ o\ b \preceq b\ o\ a.$

```
Send
s0 :while (*) drainHead();
s1: tmp = read(flag);
s2: assume tmp == toSend;
s3: assert isBufferEmpty();
s4: ti1 := write(x,val);
s4.1: [assert buf.size == 1
         && wb[tid].hd == ti1;
 while (*) drainIfHeadLessThan(ti1);]
becomes
skip
s4.2: [assert buf.size == 1
         && wb[tid].hd == ti1;
         while (*) drainHead();]
 becomes
  [assert buf.size == 1
         && wb[tid].hd == ti1;
         if (*) isAtHeadAndDrain(ti1);]
s5: ti2 := write(flag,toRecv);
s6: while (*) drainHead();
```

```
Send'
s0 :while (*) drainHead();
s1: tmp = read(flag);
s2: assume tmp == toSend;
s3: assert isBufferEmpty();
s4: ti1 := write(x,val);
s5: [assert buf.size == 1
       && wb[tid].hd == ti1;
       if (*) isAtHeadAndDrain(ti1);]
s6: ti2 := write(flag,toRecv);
s7: while (*) drainHead();
```

Figure 3.9: Abstraction and Reduction to transform drainHead* to if*isAtHeadAndDrain.

```
Send

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

       if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s7: while (*) drainHead();
```

```
Send'

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

       if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s6.1:  [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd =head= ti1 + 1]

s6.2:  while (*) drainIfHeadLessThan(ti1 + 1);

s6.3:

if (*) isAtHeadAndDrain(ti2);
```

Figure 3.10: Peeling out the memory flush at index ti2 from drain actions earlier than tail index of buffer.

```
Send

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

      if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s6.1:  [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd == ti1 + 1]

s6.2:  while (*){

           drainIfHeadLessThan(ti1 + 1);

        }

s6.3:  if (*) isAtHeadAndDrain(ti2);
```

```
Send'

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

      if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s6.1:  [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd == ti + 1]

s6.2:  while (*) drainIfHeadLessThan(ti1 + 1);

s6.3:  [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd == ti1 + 1 ]

s6.4:

if (*) isAtHeadAndDrain(ti2);
```

Figure 3.11: Detaching drainIfHeadLessThan from isAtHeadAndDrain

```
Send

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

      if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s6.1:  [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd == ti + 1]

s6.2:  while (*)

               drainIfHeadLessThan(ti1 + 1);

s6.3: [assert ti2 == ti1 + 1;

          assert wb[tid].hd == ti1

          || wb[tid].hd == ti + 1]

s6.4:  if (*) isAtHeadAndDrain(ti2);
```

```
Send'

s0 :while (*) drainHead();

s1: tmp = read(flag);

s2: assume tmp == toSend;

s3: assert isBufferEmpty();

s4: ti1 := write(x,val);

s5: [assert buf.size == 1

       && wb[tid].hd == ti1;

      if (*) isAtHeadAndDrain(ti1);]

s6: ti2 := write(flag,toRecv);

s6.1:  [assert ti2 == ti1 + 1;

         assert wb[tid].hd == ti1

         || wb[tid].hd == ti + 1]

s6.2:  if (*) isAtHeadAndDrain(ti1);

s6.3:  [assert ti2 == ti1 + 1;

         assert wb[tid].hd == ti1

         || wb[tid].hd == ti + 1]

s6.4:  if (*) isAtHeadAndDrain(ti2);
```

Figure 3.12: Final transformed source for send-receive under TSO.

3.4.8 Spin Lock

In this section, we analyse the discussion points in Section 3.4.4 on a well-known locking example, Spin-Lock. Spin Lock is a non-blocking lock implementation which basically uses a compare and swap function in 3.16 which exists almost in every architecture as an atomic instruction. Informally, CAS takes two parameters. One of them called expected. We use this variable as a guard variable to check whether the

```
procedure send()
requires(addrInBuffer(flag) ==> latestValInBuffer(flag) == toRecv)
{
s0:   while (*) drainHead();
s1:   tmp = read(flag);
s2:   assume tmp == toSend;
s3:   R [assert [flag] == toSend;]  [assert isBufferEmpty();]  // wb.hd == wb.tl;
s4:   R [assert [flag] == toSend;]  ti1 := write(x,val);
s5:   R skip
s6:  [assert buf.size == 1 && wb[tid].hd == ti1; if (*) isAtHeadAndDrain(ti1);]
s7:   R [assert [flag] == toSend;]  ti2 := write(flag,toRecv);
s8:   R [assert [flag] == toSend;]  [assert ti2 == ti1 + 1; assert wb[tid].hd == ti1 || wb[tid].hd == ti1 + 1]
s9: R [assert [flag] == toSend;]  if (*) isAtHeadAndDrain(ti1);
s10: R [assert [flag] == toSend;]  [assert ti2 == ti1 + 1; assert wb[tid].hd == ti1 || wb[tid].hd == ti1 + 1]
s11: N [assert [flag] == toSend;]  if (*) isAtHeadAndDrain(ti2);
}
```

Figure 3.13: Send method transformed with TSO specific rules. Movers annotated.

```
procedure send()
requires(addrInBuffer(flag) ==> latestValInBuffer(flag) == toRecv)
{
s0: while (*) drainHead();
s1: [tmp = read(flag); assume tmp == toSend; assert isBufferEmtpy()]
s2: [assert [flag] == toSend;
      wb = wb + (x,val) + (flag,toRecv);
      assume wb[ti1] := (x, val); && wb[ti2] := (flag, toSend);
      assume ti1 <= wb.hd <= ti2;
      assume (wb.hd > ti1) ==> Mem[x] == val;
      assume (wb.hd > ti2) ==> Mem[flag] == toRecv;]
}
```

Figure 3.14: Send method after reduction.

value of lock is as expected. If it is as expected then thread executing CAS can set the the value of lock and provide mutual exclusion via letting other threads to spin.

$r0$ is the block of release method of spin-lock interface at which a write to store-buffer for *lock* variable done. As an optimization to reduce the number of CPU cycles to access to memory, updating the value of lock at $r0$ is performed on thread local store-buffer. This optimization creates the crucial point in reasoning for spin-lock example under TSO model. Assume that we have a share variable y that is protected by spin lock acquire/release. Suppose that several threads want to access y. First the lock was free, $lock = 0$. Then first thread, t, acquired the lock with setting $lock$ to 1 atomically at $a1.1$ then jump into a critical section. All other threads see the value of *lock* as 1 in memory because flush is part of CAS operation. This provides mutual exclusion because if another thread wants to get into a critical section it will read the value of *lock* as 1 and spin inside block $a.1$.

During critical section of thread t, it can read and write y while others are spinning. Writes are put into local store-buffer at first. Some of them may be flushed into memory from store-buffer. Naturally, one of important questions in reasoning for spin-lock under TSO is :

- Does it matter how many writes flushed into memory?

The answer is definitely no because no other thread is attempting to read y. Thread t exits critical section with updating *lock* to 0. This write is put into store-buffer first. At this point there exists another important point to consider is :

- Buffered writes to y would be sent to memory, because all writes to y are prefix to buffered write to *lock* and store-buffer is has FIFO property

After flush the buffered write to *lock*, another thread can acquire the lock.

Following figures from 3.15 to 3.31 include refinement steps with their explanations for spin lock example. In Figure 3.15, we show acquire and release methods of spin-lock implementation. Release method is just an update to shared variable lock. Acquire spins against result of CAS operation. If lock is acquired by other thread, which means that lock is already set to 1 and this results in spinning of current thread until it reads the value of lock as expected, 0, which means operationally releasing of lock by another thread.

```
acquire                                    | release
                                           |
a0: isSuccessful := 1;                     | r0: lock := 0;
                                           |
a1: while(isSuccessful == 1){              |
                                           |
a1.1   isSuccessful := CAS(0,1);           |
                                           |
a1.2   assume lock == 0;                   |
                                           |
    }                                      |
```

Figure 3.15: Spin Lock implementation without any TSO encoding.

```
procedure :isatomic true CAS( expected: int, newVal: int) returns (result: int)
{
    if (*) {
         assume lock.value == expected;
         lock.value := newVal;
         result := true;
    }else{
        assume lock.value != expected;
        result := false;
    }
}
```

Figure 3.16: Compare and Swap implementation with QED.

```
acquire

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a4: while(*){call drainHead();}

a5: while(*){

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.3:  while(*){call drainHead();}

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

     }//a4 ends

a6:  while(*){call drainHead();}

a7:  assume isSuccessful;

a8:  while(*){call drainHead();}
```

```
release

r0: while (*) call drainHead();

r1: call  indx := write(lock,0);

r2: while (*) call drainHead();
```

Figure 3.17: Spin Lock implementation with TSO encoding.

```
acquire

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a4: while(*){call drainHead();}

a3: isSuccessful := false;

a4: while(*){

a5.1:  while(*){call drainHead();}

a5.3:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    } //a4 ends

a6:  while(*){call drainHead();}

a8:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

```
release

r0: while (*) call drainHead();

r1: call  indx := write(lock,0);

r2: while (*) call drainHead();
```

Figure 3.18: Swap local actions with drainHead*.

```
acquire

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a4: while(*){call drainHead();}

a3: isSuccessful := false;

a4: while(*){

a5.1:  while(*){call drainHead();}

a5.3:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    }//a4 ends

a6:  while(*){call drainHead();}

a8:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

```
release

r1: call  indx := write(lock,0);

r0: while (*) call drainHead();

r2: while (*) call drainHead();
```

Figure 3.19: Swap non-blocking local left actions with drainHead*. $r1$ is swapped with $r0$

```
acquire                                     release

a1: var isSuccessful:bool;                  r1: call  indx := write(lock,0);

a2: while(*){call drainHead();}             r0: while (*) call drainHead();

a3: isSuccessful := false;

a4: while(*){

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    }//a4 ends

a6:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

Figure 3.20: Merge drainHead*. Pair of drainHead* statements in Figure 3.19, $(r0, r2)$, $(a8, a6)$, $(a5.1, a5.3)$ $(a2, a4)$, can be merged into one drainHead*

```
acquire_0iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a6: while(*){call drainHead();}

a7: assume isSuccessful;
```

```
acquire_1iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a6:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a4: while(*){

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    }a4 ends


a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a6:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

Figure 3.21: Peel away command on body of $a4$ block.

```
acquire_1iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a5.1:  while(*){call drainHead();}

a3: isSuccessful := false;

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a6:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

```
acquire_$>=$2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a5.1:  while(*){call drainHead();}

a3: isSuccessful := false;

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a4: while(*){

a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    }//a4 ends


a5.1:  while(*){call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a6:  while(*){call drainHead();}

a7:  assume isSuccessful;
```

Figure 3.22: Swap $a3$ with $a5.1$ in Figure 3.21 in 1iter or >=2iter functions. Swap is followed by merging drainHead*.

```
acquire_0iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a6: while(*){call drainHead();}

a3: isSuccessful := false;

a7:  assume isSuccessful;
```

Figure 3.23: Swap $a3$ with $a6$ in 0iter methods. This is valid local, non-blocking swap.

Swap is followed by merging drainHead*.

```
acquire_0iter
a1: var isSuccessful:bool;
a2: while(*){call drainHead();}
[
a3: isSuccessful := false;
a7:  assume isSuccessful;
]
```

Figure 3.24: $a3$ and $a7$ form an atomic block because $a3$ is a right mover. This atomic block forms [assume false].

Due to *assert* $isBufferEmpty()$ in Figure 3.25 inside block $a6$, we can destroy empty buffer and $a6$ becomes *assert* $isBufferEmpty()$ which is a left-mover. Since *assume* $isBufferEmpty()$ exists at $a5.4$, assertion generated after destroying $a6$ can be discharged. This is the summary of $1iter$ procedure. No more atomicity refinement can be done on procedure $1iter$ in 3.26. $a5.2$ in Figure 3.22 inside >=2iter method becomes *skip* due to $a3$ inside the same method. Then we attach *assert* $isBufferEmpty()$ to all drainHead* inside block $a4$ and we follow the same tactic for block $a6$ and $a5.1$ The summary of block $a5.1$ in >=2iter method at Figure 3.27 becomes *assert* $isBufferEmpty()$; which is a right-mover action. $a5.1$ and $a5.2$ in Figure 3.28 are right-mover actions. Actions in between $a5.4$ and $a5.6$ define exactly the same semantics as CAS and it is atomic. As a result of right mover actions, loop body of block $a4$ becomes atomic. In Figure 3.29, we introduce an auxiliary variable $noThread$ which is an unique thread identifier and $lockOwner$. We annotate

```
acquire_1iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a6:  while(*){

        assert isBufferEmpty();

        call drainHead();

     }

a7:  assume isSuccessful;
```

Figure 3.25: $a5.2$ in 1iter method in Figure 3.22 becomes *skip* because of $a3$ in that method. Insert $assert\ isBufferEmpty()$ as a first statement into block $a6$.

the atomic block $a4$ with $lockOwner := tid;$. At this iteration of refinement, we introduce invariants:

- $lock.mostRecentValue == 0 ==> lockOwner == noThread;$
- $lock.mostRecentValue! = 0 ==> lockOwner == lock.mostRecentValue;$

to reduce whole block $a4$ to a single atomic block.

```
acquire_1iter
a1: var isSuccessful:bool;
a2: while(*){call drainHead();}
[
a3: isSuccessful := false;
a5.4:  assume isBufferEmpty();
a5.5:  call isSuccessful := CAS(0,1);
a5.6:  assume isBufferEmpty();
a7:  assume isSuccessful;
]
```

Figure 3.26: Refined 1*iter* procedure after transformation.

Conclusion and Challenges. The crux of reasoning for spin lock example is the optimization implemented in release method. There is not any memory barrier after updating *lock* there is not any memory barrier or fence that enables other threads see the memory value of *lock* as 0 intantaneously. This optimization complicates pre/post conditions of the methods because we need to refer to the state of the releasing thread's buffer in pre-conditions of acquire method. We need to consider both empty and non-empty state of the releasing thread's buffer.

One of challenges in reasoning for $while(*)...; CAS; ...$ pattern is abstracting the spinning behaviour without losing any executions. We use proof rule shown in Figure 3.21 allows to split the body of acquire method into three separate ones where each of them abstracts the 0, 1 or >= 2 iterations of while(*) block including CAS, $a4$ block in Figure 3.21. This allows to make reasoning on each executional behaviour

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a4: while(*){

a5.1:  while(*){

           assert isBufferEmpty();

           call drainHead();

        }

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

    } // a4 ends


a5.1:  while(*){ assert isBufferEmpty(); call drainHead();}

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a6:  while(*){

        assert isBufferEmpty();

        call drainHead();

     }

a7:  assume isSuccessful;
```

Figure 3.27: Abstraction on drainHead* with *assert isBufferEmpty*().

separately.

Another important aspect in this experiment after some refinement steps done on pre/post conditions of methods is abstraction on mutual exclusion. We abstract mutual exclusion via introducing an invariants on an auxiliary variable $lockOwner$ after annotating block $a4$ with assighment aux in Figure 3.29. The value of $lock$ can be either 0 or 1. If it is one then there exists a thread that owns the lock which is provided with the following two invariants :

- $lock.mostRecentValue == 0 ==> lockOwner == noThread$; where $noThread$ is a unique thread identifier.
- $lock.mostRecentValue! = 0 ==> lockOwner == lock.mostRecentValue$;

By introducing these two invariants, we can localize the actions inside block $a4$ and reduce this loop into a right mover atomic action shown in between $a5.1$ - aux in Figure 3.30.

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a4: while(*){
 [
a5.1:  assert isBufferEmpty();

a5.2:  assume !isSuccessful;
  [
a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();
  ]
 ]
    }//a4 ends


a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a7:  assume isSuccessful;
```

Figure 3.28: Reducing loop body of block $a4$ to an atomic action.

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

a3: isSuccessful := false;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a4: while(*){

 [

a5.1:  assert isBufferEmpty();

a5.2:  assume !isSuccessful;

  [

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

  ]

aux:   lockOwner := tid;

 ]

    }a4 ends

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

a7:  assume isSuccessful;
```

Figure 3.29: Introducing ownership variable and invariants for lock.

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

[

a3: isSuccessful := false;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();


a5.1:  assert isBufferEmpty();

a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

aux:   lockOwner := tid;


a5.2:  assume !isSuccessful;

a5.4:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

]

a7:  assume isSuccessful;
```

Figure 3.30: drainHead*; atomic{ *StmtList* } pattern.

```
acquire_>=2iter

a1: var isSuccessful:bool;

a2: while(*){call drainHead();}

[

a3: isSuccessful := false;

a5.6:  assume isBufferEmpty();

a5.5:  call isSuccessful := CAS(0,1);

a5.6:  assume isBufferEmpty();

aux:   lockOwner := tid;

]

a7:  assume isSuccessful;
```

Figure 3.31: Final pattern of Spin Lock under TSO.

Chapter 4

CONCLUSION

In chapter 2, we introduce a static proof method and tool for programs running on snapshot-isolation and similar relaxed transactional semantics. We provide a platform built on top of VCC to mechanically verify state invariants, pre- and post-conditions for operations and assertions by reasoning on the code for the operation, even though the underlying platform and additional programmer tweaks do not provide serializability.

We investigated the following approach to the problem of static axiomatic verification of code in this setting as follows. In the VCC verification tool that supports C, we annotate a given C program using auxiliary variables. The purpose of the annotation is to augment the program with an encoding of an operational model of the relaxed consistency semantics. The executions of the annotated program, when projected onto the original program variables, are exactly those of the original program running under a particular relaxed consistency semantics. Verification of code is then done on this augmented program.

We do not, however, ask the user to reason directly on this rather complicated augmented program. The mental model we present to the user verifying a program P is described below. For a program A, let $Encode_A()$ denote the program augmented with the encoding of the relaxed operational semantics.

- We ask the user to first verify using VCC their original program $\mathit{Seq}(P)$ treating every operation as if it were sequential. To accomplish this, the user may need to provide loop invariants, function pre- and post- conditions.

- Using the programmer annotations from the sequential code, we derive an abstracted program P_{abs}. More precisely, $\mathit{Encode}_{P_{abs}}()$ is an abstraction of $\mathit{Encode}_{P}()$. P_{abs} is an abstracted program that is very similar to P but is conflict serializable.

- The user now verifies the desired properties using sequential techniques on P_{abs}. All safety properties verified hold true of P running on the relaxed consistency mode.

In addition to modelling relaxed consistency semantics, we devise a proof approach and ecapsulate it in a tool. This is the first tool for verifying code running on these relaxed transactional semantics. In this work, we formalize, encode in VCC, and, thus mechanize the verification of C code running on such platforms. Practically as important is a template for a verification approach that explicitly factors in the allowed interference/non-atomicity using non-determinism and two-state invariants for data types. Simply put, the correctness of programs written for such settings depends on invariants for each data type that are preserved despite the interference.

We show applicability via applying it to the Genome, Labyrinth and Self-Organizing Map benchmarks from STAMP [12], a widely-used collection of concurrent benchmark programs containing pre-annotated transactional code blocks, and a StringBuffer pool example. All four of these examples are correct applications but their executions are not conflict serializable. The STAMP examples had been implemented in a way that is correct under SI and using programmer-defined conflict detection previously [2]. We made precise and formally verified the correctness arguments for these implementations and for the `StringBuffer` example.

In chapter 3, we introduce a static proof method and tool for programs running on total-store-order memory model. We build our interactive verification enviroment on top of QED [24]. QED is a concurrent verfication method and tool using atomicity as a refinement tool to make reasoning for concurrent programs under SC. We extend QED for TSO model with new proof rules and with TSO encoding.

We introduce thread local store-buffers explicitly into source and make store-buffers a part of program state. We build our proof methods explained in Section 3.4.4 and use these rules to transform a program under TSO model. The aim of this transformation is to make TSO program easier to reason about in terms of SC model. The verification and refinement flow is as the following:

- We ask the user to first construct program P
- We transform P into a TSO encoded program, $Encode_P()$, program augmented with TSO encoding A
- We abstract A into a refined, abstracted version, P_{abs}, $Encode_{P_{abs}}()$, which enables users to make reasoning easier for SC

We did experiments to show applicability of this approach on non-blocking spin-lock example and send-receive example showing producer/consumer relation. We explore that different programming patterns show different refinement patterns when atomicity is used as a refinement tool. We try to identify the characteristics of refined according to the following points:

- Pre/Post Conditions of a procedure
- Atomicity of a procedure body
- Behavioural structure of thread local store-buffer, bag or sequence
- Effects of coarser-grain atomic blocks on specification of correctness

BIBLIOGRAPHY

[1] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA* 1993.

[2] R. Titos, M. Acacio, J. Garcia, T. Harris, A.Cristal, O. Unsal, I. Hur and M. Valero. Hardware transactional memory with software-defined conflicts. In *HiPEAC* 2012.

[3] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *WTW* 2006.

[4] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB* 2006.

[5] Y. Sovran, R. Power, M. Aguilera and J. Li.: Transactional storage for geo-replicated systems. In *SOSP* 2011.

[6] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein and P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB* 2008.

[7] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP* 1995.

[8] M. Barnett, K.R.M. Leino, W. Schulte. The spec# programming system: an overview. In *CASSIS* 2005.

[9] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe and R. Stata. Extended static checking for java. In *PLDI* 2002.

[10] M. Fähndrich Static verification for code contracts. In *SAS* 2010.

[11] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: ICSE-Companion 2009. (may 2009) 429 –430

[12] C. Minh, J. Chung, C. Kozyrakis and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC* 2008.

[13] C. Papadimitriou. The theory of database concurrency control. In *Computer Science Press* 1986.

[14] R.J. Dias, D. Distefano, J.C. Seco and J. Lourenço. Verification of snapshot isolation in transactional memory java programs. In *ECOOP LNCS* 2012.

[15] H. Attiya, G. Ramalingam and N. Rinetzky. Sequential verification of serializability. In *SIGPLAN Not.* 2010.

[16] M. Alomari, A.D. Fekete and U. Röhm. A robust technique to ensure serializable executions with snapshot isolation dbms. In *ICDE* 2009.

[17] M.J. Cahill, U. Röhm and A.D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD* 2008.

[18] A. Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD thesis 1999.

[19] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha. Making snapshot isolation serializable. In *TODS* 2005.

[20] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. In *TOPLAS* 1990.

[21] M.F. Atig, A. Bouajjani and G. Parlato. Getting rid of store-buffers in tso analysis. In *CAV* 2011.

[22] J. Alglave, D. Kroening, V. Nimal and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP* 2013.

[23] O. Subasi, T. Elmas, A. Cristal, T. Harris, S. Tasiran, R. Tutos-Gil and O. Unsal. On justifying and verifying relaxed detection of conflicts in concurrent programs. In *WoDet* 2012.

[24] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL* 2009.

[25] R. Lipton. Reduction: a method of proving properties of parallel programs. In *CACM* 1975.

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. Comp.* 1979.

[27] Arvind and J. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *ISCA* 2006.

[28] M. Dubois and C. Scheurich. Access Dependencies in Share-Memory Multiprocessors. In *IEEE Transactions on Software Engineering* 1990.

[29] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. In *Computer* 1996.

[30] V.A. Saraswat, R. Jagadeesan, M. Michael and C. Von Praun. A theory of memory models. In *PPoPP* 2007.

[31] S. Adve and H. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. In *CACM* 2010.

[32] S. Adve and M. Hill. Weak Ordering- A New Definition. In *ISCA* 1990.

[33] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. In *Morgan and Kaufmann Press* 2008.

[34] M. Hill. Multiprocessors should support simple memory-consistency models. In *IEEE Computer* 1998.

[35] S. Owens, S. Sarkar and P Sewell. A better x86 memory model: x86-TSO. In *TPHOLs* 2009.

[36] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP* 2010.

[37] J. Burnim, K. Sen and C. Stergiou. Sound and complete monitoring of sequential consistency in relaxed memory models. In *Tech. Rep. UCB/EECS* 2010.

[38] A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN* 2010.

[39] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in PSO memory systems. In *TACAS* 2013.

[40] T. L. Gall, B. Jeannet and T. Jron. Verification of communication protocols using abstract interpretation of FIFO queues. In *AMAST* 2006.

[41] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *SPAA* 1995.

[42] T. Huynh and A. Roychoudhury. A memory model sensitive checker for CSharp. In *FM* 2006.

[43] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems* 1993.

[44] S. Burckhardt, R. Alur, and M. Martin. Bounded verification of concurrent data types on relaxed memory models: A case study. In *CAV* 2006.

[45] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI* 2007.

[46] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *CAV* 2004.

[47] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS* 2004.

[48] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *MSR-TR-2008-12* 2008.

[49] M. Kuperstein, M. Vechev and E. Yahav. Automatic inference of memory fences. In *FMCAD* 2010.

[50] M. Kuperstein, M. Vechev and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI* 2011.

[51] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI* 2008.

[52] M. Vechev, E. Yahav and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL* 2010.

[53] M. Vechev, E. Yahav, D. F. Bacon and N. Rinetzky. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* 2007.

[54] M. Vechev, E. Yahav and G. Yorsh. Inferring synchronization under limited observability. In *TACAS* 2009.

[55] A. Dan, Y. Meshman, M. Vechev and E. Yahav. Predicate Abstraction for Relaxed Memory Models. In *SAS* 2013.

[56] Y. Meshman, A. Dan, M. Vechev and E. Yahav. Synthesis of Memory Fences via Refinement Propagation. In *SAS* 2014.

[57] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI* 2012.

[58] B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics In *OOPSLA* 2013.

[59] X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS* 2003.

[60] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. In *IEEE Trans. Comput.* 2001.

[61] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *TOPLAS* 1988.

[62] J. Alglave, L. Maranget, S. Sarkar and P. Sewell. Fences in Weak Memory Models. In *CAV* 2010.

[63] J. Alglave, D. Kroening, V. Nimal and D. Poetzl. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *CAV* 2014.

[64] J. Alglave, D. Kroening, V. Nimal and M. Tautschnig. Software Verification for Weak Memory via Program Transformation. In *ESOP* 2013.

[65] J. Alglave. A Shared Memory Poetics. PhD. Dissertation, University of Paris 7, 26 November, 2010.

[66] T. Elmas. Techniques for Runtime Monitoring and Static Verification of Concurrent Software. PhD. Thesis 2010.

[67] M. Faouzi, A. Bouajjani and G. Parlato. Getting Rid of Store-Buffers in TSO Analysis. In *CAV* 2011.

[68] P. A. Abdulla, M. F. Atig, Y. F. Chen, C. Leonardsson and A. Rezine. Memorax, a Precise and Sound Tool for Automatic Fence Insertion under TSO. In *TACAS* 2013.

[69] P. A. Abdulla, M. F. Atig, Y. F. Chen, C. Leonardsson and A. Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS* 2012.

[70] T.A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Lazy abstraction In *POPL* 2002.

[71] V. Vafeiadis and F.Z. Nardelli. Verifying fence elimination optimisations In *SAS* 2011.

[72] A. Bouajjani, E. Derevenetc and R. Meyer. Checking and Enforcing Robustness against TSO. In *ESOP* 2013.

[73] T. Ridge A Rely-Guarantee proof system for x86-TSO. In *VSTTE* 2010.

[74] S. Jagannathan, V. Laporte, G. Petri, D. Pichardie and J. Vitek. Atomicity refinement for verified compilation. In *PLDI* 2014.

[75] E. Koskinen, M. Parkinson and M. Herlihy. Coarse-grained transactions In *POPL* 2010.

[76] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS* 2009.

[77] M.C. Rinard and P.C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. In *TOPLAS* 1997.

[78] P. Prabhu, S. Ghosh, Y. Zhang, N.P. Johnson and D.I. August. Commutative set: a language extension for implicit parallel programming. In *PLDI* 2011.

[79] R.J. Lipton. Reduction: a method of proving properties of parallel programs. In *CACM* 1975.

[80] H. Liang, X. Feng and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL* 2012.

[81] A.J. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL* 2011.

[82] X. Feng. Local rely-guarantee reasoning. In *POPL* 2009.

[83] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. In *CACM* 2010.

[84] H.J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *PLDI* 2008.

[85] M. Batty, S. Owens, S. Sarkar, P. Sewell and T. Weber. Mathematizing C++ concurrency. In *POPL* 2011.

[86] J.C. Blanchette, T. Weber, M. Batty, S. Owens and S. Sarkar. Nitpicking c++ concurrency. In *PPDP* 2011.

[87] S. Sarkar, P. Sewell, J. Alglave, L. Maranget and D. Williams. Understanding POWER multiprocessors. In *PLDI* 2011.

[88] S.M. Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M.M.K. Martin, P.Sewell and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV* 2012.

[89] J. Alglave, A.Fox, S.Ishtiaq, M.O. Myreen, S.Sarkar, P.Sewell and F.Z. Nardelli. The semantics of power and ARM multiprocessor machine code. In *DAMP* 2009.

[90] J. Sevcik, V. Vafeiadis, S. Jagannathan and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. In *JACM* 2013.

[91] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA* 2013.

[92] R. Ferreira, X. Feng and Z. Shao. Parameterized memory models and concurrent separation logic. In *ESOP* 2010.

[93] M. Batty, M. Dodds and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL* 2013.

[94] I. Kuru and S. Tasiran. QED4TSO : Atomicity Refinement for TSO Programs. In *Preparation* 2015.

[95] I. Kuru, B. Ozkan, S. Mutluergil, S. Tasiran, T. Elmas and E. Cohen Verifying Programs under Snapshot Isolation and Similar Consistency Models. In *TRANSACT* 2014.

[96] M. Barnett, B. Chang, R. DeLine, B. Jacobs and K. Leino. Boogie: A Modular Reusable Verifier fo Object-Oriented Programs. In *FMCO* 2005.

[97] H. Muller and W. Kathy Threads and Swing. In *Java Sun Swing Doc* 1998.

[98] S. Oaks and W. Henry. Java Threads. In *O'Reilly* 1997.

[99] M. Scott. Transactional Memory Series. In *HIPEAC ACACES* 2010.

VITA

İsmail Kuru is from North-East region of Turkey, Sürmene, Trabzon. He received his B.S. in Computer Engineering from Dokuz Eyül University (Izmir, Turkey) in June 2010. He started his M.S study in Technical University of Munich in September 2010. During his existence in Europe collaborated with Prof. Albert Cohen from Inria-Paris in Teraflux project and Dr. Josef Weidendorfer from TU Munich in MAPCO project. He participated in Google Summer of Code'11 and contributed to GCC. He has been pursuing his M.S study since September 2011 under supervision of Prof. Serdar Taşıran at Koç University. During his M.S study in Koç University, he has been a research and teaching assistant at Koç Research Center for Multicore Software Engineering with Microsoft Research PhD. scholarship. His research interests include formal methods and tool design, analysis, and verification of concurrent software. To be more specific, they include building type systems, proof methods to provde important guarantees for highly critical systems softwares. His research is supported by Microsoft Research via PhD. scholarship program in EMEA and Scientific and Research Council of Turkey (TÜBİTAK). Ismail was intern at University of Washington under supervision Prof. Serdar Taşıran, Dr. Shaz Qadeer where he was hosted by Prof. Dan Grossman during September 2013 - March 2014. He was also intern at Microsoft Research Cambridge, UK during April 2014 - July 2014 under supervision of Dr. Matthew Parkinson.