Software lock elision for x86 machine code

Amitabha Roy

University of Cambridge
Computer Laboratory
Emmanuel College

April 2011

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Software lock elision for x86 machine code

Amitabha Roy

Summary

More than a decade after becoming a topic of intense research there is no transactional memory hardware nor any examples of software transactional memory use outside the research community. Using software transactional memory in large pieces of software needs copious source code annotations and often means that standard compilers and debuggers can no longer be used. At the same time, overheads associated with software transactional memory fail to motivate programmers to expend the needed effort to use software transactional memory. The only way around the overheads in the case of general unmanaged code is the anticipated availability of hardware support. On the other hand, architects are unwilling to devote power and area budgets in mainstream microprocessors to hardware transactional memory, pointing to transactional memory being a "niche" programming construct. A deadlock has thus ensued that is blocking transactional memory use and experimentation in the mainstream.

This dissertation covers the design and construction of a software transactional memory runtime system called SLE_x86 that can potentially break this deadlock by decoupling transactional memory from programs using it. Unlike most other STM designs, the core design principle is *transparency* rather than *performance*. SLE_x86 operates at the level of x86 machine code, thereby becoming immediately applicable to binaries for the popular x86 architecture. The only requirement is that the binary synchronise using known locking constructs or calls such as those in Pthreads or OpenMP libraries. SLE_x86 provides speculative lock elision (SLE) entirely in software, executing critical sections in the binary using transactional memory. Optionally, the critical sections can also be executed without using transactions by acquiring the protecting lock.

The dissertation makes a careful analysis of the impact on performance due to the demands of the x86 memory consistency model and the need to transparently instrument x86 machine code. It shows that both of these problems can be overcome to reach a reasonable level of performance, where *transparent* software transactional memory can perform better than a lock. SLE_x86 can ensure that programs are ready for transactional memory in any form, without being explicitly written for it.

Acknowledgements

I gratefully acknowledge my supervisor Steven Hand and adviser Tim Harris for all the advice, discussion and feedback they have provided throughout my time at Cambridge. In particular, I am thankful for all the drafts they have read through patiently. Their support has helped me gain considerable expertise and confidence. This thesis has also benefited from feedback by Robert Mullins and Ian Watson. I am also grateful to the Cambridge and Nehru trusts, the Overseas Research Scholarship, the Freshwater Fund and the Lundgren Research Award for funding my studies at Cambridge.

A ton of thanks go out to my family, particularly my mother and father for having believed in me and for putting up with my insistence on exploring the unknown instead of settling down. This PhD simply would not have been possible without the both of you. Thanks also go out to the rest of the "troops" and yes, all the cats !

This thesis is indebted to Mohita, my wife, who I met here at Cambridge. I plead guilty to spending, perhaps, too much time "solving things in my head" as you have so rightly observed. I promise to henceforth spend much more time with you. Indeed, Cambridge would have a been an awfully lonely place without you by my side. Thanks also go out to your family for having welcomed me into their fold.

Finally I'd like to acknowledge my friends for the Mexican van (Derek, Eiko, Anil, Theo, Malte and Chris), for all the squash (Nishant), the action movies (Pradipta), the table football (Anastasios, Fei, Fernando and Periklis) and indeed too many to name here.

Some of the experiments for this thesis were performed on the COSMOS Consortium supercomputer within the DiRAC Facility jointly funded by STFC, the Large Facilities Capital Fund of BIS and the University of Cambridge. I am grateful to them for the use of the SGI Altix UV1000 supercomputer.

Contents

1 Introduction **17**

1.1 Motivation 17

1.2 Contributions 20

1.3 Outline 20

1.4 Publications 21

2 Background **23**

2.1 Hardware transactional memory 24

2.2 Software transactional memory 27

2.2.1 Non-blocking data structures 28

2.2.2 Non-blocking STMs 28

2.2.3 Lock-based STMs 30

2.2.4 Word-based and object-based STMs 32

2.3 STM instrumentation 33

2.3.1 Library 33

2.3.2 Compiler 35

2.3.3 Binary rewriting 36

2.4 Atomicity specifications 36

2.4.1 Atomic block 37

2.4.2 Lock elision 38

2.5 Weak vs strong atomicity 41

2.6 Single lock atomicity and memory consistency 43

2.7 Hybrid transactional memory 44

2.8 Performance benefits of transactional memory 45

2.9 Software lock elision for x86 machine code 46

3 x86-safe software transactional memory 49

3.1 x86 memory consistency model 49

3.2 Counterexamples 55

3.3 A serialising design: STM_x86_strict 58

3.3.1 STM primitives 58

3.3.2 Speculation phase 59

3.3.3 Commit phase 61

3.3.4 SLA speculation 63

3.4 Recovering scalability: STM_x86 67

3.4.1 Publication safety 72

3.4.2 Aborts 73

3.5 Comparison with language level memory models 73

3.5.1 Memory update consistency 74

3.5.2 Speculation safety 75

3.5.3 Dynamic separation 75

3.6 Mapping critical sections to SLA 75

3.7 Mixing locking with transactions 78

3.8 Implementation 80

3.8.1 Metadata and logging 80

3.8.2 Arbitrary granularity logs 80

3.8.3 Lock blacklists 83

3.9 Evaluation of STM_x86 83

3.10 Discussion 87

4 x86 machine code instrumentation 89

4.1 Approach 89

4.1.1 Static binary rewriting 89

4.1.2 Dynamic binary rewriting 90

4.1.3 Combining static and dynamic techniques 91

4.2 x86 instrumentation modes and backends 91

4.3 Active mode 93

4.3.1 Identifying critical sections 94

4.3.2 Basic block discovery 95

4.3.3 Basic block instrumentation 96

4.3.4 PIC operations 98

4.4 Passive mode 99

4.4.1 Preparation 99

4.4.2 Intercept and dispatch 100

4.5 Exceptions 101

4.6 Checkpoints 102

4.6.1 Registers 102

4.6.2 Memory 102

4.7 SLE_x86 in practise 104

4.7.1 Backends 105

4.8 Evaluation of the instrumentation system 105

4.9 Evaluation of SLE_x86 107

4.10 Discussion 108

5 Critical section characterisation 111

5.1 Disjoint access parallelism 112

5.2 Lock contention 113

5.3 Profiler operation 115

5.4 Profiling 116

5.5 Post processing 117

5.6 Characterising a Microbenchmark 119

5.7 Characterising STAMP 120

5.8 Discussion 123

6 Thread-private data tags 125

6.1 Generic capabilities 125

6.2 Associating tags with locations 128

6.3 Applications 129

6.3.1 OpenMP thread private data 129

6.3.2 Stack data 130

6.4 Adaptive tagging for STAMP 131

6.4.1 Tag metadata and allocation sites 133

6.4.2 The non-transactional access problem 134

6.4.3 Reducing undo-logging overheads 136

6.4.4 Inlining checks 137

6.5 Evaluation 137

6.6 Discussion 142

7 Applicability 145

7.1 Scalability 146

7.2 Impact on software development 148

7.3 Condition variables and fine-grained locks 151

7.3.1 Facesim 153

7.3.2 Fluidanimate 153

7.4 Discussion 156

8 Conclusion 157

8.1 Summary 157

8.2 Future research 159

Appendix A: SLE_x86 restrictions 160

Appendix B: Experiment configurations 162

List of Figures

2.1 Sorted linked list using Herhily et al's HTM instructions 25

2.2 Sorted linked list using Hammond et al's TCC 26

2.3 Linked list using the TL2 library 34

2.4 Sorted linked list using atomic blocks 37

2.5 A barrier for two threads 38

2.6 Privatisation 42

2.7 Publication 43

2.8 Memory ordering differences 44

3.1 TSO machine 50

3.2 Loads must be ordered after earlier transactions 55

3.3 Stores must be ordered across atomic blocks 57

3.4 `mfence` in a transaction 63

3.5 Visibility of writes from a transaction 64

3.6 Privatisation 68

3.7 Optimisation trades generality for scalability 72

3.8 Publication 73

3.9 Memory consistency model vs weak atomicity 74

3.10 Memory consistency implications for an STM 74

3.11 Symmetric dependent visibility 76

3.12 Buggy signalling 77

3.13 Type of asymmetric data race 78

3.14 Bypassing writes to reads 81

3.15 STM performance on the STAMP benchmarks(1) 84

3.16 STM performance on the STAMP benchmarks(2) 85

3.17 TL2: Maximum variation in execution time as a fraction of the median 86

3.18 Is using STM_x86 better than using the lock ? ('*' means also better than sequential) 86

4.1 Possibly nested locking 90

4.2 Execution in active mode 93

4.3 The same function in multiple contexts 94

4.4 Shared memory instrumentation for a basic block 97

4.5 Execution in passive mode 99

4.6 Intercept and dispatch 101

4.7 Checkpointing the stack 103

4.8 Putting it together: Using SLE 104

4.9 STAMP: Static characteristics 106

4.10 Basic blocks added to PIC in each active execution iteration 106

4.11 Binary instrumentation overhead 107

4.12 SLE on the STAMP benchmarks(1) 108

4.13 SLE on the STAMP benchmarks(2) 109

4.14 Is using SLE_x86 better than using the lock ? ('*' means also better than sequential) 109

5.1 Profiler phases 116

5.2 Example profiling output 117

5.3 Red-Black tree memory access characterisation 120

5.4 STAMP critical section memory operations 121

5.5 A fragment of code from the labyrinth benchmark 122

5.6 STAMP critical section characteristics 123

6.1 Association of tag metadata with pages 129

6.2 Example of OpenMP ThreadPrivate from Quake 129

6.3 Sharing stack locations in the Bodytrack benchmark from PARSEC 132

6.4 A fragment of code from the Genome benchmark 133

6.5 Extended tag metadata for adaptivity 134

6.6 Extended tag metadata with mirror mappings 136

6.7 Inlined tag check 137

6.8 SLE (with private data tagging) on the STAMP benchmarks(1) 138

6.9 SLE (with private data tagging) on the STAMP benchmarks(2) 139

6.10 Yada data structure: original (top) and separated (bottom) 140

6.11 Performance of Yada after data structure decomposition 141

6.12 Labyrinth source fragment: original (top) and annotated(bottom) 141

6.13 Performance of Labyrinth with annotation 142

6.14 Is using Best Of(SLE_x86, SLE_x86 with PDT) better than using the lock ? ('*' means also better than sequential) 142

7.1 Microbenchmark profiles(8 threads) 147

7.2 Scaling on a Corei7 NUMA system 147

7.3 Scalability at CPU socket intervals 148

7.4 Quake using SLE 149

7.5 Quake using different implementations of critical sections 150

7.6 Quake profile 150

7.7 Quake using different implementations of critical sections (older Xeon system) 151

7.8 PARSEC critical section memory operations 154

7.9 Using SLE on Facesim 155

7.10 Using SLE on Fluidanimate 155

7.11 Using SLE on Fluidanimate with Single Lock Atomicity 156

Chapter 1

Introduction

This dissertation describes the design, construction and evaluation of a system that applies software transactional memory (STM) to an x86 binary synchronising with locks. The system aims to improve scalability when coarse-grained locks are a bottleneck in the program while guaranteeing that the result of execution of the program is indistinguishable from that were locks to be used.

In this chapter I outline the motivation for this work. I also summarise the contributions of the dissertation and list the contents of the following chapters that describe the contributions in detail.

1.1 Motivation

Locks are the most popular method of synchronisation in multithreaded programs. With increasing thread counts in systems due to the availability of multicores, coarse-grained locks can become a scalability bottleneck in programs. This has led to a tension between coarse-grained and fine-grained locking in programs. Coarse-grained locking is easier to get right since fine-grained locking is (a) prone to programming errors such as deadlock and (b) makes composable software engineering hard since calls that cross multiple modules need to be aware of held locks in order to avoid deadlock. On the other hand fine grained locks provide better scalability since critical sections do not need to wait on the same lock if they need to access different shared memory locations.

A simple example serves to illustrate the problem of locking granularity. Consider the problem of implementing a concurrent red-black tree. A straightforward solution would be to start from a sequential implementation of red-black trees available in a standard textbook [CLRS01] and protect the tree with a single coarse grained lock. A more scalable solution is to use the considerably more complicated relaxed version of red-black trees [HOSS97] that relaxes the data structure by decoupling insertion and deletion from re-balancing. One could then refer to Fraser's dissertation [Fra03] that includes a fairly non-trivial fine grained locking protocol for relaxed red-black trees.

The parallel programming research community has been investigating transactional memory [HM93] as the solution to this problem. Hardware transactional memory provides special hardware support for executing blocks of code transactionally. Software transactional memory implements transactional memory entirely in software making the benefits of transactions available to programs running on current microprocessors without the need to wait for hardware transactional memory to become available. When using transactional memory, rather than explicitly waiting for and acquiring any coarse-grained locks, execution proceeds speculatively and uses a two phase commit protocol similar to database style transactions. Transactions thus only need to wait for each other when there are conflicting accesses to the same piece of shared data. In the example, two updates to a red-black tree can proceed in parallel, if they modify different subtrees. This is usually the case for large enough trees and thus optimistic concurrency control can provide the scalability of fine grained locking without the associated complexity (a fact that Fraser also demonstrated [Fra03]). However, in spite of almost a decade of research into transactional memory *there is no transactional memory hardware nor any examples outside the research community of the use of software transactional memory*.

Using software transactional memory is too disruptive to software engineering for large pieces of software. The stumbling block to adoption is a clean integration of transactional memory into the software stack. Integrating transactional memory into a language requires defining the semantics of a construct that can expose transactional memory to the programmer (such as an *atomic block* [HF03]) and ensuring that the software transactional memory implementation preserves those semantics *and* the language memory model (which in the case of many widely used languages is only just being defined). Further, an STM requires either tedious and error-prone instrumentation of source to use an STM library or equally tedious annotation of source to use experimental STM compilers that are yet to agree on language extensions. Finally precompiled library calls and system calls (and I/O in general) are difficult to use in transactions unless one is prepared to sacrifice scalability by serialising transaction execution in order to execute *irrevocably* [WSAT08]. At the same time, overheads associated with software transactional memory fail to motivate programmers to expend the needed effort to use software transactional memory. The only way around the overheads in the case of general unmanaged code is the anticipated availability of hardware support.

On the other hand, architects are unwilling to devote power and area budgets in mainstream microprocessors and deal with the complexity of hardware transactional memory implementation. They point to transactional memory being a “niche” programming construct. Further, the limitations of hardware means that a hybrid solution where some transactions can be executed in software is necessary. Unfortunately, the lack of mainstream acceptance of software transactional memory means that even if hardware transactional memory were to become available it would be some time before programs could be modified to take advantage of it.

A deadlock has thus ensued that is blocking transactional memory use and experimentation in the mainstream. These problems have prompted some researchers to label software transactional memory a research toy [CBM$^+$08] and raise serious questions about the future of transactional memory in general.

The crux of the problem with integrating transactional memory into language level synchronisation is that it is contrary to the systems design principle of "separating mechanism from policy". Transactional memory is a mechanism for synchronisation with optimistic concurrency control and is not a sound and easy way to *specify* (or think about) synchronisation. Hence this dissertation proposes leaving the well known and ubiquitous lock as the means for specifying synchronisation in programs. A coarse grained lock can (and should) be used for easy and maintainable synchronisation.

This leaves the question of when to apply the mechanism of "transactional memory". This thesis proposes applying it as late as possible: to machine code at runtime and making it an *optional* mechanism of execution. Software transactional memory is used to elide[1] lock acquisition and speculatively execute the critical section as a transaction. Applying it late means that the software development environment stays unchanged. It also means that the mechanism is language and compiler neutral (and requires no changes to either). Legacy code (including libraries) is no longer a problem. Keeping it optional means that the program can be simply executed with locks for debugging as usual. Locking can also be selectively used for critical sections that do operations that are difficult to reconcile with transactional memory such as system calls, *without* requiring other unrelated transactions to stop running as with current proposals for irrevocability in STMs.

These benefits are attractive enough for various other researchers to design and implement lock elision systems. Other proposals for lock elision so far either involve unavailable hardware [RG01]; are limited to Java [ZWAT$^+$08]; or require the programmer to convert code to use a new synchronisation primitive together with a special compiler [USB09].

This dissertation presents the design and implementation of a software lock elision system (SLE_x86) that uses as a starting point an x86 binary synchronising with locks. Although the implementation I describe is x86 specific the general approach could be replicated for other architectures. It provides a clean separation of mechanism from policy while being usable on current off-the-shelf hardware. Additionally I show how the infrastructure built for SLE_x86 can be used to build a runtime profiler for x86 binaries. This provides statistics about critical sections in the binary and insights into their possible interaction with transactional memory; particularly to explain the benefit or lack thereof that can be obtained through the use of software lock elision.

[1] Elision *n.*: A deliberate act of omission.

The driving design principle for SLE_x86 is *applicability* and *transparency*. It is a widely and easily applicable software transactional memory runtime system, since it requires *nothing* from the programmer in terms of awareness of transactional memory. It makes software transactional memory backward-compatible with existing code. Unlike other STM designs, performance is only a secondary design goal for SLE_x86 and the design sacrifices performance when needed for transparency.

1.2 Contributions

It is my thesis that software transactional memory can be automatically, transparently and correctly applied at runtime to critical sections in binaries that synchronise with locks and are not constructed to have any awareness of software transactional memory.

The first key contribution of this dissertation is the design and construction of a software transactional memory that preserves the x86 memory consistency model. I start by showing that it is *impossible* for a non-trivial STM to preserve the x86 memory consistency model for all programs. I then show that it is possible to recover scalability by excluding programs that contain a certain type of data race. I argue that such races likely correspond to program bugs and there exist practical techniques to detect such races.

The second key contribution of this dissertation is a lightweight low-overhead mechanism to instrument critical sections in lock-based x86 programs. My instrumentation infrastructure is built with two key design goals in mind. The first is that the cost of inserting instrumentation should be zero. The second is that the execution overhead outside critical sections should be zero. These ensure that software lock elision is “pay to use” in terms of overhead. The instrumentation infrastructure also incorporates a number of novel methods to safely filter out thread-private accesses to further reduce overhead.

1.3 Outline

I now outline the organisation of the rest of this dissertation.

In Chapter 2 I survey background literature and place the ideas of this dissertation in proper context. I also delineate my contributions in this dissertation with reference to previous work in lock elision and profilers that attempt to to quantify program suitability for the use of optimistic concurrency control.

In Chapter 3 I consider the requirements for building an STM runtime that can be used to elide locks and execute critical sections concurrently. Preserving the semantics of x86 machine code requires considering the behaviour of the program in terms of the memory model of the underlying hardware. The STM runtime should thus strive to preserve the memory model even while applying optimistic concurrency control to critical sections. There have been no previous STM designs that have considered a hardware memory consistency model and the focus of the chapter is on designing an STM algorithm that preserves the x86 memory consistency model. I also include an evaluation of the STM against designs that are *more* scalable but only support weaker *language level* memory consistency models.

In Chapter 4, I present a lightweight and low overhead technique to instrument shared memory accesses within critical sections of x86 programs. My technique builds on an existing stable binary rewriting engine but removes associated overheads. The chapter also includes an evaluation of the instrumentation system on its own (in terms of overhead over manual instrumentation) and an evaluation of a complete system for software lock elision in combination with the STM design of the previous chapter.

In Chapter 5 I present the design and construction of a profiler that makes use of this instrumentation to profile the suitability of an x86 binary for using transactional memory. I identify the two key metrics that need to be measured (lock contention and critical section disjoint access parallelism) and show how they can be measured with minimum error from executions of the binary. This profiler is extremely useful to debug the performance of SLE and, being TM agnostic, is useful to characterise the suitability of x86 binaries for TM in general.

In Chapter 6 I tackle one of the key problems with the SLE system built and evaluated in the previous chapters: instrumentation of thread-private data leads to significant overheads for some benchmarks over an STM applied through manual instrumentation. I present a technique for automatically distinguishing thread-private data and safely eliminating STM related overheads for them.

In Chapter 7 I evaluate software lock elision in a more general setting. In previous chapters, I focus on a single comprehensive benchmark (the STAMP benchmark suite) geared towards exercising speculation. In this chapter I compare with compilers that support STM at a language level, on systems architected at a hardware level for scalability and finally on the more real world benchmark of the Quake game server.

In chapter 8, I conclude and provide some suggestions for further areas of research.

1.4 Publications

Some of the research described in this dissertation also appears in the following publications:

1. Amitabha Roy, Steven Hand and Tim Harris. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the Conference on Virtual Execution Environments*, 2011.

2. Amitabha Roy, Steven Hand and Tim Harris. Poster: Weak atomicity under the x86 memory consistency model. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2011.
3. Amitabha Roy, Steven Hand and Tim Harris. Exploring the Limits of Disjoint Access Parallelism. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2009.
4. Amitabha Roy, Steven Hand and Tim Harris. A runtime system for software lock elision. In *Proceedings of the European Conference on Computer Systems*, 2009.

Of these, the last mentioned publication described a system with the exact *opposite* design goals to the one in this dissertation. It covered the design of a software lock elision system that emphasised performance over transparency, requiring the programmer to manually annotate source code and only supporting the weaker C++ memory model.

Chapter 2

Background

Transactional memory for general purpose software was inspired by database transactions and aims to bring the benefits of optimistic concurrency control to synchronisation problems in multithreaded code. Starting with the earliest proposals [HM93, ST95], the fundamental idea behind transactional memory has been to not wait for access to data, on the premise that critical sections will not, in the common case, make conflicting accesses.

The advantages of this kind of synchronisation can be illustrated by data structures that have potential for disjoint access parallelism, such as red-black trees that have historically been an important benchmark for the transactional memory community. If one were to consider a large tree and threads making simultaneous lookup, update and delete operations, it is highly likely that a read access will not conflict with a write access since the updates would be confined to a subtree not accessed by a reader. The data structure and its access patterns thus exhibit disjoint-access-parallelism [IR94] i.e. parallelism emanates from the fact that concurrent accesses are likely made to disjoint sets of locations.

Given this scenario, protecting the entire tree with a lock is likely too conservative. However, omitting the lock would be wrong as there is no guarantee that accesses to the tree from simultaneously executing critical sections are always disjoint. It is possible (but hopefully uncommon) that they will conflict. Transactional memory provides an ideal synchronisation mechanism for such a scenario. Entire computations and shared memory accesses representing one of the possible operations on the tree are encapsulated in a "transaction" and presented to the transactional memory system. The transaction is then executed using optimistic concurrency control. All transactions proceed in parallel. If no conflict is detected the transactions can complete simultaneously. On the other hand if a conflict is detected the transactional memory system has the capability to roll back the transaction that has encountered a conflict and restart it. The effect of such a system on the concurrent accesses to the red-black tree – when the tree is used to implement the set abstract data type – is to present a linearised schedule to the programmer [HW90]. A linearised schedule is a total order on the access operations, consistent with the behaviour of a set, that also respects the constraint that an operation that finishes before another operation begins is also ordered before it in the linearised schedule. The transactional memory system thus presents to the programmer the illusion that threads execute with mutual exclusion even though they are in fact accessing the data structure in parallel.

The rest of this chapter surveys various transactional memory systems starting with hardware transactional memory and then moving on to software transactional memory. I then focus on software transactional memory and discuss ways to apply transactional memory instrumentation to programs. Next, I discuss means for specifying synchronisation to the underlying transactional memory system and its interaction with the memory consistency model. Finally, I discuss the system built in this dissertation in comparison to other transactional memory research.

2.1 Hardware transactional memory

One of the earliest proposals for hardware transactional memory [HM93] proposed allowing a sequence of memory operations to be made atomic by piggy-backing on existing and widely used cache coherence protocols. Herlihy et al. proposed additional instructions for transactional load (`LT`) and transactional store (`ST`), both of which operate on memory addresses. They also added one to validate the currently executing transaction (`VALIDATE`) and finally one to commit the currently executing transaction (`COMMIT`). Using these extensions to the instructions set architecture, one can write code to add a node to a concurrent sorted linked list, as illustrated in Figure 2.1. The example uses the `LT` instructions to load pointers to successive nodes until the right successor has been determined. It then uses the `ST` instruction to insert the new node. If the commit using the `COMMIT` instruction succeeds it breaks out of the loop. Otherwise, it waits for a time dictated by an exponential backoff (to avoid transactions starving each other). Also interesting is the use of the `VALIDATE` instruction. It ensures that the data set of the transaction (locations read and written) have not been accessed by other threads and thus the thread's view of shared memory (in this case the linked list nodes determined to have a smaller value than that of the node being inserted) is *consistent*.

The implementation proposed by Herlihy et al. consisted of an extra transactional cache that was smaller than and held data exclusive from that in the normal cache of the processor. Transactional loads and stores placed their accessed cache lines in the transactional cache. Lines in the transactional cache accessed by the current transaction (such as those holding linked list nodes in the example) were either in `XCOMMIT` or `XABORT` states. The former state labeled lines that were to be discarded on a successful commit, while the latter state was for lines that were to be discarded on an abort. Every line filled in the transactional cache would have a second copy added, with one placed in the `XCOMMIT` and one in the `XABORT` state. Crucially, stores would only be made to lines in the `XABORT` states and thus stores were buffered and discarded in case

```
typedef struct node_st {
    int value;
    struct node_st *next;
} node_t;

node_t * head;

void add_node(node_t * new)
{
    node_t **pcur;
    node_t *cur;

    while(true) {
        pcur = &head;
        while(VALIDATE()) {
            cur = LT(pcur);
            if(cur == NULL) {
                ST(pcur, new)
                break;
            }
            if(LT(&cur->value) >= new->value) {
                new->next = cur;
                ST(pcur, new);
                break;
            }
            pcur = &cur->next;
        }
        if(COMMIT())
            break;
        exponential_backoff();
    }
}
```

Figure 2.1: Sorted linked list using Herhily et al's HTM instructions

the transaction aborted. This capability to roll back speculative changes is fundamental to the way transactional memory (both the hardware and software variants) operates.

Subsequently, hardware transactional memory matured to the point where it was proposed as the *sole* means of synchronisation with coherency and consistency built around the idea of a transaction [HWC$^+$04]. Hammond et al. proposed that *all* code be run in transactions with transaction boundaries marking points where speculative changes are committed back to main memory. Their objective was to simplify the cache coherence and memory consistency for large scale chip multiprocessors. They did away with the conventional snoopy or directory based protocols. Instead, processors arbitrated for commit ordering and broadcasted writes within a transaction as a broadcast packet. Processor caches only snooped the broadcast packets which could

```
typedef struct node_st {
    int value;
    struct node_st *next;
} node_t;

node_t * head;

void add_node(node_t * new)
{
    node_t **pcur;
    node_t *cur;
    TRANSACTION_BEGIN_MARKER()
    pcur = &head;
    while(1) {
        cur = *pcur;
        if(cur == NULL) {
            *pcur = new;
            break;
        }
        if(cur->value >= new->value) {
            new->next = cur;
            *pcur = new;
            break;
        }
        pcur = &cur->next;
    }
    TRANSACTION_END_MARKER()
}
```

Figure 2.2: Sorted linked list using Hammond et al's TCC

result in the currently running transaction being aborted in the event of a conflicting write from a previously committed transaction. The HTM system of Hammond et al. results in the broadcast of occasional large packets (4KB-8KB based on their simulations). This requires larger amounts of bandwidth from the interconnect but is relatively latency insensitive as compared to traditional cache coherence protocols that require low latency transfers of smaller amounts of data. Furthermore, within a transaction there was no need to order reads or writes, with the transactional memory system providing the illusion of sequential consistency to committed transactions. Another interesting contrast between the proposal of Hammond et al. and that of Herlihy made a decade earlier is the software interface. Hammond et al. do away with transactional loads and stores. Instead all operations within transactions are handled transactionally (which is the only mechanism to access memory). The linked list example thus simplifies to Figure 2.2. The simplification is due to the fact that Hammond et al's system was targeted towards general purpose concurrent code rather than simply lock-free data structures.

Both the proposals of Herlihy et al. and Hammond et al. chose to retain the original as well as modified versions of cache lines in a transaction. On a commit the original version was discarded while on an abort the modified version was discarded. In the case of Herlihy's proposal both versions were kept in the transactional cache while Hammond chose to depend on main memory for the original (unmodified) version. Moore et al. [MBM$^+$06] made the observation that in the common case when the commit succeeds, keeping the old version around was wasteful. They came up with an HTM that applied changes directly to cache lines and chose to log old values into a special "before-image" log. Commits reduce to simply flash clearing marking bits for cachelines written in the transaction and discarding the log.

A common problem with early hardware transactional memory proposals had been cases where transactions overflowed the cache (or a set in a set-associative cache). Rajwar et al. proposed a solution to this problem by virtualising the transactional logs into virtual memory of the process [RHL05]. In the (assumed) common case where transactions fit in cache, they use a standard HTM. If the transaction overflows the cache, the data and associated tags are placed in a log in the process's virtual memory. Incoming snoops would need to lookup this log (using a firmware or software walker). They further proposed reducing this cost by maintaining a bloom filter to filter out incoming snoops before initiating an expensive walk of the virtual memory log in software.

In spite of the wealth of research into hardware transactional memory there has thus far been only one manufactured mainstream microprocessor that incorporated general purpose HTM. The Rock microprocessor from the SPARC family (now cancelled) incorporated a best-effort hardware transactional memory [DLMN09]. Modestly sized hardware transactions can be executed using Rock's hardware transactional memory. Interestingly functions calls, "difficult" instructions such as division, and TLB misses are not allowed in a transaction (in addition to capacity limits) and the authors point to this being a serious impediment in their attempts to apply the HTM to various benchmarks. In general the HTM is "best-effort" with no guarantee that a transaction can be executed entirely in hardware. The authors recommend using it to assist an alternative software transactional memory implementation that guarantees eventual transaction commit to obtain best performance without undue restrictions on the contents of transactions (I discuss hybrid transactional memory implementations later in the chapter).

Due to the absence of hardware transactional memory support in mainstream processors, software transactional memory has also received considerable attention in the research community. I discuss software transactional memory in the next section.

2.2 Software transactional memory

Software transactional memory was proposed by Shavit and Touitou in 1995 [ST95], clearly inspired by the then recent proposal of hardware transactional memory by Herlihy et al. However, they wanted to implement general purpose transactional memory entirely in software by making use of an atomic compare-and-swap instruction[1] that was available in most architectures. Furthermore, although the proposal of Herlihy et al. was termed “lock-free” since it did away with mutual exclusion locks, it was in fact blocking because threads could end up infinitely aborting each other. Shavit et al’s solution was implemented entirely in software and allowed threads to execute transactions with a non-blocking guarantee: some thread would make progress as long as any (not necessarily the same) thread was able to execute instructions.

2.2.1 Non-blocking data structures

Software transactional memory has essentially evolved from research into non-blocking data structures. In general non-blocking data structures are those where the suspension of one or more executing threads does not prevent other threads from making progress. This distinguishes such data structure designs from those using a lock, since suspending a thread holding a lock means that other threads requiring the same lock cannot make progress. In the worst case all threads would need the lock leading to no progress in the system. There are three types of non-blocking guarantees:

- Wait freedom guarantees that every thread completes it operation within a constant number of steps regardless of the suspension of other threads.
- Lock freedom guarantees that as long as some thread in the system takes steps, some (not necessarily the same) thread will complete its operation on the data structure.
- Obstructions freedom guarantees that any thread that continues taking steps will complete its operation regardless of the suspension of other threads.

Wait freedom, lock freedom and obstruction freedom thus provide successively weaker guarantees. Various algorithms have been proposed for data structures with non-blocking guarantees, such as lock free double-ended queues [Mic03] and their obstruction free variant [HLM03]. The primary difficulty with these designs is that they are limited to specific data structures and algorithms. One of the motivators for early STMs was that they could be used to implement any data structure with the needed non-blocking guarantee, starting from simple single-threaded algorithms for that data structure.

2.2.2 Non-blocking STMs

Shavit et al.’s work spawned a number of STM designs that incorporated some form of non-blocking guarantee. These were classified (in retrospect) into two primary classes.

[1] `cas(loc, old, new)`: atomically check if the current contents of `loc` is `old` and if so set it to `new`. Return the contents of `loc` that was read.

The first class of STM designs comprises of lock-free ones. As described above an STM design is lock-free if some thread makes progress towards completing a transaction as long as *any* thread is able to execute instructions. The underlying feature of lock-free STM designs is “recursive helping”. A thread on encountering an obstruction in executing its transaction (possibly due to the obstructing thread being unable to run) helps the obstructing thread complete its transaction before continuing execution of its transaction. Shavit et al.’s original proposal was lock-free but could deal with only static sized transactions (where the set of reads and updates were known a-priori). Also Shavit et al.’s proposal was evaluated on a simulator and thus left some question about the practical applicability of lock-free STM designs. A more practical STM for dynamic data structures that is also lock-free is Keir Fraser’s Object-Based Software Transactional Memory (OSTM) [Fra03]. One of the key contributions of Fraser’s thesis was a comparison of red-black tree and skiplist data structures on actual NUMA systems when built using the best available fine grained locking mechanisms versus OSTM. He showed that OSTM was able to outperform locks except at low thread counts, thus establishing the practical possibilities for software transactional memory to scale general purpose code.

An alternative to lock freedom for software transactional memory is obstruction freedom. This is a stronger guarantee than that provided by a simple lock since if any other thread holds a necessary lock, suspending it will preclude any thread from completing. On the other hand this is a weaker guarantee than lock freedom. There is no guarantee that any thread makes progression towards completing its transaction: on encountering an obstruction, obstruction free implementations either choose to abort the current transaction or abort the obstructing transaction. It is thus possible for transactions to abort each other indefinitely, leading to livelock. However, obstruction freedom admits substantially simpler implementations than lock-free STMs. One of the earliest proposals for obstruction free STMs was Dynamic Software Transactional Memory (DSTM) [HLMS03]. In addition to proposing obstruction freedom as a simpler non-blocking guarantee, this was one of the first STMs to allow dynamically allocated objects to be accessed transactionally, thus allowing concurrent versions of dynamic data structures such as linked lists and trees to be implemented.

A key challenge in designing transactional memory with any kind of progress guarantee is handling STM metadata. A fundamental building block in non-blocking STM designs is the capability to atomically switch the state of entire objects from the current to the next version when a transaction modifying it commits. For example in the case of DSTM each object consisted of a header that pointed to a transactional locator. The locator object consisted of a pointer to the transaction that had last opened it for write, a pointer to an old version and pointer to a new version. If the transaction pointed to had committed, then the new version was the correct version to use for transactions reading the object. On the other hand if the transaction had aborted, then the old version was the correct version to use. If the transaction was still running, DSTM (in the spirit of obstruction freedom) allowed the accessing transaction to either wait or abort the obstructing transaction. This double indirection is a source of significant overhead and hence later obstruction free designs (such as Rochester Software Transactional Memory [MSH$^+$06]) reduced the number of indirections to one. Non-blocking Zero Indirection Software Transactional Memory (NZSTM) [TWGM07] further reduced this overhead with a metadata design that – in the common case of an uncontended object – required no indirection.

Another interesting challenge, specific to obstruction free STMs, is tackling livelock. The job of deciding whether an obstructing transaction should be aborted or waited for was delegated to a ‘contention manager’, which is responsible for ensuring progress in the system. Obstruction free STMs spawned extensive research into contention management (a good example being that in RSTM [SS05]), which could be plugged into the obstruction free implementation, regardless of the actual STM design. Contention managers usually employed randomised backoff with some form of heuristic to decide when to abort an obstructing transaction. The heuristics are driven by feedback about commit and abort events and present a rich design space. For example, ‘timestamp’ based contention managers give precedence to older transactions. On the other hand ‘karma’ style contention managers take into account the amount of work done by competing transactions when deciding which one to abort.

2.2.3 Lock-based STMs

An alternative to STM systems that provide some kind of non-blocking guarantees are *lock-based* STMs. Ennals [Enn05] argued that non-blocking guarantees were an unnecessary source of overhead in STM design. He based his reasoning on two key observations. The first is that a runtime could tailor the number of running threads to the number of cores (hardware threads) available. This makes it unlikely that an operating system would swap out a thread running a transaction thereby leading to an obstruction. The second is that thread failure still remains a problem for non-transactional versions of the program or non-transactional parts of the same program. Thus, guarding against thread failure with a non-blocking guarantee is not very useful in the practical sense. The most important contribution of Ennals’ work was however the comparison of his lock-based (blocking) STM design against non-blocking STMs. For example, he showed that his algorithm consistently takes only around $50\% - 60\%$ of the time taken by Fraser’s lock free implementation. As a consequence of avoiding object indirection, he also showed that his algorithm incurred only around 50% of the cache misses and 22% of the TLB misses of Fraser’s algorithm for red-black trees.

Lock-based STMs represented a low enough overhead to consider applying STMs to real world programs and spawned much research into building and tuning lock-based STMs. Ennals’ work was followed up by Dice et al.’s Transactional Locking 2 (TL2) [DSS06] algorithm that made important advances on the practicality front over the original proposal. It proposed to decouple metadata from data by using out-of-band metadata. This enabled the STM to be

Algorithm 1 TL2 algorithms

```
TransactionBegin:
ReadVersion := GlobalClock

TransactionalWrite(loc, value):
Append (loc, value) to WriteSet

TransactionalRead(loc):
if (loc, value) in WriteSet then
    // Correctly handle read after write cases
    Return most recent value from WriteSet
else
    current_metadata := metadata for loc
    value := contents of memory at loc
    Check metadata for loc is still at current_metadata
    Check metadata for loc is unlocked
    Check metadata for loc is not greater than ReadVersion
    Append loc to ReadSet
    return value

Commit:
for all (loc, value) ∈ WriteSet do
    lock metadata for loc, using bounded spinning to avoid deadlock
Atomically increment GlobalClock by 2, setting WriteVersion to post-increment value
for all loc ∈ ReadSet do
    check metadata for loc is less than or equal to ReadVersion
for all (loc, value) ∈ WriteSet in order do
    set location loc to value
for all (loc, value) ∈ WriteSet in order do
    unlock metadata for loc setting it to WriteVersion
```

applied “mechanically” through the insertion of read and write “barriers” without requiring extensive data structure changes that would have been necessitated by Ennals’ or previous non-blocking proposals. TL2 is also the basis for the STM design I present in this dissertation and hence I discuss the operation of the TL2 STM in some detail.

TL2 uses an out-of-band array of locks and associates each location in memory with a lock in the array, using a straightforward hash function. Each lock is a simple counter that acts both as a version number and a write lock: if the least significant bit is set then the lock is held. The rest of the bits represent the version number. The other globally shared entity is a “timestamp counter” incremented every time a transaction commits. The timestamp counter is used to ensure that speculating transactions see a consistent view of memory. This is referred to as a global clock in the TL2 design. The key steps involved in executing a transaction are listed in Algorithm 1 (if any of the checks listed at any stage fail, the transaction is aborted). Each thread maintains a local read-set of addresses loaded and a local write-set of address-value pairs.

The transaction begins by calling `TransactionBegin`. Writes within the transaction call `TransactionalWrite` while reads within the transaction call `TransactionalRead`. At the end of the transaction `Commit` is called in order to commit changes back to shared memory, atomically with respect to other transactions.

A number of optimisations are possible to TL2's basic algorithm. Dice et al. themselves proposed a few. For example, checking the local write-set for bypassing values to reads can be done by first checking a bloom filter to filter out the common case where a read does not read any previous write. A number of optimisations are also possible to reduce contention for the global clock. Dice et al. pointed out that incrementing the global clock was unnecessary for transactions that did not perform any writes. A more extensive analysis of commit sequences where an expensive CAS to the globally shared clock could be avoided has also been done [ZBS08].

Another important set of variations to the basic TL2 design can be obtained by changing the point at which locks on written locations are acquired. TL2 followed a lazy approach: locks are acquired only at commit time. An alternative is eager or encounter time locking. In such such a scheme, the lock is acquired when the first write to a location is encountered and writes are performed directly to shared memory. Reads thus no longer need to indirect into the write-set. There is also no need to acquire locks in the commit phase although locks do need to be released. However with an eager scheme it is necessary to maintain an undo log where the old value of locations are logged. On a transaction abort, the old values are restored from the undo log.

The multi-core runtime STM (McRT-STM) [SATH$^+$06] incorporates both eager and lazy locking but Saha et al. conclude that the eager approach performs better than the lazy one. On the flip side, evaluation of both an eager as well as the original lazy versions of the original TL2 algorithm [CMCKO08] reveals that the lazy version can outperform the eager version on some benchmarks. This is because the eager version is more prone to livelock between competing transactions while the lazy version often allows one of the conflicting transactions to finish, thus reducing the amount of wasted work. Other work has shown that an eager STM suffers more in high conflict scenarios due to the cost of applying the undo-log in order to roll back transactions [DS07]. There is thus no clear consensus on which approach is better, with the accepted conclusion being that no one size fits all benchmarks. Hence, considerations other than pure performance typically drive the decision to pick between the approaches.

2.2.4 Word-based and object-based STMs

An important differentiator between STMs of both the non-blocking and lock-based variety is whether they are word-based or object-based. Word-based STMs divide memory into fixed-size chunks and associate external STM metadata with each chunk using a hash function. In contrast, object-based STMs treat memory as composed of variable sized objects with the metadata either embedded in an object or placed externally and associated with the base address of the object. Fraser [Fra03] proposed both object-based and word-based non-blocking designs in his thesis. RSTM [MSH$^+$06] on the other hand is exclusively object-based. TL2 [DSS06] is a word-based STM while McRT-STM [SATH$^+$06] incorporates both an object-based and word-based STM.

Object-based STMs present both advantages and disadvantages over word-based STMs. A key advantage is that the amount of STM metadata manipulated by a transaction decreases if multiple fields of the same object are accessed. On the other hand, object-based STMs require a precise mapping of field accesses to their containing objects. This is not possible when automatically inserting STM calls in code generated from unmanaged environments such as C/C++ unless there is some amount of source code information present to be taken into consideration by the instrumentation system. In general word-based STMs are preferred in such unmanaged environments.

There have been STM designs that try to combine the properties of word-based and object-based STMs. Riegel et al. [RBdB08] proposed a data structure analysis (DSA) pass in their compiler, which could automatically identify object boundaries in compiled code. They then used this information to associate external metadata with object bases rather than with individual fields. Roy et al. [RHH09b] proposed a word-based STM with variable word size. Their proposal however depended on the programmer specifying the object size and base address to their library-based STM.

2.3 STM instrumentation

A key problem with practically applying software transactional memory at scale is that of instrumentation. Using software transactional memory in any program involves at the very least delimiting the start and end of a transaction and, more difficult, indirecting every shared memory access to the STM runtime system. There are two prevalent approaches to the problem. The first uses the STM runtime system as a library and inserts appropriate calls to it directly in the source. The second approach makes use of an STM compiler and depends on annotations in the code to delimit transactions; the compiler can then automatically insert instrumentation for shared memory accesses. A third (not so widely used) option is to insert instrumentation at runtime using a dynamic binary rewriting engine. I discuss these three alternatives next.

2.3.1 Library

Library-based STMs expose an API to the programmer to insert calls at the start and end of a transaction, as well as calls at every shared memory access. For example, Figure 2.3 shows how the linked list example looks when using the TL2 STM API. The start and end of transactions

```
typedef struct node_st {
    int value;
    struct node_st *next;
} node_t;

node_t * head;

void add_node(node_t * new)
{
    node_t **pcur;
    node_t *cur;
    TxStart(...); // TransactionBegin
    pcur = &head;
    while(1) {
        cur = TXLoad(..., pcur); // TransactionalRead
        if(cur == NULL) {
            TxStore(..., pcur, new); // TransactionalWrite
            break;
        }
        if(TxLoad(..., &cur->value) >= new->value) {
            new->next = cur;
            TxStore(pcur, new);
            break;
        }
        pcur = &cur->next;
    }
    TxCommit(...); // Commit
}
```

Figure 2.3: Linked list using the TL2 library

are delimited by `TxStart` and `TxCommit` calls while shared memory loads and stores are indirected through `TxLoad` and `TxStore` calls respectively.

Most of the early STM implementations were made available as library-based STMs (since the alternative of compilers was yet to mature). Library-based STMs suffer from two key problems. The first is the large amount of tedious instrumentation that must be inserted by hand. This can often be a source of errors that negates much of the supposed simplicity of using transactional memory. The second problem with library-based STMs is of rolling back aborted transactions. Library-based STMs use `setjmp` and `longjmp` calls in order to checkpoint and restore state. While this restores registers, it does not restore variables on stack that are known to be thread-private and not instrumented by the programmer. In Figure 2.3 for example, the `pcur` and `cur` variables are changed in the transaction but are not restored on an abort. The programmer is required to ensure that there are no live variables when entering the transaction that are not indirected through the STM.

On the plus side however, library-based STMs empower the programmer to tweak and optimise the instrumentation. For example, the programmer can avoid STM calls for data that is known to be thread-private and not live when entering the transaction. The STAMP benchmark suite [CMCKO08] contains a number of examples of such “programmer-driven optimisation”. In Chapter 6 I discuss techniques by which some of the gap between automatically generated instrumentation and this kind of optimised instrumentation can be closed.

2.3.2 Compiler

Compiler-based software transactional memory uses the compiler to insert STM instrumentation into programs.

Harris et al. were the first to integrate transactional memory support in a Java compiler [HF03]. Subsequently, Adl-Tabatabai et al. implemented transactional memory support in a Java compiler [ATLM$^+$06] using the McRT-STM (discussed before). Their solution provides language extensions to Java that allowed the programmer to delimit transactions. They used a compiler toolkit that translated these language extensions to library calls into their STM. In the JIT phase, they generated a transactional and a non-transactional version of each method and operated with the guarantee that the transactional version would be called only within a transaction. The transactional version had STM-related instrumentation. An interesting aspect of their instrumentation was that they allowed both object-based and word-based software transactional memory to be used on a type-by-type basis.

Harris at al. [HPST06] presented an STM compiler for the Common Intermediate Language (CIL), meant to execute in a managed runtime similar to .NET. Their implementation was for Bartok, an experimental compiler for CIL. The crucial focus of that work was in optimising memory transactions. They showed that by properly decomposing the STM interface it was possible to perform significant optimisations over a naive insertion of STM calls. For example, with their object-based STM, opening an object for writing could be hoisted out of a loop, while leaving the actual updates to object fields within the body of the loop.

Tanger is a transactifying compiler for C/C++ developed by Felber et al. [FFM$^+$07]. Tanger is a not a compiler by itself but rather a pass in the LLVM [LA04] compiler infrastructure. It instruments code in transactions to use a word-based STM. They showed that the compiler was able to insert and optimise STM instrumentation to a level that was equivalent to hand instrumented code, a major step in terms of automation from library based STMs.

Wang et al. [WCW$^+$07] presented an optimising compiler for C/C++ code that supported transactions. They used a variant of the McRT-STM and presented various optimisations that can be done to mitigate the overhead of the STM. A key contribution of Wang et al.’s work however is their focus on *safety*. They showed that integrating an STM into an unmanaged environment such as C/C++ that allowed pointer arithmetic and did not guarantee type safety is extremely challenging. Some of the problems they pointed out, such as the added difficulties in supporting a programming construct called privatisation are also relevant to this dissertation. I return to these problems later in this chapter.

2.3.3 Binary rewriting

A key problem with the compiler-based approach is that it is ineffective when source code is not available. Such a situation occurs, for example, when a transaction calls a legacy library function that has not been 'transactified'. An alternative to compiler-based instrumentation is to start with compiled code and insert instrumentation directly at the machine code level. Some of the early work in using binary rewriting focused on *supporting* STM compilers.

Felber et al. [FFM$^+$07] proposed using a static binary rewriting scheme to instrument such legacy x86 libraries using a tool they called Tarifa (their LLVM module was called Tanger). Tarifa uses a static rewriting scheme (along the lines of ATOM [SE94]) where the legacy library is first disassembled, instrumentation is inserted and finally the instrumented assembly code is assembled back into machine code. Their conclusion was that, while using binary instrumentation did not introduce any fundamental scalability limits, nevertheless it added significant overhead to code generated and optimised by Tanger.

Wang et al. [WYW08] developed a tool called LDBTOM (Lightweight Dynamic Binary Translation and Optimization Module) to allow legacy x86 code to be called within a transaction compiled with an STM capable compiler. LDBTOM was developed to complement the STM compiler developed by the same authors. Wang et al. also report a significant overhead over compiler generated code.

Olszewski et al. [OCS07] developed JudoSTM on top of their binary rewriting engine, Judo. JudoSTM depended on marker functions being inserted in the source code to delimit transactions and inserted transactional barriers automatically. Interesting aspects of their STM included the use of value-based validation and generation of transaction-instance-specific commit sequences – sequences of x86 instructions for each executed transaction – specific to the read and write set. They obtained extremely efficient commit in return for the instruction cache miss costs. JudoSTM reported comparable performance to RSTM (manually instrumented) for counter, linked list and hash table benchmarks.

2.4 Atomicity specifications

Transactional memory is a way to execute sections of code atomically. While the previous sections have surveyed *mechanisms* to execute transactionally this section focuses on *policy* i.e. atomicity related synchronisation constructs used in programs. There have been two primary constructs that assist programmers in specifying synchronisation using transactional memory.

```
typedef struct node_st {
    int value;
    struct node_st *next;
} node_t;

node_t *head;

void add_node(node_t * new)
{
    node_t **pcur;
    node_t *cur;
    atomic {
        pcur = &head;
        while(1) {
            cur = *pcur;
            if(cur == NULL) {
                *pcur = new;
                break;
            }
            if(cur->value >= new->value) {
                new->next = cur;
                *pcur = new;
                break;
            }
            pcur = &cur->next;
        }
    }
}
```

Figure 2.4: Sorted linked list using atomic blocks

2.4.1 Atomic block

The notion of building language-level atomic blocks using transactional memory was introduced by Harris et al. [HF03] in the context of the Java programming language. They advocated the atomic block for declarative concurrency control as a first class language feature.

Although Harris et al.'s original proposal for atomic blocks was in the context of Java, atomic blocks are a popular means for expressing synchronisation when using transactional memory. Both library-based STMs such as TL2 as well as compilers for unmanaged environments such as Wang et al.'s compiler chose to expose transactional memory to the programmer through atomic blocks. Figure 2.4 shows how the sorted linked list example of this chapter can be expressed using atomic blocks.

Atomic blocks have the key advantage of composability when compared to the more imperative style of expressing synchronisation using locks. Consider the problem of removing a node from

```
                   int x=0

// Thread 1                  // Thread 2
lock(l1);                    do {
    lock(l2);                    lock(l2);
        x = 100;                     t2 = x;
    unlock(l2);                  unlock(l2);
    do {                     } while (t2 != 100);
        lock(l2);            lock(l2);
            t1 = x;              x = 200;
        unlock(l2);          unlock(l2);
    } while (t1 != 200);
unlock(l1);
```

Figure 2.5: A barrier for two threads

a sorted linked list and reinserting the node into another, while making the operation appear atomic. A concurrent thread searching both lists for the node (value) should be guaranteed to find it. Thread-safe building blocks for the individual operations are available either using atomic blocks or locks.

Using atomic blocks the solution would simply involve enclosing both operations in a larger atomic block[2], thereby *composing* the smaller fragments into a larger one. Using locks protecting the individual lists on the other hand, one would have to worry about lock acquisition order to avoid deadlock.

2.4.2 Lock elision

Despite the popularity of atomic blocks as a language level interface to transactional memory, researchers have explored lock elision as an alternative. Lock elision assumes that locks continue to be the method of synchronisation. Transactional memory provides a best effort execution of the enclosed critical section by eliding (not acquiring) the lock and hence not waiting for it. If execution of the critical section using the lock fails for some reason, it can be re-executed with the lock held, without using transactional memory. Lock elision has three important advantages over atomic blocks.

The first advantage is that existing synchronisation constructs that use locks need not necessarily have a straightforward transformation to atomic blocks. Starting from legacy code, one might be tempted to simply remove the lock and unlock calls and instead enclose critical sections in atomic blocks. This is not always correct. Consider a barrier between two threads, as shown in Figure 2.5. The lock `l2` protects the shared variable `x` that is used for synchronisation

[2] Atomic blocks allow themselves to be nested.

between the threads. In the case of `Thread 1` the entire barrier code is executed while holding lock `l1`. If the locks were to be replaced with (nested) atomic blocks in a straightforward transformation, no forward progress is possible since the atomic blocks cannot be serialised in a way that allows forward progress[3]. On the other hand, the commonly used Single Lock Atomicity (SLA) semantics of atomic blocks can be simulated with a single process-wide recursive lock.

The second reason is legacy code. In spite of much research focus on atomic blocks, locks continue to be the most popular method of synchronisation for programmers. Lock elision allows the scalability benefits of transactional memory to be obtained for legacy code without requiring it to be be rewritten using atomic blocks. Even with an STM compiler and rewritten source code, legacy libraries become a problem if they acquire locks.

A third reason for lock elision to be preferred is flexibility. With atomic blocks as the sole means of synchronisation, every critical section must execute transactionally. This can be detrimental in some scenarios. For example, if a critical section increments a shared counter, it will inevitably suffer from a high conflict rate. Lock elision on the other hand decouples transactional memory from synchronisation. It is possible to choose to elide some locks but not others for either performance (as some researchers have shown) or (as I show in this dissertation) for correctness reasons.

Hardware lock elision

The idea of lock elision was first proposed by Rajwar et al. [RG01] and described in detail by Ravi Rajwar in his PhD thesis [Raj02]. Their proposal for speculative lock elision was implemented using hardware transactional memory. Critical sections were detected by looking for silent store pairs: two atomic updates to the same location whose cumulative effect was to leave that location unchanged. The critical section itself was executed using hardware transactional memory. The memory occupied by the lock variable was explicitly added to the read-set of the transaction. Thus, any other thread could choose to acquire the lock (writing to it) and execute the critical section non-transactionally, and the system would ensure that any other transactional executions were invalidated.

Another example of lock elision using hardware transactional memory is TxLinux [RHP$^+$07]. In this work, the researchers replaced conventional spinlocks in the Linux kernel with a “cooperative transactional spinlock” (`cxspinlock`) that could either be acquired transactionally or acquired normally as a spinlock. Normal spinlocks were used when the critical section protected by the spinlock did I/O.

Azul systems[4] support lock elision using HTM in their proprietary systems. Lock elision is aimed at accelerating code running in Java Virtual Machines. Since the system is proprietary,

[3] This is a race-free variant of a similar example in [MBL06].

[4] `http://www.azulsystems.com/events/vee_2009/2009_VEE.pdf`

few details are available about it.

Researchers working with hardware transactional memory in the Rock Microprocessor also experimented with eliding locks [DLMN09]. They were motivated by the possibility of applying their HTM to legacy software written to use locks and implemented Transactional Lock Elision (TLE) for data structures in the C++ STL and Java. They showed that far better scalability resulted when using transactions to elide lock acquisitions although they had to modify the data structure implementations to get around some of the failings of the Rock HTM, as discussed previously.

Software lock elision

Ziarek et al. [ZWAT$^+$08] proposed a unification of Java's monitors with transactional memory. In their "unified execution environment" programmers could choose to synchronise using either the Java `synchronized` keyword or atomic blocks. Both kinds of critical sections were executed using transactions. In the event that the critical section could not be executed transactionally (eg. if it made native calls) execution would fall back to pessimistic locking along the lines of traditional Java monitors.

Nakaike et al. proposed eliding the lock for read-only critical sections in Java [NM10]. They implemented a sequence lock[5] where the lock consisted of a counter, with an odd count indicating that the lock is held. Readers simply took a snapshot of the counter and, if it was even, proceeded into their critical section eliding lock acquisition. At the end of their critical section and at various validation points they would ensure that the counter value remained unchanged which guaranteed that no writer could have overwritten any data which has been read. For read-mostly microbenchmarks they showed significant speedups over reader-writer locks since they avoided an expensive atomic operation to acquire the lock in the case of readers.

Roy et al. [RHH09b] proposed software lock elision for C/C++ programs. They wrapped existing locks in a special elidable lock. The critical section was instrumented using a library-based STM. Critical sections could execute either transactionally or after acquiring the lock. An interesting feature of this work was the capability of threads that pessimistically acquired the lock to make progress even if other threads might have speculated past it. This was achieved by using an implementation of revocable locks [HF05] for the fine-grained locks in the underlying STM.

Usui et al. [USB09] proposed an adaptive lock design that could dynamically choose to execute the enclosed critical section using either a lock or a transaction. Their solution involves extending the C language with an `atomic(l)` construct, allowing the enclosed compound statement to be executed either by acquiring the lock `l` or by using a transaction. The choice is based on dynamic (and changing) runtime information. The key focus of their work is on

[5] The authors attribute the idea to sequence locks used in the Linux kernel.

deriving and efficiently implementing a *dynamic* cost-benefit analysis of performance, focusing on whether transactions or locks would be better for a particular critical section.

2.5 Weak vs strong atomicity

A traditional area of difficulty for transactional memory researchers is considering cases where the same memory is accessed concurrently both within and outside a transaction. Depending on the desired behaviour in such a situation Martin et al. [MBL06] classified transactional memory as either providing *strong atomicity* or *weak atomicity*[6].

TM implementations that provide strong atomicity guarantee that the non-transactional access is serialised either before or after any transaction that accesses the same memory. HTM implementations by virtue of modifying existing cache coherence protocols automatically provide strong atomicity. STM implementations on the other hand must pay a price in order to enforce strong atomicity. One approach is to expand all non-transactional accesses into "mini-transactions". Shpeisman et al. [SMAT$^+$07] implemented strong atomicity in a Java STM using this technique. They used a static analysis to determine when objects are never accessed in transactions and dynamic escape analysis to determine when objects never leave transactions in order to reduce the number of STM barriers required (which would otherwise be prohibitive). Abadi et al. [AHM09] present a strongly atomic STM for C# (a managed environment). Instead of attempting to statically reduce barriers in non-transactional code, they used standard memory protection hardware (paging) to detect when code outside transactions accesses data currently being accessed transactionally and patched it into a mini-transaction.

On the other hand, TM implementations providing weak atomicity allow non-transactional accesses to interleave with transactional ones. Most STM implementations provide weak atomicity due to the cost and complexity of providing strong atomicity. Allowing such interleaving however leads to a number of difficult issues from the perspective of the semantics provided to programs. One such problem that has caused concern for almost every weakly atomic STM implementation is *privatisation*. Privatisation has been extensively studied in STM literature [SMAT$^+$07, SMDS07]. Privatisation represents the broad range of issues that arise when objects move from being shared and accessed within transactions to being private to a single thread and accessed without any synchronisation. This is illustrated in Figure 2.6

In this example, `thread 1` privatises a node from a shared linked list by removing it from the list. It next updates the value of the node and finally frees the memory belonging to the node for further reuse. There are two privatisation related problems that arise here.

If the transaction on `thread 2` commits before the transaction in `thread 1`, it might still be in the process of writing back its changes while `thread 1` finishes its transaction and does an

[6] Some researchers instead use the terms *strong isolation* or *weak isolation* respectively.

```
// Thread 1                  // Thread 2
deleteNode(...) {            updateNode(...) {
  ListNode *node;              ListNode *node;
  atomic {                     atomic {
      ...                          ...
      node = ...;                  node = ...;
      ...                          ...
  }                                r = 1/node->value;
  node->value = 0;             }
  free(node);                }
}
```

Figure 2.6: Privatisation

unprotected write to the node. It might thus overwrite the update from `thread 1`. The update from `thread 2` might also come after the node has been freed from `thread 1` and possibly used for a completely different data-type, which could lead to memory corruption.

The second class of problems arises if the transaction on `thread 1` commits first. In this case the transaction on `thread 2` becomes a *zombie* transaction – one that is doomed to abort – since it is working on a node that has been removed from the linked list. `Thread 2` might suffer an unanticipated arithmetic fault if it ends up reading the node's value as 0 (as updated by `thread 1`) for the division.

A variety of solutions have been proposed in STM designs for both kinds of privatisation related problems. Explicit solutions to the privatisation problem involve requiring the programmer to indicate when data moves from being shared to thread private [DSS06, DMS10, SMDS07]. On the other hand, implicit solutions to the privatisation problem add a privatisation related fence to the commit phase of the STM design [SMDS07, WCW$^+$07].

A mirror to the privatisation problem is publication. Figure 2.7 illustrates this construct, where `thread 1` makes a location shared by publishing it. Consider the case where the access to `published` in `thread 2` occurs after the access to `published` in `thread 1` and thus ends up with `local_published == 1`. In this case an STM needs to be careful that the racing access to `value` in `thread 2` not occur before the update in `thread 1`. Otherwise, the undesired result will occur.

Instead of linking programming idioms to STM internals some researchers have suggested Single (Global) Lock Atomicity (SLA) as a semantic requirement for STM implementations [HLR10]. An STM exports SLA semantics to users if any execution can be mapped to one where every transaction begins by acquiring a hypothetical process-wide mutual exclusion lock and ends by releasing it. SLA supports both privatisation and publication and presents easy STM implementation-independent semantics to programmers. Menon et al. [MBS$^+$08a] in-

```
            int published = 0, value =0;

   // Thread 1               // Thread 2
   value = 1;                atomic {
   atomic {                      ...
       ...                       local_value = value;
       published = 1             local_published = published;
       ...                   }
   }                         if(local_published == 1)
                                 some_function(local_value);

// Impossible: local_published == 1 and local_value == 0
```

Figure 2.7: Publication

vestigated an STM that supported SLA for Java. They concluded that this added significant overhead to the STM and thus also investigated weakenings of SLA that supported common programming idioms (such as the two above) and incur lower overhead.

2.6 Single lock atomicity and memory consistency

Constructing an STM that supports single global lock atomicity requires establishing that every execution can be mapped to one where transactions are delimited by the acquisition and release of a hypothetical global lock. In addition, optimistic concurrency control must not result in executions that are forbidden by the underlying memory consistency model. For a weakly atomic STM, transactions are atomic with respect to each other (and thus sequentially consistent with respect to each other). However, interactions between non-transactional and transactional accesses can reveal the inner workings of the STM and a departure from the memory consistency model if the STM is not carefully designed. Consider the simple example in Figure 2.8.

Under the C++ memory model, *any* result is allowed for this racy program. This is commonly referred to as “catch-fire” semantics and the racing reads are allowed to return any arbitrary value and the program to crash. In effect, the C++ memory model forbids data races. An STM designed for use in C/C++ (such as TL2) thus simply need not care about such data races. Such STMs usually focus only on preserving atomicity between transactions and correctly supporting privatisation.

Under the Java memory model no out-of-thin air values should be returned i.e. a read should return either the initial value of a location or the value written by an executed write. This is expressed as a requirement for causality in executions. Nevertheless an STM is still afforded considerable flexibility. For example an STM working at a cache line granularity (such as

```
               int X = 0, Y = 0;

      // Thread 1                //Thread 2
      atomic {
          X = 10;                t1 = Y;
          Y = 10;                t2 = X;
      }


     C++: Catch fire due to data race, any result allowed
     Java: Intra-thread reordering allowed
     x86: No intra-thread reordering
```

Figure 2.8: Memory ordering differences

McRT-STM) could reorder the writes to `X` and `Y` by virtue of `X` being located later in a cache line than `Y` and thus written out later in the commit phase. This is perfectly acceptable in the Java memory model. Consequently the result `t1 == 10` and `t2 == 0` is allowed.

The x86 memory model however is much stronger. The execution of all programs – including those with data races – is precisely specified. In addition, there are intra-thread ordering constraints. In the example the writes to `X` and `Y` *cannot* be reordered and the result `t1 == 10` and `t2 == 0` is forbidden.

One of the key results of this dissertation (Chapter 3) is that a weakly atomic STM providing SLA for the x86 memory model is forced to serialise all transactions. I also show in the same chapter how a practical STM can be built for the x86 memory model by relaxing the requirements that it be applicable to any arbitrary program.

2.7 Hybrid transactional memory

Software transactional memory in general imposes high overhead when compared to hardware transactional memory due to the cost of adding instrumentation for shared memory accesses. On the other hand most proposals for hardware transactional memory recognise that structures such as caches in hardware are always limited and thus hardware transactional memory can only be "best-effort" and cannot guarantee atomic execution of large transactions. The unbounded designs of Rajwar et al. that proposed virtualising transaction logs in order to solve this problem involve even more hardware complexity and thus represents an even bigger barrier to the availability of hardware TM.

Motivated by these observations, Damron et al. proposed *hybrid transactional memory*: using a combination of software transactional memory with hardware transactional memory [DFL$^+$06]. The common case is execution of transactions entirely in hardware, representing the best possible performance. In the uncommon case where a transaction cannot be handled in hardware, execution could fall back to software transactional memory. The key idea of Damron et al. was to automatically add STM-related metadata to the read sets of hardware transactions. This ensured that hardware transactions were atomic with respect to software transactions that might access the same locations.

Damron et al.'s proposal provided strong atomicity to the hardware transactions due to the presence of a coherence mechanism that would detect non-transactional accesses. On the other hand, it provided only weak atomicity to software transactions, allowing interleaving of non-transactional accesses. Minh et al. [MTC$^+$07] proposed a design for a hybrid transactional memory system that used a weakly atomic STM (TL2) but still provided strong atomicity. This was achieved by using a special bloom filter to encode the read and write sets of the software transaction and checking coherence requests from non-transactional accesses on other threads against it. On a match the software transaction is aborted.

2.8 Performance benefits of transactional memory

Transactional memory aims to bring the performance of fine-grained locking to programs written with coarse-grained synchronisation, which is presumably easier and less prone to problems such as deadlock. Traditionally, data structures such as red-black trees have been used to demonstrate how transactional memory can improve scalability and hence performance over coarse grained locking at higher thread counts.

In the process of examining the more general applicability of transactional memory, research has focused on the conditions under which transactional memory can actually deliver better performance than alternative methods of synchronisation such as locks.

Rossbach et al.'s work on TxLinux [RHP$^+$07] used simulated hardware transactional memory with no overhead for transactional accesses to replace locking in the Linux kernel. Surprisingly, they obtained only a 5% speedup over the lock-based version of Linux. The reason for this is that Linux is well-tuned enough that little time is wasted waiting for a lock: i.e. lock contention is low. There was thus little advantage in allowing multiple threads to execute critical sections simultaneously since they rarely wanted to do so.

The amount of *disjoint-access parallelism* [IR94] is also an important determinant of transactional memory performance. Transactions that increment a single shared counter will conflict. On the other hand transactions that update a large red-black tree are unlikely to conflict. A red-black tree thus has a far larger amount of disjoint-access parallelism than a counter. Von Praun et al. [vPBC08] studied the amount of disjoint-access parallelism available in programs by collecting memory access traces from sequential execution and analysing them to detect possible conflicts.

Some researchers have considered quantifying the benefit that transactional memory can bring to programs using locks by combining measurements of lock contention and disjoint-access parallelism. Roy et al. [RHH09a] built a tool that measured lock contention and disjoint-access parallelism using dynamically instrumented programs. Porter et al. [PW10] built a similar tool, but they used execution on a simulator for measurements. A key benefit of this approach is that it can also be applied to operating system kernels while Roy et al.'s techniques are limited to application software. On the other hand Roy et al. used real execution, which yields more accurate lock contention data.

Usui et al. [USB09] built dynamic adaptivity into their solution for software lock elision. They measured lock contention and transaction conflicts to determine whether locks or transactions were a better way to execute a particular critical section. Crucially they also considered the overhead of software transactional memory in their cost-benefit analysis; previous approaches had either ignored this overhead, or focused only on hardware transactional memory.

One of the key contributions of this dissertation is a profiler that accurately analyses and characterises critical sections in x86 binaries. It uses the same infrastructure as that built for software transactional memory. In addition to memory access characteristics of critical sections, it also produces data related to possible conflicts and lock contention, which can be used to judge possible performance gains from using transactional memory.

2.9 Software lock elision for x86 machine code

This dissertation focuses on building software lock elision for x86 machine code. There are a number of decisions that I made in the design for SLE in relation to previous work.

I use word-based STM as the basis for software lock elision. This is because detecting object boundaries in x86 machine code is not feasible. I use a weakly atomic design. Using strongly atomicity would require either hardware support or expensive insertion of barriers for *all* shared memory accesses in the binary rather than only instrumentation of accesses in a critical section. Previous weakly atomic STM designs had focused on providing single lock atomicity for language level memory consistency models. In Chapter 3 I first solve the problem of designing an STM that provides SLA for the stricter x86 memory consistency model.

I chose to use dynamic binary rewriting to insert instrumentation for the STM. Unlike a library-based STM this is fully automatic; and unlike a compiler-based STM it is language, compiler and debugger agnostic thus requiring no changes to the software development environment, one of the stated goals of this dissertation. Unfortunately dynamic binary rewriting adds large amounts of overhead. In Chapter 4 I present the design for a lightweight binary instrumentation infrastructure that avoids much of the overhead traditionally incurred when using heavyweight dynamic binary rewriting engines.

The next two chapters discuss this work supporting efficient SLE. Chapter 5 discusses a profiler that is built from the instrumentation infrastructure for SLE. It explains some of the performance anomalies seen with SLE in Chapter 4. Also, since it operates directly on lock-based binaries, it can be used to evaluate the possible impact of transactional memory on programs in a TM agnostic manner. In Chapter 6 I discuss a way to make SLE more efficient by eliminating thread-private data from the STM logs. In Chapter 7, I discuss SLE in a more general setting in terms of applicability before concluding.

For all the benchmarks in this dissertation, the hardware and toolchain used is available in Appendix B.

Chapter 3

x86-safe software transactional memory

The first step towards SLE for x86 machine code is to build software transactional memory that preserves the x86 memory consistency model and can be used to replace critical sections in x86 machine code with transactions. This chapter describes the design and construction of such an STM, that I refer to as STM_x86. There are two subproblems that need to be solved for this. The first is to build software transactional memory that preserves the x86 memory consistency model and provides Single (Global) Lock Atomicity (SLA) to transactions. This is done in Sections 3.1 to 3.5. The second subproblem, discussed in Section 3.6, is to show how critical section execution can be mapped onto execution with SLA.

3.1 x86 memory consistency model

In order to describe the construction of an STM that preserves the x86 memory consistency model (x86-MM) and provides SLA, it is important to use a well-defined model for x86-MM. I use the recent model proposed by Owens et al. [OSS09] that casts the x86 as a total store order machine. This model adequately considers the (informal) x86 manuals and formalises them to a level that is usable for this dissertation.

Owens et al. describe the x86 microprocessor as essentially a sequential machine with a write buffer per hardware thread. I use the terms thread and processor interchangeably in this dissertation. In other words, they cast the x86 as a Total Store Order (TSO) machine. This means that loads and stores execute in order. However, each thread has a local write buffer that allows writes to be visible to local reads before they are visible to other threads.

A TSO machine model is depicted in Figure 3.1. The registers in a processor are abstracted in the `Computation` block. Each processor includes a write buffer and there is a globally visible shared memory. An important addition to a simple write-buffered machine is the `lock`. Exactly one processor can hold the `lock` at a time. A processor is said to be *blocked* if some

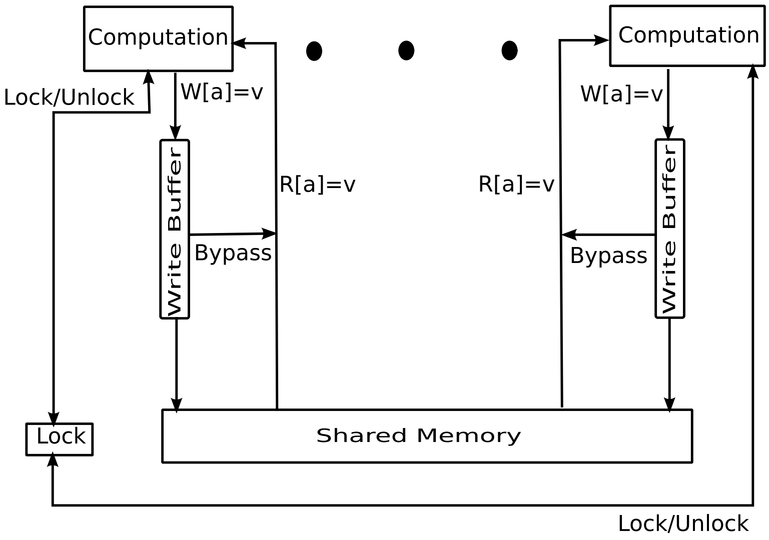


Figure 3.1: TSO machine

other processor holds the lock. This “TSO lock” is acquired by the processor before executing a locked x86 instruction and released by the processor at the end of the locked instruction.

Any x86 processor can perform the following steps:

- A processor can read from a register.
- A processor that is not blocked can read from shared memory if there is no matching location in its own write buffer.
- A processor that is not blocked can read its own most recent write from its write buffer.
- A processor can write to a register.
- A processor can add a write to its own write buffer.
- A processor that is not blocked can move the oldest write from its write buffer to memory.
- A processor can execute a barrier (`mfence`) if its write buffer is empty. [1]
- If a processor's write buffer is empty then the processor can acquire a `lock` (if it is available), or release a lock (if the processor holds it).

I now define formal terms that I use in the rest of this chapter. This chapter is concerned with programs executed on the TSO machine. Formally, a program can consist of the following operations:

1. **Read:** A read, denoted as `Read(p, x, Value)`, meaning that a read from location `x` by processor `p` returns `Value`.

[1] The `lfence` and `sfence` x86 instructions are no-ops for the writeback memory type.

2. **Write:** A write, denoted as `Write(p, x, Value)`, meaning a write to location `x` by processor `p` of `Value`.

3. **LockedRMW:** A locked read-modify-write instruction denoted as `LockedRMW(p, x, Read, Write)` meaning the Read and Write performed atomically on location `x` by processor `p`. This models locked x86 instructions as `lock cmpxchg` that atomically read from and write to a memory location with the written value being a function of the read value.

4. **Fence:** An `mfence` instruction on processor `p` denoted as `mf(p)`.

Execution Trace: An execution trace of a multithreaded program is a sequence of operations for each processor.

Execution of a program on the TSO machine results in events. An event generated by one processor is "observable" by other processors. There are 4 kinds of events.

1. `MemRead(p, x, Value)` meaning that a read from memory of location `x` returns `Value` on processor `p`. This is generated by the execution of a read that does not get a value from the local write buffer.

2. `MemFlush(p, x, Value)` meaning that a store leaves a write buffer and gets written to shared memory. This does not necessarily correspond to execution of any instruction (write buffer flushes are asynchronous).

3. `ILock(p)` Processor `p` acquires the lock. Again, the write buffer on that processor must be empty.

4. `IUnlock(p)` Processor `p` releases the lock. Again, the write buffer on that processor must be empty.

Event Trace: An event trace is a sequence of events.

Event Order $\prec_e$: $e_1 \prec_e e_2$ if e_1 precedes e_2 in the event trace.

Each *step* of the TSO machine is a tuple consisting of an operation and an event. Either the operation or the event (but not both) can be empty. A step can be one of the following:

1. (Read(p, x, Value), MemRead(p, x, Value)): A processor reads a value from memory.

2. (Read(p, x, Value), None): A processor reads a value from its local write buffer generating no event.

3. (Write(p, x, Value), None): A processor appends a value to its local write buffer generating no event.

4. (None, MemFlush(p, x, Value)): A processor flushes a write buffer entry to memory.

5. A locked RMW operation generates a sequence of steps:
   (None, Ilock(p))
   (Read(p, x, Value1), MemRead(p, x, Value1))
   (Write(p, x, Value2), None)
   (None, MemFlush(p, x, Value2))
   (None, IUnlock(p))

6. (mf(p), None): A processor executes a memory fence, generating no event

A *valid execution* on the TSO machine is simply a sequence of steps in accordance with the rules of the TSO machine specified at the beginning of the section. The term "execution" always refers to a "valid execution" unless otherwise specified.

Owens et al. also introduce a progress-related condition which restricts the allowable paths on TSO such that: any entry in any write buffer is ultimately written back to memory. This means that any execution must ultimately end in empty write buffers for all processors.

Single lock atomicity requires the notion of a software "lock" that is acquired at the start of a transaction and released at the end of it. I model that lock (referred to as the SLA lock) as a test and set lock that is acquired and released through x86 compare and swap instructions (`cmpxchg`). SLA execution thus generates `ILock(p)` and `IUnlock(p)` events at both the start and end of a transaction.

Finally, any practical STM design providing optimistic concurrency control needs to modify program execution from that attained when using pessimistic SLA locking. The STM produces "equivalent" executions from the perspective of the programmer. It is important to precisely define what is meant by this equivalence.

Equivalent Execution: A valid execution is equivalent to another valid one if one can be derived from the other by deleting events of the form (Read(p, x, Value), None) and by delaying events of the form (Write(p, x, Value), None) up to a point before the corresponding (None, MemFlush(p, x, Value)) is generated for it from the write buffer.

The notion of equivalent executions stems from the fact that these deletions and reorderings are invisible to processors other than the one on which it is performed and leaves shared memory untouched. For the processor on which reordering/deletion is performed, the STM machinery ensures that the intermediate values of registers are as expected in the original execution for any deleted memory reads.

Another important flexibility needed for STMs is the capability to introduce arbitrary reads and writes to STM metadata locations. The execution when running with an STM would include these "additional" steps. The programmer is concerned only with a subsequence of the execution that affects program locations (distinct from STM locations). This can be formalised as follows.

Consider a partitioning of memory into "special" locations and "general" locations and an execution which is a sequence of steps. Consider an execution formed by a *subsequence* of this execution such that it consists only of accesses to general locations (and possibly memory fences) and there is no updating step (memory write or write buffer flush) to a general location in the steps of the execution outside the subsequence. The subsequence is also restricted to contain all steps originating from locked read-modify-write operations to general locations and must not contain any steps originating from locked read-modify-write operations to special locations.

Theorem 1. *The subsequence forms a valid execution.*

Proof. Consider a TSO machine M executing all the steps in the sequence. I will show that there exists a TSO machine M′ that executes the steps in the subsequence. The proof proceeds by induction on the steps of the sequence. I also show that the machines maintain the inductive invariants: 1) The write buffers of M′ consists of the write buffers of M with all writes to special locations deleted 2) The main memory of M′ has the same values as the main memory of M for all general locations 3) If a processor holds the TSO lock in M′ then that same processor holds the TSO lock in M.

Clearly the invariants hold before any step is executed by either machine. If the current step in the sequence is not part of the subsequence then M executes it while M′ takes no step. It must be shown that the invariants are still preserved at the end of the step. If the step does not modify the write buffer or main memory or acquire the TSO lock, clearly the invariant is preserved. The other possible steps are:

1. (Write(p, SpecialLoc, Value), None): Appends a write to a special location to the write buffer. The invariants are preserved.

2. (None, MemFlush(p, SpecialLoc, Value)): Updates the value of SpecialLoc in memory. The invariants are preserved.

3. (None, ILock(p)): The invariants are preserved since M' takes no step. Note that the write buffer for p must have been empty in both the machines due to the invariants holding before the step is taken.

4. (None, IUnlock(p)): M′ takes no step. It is important to show that no processor in M′ can be holding the lock. Assume otherwise that some processor in M′ were holding the lock. This processor must be p otherwise the invariant would be violated before the step

(p must be holding the lock in M in order to release it). p can only acquire the lock in M′ due to a step from the *subsequence* corresponding to a locked RMW. Two locked RMWs cannot be interleaved in the sequence and hence, this (None, IUnlock(p)) must belong to a locked RMW to a general location, which would then require M′ to take a step.

5. (mf(p), None): M′ takes no step. The invariants are preserved. Note that the write buffer of p must have been empty in both the machines due to the invariants holding before the step is taken.

Next, consider a step that is part of the subsequence. Both M and M′ must be able to perform the step. Further, the invariant must be preserved at the end of the step. The possible steps are:

1. (Read(p, GeneralLoc, Value), MemRead(p, GeneralLoc, Value)): Due to the invariant both machines can perform the read. The invariants are preserved after the step.
2. (Read(p, GeneralLoc, Value), None): Same reasoning as above.
3. (Write(p, GeneralLoc, Value), None): Both machines perform the step. The invariants are preserved.
4. (None, MemFlush(p, GeneralLoc, Value)): Since M can perform this step, p is not blocked. Due to the invariant the same holds true for M′ that performs the step. The invariants are preserved.
5. (None, ILock(p)): Again M′ can perform this step at the end of which p holds the lock for both M and M′. This lock acquisition must begin a locked RMW to a general location that cannot be interleaved with any other locked RMW.
6. (None, IUnlock(p)): Both M and M′ release the lock in an identical fashion. The invariants are preserved.
7. (mf(p), None): Both M and M′ take the step. The invariants are preserved. Note that the write buffer of p must have been empty in both the machines due to the invariants holding before the step is taken.

□

This theorem allows an STM to add operations to "special" STM locations while ensuring that a subsequence corresponding to the executed program remains a valid execution. Note that the proof above does not say anything about the state of registers. Ensuring that register read-writes and computation proceeds as expected from the perspective of a thread remains the responsibility of the STM instrumentation machinery. Also, while the programmer can ignore the STM when considering the execution of the program, other observers such as another processor on the system bus or a debugger would see the effects of the STM.

```
Initially:  X == 0, Y == 0, done1 == false, done2 == false

// Thread 1                    // Thread 2
atomic {                       atomic {
    3: Write(X, 100)               5: Read(Y, 0)
    4: Write(done1, true)          6: Write(done2, true)
}                              }


// Thread 3                    // Thread 4
1: Write(Y, 300)               Read(done1, true)
2: Write(X, 300)               Read(done2, false)

          // Forbidden by x86-MM: Finally X == 100
```

Figure 3.2: Loads must be ordered after earlier transactions

3.2 Counterexamples

In this section, I consider the consequences of supporting the x86 memory consistency model in a weakly atomic STM providing transactions with SLA semantics. My approach is to start with the assumption that if two transactions running on different threads access disjoint sets of memory locations, an STM providing optimistic concurrency control is free to run them without reference to each other. I then provide counterexamples to this assertion, which aim to show that this leads to executions that violate the x86 memory consistency model. The result of these counterexamples is that STM designs are constrained to run transactions in strictly serial order.

The examples illustrate executions abstracted as program operations. I use “atomic blocks” to delimit sections of code to be run as transactions.

First, consider the example in Figure 3.2. `Thread 4` observes that `Thread 1` must have acquired the SLA lock before `Thread 2`. When `Thread 2` reads the value of `Y` as 0, the flush to memory of the update to `Y` followed by the flush to memory of the update to `X` by `Thread 3` is yet to happen. However at this point (due to SLA semantics) the update to `X` by `Thread 1` has already happened. Hence the update to `X` by `Thread 1` must precede the update to `X` by `Thread 3`. Hence the final value of `X` cannot be 100.

That this is an illegal execution can be formally shown as follows. Assume that the disallowed execution were to occur; then each of `thread 1` and `thread 2` must generate the following events (recall locked instructions flush the write buffer):

ILock(thread 1) $\prec_e$ IUnlock(thread 1)
$\prec_e$ MemFlush(thread 1, X, 100)
$\prec_e$ MemFlush(thread 1, done1, true) $\prec_e$ ILock(thread 1)
$\prec_e$ IUnlock(thread 1)

ILock(thread 2) $\prec_e$ IUnlock(thread 2) $\prec_e$ MemRead(thread 2, Y, 0)
$\prec_e$ MemFlush(done2, true) $\prec_e$ ILock(thread 2) $\prec_e$ IUnlock(thread 2)

Furthermore, Thread 4 observes done1 == true and done2 == false. This is only possible if

MemFlush(thread 1, done1, true)
$\prec_e$ MemFlush(thread 2, done2, true)

In combination with the above, one can thus conclude:

MemFlush(thread 1, X, 100) $\prec_e$ MemRead(thread 2, Y, 0)

As the final value of X is 100, this means that:

MemFlush(thread 3, X, 300) $\prec_e$ MemFlush(thread 1, X, 100)

Since the write buffer is ordered:

MemFlush(thread 3, Y, 300) $\prec_e$ MemFlush(thread 3, X, 100)

Hence I can conclude:

MemFlush(thread 3, Y, 300) $\prec_e$ MemRead(thread 2, Y, 0)

This is a violation of x86-MM, since the read from Y must see the write to it.

The transactions on Thread 1 and Thread 2 access completely disjoint sets of locations. A weakly atomic STM executing the two transactions could easily allow the updates to done1 and done2 to happen in the order indicated but allow the read to Y to proceed early on Thread 2. This would result in the illegal execution.

```
        // Initially:  X == 0, Y == 0, Z == 0, W == 0


// Thread 1          // Thread 2          // Thread 3
atomic {             atomic {             Read(X, 100)
    Write(X, 100)        Write(Z, 100)    Read(Z, 0)
    Write(Y, 100)        Write(W, 100)    Read(W, 100)
}                    }                    Read(Y, 0)

            // Execution not allowed by x86-MM and SLA
```

Figure 3.3: Stores must be ordered across atomic blocks

The next example is Figure 3.3. Stores from the same thread are ordered under the x86-MM. The lock acquisition and release at the beginning and end of an SLA transaction must flush the write buffer. Hence for SLA semantics either all stores on `Thread 2` must be ordered after all stores on `Thread 1`, or vice-versa. This leads to the assertion (which says that `Thread 3` must not observe interleaved stores). This can again be formally shown as follows:

We immediately have, for any execution:

`MemFlush(thread 1, X, 100)` $\prec_e$ `MemFlush(thread 1, Y, 100)`

`MemFlush(thread 2, Z, 100)` $\prec_e$ `MemFlush(thread 2, W, 100)`

In order for `thread 3` to observe the forbidden result:

`MemFlush(thread 1, X, 100)` $\prec_e$ `MemFlush(thread 2, Z, 100)`

`MemFlush(thread 2, W, 100)` $\prec_e$ `MemFlush(thread 1, Y, 100)`

Hence the write buffer flushes from `thread 1` and `thread 2` must have been interleaved:

`MemFlush(thread 1, X, 100)` $\prec_e$ `MemFlush(thread 2, Z, 100)`
$\prec_e$ `MemFlush(thread 2, W, 100)` $\prec_e$ `MemFlush(thread 1, Y, 100)`

If `thread 1` executed its transaction first under SLA then it is required (reasoning through the lock acquires and releases as in the previous example) that:

$$\texttt{MemFlush(thread 1, X, 100)} \prec_e \texttt{MemFlush(thread 1, Y, 100)}$$
$$\prec_e \texttt{MemFlush(thread 2, Z, 100)} \prec_e \texttt{MemFlush(thread 2, W, 100)}$$

On the other hand if `thread 2` acquires the SLA lock first, then:

$$\texttt{MemFlush(thread 2, Z, 100)} \prec_e \texttt{MemFlush(thread 2, W, 100)}$$
$$\prec_e \texttt{MemFlush(thread 1, X, 100)} \prec_e \texttt{MemFlush(thread 1, Y, 100)}$$

Hence the interleaved result is forbidden.

Again, the transactions access completely disjoint sets of memory locations. Running them on a weakly atomic STM could end up interleaving the stores.

The two counterexamples have serious consequences for transaction execution. A load in a transaction must wait until all concurrently executing transactions are finished to avoid the situation in the first counterexample. A store in a transaction must wait until all concurrently executing transactions are finished to avoid the situation in the second counterexample. In short, transaction execution must be serialised.

These examples illustrate the difficulty of building an STM that preserves x86-MM for all executions. The only way around this is to restrict the class of programs that can be handled, in essence excluding difficult constructs that prevent the effective use of optimistic concurrency control.

3.3 A serialising design: STM_x86_strict

In this section I present a design for STM_x86, which uses lazy version management, lazy conflict detection, and a global version number. It is similar in principle to TL2 [DSS06]. It aims to provide SLA and x86-MM for *all programs* and thus ends up serialising transactions. In the next section I discuss an optimisation to this basic approach in order to recover scalability.

3.3.1 STM primitives

The rest of this chapter makes references to a set of STM primitives that I define below:

- `Metadata(loc)`: Each location is mapped (many to one) to a metadata location, which holds a simple sequence number.
- `Threads`: The set of threads in the system

- `ReadLog(t)`: An ordered set of (location, value) pairs on thread t
- `WriteLog(t)`: An ordered set of (location, value) pairs on thread t
- `SnapshotSeqNo(t)`: A sequence number maintained locally by thread t
- `Epoch(t)`: A sequence number updated only by thread t but readable by all others
- `DirtyList(t)`: A set of metadata locations corresponding to locations in the Write Log
- `SLASeqNo`: A globally shared sequence number
- `NextSeqNo`: Another globally shared sequence number
- `StableSeqNo`: Another globally shared sequence number
- `NextSLAExec`: Another globally shared sequence number

The STM primitives are held in memory and are only accessible by the STM. Before any transaction can be executed, `StableSeqNo` and all metadata locations are initialised to zero. All other shared sequence numbers that can be updated by more than one thread are initialised to *two*. The value of `Epoch` on every thread is initialised to zero at thread creation time. Transactions operate in two phases. In the first *speculation phase*, the transaction is executed speculatively and execution can be rolled back in the event of a conflict. In the second *commit phase*, the effects of the transaction are applied atomically (with respect to other transactions) and made visible through shared memory.

3.3.2 Speculation phase

The fundamental task of the speculation phase in STM_x86 is to capture an execution of the transaction. It aims to collect the set of steps representing the execution of the transaction. This is conceptually achieved for the thread `t` in the ordered logs `StepLogHead(t)` and `StepLogTail(t)`. Appending `StepLogTail(t)` to `StepLogHead(t)` produces a sequence of steps corresponding to the execution of the transaction (excluding the enclosing SLA lock and unlock) on a TSO machine where the write buffer of processor `t` is constrained to flush writes to memory only at the end of the transaction. The `StepLog` is abstract and for illustration only; operations to it use italics in the algorithms. The actual implementation uses an ordered read log where reads from shared memory are logged and an ordered write log where writes to shared memory are logged. No writes are actually performed to shared memory, thus ensuring that no effects are leaked from the speculative transaction that might be seen by other threads.

A thread `t` that begins a transaction calls Algorithm 2. A write of `Value` to location `loc` in the transaction is appended to the write log by calling Algorithm 3. A read from `loc` in the transaction is accomplished by calling Algorithm 4. This first checks the local write log to see if there is a write to `loc`, which is then forwarded to the read (lines 1–2). If this is not so, then a read is made from shared memory and, after a series of STM-related checks (lines 5–8), the result of the read is appended to the read log.

Algorithm 2 SpeculationBegin(t)

```
Epoch(t) := Epoch(t) + 1
Memory Fence
SnapshotSeqNo(t) := StableSeqNo
Initialise WriteLog(t) to empty
Initialise ReadLog(t) to empty
Initialise DirtyList(t) to empty
Initialise StepLogHead(t) to empty
Initialise StepLogTail(t) to empty
```

Algorithm 3 SpeculativeWrite(t, loc, Value)

```
DirtyList(t) := DirtyList(t) $\cup$ Metadata(loc)
Append (loc, Value) to WriteLog(t)
Append (Write(t, loc, Value), None) to StepLogHead(t)
Append (None, MemFlush(t, loc, Value)) to StepLogTail(t)
```

Algorithm 4 SpeculativeRead(t, loc)

```
 1: if $\exists$ (loc, value) $\in$ WriteLog(t) then
 2:     get most recent (loc, result) $\in$ WriteLog(t)
 3:     Append (Read(t, loc, result), None) to StepLogHead(t)
 4: else
 5:     result := contents of memory at loc
 6:     if Metadata(loc) is odd then
 7:         abort
 8:     if Metadata(loc) $>$ SnapshotSeqNo(t) then
 9:         abort
10:     Append (loc, result) to ReadLog(t)
11:     Append (Read(t, loc, result), MemRead(t, loc, result)) to StepLogHead(t)
12: return result
```

An implementation of the logging algorithms must take into account that the x86 allows accesses of different size and alignment that can lead to reads partially overlapping with writes. Extensions to the logging algorithms that handle these cases are given later in this chapter.

Algorithm 5 Commit(t)

```
ticket := x86FetchAdd(SLASeqNo, 2)
while x86CAS(NextSeqNo, ticket, ticket) $\neq$ ticket do
    wait
for all metadata $\in$ DirtyList(t) do
    if metadata is odd then
        abort
    metadata := metadata + 1
x86FetchAdd(NextSeqNo, 2)
for all (loc, Value) $\in$ ReadLog(t) (in order) do
    if Metadata(loc) is odd then
        if Metadata(loc) $\notin$ DirtyList(t) then
            abort
    if Metadata(loc)$>$ SnapshotSeqNo(t) then
        abort
    if Contents of Memory at loc $\neq$ Value then
        abort
Epoch(t) := Epoch(t) + 1
// SLA lock acquire
while x86CAS(NextSLAExec, ticket, ticket) $\neq$ ticket do
    wait
for all (loc, Value) $\in$ ReadLog(t) (in order) do
    if Contents of Memory at loc $\neq$ Value then
        abort
for all (loc, Value) $\in$ WriteLog(t) (in order) do
    set contents of loc := Value
StableSeqNo := ticket
// SLA lock release
x86FetchAdd(NextSLAExec, 2)
for all metadata $\in$ DirtyList(t) do
    metadata := ticket
for all x $\in$ Threads do
    EpochOther := Epoch(x)
    if EpochOther is odd then
        while EpochOther = Epoch(x) do
            wait
```

3.3.3 Commit phase

At the end of a transaction the thread `t` executes `Commit(t)`, described in Algorithm 5. The algorithm makes use of two instructions native to x86:

- `x86FetchAdd(loc, increment)`: x86 mnemonic `xadd`, this atomically adds `increment` to `loc` and returns the original value of `loc`
- `x86CAS(loc, old, new)`: x86 mnemonic `cmpxchg`, this atomically sets `loc` to new if the current value is `old`; it also returns the current contents of `loc`.

The core of the algorithm is straightforward: the transaction is assigned a global sequence number (in SLA execution) from SLASeqNo. This is the next unassigned even number (see line 1), and is stored in the local variable ticket. Lines 2–17 execute the standard two phase commit associated with word based STMs (such as TL2). Metadata locations corresponding to updated locations in shared memory are locked and for locations read, a check is made that both the contents of the location and their metadata is unchanged.

The algorithm then proceeds to *re-execute* the critical section using SLA in lines 18–26. Rather than re-executing the computation it aims to generate the steps from StepLogHead(t) followed by the steps from StepLogTail(t) on the TSO machine. This assumes that the transaction is deterministic and memory writes are purely determined by memory reads.

In an execution of the program using the STM I consider only the (unmodified by the weakly atomic STM) steps outside any transaction and the steps from lines 20–24 of Commit. There are no writes outside this subsequence to program locations and STM locations (global and thread-private) are not manipulated in this subsequence. Theorem 1 says that this is therefore a valid execution and I focus only on this subsequence.

An execution where the critical section has been replaced by the subsequence consisting of the steps in StepLogHead(t) followed by the steps in StepLogTail(t) is equivalent (definition in Section 3.1) to one where the bypassed reads are deleted and write operations moved down till some point before the corresponding memory flushes. Such an execution can be achieved by first replaying the read log (which does not contain bypassed reads) and then the write log (performing the writes to shared memory) followed by explicitly flushing the write buffer using the locked-RMW operation in line 26 of Commit.

Being able to maintain a "trace" of the transaction in the separated StepLogs depends on there being *no mfence or x86 locked instruction in the transaction.* Such an instruction would necessitate the flushing of the write buffer and hence disallow the reordering of loads before the stores. This leads to:

> **Restriction 1:** No LockedRMW or Fence can appear in the execution of the transaction.

The reason for this restriction can be illustrated by the example in Figure 3.4. The disallowed execution can occur if the STM were to ignore the mfence, since it could complete Thread 1's read of y before Thread 2 can update it. Hence, on encountering either of the above instructions during speculation, the STM must abort the transaction and retry in a non-speculative fashion (Section 3.6).

```
    // Initially:  x == 0, y == 0

// Thread 1            // Thread 2
atomic {
    Write(x, 1)        Write(y, 1)
    mfence             mfence
    Read(y, 0)         Read(x, 0)
}


  // Execution not allowed by x86-MM and SLA
```

Figure 3.4: `mfence` in a transaction

3.3.4 SLA speculation

The STM needs to ensure that speculating threads see a view of memory that is consistent with SLA execution. This is critical for my intended applications. Executing from an inconsistent state (due to updates by other committing threads) can lead to a thread faulting, entering an infinite loop or even corrupting STM state. Weakly atomic STM implementations for managed environments such as Java [ATLM$^+$06] can use sandboxing, integration with a garbage collector and other techniques to ensure that execution can recover from an inconsistent read set. This is not an option for x86 machine code.

The first point to consider in this context is the effect of execution of a speculative thread on the rest of the system. Since all writes are buffered, they do not leak out from a speculating transaction. A speculating thread executes memory reads and writes from the execution trace in order but buffers all writes. This is equivalent to a TSO execution where the write is held in the write buffer until the end of the transaction. However this does not guarantee progress. When executing with the STM, writes from a transaction are not visible until the end of the transaction. As such, a transaction might stall waiting for a signal from another thread, or simply enter an infinite loop. x86-MM dictates that all performed stores be ultimately visible to other threads. This problem with progress leads to the second restriction on transactions that can be executed with this STM:

> **Restriction 2:** Execution cannot depend on stores in a transaction being made visible to other threads before the transaction completes.

This restriction is illustrated by the example code fragment in Figure 3.5: `Thread 2` cannot make progress until the write from the transaction in `Thread 1` is made visible.

The next point to consider is the effect that the rest of the system can have on a speculating thread. In this STM, transactions commit in SLA order dictated by the assigned sequence number. In addition, we desire that a speculating thread `t` executes with a read set consistent

```
      Initially:  go1 == false, go2 == false


   // Thread 1                        // Thread 2
   atomic {
       Write(go2, true)               loop
       loop                            while(Read(go2, false))
        while(Read(go1, false))       Write(go1, true)
   }
```

Figure 3.5: Visibility of writes from a transaction

with SLA sequence number SnapshotSeqNo(t) + 2 (recall that the SLA sequence number is always even).

First, I need to show that all SLA transactions with sequence number at most SnapshotSeqNo(t) have made their writes visible.

Theorem 2. *Consider a speculating thread* t *with sequence number* SnapshotSeqNo(t)*. Consider another thread* x $\neq$ t *that has successfully committed with sequence number* w $\leq$ SnapshotSeqNo(t)*. If* x *writes an update to* loc *through* MemFlush(x, loc, Value) *and* t *reads* StableSeqNo *in line 3 of* SpeculationBegin *generating the TSO event* MemRead(t, StableSeqNo, SnapshotSeqNo(t)) *then:*

MemFlush(x, loc, Value) $\prec_e$
MemRead(t, StableSeqNo, SnapshotSeqNo(t))

Proof. StableSeqNo must have been updated by execution of line 25 of Commit generating: MemFlush(x, StableSeqNo, w). We know that w $\leq$ SnapshotSeqNo(t) and, since StableSeqNo is monotonically increasing, this update must have preceded the read of StableSeqNo by thread t, i.e.

MemFlush(x, StableSeqNo, w)
$\prec_e$ MemRead(t, StableSeqNo, SnapshotSeqNo(t))

However, this memflush itself is preceded by the actual write to loc:

$$\texttt{MemFlush(x, loc, SomeValue)} \prec_e \texttt{MemFlush(x, StableSeqNo, w)}$$
$$\prec_e \texttt{MemRead(t, StableSeqNo, SnapshotSeqNo(t))}$$

The write is thus visible to the read. □

Next, consider reads generated by speculating transactions in line 5 of `SpeculativeRead`. I now show that if this read receives a value from a transaction, it must be from a transaction whose sequence number is at most the snapshot sequence number of the reading transaction.

Theorem 3. *Consider a speculating thread* `t` *that generates a read event due to execution of line 3 of* `SpeculativeRead`*:* `MemRead(t, loc, Value)`*. Assume that this read returns the result of the write from a committed transaction on thread* $\texttt{x} \neq \texttt{t}$ *due to execution of line 24 of* `Commit` *generating the TSO event:* `MemFlush(x, loc, Value)`*. Let* `s` *be the sequence number used by that instance of* `Commit` *on thread* `x` *(i.e. the value held in its* `ticket`*). Then we have:*

$$\texttt{s} \leq \texttt{SnapshotSeqNo(t)}$$

Proof. Consider the set of events generated along with `MemFlush(x, loc, Value)`. There must be updates to Metadata(loc). We thus have

$$\texttt{MemFlush(x, Metadata(loc), OddValue)} \prec_e \texttt{MemFlush(x, loc, Value)}$$
$$\prec_e \texttt{MemFlush(x, Metadata(loc), s)}$$

Due to the serialisation in accessing metadata in `Commit` there can be no intervening updates to the metadata from other threads until an even value is written to this metadata and made visible to other threads.

Similarly from the perspective of the reader, the `MemRead(t, loc, Value)` is generated. The following checks in lines 5–8 of `SpeculativeRead` must generate `MemRead(t, Metadata(loc), k)`, where `k` is some value for the metadata. We have

$$\texttt{MemRead(t, loc, Value)} \prec_e \texttt{MemRead(t, Metadata(loc), k)}$$

Furthermore, the memread of `loc` must see the memflush and hence the memread of `Metadata(loc)` must follow the memflush that locks it. However, `k` must be even for the checks on the reading transaction to succeed. There can be no intervening updates to the metadata from other threads while it is "locked" by `Commit` to an odd value. Hence the memread of `Metadata(loc)` must occur after the Memflush that unlocks the metadata in order that an even value be observed.

$$\texttt{MemFlush(x, Metadata(loc), s)} \prec_e \texttt{MemRead(t, Metadata(loc), k)}$$

Metadata numbers are strictly increasing and so $\texttt{s} \leq \texttt{k}$. We also check in `SpeculativeRead` that $\texttt{k} \leq \texttt{SnapshotSeqNo(t)}$. Hence we have: $\texttt{s} \leq \texttt{SnapshotSeqNo(t)}$.

□

Finally if any thread executes a transaction with SLA sequence number more than the snapshot sequence number of a speculating thread then it must not be allowed to generate any writes *including those outside the scope of a transaction* that may be visible to this speculating thread. This safety property is the most difficult to provide since it involves constraining the visibility of operations outside a transaction using only a weakly atomic STM (recall that weakly atomic STMs cannot constrain execution outside a transaction). In my STM design, a thread that commits with a later sequence number to a speculating thread stops at line 33 in `Commit`. Updates made in the transaction are invisible to the speculating thread (by virtue of the previous theorem). I only need to prove that the odd quantity in the `Epoch` of the speculating thread is correctly observed by the thread that has just executed its `Commit`.

Theorem 4. *Consider a speculating thread* `t` *and another thread* $\texttt{x} \neq \texttt{t}$ *that has committed with a sequence number larger than* `SnapshotSeqNo(t)`. *It generates* `MemRead(x, Epoch(t), PossiblyOddValue)` *at line 30 of* `Commit`. *At some point the speculating thread has generated* `MemFlush(t, Epoch(t), SomeOddValue)`. *We then have:*

$$\texttt{MemFlush(t, Epoch(t), SomeOddValue)} \prec_e$$
$$\texttt{MemRead(x, Epoch(t), PossiblyOddValue)}$$

Proof. Due to the memory fence at line 2 of `SpeculationBegin`, we have:

$$\texttt{MemFlush(t, Epoch(t), SomeOddValue)} \prec_e$$
$$\texttt{MemRead(t, StableSeqNo, SnapshotSeqNo(t))}$$

Since the sequence number written by the committing transaction to `StableSeqNo` is larger than `SnapshotSeqNo(t)`, we have:

$$\begin{aligned}
&\texttt{MemFlush(t, Epoch(t), SomeOddValue)} \prec_e \\
&\texttt{MemRead(t, StableSeqNo, SnapshotSeqNo(t))} \prec_e \\
&\texttt{MemFlush(x, StableSeqNo, SomeValue)}
\end{aligned}$$

Due to the locked instruction at line 26 of `Commit`, the write to StableSeqNo must be flushed before the read from the speculating thread's epoch is made at line 30. We thus have as an extension to the event order above:

$$\begin{aligned}
&\texttt{MemFlush(t, Epoch(t), SomeOddValue)} \prec_e \\
&\texttt{MemRead(t, StableSeqNo, SnapshotSeqNo(t))} \prec_e \\
&\texttt{MemFlush(x, StableSeqNo, SomeValue)} \prec_e \\
&\texttt{MemRead(x, Epoch(t), PossiblyOddValue)}
\end{aligned}$$

This completes the proof. □

Privatisation safety

The need for safe speculation can be illustrated through the programming idiom of privatisation (discussed in Chapter 2). Figure 3.6 replicates the example from that chapter where `Thread 1` privatises a node from a shared linked list and subsequently accesses it directly. Consider the two kinds of problems that can arise here.

Firstly, if `Thread 2` commits first and is in the process of writing back its updates, `Thread 1` might then commit and free the node leading to memory corruption. In my STM, this is not possible since the updates from `Thread 2` are explicitly ordered (through `NextSLAExec` in `Commit`) before the updates in `Thread 1`. Otherwise, if `Thread 1` commits first, this leaves `Thread 2` as a "zombie" transaction (doomed to failure). However, I have just proved that `Thread 2` must operate on a consistent read set and hence `Thread 2` cannot see any updates from `Thread 1`. In particular it cannot suffer an unanticipated arithmetic fault on performing the division.

3.4 Recovering scalability: STM_x86

In accordance with the conclusion of Section 3.2, Algorithm 5 serialises SLA transaction execution. It includes an apparently redundant repetition of reads at line 15, since it is followed a

```
// Thread 1                   // Thread 2
deleteNode(...) {             updateNode(...) {
    ListNode *node;               ListNode *node;
    atomic {                      atomic {
        ...                           ...
        node = ...;                   node = ...;
        ...                           ...
    }                                 r = 1/node->value;
 node->value = 0;                 }
 free(node);                  }
}
```

Figure 3.6: Privatisation

little later by the actual replay of reads at line 21. However the first batch of reads is performed in parallel by committing threads while the second batch must be serialised. If, for a given execution, both reads return the same value then the second set can be eliminated, reducing the length of the serialised portion of commit and improving scalability. This, however would also mean that for executions where the reads differ, the STM can lead to a divergence from x86-MM. In this section I precisely identify the class of executions where this can occur.

I begin by first defining the notion of a data race [AH98] that I use in this chapter. This is based on defining a partial order on operations in the execution trace of a program.

> If an operation `y` follows an operation `x` on the same thread then I say that `y` is ordered in program order after `x`: $\mathtt{x} \rightarrow_{\text{po}} \mathtt{y}$
>
> If a LockedRMW operation `y` *implementing the SLA lock acquire at line 18 of* `Commit` reads the result of a LockedRMW operation `x` *implementing the SLA lock release at line 26 of* `Commit`, I say that `y` occurred in synchronisation order after `x`: $\mathtt{x} \rightarrow_{\text{so}} \mathtt{y}$
>
> Finally, the happens-before order is the transitive closure of these two:
> $\rightarrow_{\text{hb}} \overset{\text{def}}{=} (\rightarrow_{\text{po}} \cup \rightarrow_{\text{so}})^{+}$

Assume that in an execution with SLA, the result of the earlier read from location `loc` in `Commit` differs from the same read done later. Let the earlier read generate event `MemRead(t, loc, Value)` during execution and the later read generate `MemRead(t, loc, OtherValue)` during execution. This means that there must have been an intervening `MemFlush(x, loc, OtherValue)` during execution from some other thread (x $\neq$ t), i.e.

$$\begin{aligned}&\texttt{MemRead(t, loc, Value)} \prec_e \texttt{MemFlush(x, loc, OtherValue)}\\ &\prec_e \texttt{MemRead(t, loc, OtherValue)}\end{aligned}$$

Theorem 5. `MemFlush(x, loc, OtherValue)` *cannot originate from a transaction.*

Proof. Assume that the MemFlush *did* originate from a transaction. First, consider the case where the writing transaction executes in SLA order after the transaction that does the read. Its writes are ordered through the operations on `NextSLAExec` and thus we have:

$$\begin{aligned}&\texttt{MemRead(t, loc, OtherValue)}\\ &\prec_e \texttt{MemFlush(t, NextSLAExec, seq)}\\ &\prec_e \texttt{MemRead(x, NextSLAExec, LaterSeq)}\\ &\prec_e \texttt{MemFlush(x, loc, OtherValue)}\end{aligned}$$

which contradicts the required order.

Next, consider the case where the writing transaction executes in SLA order before the transaction that does the read. The wait on `NextSeqNo` and the metadata checks establish:

$$\begin{aligned}&\texttt{MemRead(t, NextSeqNo, LaterSeq)}\\ &\prec_e \texttt{MemRead(t, Metadata(loc), EvenValue)}\\ &\prec_e \texttt{MemRead(t, loc, SomeValue)}\\ &\prec_e \texttt{MemRead(t, loc, OtherValue)}\end{aligned}$$

From the perspective of the writing transaction we have:

$$\begin{aligned}&\texttt{MemFlush(x, Metadata(loc), OddValue)}\\ &\prec_e \texttt{MemFlush(x, NextSeqNo, EarlierSeq)}\\ &\prec_e \texttt{MemFlush(x, loc, OtherValue)}\\ &\prec_e \texttt{MemFlush(x, Metadata(loc), SomeEvenValue)}\end{aligned}$$

As a consequence of ordering through the sequence number we have:

$$\begin{aligned}
&\texttt{MemFlush(x, Metadata(loc), OddValue)}\\
&\prec_e \texttt{MemRead(t, Metadata(loc), EvenValue)}\\
&\prec_e \texttt{MemRead(t, loc, SomeValue)}\\
&\prec_e \texttt{MemRead(t, loc, OtherValue)}
\end{aligned}$$

There can be no updates to the metadata while the writing transaction has locked it. The reading transaction sees an even value. Hence the unlock must precede it:

$$\begin{aligned}
&\texttt{MemFlush(x, Metadata(loc), OddValue)}\\
&\prec_e \texttt{MemFlush(x, loc, OtherValue)}\\
&\prec_e \texttt{MemFlush(x, Metadata(loc), SomeEvenValue)}\\
&\prec_e \texttt{MemRead(t, Metadata(loc), EvenValue)}\\
&\prec_e \texttt{MemRead(t, loc, SomeValue)} \prec_e \texttt{MemRead(t, loc, OtherValue)}
\end{aligned}$$

which again contradicts the required order. □

There is another restriction in the execution on `MemFlush(x, loc, OtherValue)`. It should not be ordered using a happens-before relation to the read in the transaction. Let `Write(x, loc, OtherValue)` be the originating write in the execution trace for the event `MemFlush(x, loc, OtherValue)` in the event trace. Similarly, let `Read(t, loc, OtherValue)` be the originating read for `MemRead(t, loc, OtherValue)`.

Theorem 6. `Read(t, loc, OtherValue)` $\rightarrow_{hb}$ `Write(x, loc, OtherValue)` *does not hold.*

Proof. Due to the definition of the happens-before order, an operation can be ordered before an operation on another thread through a sequence that involves synchronisation operations across threads. Thus, the ordering between the read and the write are established through such a sequence:

$$\begin{aligned}
&\texttt{Read(t, loc, OtherValue)} \rightarrow_{\mathrm{hb}} \texttt{Write(x, NextSLAExec, v1)} \rightarrow_{\mathrm{hb}}\\
&\texttt{Read(t, NextSLAExec, v2)} \rightarrow_{\mathrm{hb}} \texttt{Write(x, loc, OtherValue)}
\end{aligned}$$

The corresponding event orders would then be:

$$\texttt{MemRead(t, loc, OtherValue)} \prec_e \texttt{MemFlush(x, NextSLAExec, v1)} \prec_e$$
$$\texttt{MemRead(t, NextSLAExec, v2)} \prec_e \texttt{MemFlush(x, loc, OtherValue)}$$

which contradicts the required order. □

Theorem 7. `Write(x, loc, OtherValue)` $\rightarrow_{hb}$ `Read(t, loc, OtherValue)` *does not hold.*

Proof. Assume that the happens-before relations *does* hold. We have already shown that `Write(x, loc, OtherValue)` cannot be in a transaction in the previous theorem.

Again, due to the definition of the synchronisation order we have:

`Write(x, loc, OtherValue)` $\rightarrow_{\text{hb}}$ `Write(x, NextSLAExec, v1)` $\rightarrow_{\text{hb}}$ `Read(t, NextSLAExec, v2)` $\rightarrow_{\text{hb}}$ `Read(t, loc, OtherValue)`

The write thus precedes a locked operation on `NextSLAExec` that flushes the write buffer on thread `x`. This write must precede in TSO event order a later read from `NextSLAExec` that in turn is followed by a read from `loc`.

Hence we have:

$$\texttt{Write(x, loc, OtherValue)} \prec_e \texttt{Write(x, NextSLAExec, v1)} \prec_e$$
$$\texttt{Read(t, NextSLAExec, v2)} \prec_e \texttt{Read(t, loc, OtherValue)}$$

Now, since `Write(x, loc, OtherValue)` $\rightarrow_{\text{hb}}$ `Write(x, NextSLAExec, v1)` and the write is not in a transaction means that it must precede the transaction in the program execution trace (both are on the same thread). This also means that it precedes the operation on `NextSeqNo`. We thus have:

$$\texttt{MemFlush(x, loc, OtherValue)} \prec_e \texttt{MemFlush(x, NextSeqNo, v1)} \prec_e$$
$$\texttt{MemRead(t, NextSeqNo, v2)} \prec_e \texttt{MemRead(t, loc, SomeValue)} \prec_e$$
$$\texttt{MemRead(t, loc, OtherValue)}$$

This contradicts the required order. □

The theorems in this section imply that eliminating the later set of reads (line 21 of `Commit`) to produce STM_x86 is safe (preserves the x86-MM) when the execution of the program with SLA does not have a write that is outside any transaction and is not ordered (happens-before)

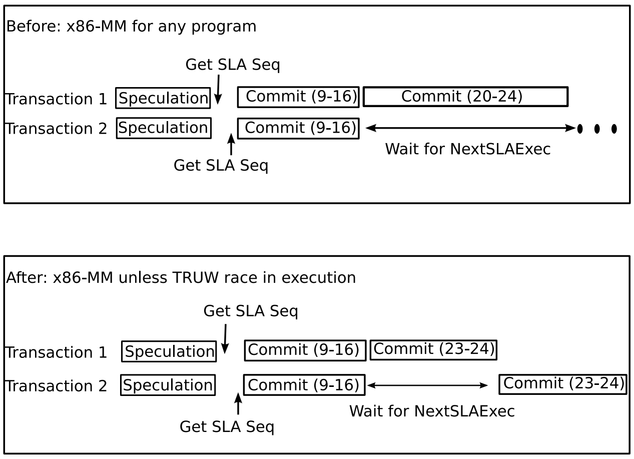


Figure 3.7: Optimisation trades generality for scalability

with a read in a transaction to the same location. Technically, this is a "data-race" between a write outside any transaction and a read inside a transaction. For convenience, I refer to such a race in an execution as a Transactional Read Unprotected Write race (TRUW) race. I thus have the third restriction on programs that can be executed with the STM:

> **Restriction 3:** The execution must not contain a TRUW race.

Given this restriction, I delete lines 20–22 from `Commit`. Committing transactions no longer need to replay their reads in SLA order. Note that this restriction excludes code such as Figure 3.2 early in this chapter, which I had used to show that an STM that preserves x86-MM needs to serialise reads in a transaction after writes in earlier transactions. That example contains a TRUW race between the read to `Y` in `Thread 2` and the write to `Y` in `Thread 3`.

This optimisation improves scalability. Figure 3.7 shows how a later transaction spends less time waiting for an earlier transaction with this change.

3.4.1 Publication safety

The previous theorems should not be taken to construe that TRUW races leading to a departure from x86-MM are always fatal from a program perspective. One example of this is the *publication* construct from Chapter 2, which is the mirror of privatisation: data moves from being thread-private to being shared transactionally. An example of publication from that chapter is replicated in Figure 3.8.

Consider the two possibilities that might occur. If `Thread 1` commits first then STM_x86 correctly provides SLA and ensures that `value` is read correctly (if it is not, the check at line 15 of `Commit` would fail). On the other hand, if `Thread 2` commits first then there is a TRUW race in the execution on accesses to `value`. STM_x86 no longer guarantees compliance with

```
            int published = 0, value =0;

   // Thread 1                 // Thread 2
   value = 1;                  atomic {
   atomic {                        ...
       ...                         local_value = value;
       published = 1               local_published = published;
       ...                     }
   }                           if(local_published == 1)
                                   some_function(local_value);

// Impossible: local_published == 1 and local_value == 0
```

Figure 3.8: Publication

x86 *but* in this case the race is benign since it still correctly reads `published` as zero and hence does not use `value` during execution.

3.4.2 Aborts

The references to the capability to abort a transaction thus far have not formally specified an abort. Aborts are accomplished through Algorithm 6. There is only one necessary step in abort, to increment the epoch, since the thread is no longer speculating. Depending on where the transaction aborts (such as after locking metadata in `Commit`) additional bookkeeping might be necessary (such as unlocking metadata). The simplicity of `Abort` stems from the fact that writes are buffered, and hence simply discarding the write log undoes all the effects of the transaction.

Algorithm 6 Abort(t)

```
Epoch(t) := Epoch(t) + 1
```

3.5 Comparison with language level memory models

Language level memory models (such as those for C++ and Java) are weaker than the x86-MM. The rigidity of the memory model has considerable impact on STM scalability. An STM aims to execute all transactions concurrently without reference to each other. In order to ensure that transactions appear to execute as dictated by the memory model, STMs (such as the one presented above) are forced to introduce synchronisation between transactions running on different threads even when the transactions do not conflict with one another in terms of their data accesses. The stricter the memory model, the more the synchronisation. Hence a continuum of

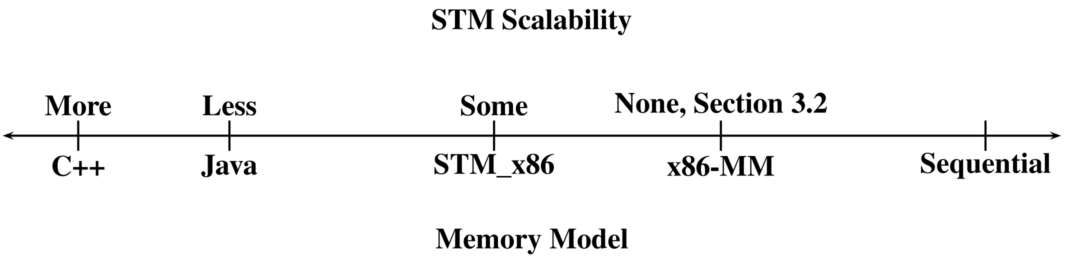


Figure 3.9: Memory consistency model vs weak atomicity

```
            volatile int flag = 0;
            volatile int value = 0;


     // Thread 1          // Thread 2
     atomic {             while(!flag);
         value = 1;       temp = value;
         flag = 1;
     }


    // Java Memory Model guarantees temp == 1
```

Figure 3.10: Memory consistency implications for an STM

weakly atomic SLA STMs exists, as shown in Figure 3.9. STM_x86 gets around the complete lack of scalability due to the strictness of the x86 memory model by providing weaker behaviour to programs with TRUW races.

Even the weaker language level memory models impose restrictions on the interactions between transactional and non-transactional accesses. Researchers building STMs for languages often identify specific (named) programming idioms that illustrate these interactions. In this section I cover some of these idioms, focusing on how STM_x86 handles them by virtue of being designed for a stronger memory model.

3.5.1 Memory update consistency

Memory update inconsistency arises when the STM does not update memory in the order expected by a programmer from the underlying memory consistency model. This term was introduced by Shpeisman et al. in the context of STM design for Java [SMAT$^+$07]. It is illustrated by the code in Figure 3.10.

Declaring the global variable as volatile requires (at a Java language level) that the updates from `Thread 1` happen in order. An STM that buffers at large enough granularities (such as McRT-STM that buffers at cacheline granularities) can write back updates out of order in `Thread 1` leading to a disallowed result.

There is no notion of volatile memory locations in x86, but x86-MM requires the same ordering on `Thread 1` and this is provided by the ordered write log of STM_x86.

3.5.2 Speculation safety

Speculation safety is a property referred to in more than one work on STM design [ZWAT$^+$08, MBS$^+$08b, SMAT$^+$07]. It reflects the fact that, fundamentally, STM designs allow speculative executions to be rolled back on encountering conflicting accesses. Any updates made by a transaction that is yet to commit successfully must not be made visible to other threads, particularly those not speculating since they cannot be rolled back. Any STM supporting speculation safety must either use strong atomicity or disallow in-place updates. STM_x86, like many language level STMs that provide speculation safety chooses to eschew in-place updates.

3.5.3 Dynamic separation

An STM that targets the C++ memory model is less restricted than that for Java. The C++ memory model explicitly gives no semantics to programs with data races [BA08] and thus, for example, the STM is free to do any re-ordering it wishes for the writes of `Thread 1` of Figure 3.10. STMs written for unmanaged C/C++ code [DSS06, RHH09b, WCW$^+$07] often simply assume a *separation* of data into that which can be accessed transactionally and that which cannot. Separation means that at any point during execution with single lock atomicity, a data item can be accessed transactionally or non-transactionally, but not both. The rationale for this is that a program that admits executions without such separation in the context of SLA semantics for transactions would likely contain a data race.

Nevertheless STMs for the C++ memory model still need to deal with concurrent transactional and non-transactional accesses to the same data item. This is because separation can be dynamic [ABH$^+$08], where an object moves from being accessed transactionally to non-transactionally and vice-versa (in a race-free manner). The privatisation and publication idioms arose when studying STM designs supporting dynamic separation. I have already shown how STM_x86 provides privatisation safety (Section 3.3.4) and publication safety (Section 3.4.1).

3.6 Mapping critical sections to SLA

I now discuss the second part of the STM design problem for SLE: executing critical sections protected by locks using transactions providing SLA semantics. A critical section is a subsequence of the execution trace of a program where at least one lock is held. I assume that lock and unlock calls can be intercepted and elided (unlike [RG01] I do not attempt to identify them

```
            bool go1, go2 = false;

     // Thread 1                    // Thread 2
     Lock(L1);                      Lock(L2);
       go2 = true;                    while(go2 == false);
       while(go1 == false);           go1 = true;
     Unlock(L1);                    Unlock(L2);
```

Figure 3.11: Symmetric dependent visibility

speculatively). I also assume that the lock and unlock operations have no side-effect other than serialising execution of critical sections protected by the same lock *and* adding a memory fence at the beginning and end of the critical section.

The approach I follow is to execute all critical sections using SLA. From a correctness standpoint there is no problem. An SLA execution can be mapped to one using locks where exactly one lock can be acquired at a time. Also STM_x86 is designed so as to insert a memory fence at the beginning and end of the transaction.

From a progress standpoint however, there may not exist any possible SLA executions of a given lock-based program. This happens when two or more critical sections depend on each other to make progress. Ziarek et al. first encountered this when attempting to replace monitors with transactions in Java [ZWAT$^+$08]. They described such constructs as requiring *symmetric dependent visibility*, a simple example being illustrated by the code in Figure 3.11.

Execution of such lock-based programs using SLA falls under the STM restriction already discussed: execution must not depend on stores in a transaction being made visible to other threads before the transaction completes.

I now examine each of the restrictions on execution using STM_x86 in the context of lock-based programs. I first express the restriction in terms of executions of the lock-based program *without* any STM in the picture (since SLA execution is one possible lock-based execution). I then show that one of the following holds true:

1. The forbidden behaviour is detected when executing with STM_x86 and correct execution is preserved by falling back to acquiring the lock (Section 3.7)

2. The forbidden behaviour implies possible executions of the *lock-based* program *without STM_x86* that should be considered buggy and hence unlikely to occur in practise.

Restriction 1: No lockedRMW or Fence in the execution trace of the transaction.

If the lock-based program does not admit any execution where a locked instruction or memory fence is executed in a critical section, then this restriction cannot be violated when executing with SLA (which is just a particular locking schedule).

```
            bool go = false;

// Thread 1                    // Thread 2
atomic {
    go = true;                 while(go == false);
    while(true);
}
```

Figure 3.12: Buggy signalling

Detecting memory fences and locked instructions during execution is easy with appropriate instrumentation, since these instructions can be statically identified. If any memory fence or locked instruction is detected in a critical section then execution falls back to the lock.

Restriction 2: Execution cannot depend on stores in a transaction being made visible to other threads until the transaction completes.

If the lock-based program does not admit any execution that depends on a store in a critical section being made visible before the critical section ends then this restriction is satisfied.

This restriction deals with critical sections that send signals to other threads (since that can be the only reason why the write needs to be visible before the end of the critical section). If the transaction terminates then the signal will be visible and there is no problem. If the transaction does not terminate, there are two possibilities: the transaction itself waits for a signal or the transaction enters an infinite loop unrelated to any other thread.

If the transaction itself waits for a signal then in STM_x86, it generates a continuous sequence of reads from the signal variable which ultimately leads to an overflow of the read buffer (which must be finite in any implementation). On an overflow of any STM buffer, execution falls back to the lock. This leaves SLE vulnerable only to programs such as Figure 3.12. I argue that such program behaviour is buggy.

Restriction 3: The execution must not contain a TRUW race.

Extending the definition of synchronisation operations in Section 3.4 to include the operations used to acquire and release the lock in the lock-based program; this holds if: the lock-based program does not admit a race between a write outside any critical section and a read in a critical section. If the lock-based program does not admit a race between a write outside any critical section and a read in a critical section then SLA execution cannot have a TRUW race.

A race between an operation in a critical section and one outside any critical section is an asymmetric data race (which is a subclass of all data races). The restriction refers to a further subclass of asymmetric data races that involve a writer that accesses a shared variable without holding a lock, while a reader accesses the same shared variable while holding a lock. I argue that such a scenario likely originates from a program bug. Asymmetric data races have been

```
           // gScript is shared

// Thread 1                          // Thread 2
EnterCriticalSection();
if(gScript == NULL) {
    baseScript = default;
}
else {                               gScript = NULL;
    baseScript = gScript;
}
ExitCriticalSection();
...
baseScript->Compile();
```

Figure 3.13: Type of asymmetric data race

the subject of research by Ratanaworabhan et al [RBK$^+$09] and Figure 3.13 shows an example from that work[2]. It is an example of the type of racy behaviour not supported by STM_x86 and is a program bug due to `thread 2` not acquiring the needed lock when accessing `gScript`. Ratanaworabhan et al. also design and implement a dynamic race detector called Tolerace that can be used to detect such races. Tolerace can be used on all x86 binaries before SLE is applied.

Alternatively, a simple modification to `Commit` allows STM_x86_strict to serve the purpose of a dynamic race detector that looks for TRUW races in SLA execution. The algorithm is run without the optimisation of Section 3.4 since we are at this point not sure if there exist executions with TRUW races. If in any execution, the metadata checks in lines 13-14 succeed but the memory location is found to be modified in either line 15 or line 21, then we have a racing write outside the control of the STM indicating a TRUW race. Note that this kind of dynamic race detection (as also Tolerace) produces no false positives but can produce false negatives. Unlike static race detections techniques, it is not perfectly sound and can miss races in execution.

3.7 Mixing locking with transactions

A crucial feature of Software Lock Elision is the capability to acquire a lock and execute the critical section directly without indirection into the STM. I refer to this as *pessimistic locking* (as opposed to optimistic concurrency control).

Pessimistic locking is enabled through a `BlackList` of locks (note that the blacklist is a set and not an ordered list, regardless of the name). A lock in the blacklist is always acquired and

[2] Taken from the Mozilla application suite.

Algorithm 7 Blacklist(t, L)

```
 1: QuiesceList := QuiesceList ∪ { L }
 2: Memory Fence
 3: for all x ∈ Threads do
 4:     EpochOther := Epoch(x)
 5:     if EpochOther is odd then
 6:         while EpochOther = Epoch(x) do
 7:             wait
 8: dummyticket := x86FetchAdd(SLASeqNo, 2)
 9: while x86CAS(NextSeqNo, dummyticket, dummyticket) ≠ dummyticket do
10:     wait
11: x86FetchAdd(NextSeqNo, 2)
12: while x86CAS(NextSLAExec, dummyticket, dummyticket) ≠ dummyticket do
13:     wait
14: x86FetchAdd(NextSLAExec, 2)
15: BlackList := BlackList ∪ {L}
16: QuiesceList := QuiesceList − {L}
```

Algorithm 8 Elide(t, L)

```
1: while L ∈ QuiesceList do
2:     wait
3: if  L ∈ Blacklist then
4:     Pessimistic Locking
5: SpeculationBegin(t)
```

never elided. Locks start off outside the blacklist and are added to it if STM_x86 encounters any condition (such as those discussed above) during execution that prevents optimistic concurrency control. Locks cannot simply be added to the `BlackList` since there may be other threads that have speculated past it. This requires a supporting `QuiesceList`. A lock in the `QuiesceList` is in the process of being blacklisted. A thread `t` can blacklist a lock `L` using Algorithm 7 after which the thread is free to acquire the lock. Any thread that is eliding a lock uses Algorithm 8, which checks the `Blacklist` status of the lock before calling `SpeculationBegin`.

`Blacklist` works by first adding the lock to the `QuiesceList`. It then needs to ensure that any speculating thread that has executed past line 1 of `Elide` for the same lock has finished the transaction. It does so by first waiting for an epoch, ensuring that the speculating thread has reached line 18 of `Commit`. It then executes a *dummy* commit, which ensures that speculating threads have safely reached line 27 of `Commit` and have thus finished accessing shared memory or have aborted. It then adds the lock to the `Blacklist` and removes it from the `QuiesceList`.

3.8 Implementation

The implementation of STM_x86 is contained in approximately 3000 lines of C code. Many of the implementation details are the same as well known word-based STMs such as TL2 [DSS06]. I highlight here the specific aspects of the implementation that are relevant to x86 machine code.

3.8.1 Metadata and logging

I use a single aligned memory word of 4 bytes to store sequence numbers. I use a metadata table with 2^{20} or approximately 1 million entries. I associate metadata with memory locations using the hash function:

$$\text{Hash}(s) = (s >> 4) \& (2^{20} - 1)$$

The representation uses C notation: $>>$ is the rightshift operator and $\&$ is the bitwise-and operator. It sequentially maps every 16 bytes of memory into the metadata table wrapping around after 2^{20} entries. The decision to treat 16 bytes of memory as one unit is guided by the fact that all aligned x86 memory accesses up to and including the 16 bytes vector SSE2 accesses would fit in 16 bytes. This minimises the possibility that an access needs to map to multiple metadata words. The hash function and mapping unit is of course tunable through compile time parameters.

An interesting difference between the STM design in this dissertation and TL2 is that I do not need atomic operations to manipulate metadata in `Commit`. This trades off the latency of atomic operations for the serialisation on `NextSeqNo` in `Commit` (Algorithm 5, line 2).

3.8.2 Arbitrary granularity logs

The logs are maintained as FIFO buffers implemented as singly linked lists. The key complication with logging in the STM is bypassing values from previous writes to later reads. This is because the x86 architecture allows accesses of varying size to any alignment. Most word-based STMs such as TL2 assume word-sized accesses aligned to word boundaries.

Consider the access pattern shown in Figure 3.14. First, a write from a transaction updates 2 bytes in memory (step 2). Next, a read from the same transactions reads those 2 dirty bytes and thus can proceed by simply reading the results of the previous write (step 3). In order to enable this, I maintain a dirty buffer consisting of aligned 16 byte chunks of memory that have been written to. I initialise the dirty buffer (step 1) from the clean contents of memory. If a containing dirty buffer is found for a read, it receives its values out of the buffer.

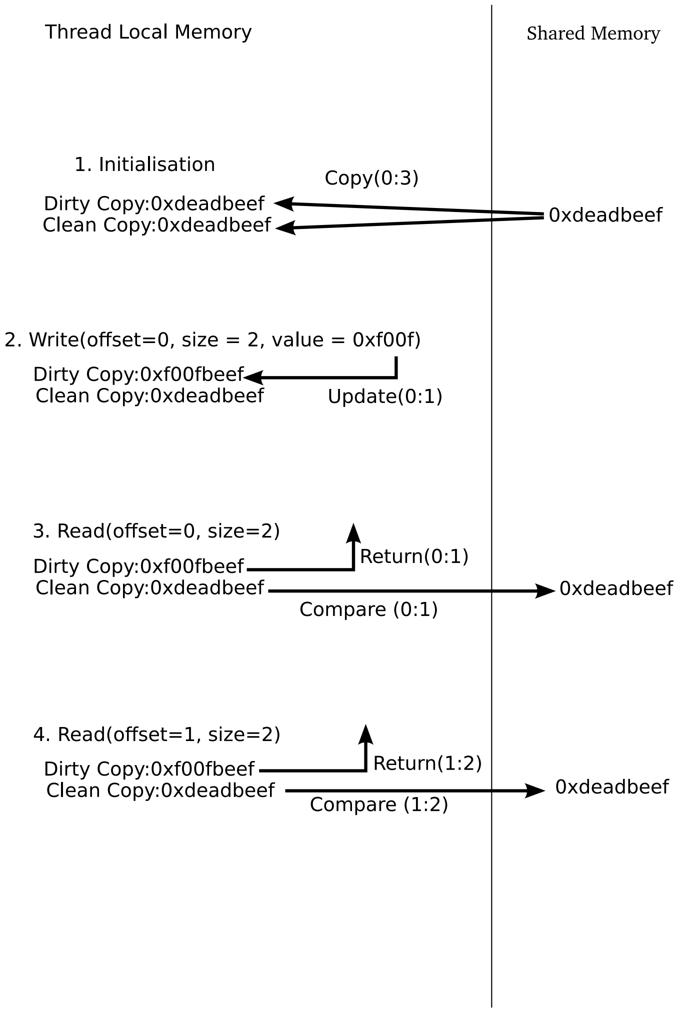


Figure 3.14: Bypassing writes to reads

The next read (step 4) however presents a significant complication since it reads partially from a previous write and partially from shared memory. Both Owens et al.'s model as well as the architecture manuals [x8609] are silent about the behaviour of the write buffer under such circumstances. I assume that the read receives part of its values from the write buffer and part from shared memory (effectively executing a split read).

Implementing this feature however proved a challenge for the STM. One option is to track which bytes of the dirty buffer are actually dirty. This means updating the data structure doing the tracking on every write. Using some micro-benchmarks, I determined that this has an unacceptable cost, since it leads to an extra operation on *every* write, while the actual number of reads hitting a dirty buffer is relatively rare.

Instead, my solution uses the result bypassed from the dirty buffer and *simultaneously* treats the whole read as being from clean data. To do this I maintain two versions of the dirty buffer. The first version acts as normal while the second version (referred to as the clean copy) is never changed after initialisation. I check that the memory contents are unchanged by comparing the accessed portion in the clean buffer with memory. Then, I perform STM metadata related checks on the read. This ensures that I use the correct checks for the clean portion of the read, if any. The conservativeness means that abort rates can possibly increase since data that has been forwarded can potentially be dirtied in memory, leading to an unnecessary abort. However, I saw no such effect in practise.

The enhancements needed to the plain `SpeculationWrite` and `SpeculationRead` are shown in Algorithm 9 and Algorithm 10 respectively. They now take an additional `size` parameter. The read and write logs are also enhanced to include the size of the access in addition to the location and value.

Algorithm 9 SpeculativeWriteEnhanced(t, loc, size, Value)

```
DirtyList = DirtyList(t) ∪ Metadata(loc)
Append (loc, size, Value) to WriteLog(t)
if ∃ (DirtyBuffer, CleanBuffer) containing loc in ByPassList(t) then
    Apply update (loc, size, Value) to DirtyBuffer
else
    Initialise DirtyBuffer, CleanBuffer containing loc from memory
    if Metadata(loc) is odd then
        abort
    if Metadata(loc) > SnapshotSeqNo(t) then
        abort
    Apply update (loc, size, Value) to DirtyBuffer
    Append (DirtyBuffer, CleanBuffer) to ByPassList(t)
```

Algorithm 10 SpeculativeReadEnhanced(t, loc, size)

```
memvalue := contents of Memory at loc
Append (loc, size, memvalue) to ReadLog(t)
if ∃ (DirtyBuffer, CleanBuffer) containing loc in ByPassList(t) then
    result := contents of loc in DirtyBuffer
    if memvalue ≠ contents of loc in CleanBuffer then
        abort
else
    result := memvalue
if Metadata(loc) is odd then
    abort
if Metadata(loc) > SnapshotSeqNo(t) then
    abort
return result
```

The implementation described thus far assumes that accesses do not cross 16 byte boundaries. For some benchmarks I examined, a simple `memcpy` often causes misaligned accesses that cross a slot boundary. Hence such split accesses must be handled. I handle such accesses as follows:

1. Split the access into two partial accesses, one per slot

2. Make a recursive call for each partial access (for writes inhibit additions to the write log for the partial access)

3. Add an entry to the appropriate read or write log for the actual access with the data

The allows me to correctly add the read and write accesses as single accesses (as expected in the TSO machine). It also cleanly handles metadata for each slot accessed. Finally, it adds extra reads for split-reads but there is no correctness related issue there, since neither individual split-read can fault given that the whole read does not.

The bypass list is a pair of singly linked lists of buffers, one for the dirty buffer and one for the clean buffer. I use a pair of bloom filters to avoid unnecessary lookups in the linked list, which is otherwise searched linearly. For large transactions, I switch to a hash index on the bypass list. The physical memory requirements for the logs is capped at 4 MB of physical memory per-thread.

3.8.3 Lock blacklists

The `Blacklist` of locks is maintained as a 1024 entry hash table indexed by the lower 10 bits of the lock address. Each hash table entry is simply a integer that holds one of three values: `0` is the default value, `1` means the lock is in the `QuiesceList` and `2` means that the lock in in the `BlackList`. The hash table size and function are configurable and I empirically chose these settings based on the benchmarks I considered for this dissertation.

I ignore conflicts in mapping locks to table entries, meaning it is possible for a blacklisted lock to cause other locks to be acquired pessimistically. While it is possible to handle conflict using, for example, a closed addressing scheme, I deemed this unnecessary. Lock blacklisting should ideally be very rare (otherwise the program is probably not suitable for software lock elision).

One exception that can necessitate a re-examination of this implementation decision is dynamically allocated locks that are also blacklisted. This can cause the hashtable to quickly fill up with blacklisted entries, leading to the program executing with pessimistic locking throughout.

3.9 Evaluation of STM_x86

In this section, I evaluate the performance of the STM runtime system presented in this chapter. The STM has been designed for safety and transparency: it provides SLA and exactly preserves the x86-MM for the targeted set of programs. In order to properly characterise the price that must be paid for working with a strict memory model, I compare its performance with TL2 [DSS06]. TL2 is built for scalability and performance, as well as being for use as a

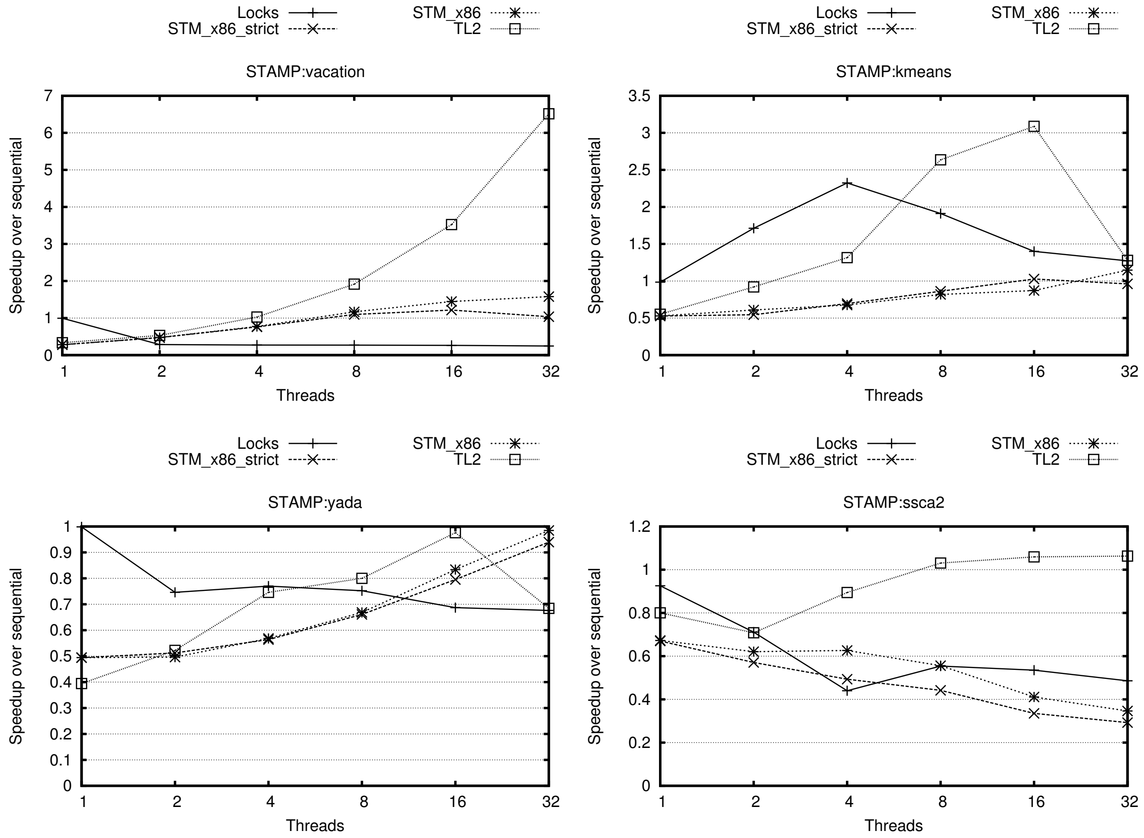


Figure 3.15: STM performance on the STAMP benchmarks(1)

library-based STM by a programmer who is familiar enough with its internals to understand its implication for program safety. It is meant for programs with dynamic separation (only guaranteeing ordering between operations in transactions) and thus most suitable for the C++ memory consistency model that forbids races. In addition, it does not guarantee privatisation safety to speculating transactions. In a sense, the STM in this chapter and TL2 lie at opposite ends of the safety/scalability scale.

Further, in order to properly characterise various aspects of the STM design in this chapter, I evaluate two different variants of it. The first is STM_x86_strict, that provides x86-MM to *all* programs without exception. The next is STM_x86, obtained by applying the optimisation of Section 3.4. This is also the STM used in the rest of this dissertation.

I use the STAMP benchmark suite [CMCKO08] for evaluation. The eight benchmarks in the STAMP suite are: a gene sequencing program (“Genome”), a bayesian learning network (“Bayes”), a network intrusion detection algorithm (“Intruder”), a k-means clustering algorithm (“KMeans”), a maze routing algorithm (“Labyrinth”), a set of graph kernels (“SSCA2”), a client-server reservation system simulating SpecJBB (“Vacation”) and finally a Delaunay mesh refinement algorithm (“Yada”). STAMP uses transactions for synchronisation and shared memory reads and writes have been *manually* instrumented. STAMP has been built to exercise STMs and includes a variety of transactions in terms of the time spent spent by the program in

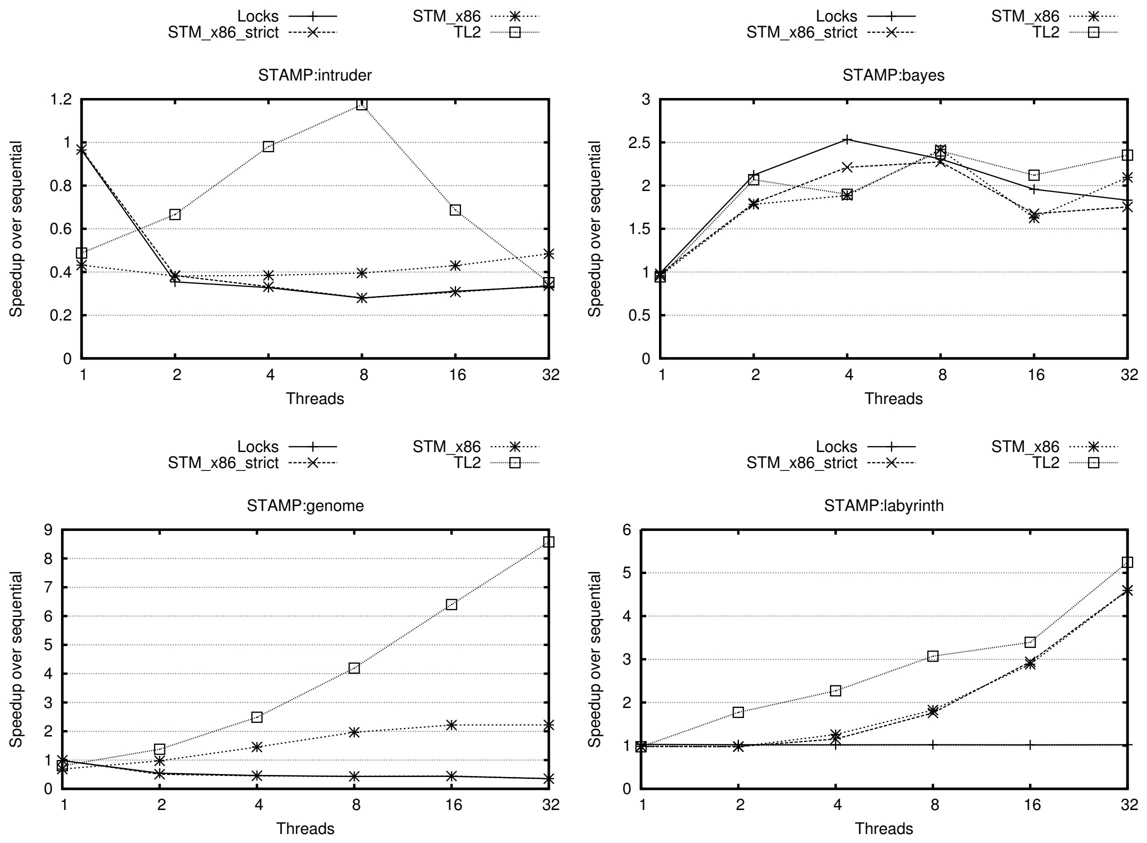


Figure 3.16: STM performance on the STAMP benchmarks(2)

a transaction and the length of the transactions. I use large (native) input sets for the STAMP benchmarks (both for experiments in this chapter as well as for those in the later ones) shown below.

Benchmark	Input
Vacation	-n2 -q90 -u98 -r1048576 -t4194304
Kmeans	-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt
Yada	-a15 -i inputs/ttimeu100000.2
SSCA2	-s19 -i1.0 -u1.0 -l3 -p3
Bayes	-v32 -r2048 -n10 -p40 -i2 -e8 -s1
Intruder	-a10 -l128 -n262144 -s1
Genome	-g16384 -s64 -n16777216
Labyrinth	-i inputs/random-x512-y512-z7-n512.txt

I use a 48-core system for the evaluation (Appendix B: Tigger) The system configuration is available in Appendix B. STAMP only allows thread counts that are a power of two. It also includes a sequential version of each benchmark. I report the execution time for the sequential version of the benchmark divided by the observed execution time for that same benchmark running with the desired STM or lock. Any measured execution time is the median of 5 runs.

	Threads					
Benchmark	1	2	4	8	16	32
labyrinth	0.024	0.006	0.048	0.131	0.165	0.060
yada	0.008	0.047	0.024	0.014	0.048	0.043
bayes	0.001	0.020	1.030	0.297	0.403	0.174
vacation	0.006	0.076	0.023	0.031	0.054	0.030
genome	0.003	0.055	0.027	0.051	0.010	0.004
kmeans	0.023	0.288	0.481	0.996	0.533	0.129
ssca2	0.005	0.099	0.025	0.004	0.005	0.008
intruder	0.003	0.049	0.021	0.040	0.025	0.208

Figure 3.17: TL2: Maximum variation in execution time as a fraction of the median

	Threads					
Benchmark	1	2	4	8	16	32
labyrinth	NO	NO	YES*	YES*	YES*	YES*
yada	NO	NO	NO	NO	YES	YES
bayes	NO	NO	NO	YES*	NO	YES*
vacation	NO	YES	YES	YES*	YES*	YES*
genome	NO	YES	YES*	YES*	YES*	YES*
kmeans	NO	NO	NO	NO	NO	NO
ssca2	NO	NO	YES	YES	NO	NO
intruder	NO	YES	YES	YES	YES	YES

Figure 3.18: Is using STM_x86 better than using the lock ? ('*' means also better than sequential)

The results are shown in Figures 3.15 and 3.16. For most of the benchmarks, the relative performance of the three STMs is as expected. STM_x86_strict is the slowest although surprisingly it is occasionally able to perform better than a lock (such as in Vacation). This is likely due to the transactions being computation heavy, which is performed in parallel. STM_x86 (the STM for the rest of this dissertation) comes next. It performs better than STM_x86_strict by virtue of the optimisation. Also, as expected, TL2 performs the best in general.

There are also some interesting exceptions to this ordering between the STMs. In the case of Yada, Intruder and Bayes the STM in this dissertation performs better than TL2. This is likely due to the fact these three benchmarks demonstrate a combination of long transactions and high conflict rates [CMCKO08]. The STM in this dissertation avoids wasted work by detecting conflicts early due to the strict insistence on a consistent read set compared to TL2. Another important factor is the variance in runtimes. I use 5 runs of every benchmark, reporting the median of the 5 runs. Figure 3.17 gives the maximum relative error from the reported median in the case of TL2. The other STMs show similar behaviour. In the case of Kmeans, the close performance of STM_x86 and STM_x86_strict coupled with the high variance explains why the latter is faster than the former for some thread counts.

The final question of interest is whether STM_x86 (transparent optimistic concurrency control) can improve performance when compared to locking. Figure 3.18 demonstrates that STM_x86 can indeed perform better than a lock in spite of being built for a strict memory model and transparency.

3.10 Discussion

Designing an STM that can be safely used at the level of x86 machine code is clearly a non-trivial endeavour. I started this chapter with a set of examples showing that the strictness of the x86 memory model is a significant challenge for an STM compared to the weaker language level memory models. I showed that STM_x86 must trade off generality and presented the design for an STM that only disallows programs that result in TRUW races under transactional execution. I argue that this likely originates from buggy programs with incorrect synchronisation and thus does not severely limit the applicability of SLE. Finally, I presented an evaluation showing the scalability tradeoffs made in order to work at the lower level of the x86-MM. In spite of the significant extra synchronisation needed to ensure transparency and applicability, STM_x86 can perform better than a lock.

The STM presented in this chapter has been used with manual source level instrumentation. In order to be actually used with SLE, it needs to be applied automatically at machine code level. This requires a means to automatically instrument machine code to intercept lock and unlock calls and all memory accesses in a critical section, which is the subject of the next chapter.

Chapter 4

x86 machine code instrumentation

As chapter 3 showed, STM_x86 preserves x86-MM but by itself lacks the means to interface with x86 machine code to provide software lock elision. This chapter presents the design and implementation of a binary instrumentation system that allows instrumentation of the lock and unlock calls in a binary as well as instrumentation of all shared memory accesses within the delimited critical sections. Together with STM_x86, the complete system (referred to as SLE_x86) provides software lock elision for x86 machine code.

I begin this chapter by discussing the two prevalent approaches to instrumenting x86 machine code: static and dynamic binary rewriting. I argue that neither of them alone is sufficient for SLE_x86. Instead, I show how the best features of each can be combined into a special purpose improved binary instrumentation system.

4.1 Approach

Research in the area of instrumenting machine code has been driven both by the need to build profiling tools that operate at the binary level, such as [RHH09a] as well as tools that actively modify execution: such as to eliminate dead code [BDA00] or even to apply software transactional memory [OCS07]. Given x86 machine code contained in a program there are two prevalent approaches to rewrite binary code into a form that executes an instrumented version of the machine code: a purely static (pre-execution) approach and the other is a purely dynamic (runtime) approach. I discuss each of these approaches next, focusing on their strengths and weaknesses with regard to my intended application of using it for software lock elision.

4.1.1 Static binary rewriting

Static binary rewriting modifies binaries before execution to produce an instrumented version. An early example of static binary rewriting is the binary rewriting tool ATOM [SE94]. Other

```
if(AnalysisOpaqueCondition())
    pthread_mutex_lock(&lock);
pthread_mutex_lock(&possibly_nested_lock);
    ...
pthread_mutex_unlock(&possibly_nested_lock);
// Should the following be instrumented ?
...
if(AnalysisOpaqueCondition())
    pthread_mutex_unlock(&lock);
```

Figure 4.1: Possibly nested locking

examples are DIABLO [VPCDB+05] and PLTO [SDAL01].

Static binary rewriting has one key advantage: there is no overhead to insert instrumentation, since this process happens before the binary is executed. From the perspective of SLE_x86 however, there are two key difficulties with using static binary rewriting.

The first problem is indirect branches. Static binary rewriting needs to analyse the control flow graph to decide which basic blocks[1] in the binary need to be instrumented. For example, in the case of software lock elision, the critical section is made up of all basic blocks reachable from the basic block containing the lock call, but without encountering an unlock call. This cannot (in general) be determined with static binary rewriting. ATOM, for example, uses understanding of the manner in which case statements are complied (by a C compiler) to work out possible targets of indirect branches. Since I wish to make SLE language agnostic, I cannot use this technique.

The second problem, specific to SLE_x86, is demarcating critical sections in the presence of nested locking. Consider the example in Figure 4.1. A purely static approach cannot determine whether the portion of code after the first unlock call should be instrumented since its inclusion in a critical section is execution dependent.

4.1.2 Dynamic binary rewriting

Dynamic binary rewriting can be used to modify binaries at execution time in order to insert instrumentation. Dynamic binary rewriting has recently gained popularity since it enables extremely useful program analysis and optimisation tools to be built. A number of dynamic binary rewriting engines have been produced, including PIN [LCM+05], FastBT [PG10], Dynamo [BDB00] and Valgrind [NS07]. They have formed the basis for useful program analysis tools such as Memcheck [SN05] and program optimisation [BDB00]. Since instrumentation is inserted dynamically, it does not suffer from the problems with static rewriting mentioned above. Dynamic binary rewriting however suffers from the problem of high overhead. There are two primary sources of this overhead.

[1] A single-entry single-exit sequence of machine code.

The first is the cost of inserting instrumentation. Code execution must be stopped in order to rewrite it with instrumentation inserted. This happens every time new instrumentation is inserted.

Another source of high overhead is maintenance of the ‘code cache’: a region of memory that holds all executed (and possibly modified) basic blocks. Since the dynamic binary rewriting engine cannot at any point guarantee that no new code requiring instrumentation will be executed, it executes all code out of a ‘code cache’. The code cache even contains code that has not been instrumented. This ensures that the dynamic binary rewriting engine maintains control of code execution and is able to see all newly executed code. Unfortunately the ‘code cache’ imposes significant overhead even for uninstrumented code. This stems from the cost of maintaining the finitely sized code cache and taking care of events such as evictions. From the perspective of software lock elision, the code cache is a completely unnecessary source of overhead. There is no need to instrument code outside critical sections, and thus no need to put it into the code cache, possibly displacing more useful instrumented blocks.

4.1.3 Combining static and dynamic techniques

The instrumentation system for SLE_x86 aims to combine the benefits of static and dynamic binary rewriting while side-stepping their problems. At the heart of the instrumentation system lies the Persistent Instrumentation Cache (PIC).

A PIC contains instrumented versions of basic blocks within critical sections of its originating binary. It is persistent and held in an on-disk file. It thus represents the instrumentation that would have been added by a static binary rewriting engine. A *complete* PIC contains instrumented versions of *every* reachable basic block within *every* possible critical section of the binary. The completeness of a PIC is clearly undecidable in the presence of indirect branches in the binary. I return to the problem of tolerating incomplete PICs during software lock elision later in the dissertation.

The PIC is generated dynamically, as I depend on execution to look past indirect branches. I also depend on execution to properly handle nested critical sections (such as that shown in Figure 4.1) by dynamically counting held locks. The PIC thus contains code that would have been generated by a dynamic binary rewriting engine.

4.2 x86 instrumentation modes and backends

Instrumented execution of x86 binaries begins with an empty PIC. Execution can then happen in one of two modes: *active mode* or *passive mode*. In active mode, new instrumentation can be added to the PIC based on newly discovered basic blocks. In passive mode no new basic blocks can be added to the PIC. However execution in passive mode is much faster since the instrumentation system avoids any overhead associated with intercepting and examining executed basic blocks to determine if they should be instrumented, as I show later in this chapter.

Crucially, active mode provides the capabilities of dynamic discovery normally associated with dynamic binary rewriting while passive mode uses the (already built) PIC in a manner that attains lower performance overheads more typical to static binary rewriting.

In both modes, the binary executes with instrumentation. The instrumentation system is oblivious to what the instrumentation does. It assumes a backend that provides functions to be called for each instrumentation hook. The x86 instrumentation system provides the following hooks:

1. `Elide`: Called whenever a lock is acquired. In addition to the lock address, `Elide` takes a set of parameters related to checkpoints (see Section 4.6).

2. `SpeculativeRead(loc, size)`: Called on a memory read from location `loc` of size `size`, within a critical section. The instrumentation hook should return the (possibly different) location from where the read is to be actually performed.

3. `SpeculativeWrite(loc, size, rmw_flag)`: Called on a memory write to location `loc` of size `size`, within a critical section. The `rmw_flag` is a boolean that that tells the backend if the write is an x86 read-modify-write rather a simple write. The instrumentation hook should return the (possibly different) location to which the operation is to be actually performed.

4. `Release`: Called whenever a lock is released. Indicates on return whether any locks are still held.

The backend must know the type and functionality of the locking being used in the program in order to be able to fall back to pessimistic locking. For example, most of the evaluation in this dissertation has been with backends written for the synchronisation operations in the `Pthreads` library [DM05].

A simple backend used in this dissertation is the *null backend*. The null backend simply acquires and releases locks on the `Elide` and `Release` calls respective. For memory access instrumentation it simply returns the location passed in. The null backend thus leaves execution unchanged. In addition to being useful for debugging I also use it in this chapter to evaluate the overhead of the binary instrumentation system itself without any of the costs associated with STM_x86.

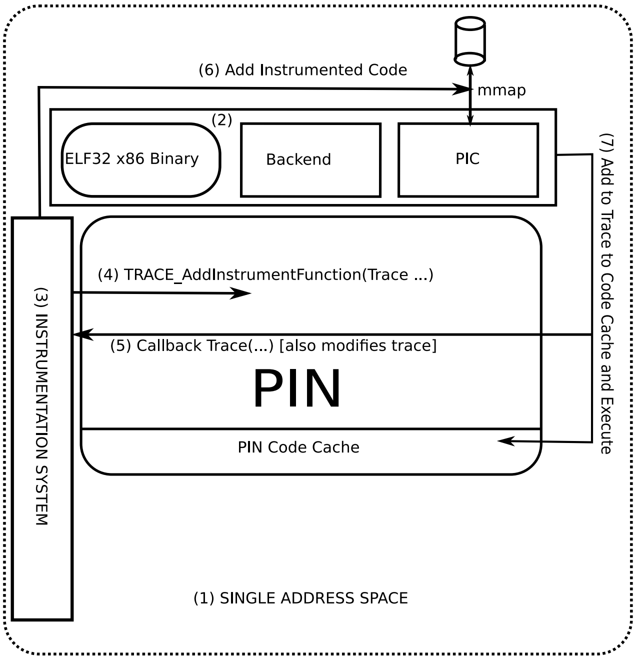

(6) Add Instrumented Code
mmap
(2)
ELF32 x86 Binary
Backend
PIC
(7) Add to Trace to Code Cache and Execute
(3) INSTRUMENTATION SYSTEM
(4) TRACE_AddInstrumentFunction(Trace ...)
(5) Callback Trace(...) [also modifies trace]
PIN
PIN Code Cache
(1) SINGLE ADDRESS SPACE

Figure 4.2: Execution in active mode

4.3 Active mode

Active mode executes the binary using a dynamic binary rewriting engine. Code is examined when it is executed for the first time. If the code is being executed in the context of a critical section, then it is instrumented and placed in the PIC. Otherwise, the code is executed uninstrumented.

Dynamic binary rewriting engines are fairly complex to build and maintain and by itself can form the topic for a PhD thesis [NS07]. I thus chose to leverage an existing dynamic binary rewriting engine for this part of the instrumentation system. I use PIN [LCM$^+$05], a widely used and stable dynamic binary rewriting engine for x86 binaries. My decision to use PIN was guided by two factors.

First, PIN provides an excellent high level interface to inspect and manipulate x86 code. It provides a C++ API that can operate at various levels of abstraction: from whole images, down to functions, basic blocks and individual instructions. PIN also includes a (not very widely used) API called X86 Encoder Decoder(XED)to directly decode, manipulate and re-encode x86 instructions (complex due to their CISC nature) from and to machine code. I made extensive use of XED to build the instrumentation system in this dissertation.

Second, PIN has a large community of users and is actively maintained. This is important because the x86 ISA is actively changing (such as the addition of SSE3 instructions) and it is important that the binary rewriting engine keep up with these additions to be useful for SLE now and in the future.

Figure 4.2 details the key software blocks and their interactions when operating in active mode.

```
char* (*fptr)(char *, char *) = strcpy;
fptr(a, b);
pthread_mutex_lock(&lock);
fptr(c, d);
pthread_mutex_unlock(&lock);
```

Figure 4.3: The same function in multiple contexts

The first key point to note is that all components are in the same address space (1). The three executable components (2) are the x86 binary, the backend and PIC (which is mapped into memory for use). These are never executed natively and instead executed out of PINs code cache (bottom).

The instrumentation system logic dealing with active mode exists as a *pintool* (3). It is written in C++ and compiled into a shared library. It is loaded with PIN and interacts with it at a *trace* granularity – a trace is a single entry multiple exit contiguous sequence of basic blocks that is begun at the target of any branch and ends at an unconditional branch. The pintool registers a callback with PIN through the `TRACE_AddInstrumentFunction` call at initialisation time (4). For every new trace encountered, PIN presents the trace for manipulation to the pintool (5). If the trace originates from a critical section, the instrumentation system logic places an instrumented version of the basic block(s) in the trace into the PIC (6). It then modifies the trace to branch to the instrumented version in the PIC. All executed code (other than that from PIN or the pintool) is placed (after possible modification) into PIN's code cache (7) from where it is actually executed.

Most of the heavy lifting of binary interception, analysis and synthesis is thus done through PIN. The remaining sections focus on features of active mode contained in the pintool that are relevant to SLE_x86.

4.3.1 Identifying critical sections

Active execution depends being able to determine whether a basic block belongs to a critical section. Unfortunately this is not a *static* characteristic of a basic block. Consider the example in Figure 4.3. In that example, the string copy library function is called through a function pointer twice, first outside any critical section and next inside a critical section. On the second call, PIN no longer presents traces in string copy for instrumentation since it has already added these instructions (without instrumentation) to its code cache after the first call. To solve this problem I needed a way to communicate the different contexts of a basic block, such as for the string copy code in the example, to PIN.

To do this, I make use of a PIN facility called *trace versioning*. This allows a trace (and hence basic blocks in it) to be assigned a version string. The same trace with two different version strings is treated as two different traces by PIN. Traces are associated with a NULL version by default. After entering a critical section, I set the trace version to a special string indicating critical section context, which is then propagated to traces executed by branches from that trace. This ensures that executed code is presented for instrumentation even if it has been encountered before (as in Figure 4.3). At the end of a critical section, I reset the trace version string to NULL.

The next problem is to determine when execution enters or exits a critical section. To do this, I first determine (using a tool, before the program is run) a translation from lock and unlock function names to corresponding function addresses. For every instruction presented by PIN, I check if it is a direct control transfer (direct x86 `jmp`/`jcc` or `call`) and if the target is the same as one of these addresses. If so, I insert the appropriate instrumentation call. The actual lock and unlock calls are deleted, leaving the backend to either elide or acquire the lock. The backend is then responsible for indicating when critical sections are begun and ended (by maintaining a count of held locks). On a critical section begin, I require a call to a specially named function in the backend (`cs_begin`); similarly, on encountering the end of a critical section, I require a call to another specially named function (`cs_end`). The instrumentation infrastructure looks for execution of these functions (which can be empty "no-ops") in order to learn when critical sections begin and end. Traces branched to by a return from `cs_begin` have their version set to critical section context. On the other hand, traces branched to by a return from `cs_end` have their version set back to NULL. This allows dynamic handling of nesting such as that required for Figure 4.1.

4.3.2 Basic block discovery

Active execution aims to instrument every basic block that can be executed in critical section context. One way to do this is to wait for execution to discover every possible trace in a critical section and then instrument it. However, this makes building a complete PIC a function of input and timing (in multithreaded execution) and in practise this leads to a large number of program iterations being needed to build a complete PIC. Even worse, in the presence of indirect branches it is impossible to decide if a PIC is complete.

However, the problem *is* tractable given only direct branches by simply traversing the control flow graph starting from a lock call. I exploit this fact by extending dynamic code discovery made available by PIN with static control flow graph traversal in the instrumentation logic of the pintool. To do this, starting with a lock call, I traverse the control flow graph of basic blocks using the depth first search (DFS) shown in Algorithm 11. The DFS does not look past indirect branches and unlock calls. For direct conditional branches it processes the fall-through basic block immediately to avoid an unnecessary branch in the PIC.

I thus depend on execution only to discover critical section boundaries and to look past indirect branches. In the evaluation at the end of this chapter I demonstrate how this combination of static traversal of the control flow graph (similar to that used in static binary rewriting tools such as ATOM) and dynamic code discovery is effective in improving the rate at which basic blocks are added to the PIC.

Algorithm 11 Depth-first search of control flow graph

```
 1: while BasicBlockStack is not empty do
 2:    bb = BasicBlockStack.pop()
 3:    Instrument bb and add to persistent instrumentation cache
 4:    ins = bb.LastInstruction()
 5:    /* ins must be a branch */
 6:    if ins ends a critical section (unlock call) then
 7:       continue
 8:    else
 9:       if ins is a direct branch then
10:          bb = BasicBlockAt(ins.target())
11:          BasicBlockStack.push(bb)
12:       if ins is a conditional branch then
13:          /* has a basic block at fall-through */
14:          bb = BasicBlockAt(ins.next())
15:          goto line 3
```

4.3.3 Basic block instrumentation

I now discuss how memory access instrumentation is added to those individual basic blocks determined to be reachable in critical section context. To illustrate this, I use an example basic block from one of the benchmarks considered in this dissertation: concurrent AVLtree. Figure 4.4 shows the original basic block on the left, with the instrumented version of the basic block on the right.

The numbered instructions on the right correspond to the numbered instructions on the left. For example, the first instruction accesses memory. This is converted into an instruction that first loads the target address into the `eax` register. The next few instructions load the size of access into the `edx` register and the rmw flag into the `ecx` register. The size and rmw flag are encoded such that the most common values (4 bytes and false) map to zero. This means the registers can be set up with a two byte instruction (exclusive or-ing the register to itself), keeping the size of instrumentation and hence instruction cache pressure down. The call to the instrumentation hook returns the (possibly) different address to use in `eax`, which is then used in the instrumented version of the basic block.

The first notable feature of the instrumentation is flag and register management. Since calls to the instrumentation hooks are expected to destroy the `eax`, `edx` and `ecx` registers as well as the flags, these need to be saved and restored as appropriate. This is accomplished by the un-numbered instructions in the instrumented version of the basic block. The save area is setup

```
# Note: AT&T format-> operation src, dst
                                                    lea    0xfffffff0(%esp),%esp
# Memory[8 + Reg[eax]] -= 1;                        mov    %eax,0x0(%esp)
1. subl   $0x1,0x8(%eax)                        1.1 lea    0x8(%eax),%eax
                                                    mov    %ecx,0x4(%esp)
                                                    mov    %edx,0x8(%esp)
                                                1.2 xor    %edx,%edx
                                                1.3 xor    %ecx,%ecx
                                                1.4 inc    %ecx
                                                # Call SpeculativeWrite
                                                1.5 call   0xff6a4730
                                                1.6 subl   $0x1,(%eax)
# Memory[12 + Reg[eax]] = Reg[esi];                 mov    0x0(%esp),%eax
2. mov    %esi,0xc(%eax)                        2.1 lea    0xc(%eax),%eax
                                                2.2 xor    %edx,%edx
                                                2.3 xor    %ecx,%ecx
                                                2.4 call   0xff6a4730
                                                2.5 mov    %esi,(%eax)
# Memory[4 + Reg[esp]] = Reg[eax];                  mov    0x0(%esp),%eax
3. mov    %eax,0x4(%esp)                        3.  mov    %eax,0x14(%esp)
# Reg[ebx] = 0;
4. xor    %ebx,%ebx                             4.  xor    %ebx,%ebx
# Memory[Reg[esp]] = Reg[edi];
5. mov    %edi,(%esp)                           5.  mov    %edi,0x10(%esp)
                                                    mov    0x8(%esp),%edx
                                                    mov    0x4(%esp),%ecx
                                                    lea    0x10(%esp),%esp
6. call   8048ba0 <rebalance_insert>            6.1 push   $0x8048e12
                                                6.2 jmp    0x123c4ba0
```

Figure 4.4: Shared memory instrumentation for a basic block

on stack (the PIC is shared between threads) by the first instruction. Liveness analysis is done at the level of the basic block to optimise away unnecessary save restores. For example, the fourth instruction overwrites the x86 flags and thus the flags are not saved.

The second notable feature of the instrumentation is the treatment of memory accessed through the stack pointer (register `esp`). The stack is usually thread private and (due to the limited number of registers on the x86) heavily accessed. If a location is known to be thread-private, accesses to it (other than locked ones) can be re-ordered without any effects observable to other threads and hence without any violation of x86-MM. This means that accesses to the stack can be performed directly without indirection into STM_x86. This is turn removes a large number of redundant calls to `SpeculativeRead` and `SpeculativeWrite`. This however means that there must be some way to return stack locations to their original value if the executing transaction aborts; I discuss how this is done in Section 4.6. Also, there must be some way to detect the case where stack locations are shared between threads (although unlikely, this is not impossible); I discuss how this is done in Chapter 6. Finally, from the perspective of instrumentation, stack accesses need to be adjusted to account for the save area created on stack. This can be seen in the example where the offset for instruction `5` is adjusted upward by 16 bytes.

The final notable feature about the instrumentation is the handling of the call instruction that terminates the basic block. The instrumented version pushes the return address before jumping to the target. This is standard practise for binary rewriting engines and originates from the need to leave return addresses unmodified on stack. In the example, the `rebalance_insert` function would see the original native address rather than the address from the PIC were it to query the return address of the function. A common occurrence of this kind of behaviour is in position independent code, where a call is made to the immediately following instruction which then queries the top of stack to discover the current instruction pointer (there is no direct way on x86 in 32 bit mode to materialise the instruction pointer in any other register).

The instrumentation of basic blocks is also complicated by the fact that the x86 ISA permits complex instructions. Some instructions allow accessing more than one location (such as a push of the contents of a memory location). Another complication arises from string operations where the length of the access cannot be determined statically (it usually depends on the contents of the `ecx` register). I handle such cases by breaking them down into simpler RISC style operations that are then instrumented.

4.3.4 PIC operations

There are four fundamental operations that active mode performs on the PIC. These are loading the PIC into memory; appending instrumented basic blocks to the PIC; executing from the PIC; and querying the PIC.

I load the PIC into memory by doing a memory map (Unix `mmap`) from the disk file containing it. This ensures that the disk file is upto-date with any additions to the PIC. Appending basic blocks to the PIC simply consists of writing out instrumented versions of basic blocks to the end of the PIC.

Executing from the PIC represents a problem due to special handling of self-modifying code by PIN. In order to detect self-modifying code, PIN looks for pages that are being executed from while being marked writable. It then marks these pages as read-only and traps any writes to it in order to detect self-modifying code. This causes large slowdowns when executing instrumented code out of PIC pages. To work around this problem, I map the same PIC page twice, once as executable but read-only and once as read-write but not executable. Appending to the PIC is done through the writable mapping while execution uses the executable read-only mapping.

The final thing that needs to be supported by the PIC are queries to map executable native addresses to instrumented basic block addresses in the PIC, if present. The core of the logic that handles queries is a map $f\,:\,\text{native address} \rightarrow \text{PIC offset}$. Such a map is easy to set up and maintain for a single run but difficult to persist across runs. The reason is that the native executable address of the basic block in the PIC can change across runs. For example, the native address might originate in a shared library that can change its load address on each active execution. To solve this problem, the map is persisted as
$f\,:\,(\text{native address relative to image base, image name}) \rightarrow \text{PIC offset}$. It is loaded and turned

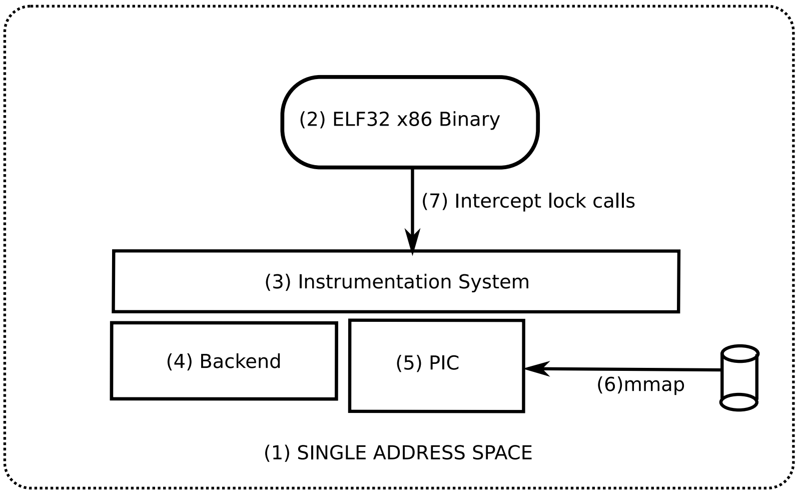


Figure 4.5: Execution in passive mode

into the required form by querying the base of each loaded image (main binary or shared library). A similar technique is used by dynamic binary rewriting engines that persist instrumentation across runs [RCCS07].

4.4 Passive mode

I now cover the operational details of execution in passive mode. Figure 4.5 provides an overview of the software components involved in passive mode. All the components run in the same address space (1). In addition to the x86 binary (2), the instrumentation system (3) and the backend (4) are loaded as shared libraries (through the Unix `LD_PRELOAD` mechanism). On initialisation the instrumentation system loads the PIC (5) by mapping the on-disk version into memory (6). By virtue of being preloaded the instrumentation system is able to intercept lock calls (7) at which point it can redirect execution into the PIC. There is no dynamic binary rewriting engine (PIN) nor its code cache. Execution proceeds directly from the native binary or the PIC.

The critical features of execution in passive mode are simplicity and low overhead. There is no overhead to add instrumentation and – as I shortly show – no overhead when executing outside a critical section.

4.4.1 Preparation

An offline tool needs to be run on the PIC before any passive execution that follows an active execution. The job of this offline tool is effectively to “stitch” together basic blocks in the PIC by patching branches across them, to target instrumented basic blocks in the PIC rather than in the native binary. As an example, consider the call instruction at the end of Figure 4.4. During active execution it targets the native binary and is intercepted and redirected via PIN. Its target must have been added (being a direct branch) by the CFG Walk algorithm to the PIC. The offline patching step patches the branch to point to the instrumented version of the target. Note that on the x86 direct branches are instruction pointer relative and thus the patching is unaffected by PIC relocation across different runs. The patching step is fast. For example, for a 5MB PIC it takes barely a few seconds to run.

4.4.2 Intercept and dispatch

The heart of passive execution is the intercept and dispatch logic. I illustrate this using the example flow in Figure 4.6: dynamic execution would normally move through the basic blocks B_0 through to B_{n+1}. B_0 ends in a call to `pthread_mutex_lock` and thus begins a critical section. Basic block B_n ends in a call to `pthread_mutex_unlock` and thus ends a critical section. The instrumented versions of the basic blocks in the critical section (B_1 till B_n) are shown on the right.

The first step is to intercept all lock calls. This is done through the Unix `LD_PRELOAD` mechanism by intercept logic which is specific to the type and functionality of the locking in use: the example deals with the `pthread_mutex_lock`. Instead of acquiring the lock, executing the critical section and releasing the lock (as on the left), the dynamic linker in the system ensures that the lock call transfers control to the interceptor. The interceptor then calls `Elide` which can decide to do one of two things. In the event that the lock is blacklisted (Section 3.7 of the previous chapter), the interceptor returns control to the normal code flow path. Otherwise, the lock is elided and control transfers to the dispatcher.

The dispatcher queries the PIC to determine the instrumented version of the basic block pointed to by the return address of the original lock call. It then *modifies* the return address on the stack to point to the instrumented version of the basic block (the PIC having been mapped into memory). On exit from the dispatcher, control transfers into the PIC and executes the instrumented version of the critical section (on the right). The unlock call is replaced with a call to `Release` which indicates on return if any locks are held. If no locks are held, control returns to the native binary. Otherwise control returns to the dispatcher, which decides the next basic block to branch to in the PIC.

The result is that no overhead exists for inserting instrumentation or for executing code that is not in a critical section, since in that case execution proceeds from the native binary.

The dispatcher is also used to resolve indirect branches that cause a lookup in the PIC. If the indirect branch cannot be resolved, an exception is raised.

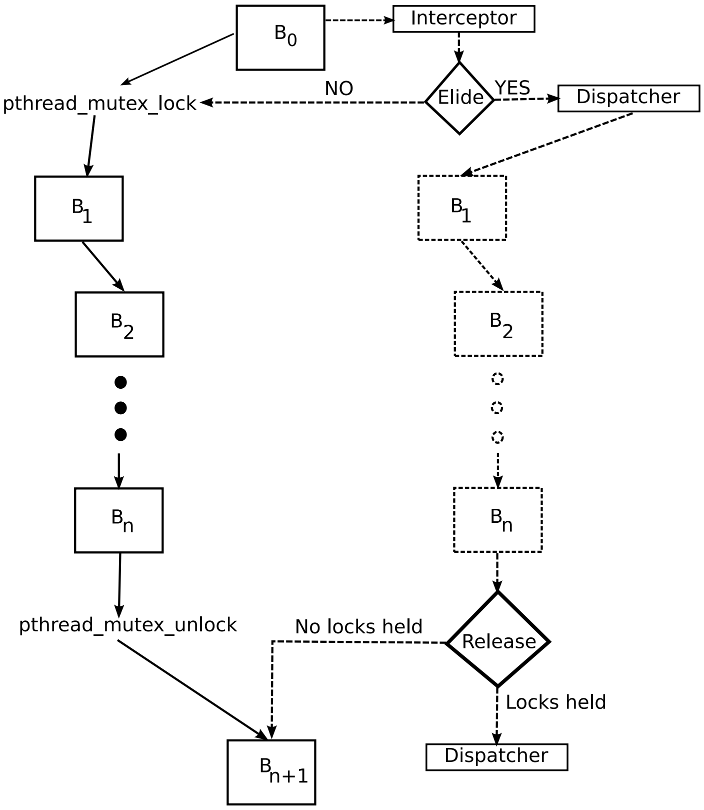


Figure 4.6: Intercept and dispatch

4.5 Exceptions

The instrumentation system raises two kinds of exceptions to the backend:

1. `FlushWriteBuffer`
2. `ExitPIC`

The first exception is raised when any locked instruction or `mfence` instruction is encountered. As I have shown in the previous chapter, the x86-MM cannot be preserved in this case. The backend implementing STM_x86 handles this by switching over to pessimistic locking. The null backend does not care about the flush of the write buffer and simply continues executing out of the PIC.

The second exception is raised when execution is forced to exit the PIC. There are two situations where this can arise: (i) During passive mode the PIC is not complete and does not contain the target of an indirect branch, (ii) A system call is made, the instrumentation system cannot instrument the kernel. The STM_x86 backend handles `ExitPIC` exceptions by falling back to pessimistic locking (none of the benchmarks studied in this dissertation raise this exception, although I have simulated it in unit tests). The null backend handles this exception by switching execution back to the native binary (since it already effectively executes native code by acquiring locks and performing memory accesses directly).

4.6 Checkpoints

A complex requirement for SLE_x86 is the ability for transactional execution to be aborted and its effects rolled back. The instrumentation system provides the capability to take a *thread* checkpoint and roll back execution of the thread to this checkpoint. A checkpoint is taken at the beginning of a critical section and discarded at the end.

There are two effects that transactions can have on system state: they affect register state and they affect memory. I first discuss checkpointing registers and then discuss checkpointing memory.

4.6.1 Registers

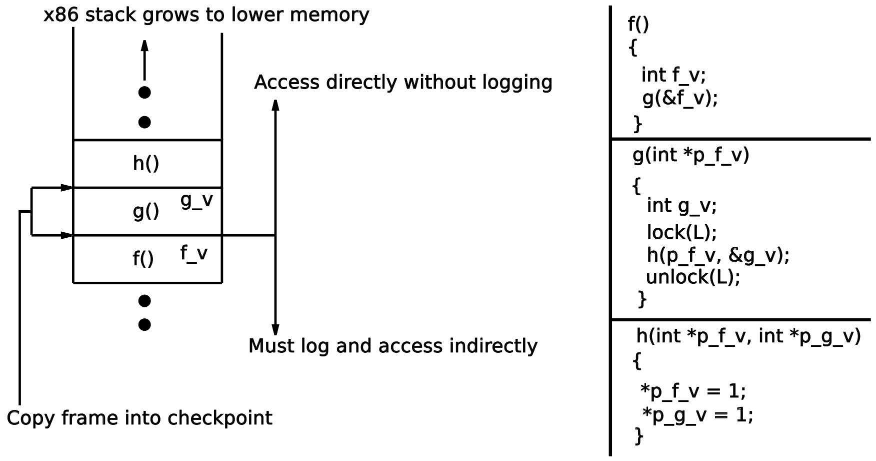
Like library-based STMs such as TL2, I use the `setjmp` and `longjmp` C library functions to checkpoint and restore register state. The `setjmp` and `longjmp` pair however, do not save and restore floating point state for the x86 CPU and this needs to be done by the instrumentation system. The calling ABI in use with the threading libraries I have built backends for (Pthreads and OpenMP) mandate that the x86 floating point register stack is empty and the flags are reset on entry and exit from the lock call. Hence, on a rollback I simply reset the floating point stack and exception flags (using the x86 `emms` instruction) to remove any effects from the aborted transaction.

4.6.2 Memory

All memory accesses indirected into STM_x86 are automatically buffered but accesses to the stack are performed without instrumentation under the assumption that they are thread private (I return to the problem of detecting the case where the stack of one thread is accessed by another in Chapter 6).

However, updates to the stack still need to be rolled back if the transaction aborts. Library-based STMs such as TL2 that are used at a source level require the programmer to avoid updating variables on the stack that are live-in to transactions. This is not possible with automatic instrumentation. Hence the instrumentation system checkpoints the region of stack that might be updated in the transaction.

One problem is to determine exactly how much of the stack to checkpoint. On the x86 architecture, the stack grows downwards (towards lower addresses). Hence, when a memory checkpoint is taken, all stack memory at addresses lower than `esp` on the stack are dead and are not checkpointed. This leaves the portion of the stack from `esp` upto the end of the stack. Clearly this could be a large region. Instead of checkpointing it all, which would unacceptably


x86 stack grows to lower memory
Access directly without logging
h()
g()
g_v
f()
f_v
Must log and access indirectly
Copy frame into checkpoint

```
f()
{
  int f_v;
  g(&f_v);
}
g(int *p_f_v)
{
  int g_v;
  lock(L);
  h(p_f_v, &g_v);
  unlock(L);
}
h(int *p_f_v, int *p_g_v)
{
 *p_f_v = 1;
 *p_g_v = 1;
}
```

Figure 4.7: Checkpointing the stack

slow down execution with software lock elision, I only checkpoint a sub-region [`esp, esp + checkpoint_size`).

The checkpoint size needs to be large enough to capture all accesses to the live region of stack made through the stack pointer during the transaction. I make the (reasonable) assumption that `esp` relative addressing is limited to the frame of the currently executing function and its parameters. It thus suffices to:

1. Checkpoint only the portion of stack occupied by frame and parameters of the function executing the lock call beginning a critical section

2. Raise an ExitPIC exception if control returns from this function before the critical section (and hence the PIC) is exited

The instrumentation system (during the active phase) determines the checkpoint size by scanning the function where the critical section is begun, looking for instructions that reserve space on the stack and those that use an offset from the stack pointer (to access parameters). This is used to conservatively estimate the frame size.

To illustrate how checkpointing works, consider the simple C example in Figure 4.7. Execution switches to the PIC at the critical section begun in function `g`. Accesses through the stack pointer during the execution of the critical section are limited to the frame of `g` and the frame of the called function `h`. The frame of `h` is dead from the perspective of the checkpoint if execution is rolled back. Access to locations in the frame of the callee `f` can only be made through pointers that are indirected into the STM. If control returns from `g` before the critical section is exited, pessimistic locking must be used (via raising the ExitPIC exception) since accesses to the stack can be made in `f` through the stack pointer outside the checkpointed region.

It is also important to ensure, from the perspective of STM_x86 that locations on the stack that can be accessed through the stack pointer within a transaction are never logged. For example,

```
#iterate
emacs program.c
gcc -O3 program.c -lpthread -o program
gdb program


#Generate PIC (iterate until no new basic blocks added)
active_exec.sh program <arguments>

#run without
./program <arguments>

# run with SLE
LD_PRELOAD=libslepassive.so ./program <arguments>
```

Figure 4.8: Putting it together: Using SLE

in Figure 4.7, the write to `g_v` in function `h` must not be logged in the write buffer because a subsequent direct access through the stack pointer in `g` would not see the update. The STM logging functions `SpeculativeRead` and `SpeculativeWrite` are constructed to ensure that locations on the stack at or below (in address terms) `esp + checkpoint_size` are never logged. This ensures, in the example, that `f_v` is correctly logged.

4.7 SLE_x86 in practise

The instrumentation system contained in the pintool, meant for active execution is approximately 4000 lines of C++ code. The implementation supports 32 bit code only. The SLE_x86 backend excluding the STM code described in the previous chapter is approximately 2000 lines of C code. The typical manner in which programs have been run for this dissertation is shown in Figure 4.8. The first step is to write, compile and debug the program. There is no awareness of SLE at this stage.[2]

The next step is to build the PIC. This is done by iteratively running the program under the control of PIN in active instrumentation mode. I provide a script that abstracts away much of the complexity of the PIN command line. The instrumentation system pintool prints the number of basic blocks added to the PIC in the run. I assume that the PIC is complete when 2 consecutive runs add no new instrumented basic blocks. This is only a heuristic since PIC completeness is undecidable in general. However it works well for the programs examined in this dissertation. Once the PIC has been built, the program can be run in passive instrumentation mode.

[2] Indeed, the fact that there is no transactional memory in the toolchain made it easy to debug some of the issues with the larger programs used in the evaluation.

One of the interesting consequences of using aggressive static basic block discovery during active instrumentation is that limited inputs can be sufficient to build the PIC. In the case of the STAMP benchmarks that have been heavily used in this dissertation I used a small input set (intended for simulators) to build the PIC. This meant I could build the PIC in under 2 minutes for all benchmarks; while a full run of the benchmarks using the native input sets took to the order of 18 minutes.

4.7.1 Backends

The backend is specific to the purpose (null or STM_x86) as well as the type of locking in use. In this dissertation, I use the null backend and the STM_x86 backend, as well as a profiler backend that I discuss in the next chapter; with both Pthreads as well as OpenMP. Each of these six individual backends share most of their code and are built via different compile-time settings. The OpenMP backend includes support for the Intel compiler OpenMP implementation that in addition to standard OpenMP locking calls uses so-called "fast dispatch" calls (additional parameters are passed to quickly locate thread private data[3]).

4.8 Evaluation of the instrumentation system

I now evaluate the instrumentation system alone. I use the null backend for this section in order to eliminate any STM related effects. I use the same STAMP benchmarks that were used in the previous chapter and the same 48-core system (Appendix B: Tigger). Instead of using transactions declared and instrumented at source level, I replace the transactions with critical sections that acquire a process-wide lock, thereby executing the transactions with single lock atomicity. The manual instrumentation for shared memory accesses is redefined (through macros) to no-ops i.e. direct access to shared memory. The compiled STAMP benchmark binaries can thus be run either using locks, the null backend (equivalent to running with locks) or (as I discuss in the next section) with software lock elision.

I begin with a static characterisation of the binaries from the perspective of binary instrumentation in the table in Figure 4.9. The 32 bit ELF binaries generated range in size from 52k for Kmeans to 196k for Yada. The number of critical sections (obtained by counting lock calls) range from 6 to 18. The size of the PIC generated by active instrumentation is usually a quarter of the size of the binary (since it does not include code outside critical sections). There are quite a few indirect calls in critical sections (counted from the generated PIC), which would have been a limitation for static techniques. A significant fraction of these are to shared libraries that

[3] ftp://download.intel.com/technology/itj/2004/volume08issue01/art02_compilers/vol8iss1_art02.pdf

Benchmark	Binary size (bytes)	Critical sections	PIC size (bytes)	Indirect calls
Bayes	181603	18	61574	57
Genome	118334	8	18677	30
Intruder	153089	6	30722	26
Kmeans	52821	6	9241	15
Labyrinth	116384	6	20617	23
SSCA2	140156	13	19968	19
Vacation	143772	6	40887	29
Yada	196715	9	56246	50

Figure 4.9: STAMP: Static characteristics

	With CFG walk			Without CFG walk		
Benchmark	Exec 1	Exec 2	Exec 3	Exec 1	Exec 2	Exec 3
Bayes	1435	0	0	723	2	0
Genome	383	0	0	221	0	4
Intruder	629	0	0	452	2	4
Kmeans	178	0	0	91	4	0
Labyrinth	443	0	0	340	2	0
SSCA2	394	0	0	111	0	0
Vacation	853	0	0	464	0	0
Yada	1113	0	0	899	1	0

Figure 4.10: Basic blocks added to PIC in each active execution iteration

would have posed a problem for a transactional memory compiler, since these ‘legacy’ libraries would have remained uninstrumented.

The next set of results focus on the active instrumentation phase when the PIC is built. The table in Figure 4.10 shows the number of basic blocks added in each iteration. In accordance with the heuristic, I stop when no basic blocks are added for two consecutive iterations. In contrast, the table also shows what would happen were static exploration of the control flow graph not in effect. Even after 3 iterations the PIC is missing basic blocks from critical sections for all the benchmarks.

I now focus on the difference between active and passive mode. The entire objective of building a PIC is to execute without the overhead of interception by a dynamic binary rewriting engine. Figure 4.11 shows the overhead of instrumentation (calling into the null backend). I report the execution time of the instrumented binary divided by the execution time of the *native* binary (running with locks). The executions are with one thread only to focus on single thread overheads. This reflects the cost of adding memory access instrumentation. The results in Figure 4.11 show that passive mode is much faster than active mode. The gains vary from 1.13X faster in Labyrinth to 2.27X faster in Yada.

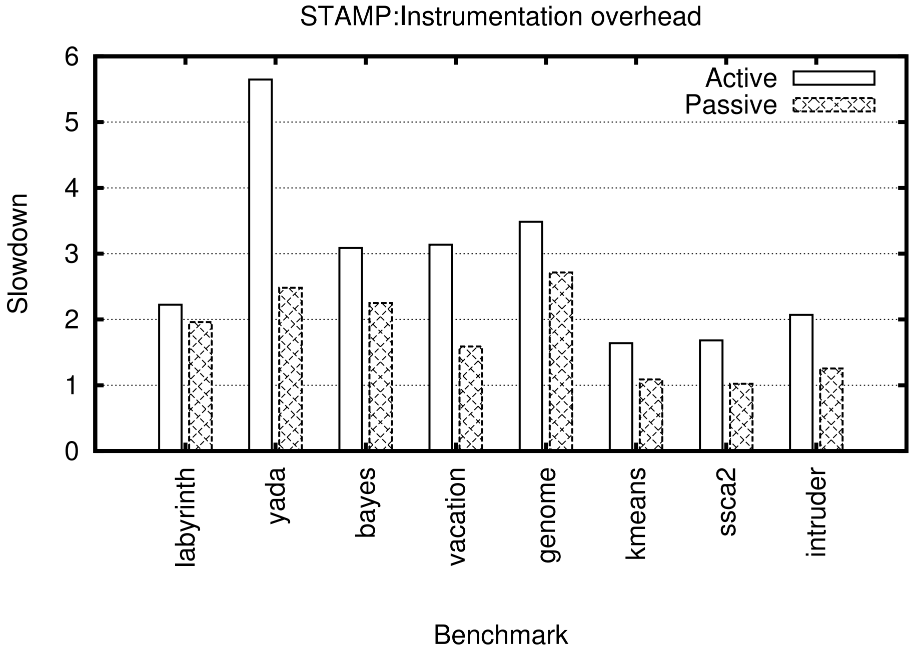


Figure 4.11: Binary instrumentation overhead

4.9 Evaluation of SLE_x86

I now turn my attention to the purpose for which the instrumentation system has been built: software lock elision. I use the STM_x86 backend running in passive mode (with a pre-generated PIC) for the experiments in this section. This is the complete SLE_x86 system. The results (also on the 48-core machine), in addition to contrasting SLE_x86 with locks also repeat the results from STM_x86 in the previous chapter. Note that manually applying the instrumentation and then compiling it results in a substantially different program from that obtained by automatically inserting the instrumentation in machine code. Thus STM_x86 and SLE_x86 are not directly comparable, although one would ideally like them to be close in terms of performance.

The results are split across Figure 4.12 and Fig 4.13. For the four benchmarks Vacation, Kmeans, SSCA2 and Intruder, the performance of SLE_x86 closely approximates the performance of STM_x86. It is interesting to note that in some cases automatic instrumentation is able to outperform that inserted manually and optimised by the compiler. The remaining four benchmarks however do not behave well with automatic instrumentation. In the case of Bayes and Labyrinth, the number of accessed shared memory locations is so large that it overflows the STM logs. This is not fatal for SLE_x86, which falls back to pessimistic locking. Hence, I label them specially as `SLE_x86(lock)` in the graphs. In the case of Yada and Genome, automatic instrumentation trails the manual one by a very large amount. The focus of the next chapter is on building a profiler that can, among other applications, explain this anomaly. In Chapter 6 I present an extension to the automatic instrumentation system that helps to solve this problem and bring the performance of SLE_x86 closer to STM_x86.

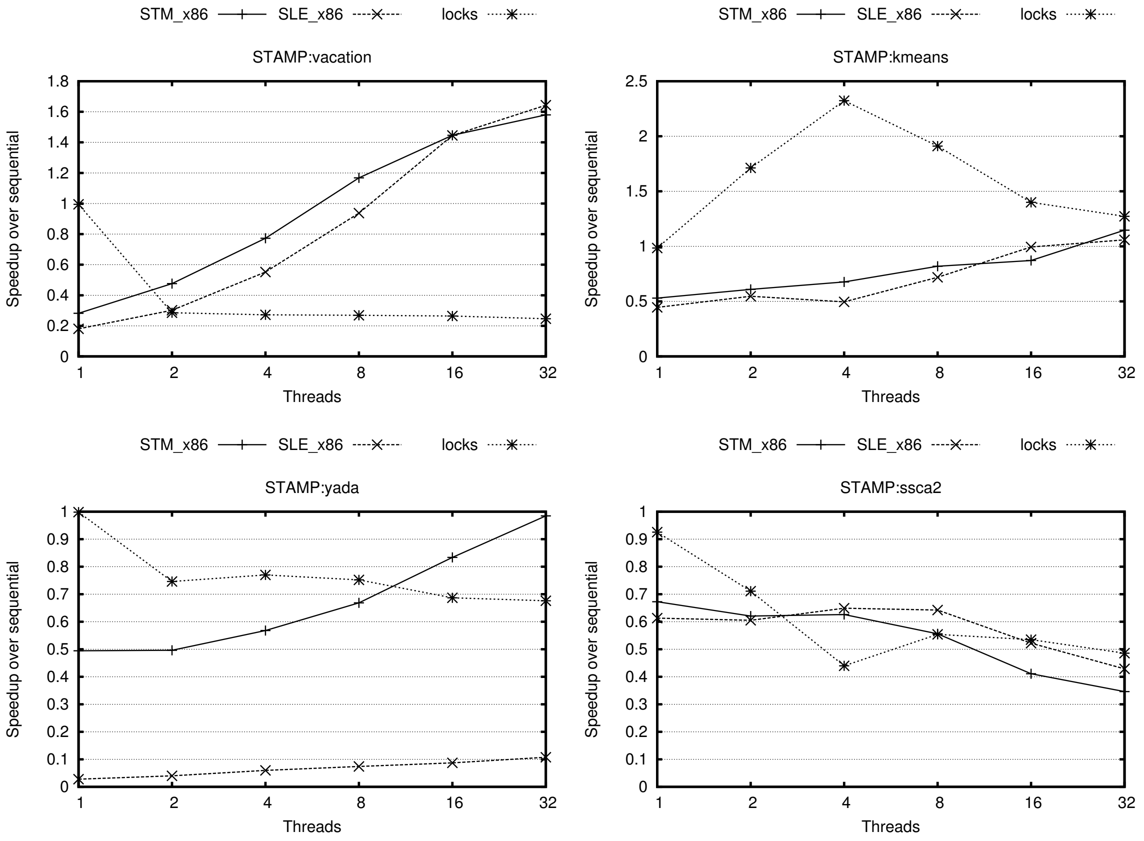


Figure 4.12: SLE on the STAMP benchmarks(1)

Finally, Figure 4.14 shows that automatic and transparent transactional memory can outperform the lock.

4.10 Discussion

This chapter has described an instrumentation system that generates a persistent instrumentation cache (PIC). The PIC is a file on disk holding instrumented x86 machine code from critical sections in the original binary. It is generated by executing the binary with instrumentation enabled in *active* mode. In active mode execution is intercepted by the PIN binary rewriting engine that enables instrumentation and placement of basic blocks from critical sections into the PIC. Once the PIC is (heuristically) complete, instrumentation can be executed in *passive* mode. Passive mode incurs far lower overhead than active mode and as I have shown, can be close to the performance of manual instrumentation.

This chapter however, also throws up a set of interesting performance-related questions. Automatic instrumentation lags manual instrumentation by a large amount for four of the STAMP benchmarks. There are also interesting TM performance questions that arise here. For example, why does Vacation benefit far more from transactional memory than SSCA2? In order to

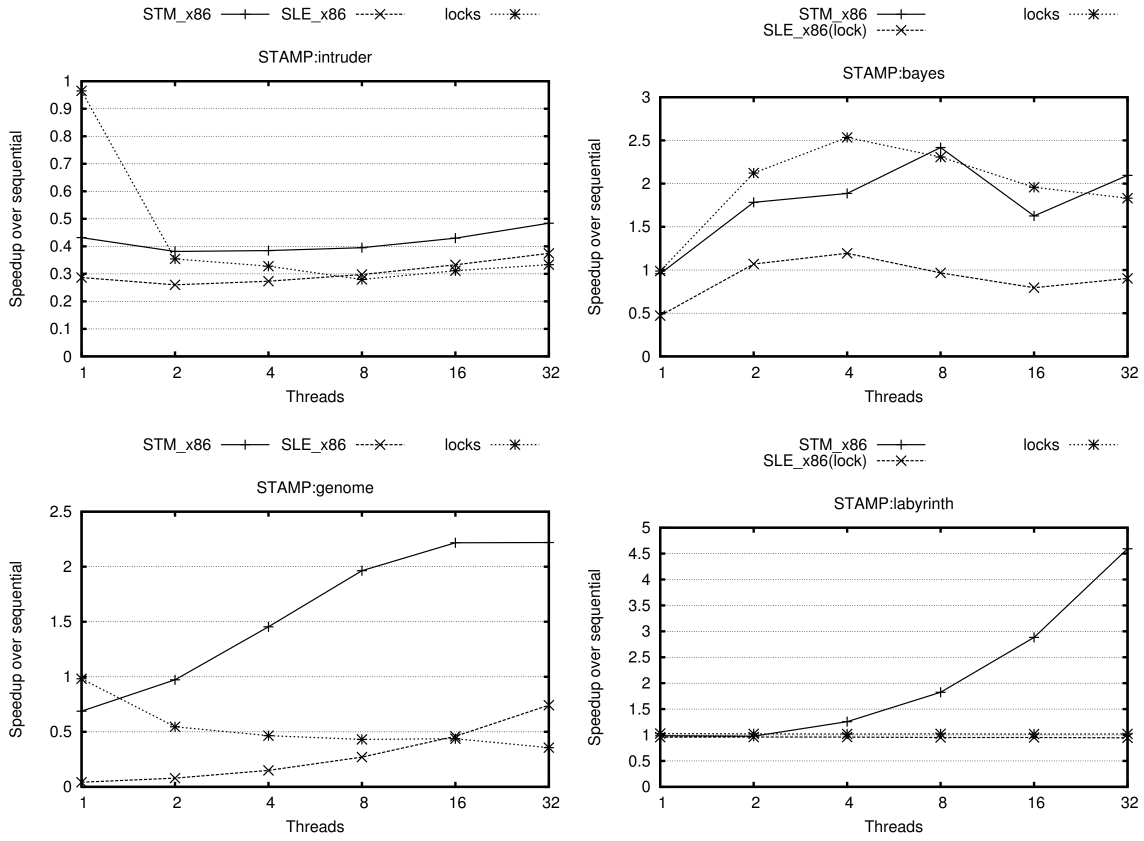


Figure 4.13: SLE on the STAMP benchmarks(2)

	Threads					
Benchmark	1	2	4	8	16	32
labyrinth	NO	NO	NO	NO	NO	NO
yada	NO	NO	NO	NO	NO	NO
bayes	NO	NO	NO	NO	NO	NO
vacation	NO	YES	YES	YES	YES*	YES*
genome	NO	NO	NO	NO	YES	YES
kmeans	NO	NO	NO	NO	NO	NO
ssca2	NO	NO	YES	YES	NO	NO
intruder	NO	NO	NO	YES	YES	YES

Figure 4.14: Is using SLE_x86 better than using the lock ? (‘*’ means also better than sequential)

answer these questions I have built a profiler, described in the next chapter, that uses the PIC and passive instrumentation to produce a profile for x86 binaries. This profile identifies critical sections and helps to explain their behaviour when executed with transactional memory.

Chapter 5

Critical section characterisation

One goal of my research is to separate mechanism from policy: software is written, compiled and run without awareness of transactional memory which can automatically be added if desired at runtime. In this chapter I show that this decoupling can extend to profiling and performance debugging. I show that x86 binaries can be profiled for suitability for transactional memory without explicitly being written to use it.

Transactional memory works well as a replacement for a lock in the presence of two favourable conditions. The first is the requirement for disjoint-access parallelism [IR94]. Informally, transactions can be executed in parallel when they do not make conflicting accesses to the same memory location. A good example would be critical sections manipulating red-black trees. Updates are usually localised in subtrees and thus for a large enough tree are unlikely to conflict with each other. An example with low disjoint-access parallelism is a shared integer counter: all updates to it will conflict.

More recently, in the context of examining how existing lock-based applications can be adapted to use transactional memory, researchers have come across a second condition: lock contention. A lock is a scalability bottleneck only if threads contend for it; if there is no contention there are no scalability benefits gained via the use of transactional memory. This was highlighted by TxLinux [RHP$^+$07], where a meagre 5% speedup was obtained when spinlocks in the Linux kernel were replaced with transactions running on a hypothetical hardware transactional memory. The explanation is that Linux is a heavily optimised piece of software and there was very little or no contention for locks. The problem is more acute with software transactional memory since its associated slowdown can significantly hurt overall performance when there is no scalability benefit to compensate for it.

The profiler in this chapter measures the suitability of transactional memory for an x86 binary by measuring each of these quantities. A lock-based execution of the binary is used to obtain the profile. There is no assumption made about any underlying (hardware or software) TM system as none is used.

5.1 Disjoint access parallelism

The only systematic attempt thus far to *quantitatively* define and measure disjoint-access parallelism from actual program runs is Von Praun's dependence analysis [vPBC08]. The objective was to consider the possible speedups that could be obtained, were program execution to be limited only by data-flow dependencies. In order to do this Von Praun collected memory access traces from program fragments that could be run in parallel (which he termed tasks) and defined a metric called dependence density across them. I describe how dependence density is defined and measured below.

Task t_2 is defined as *flow dependent* on task t_1, if t_2 reads from a location written to by t_1, expressed as the predicate flow_dep(t_1, t_2). Further let len(t) be the length of task t using some suitable metric (such as number of memory accesses).

Consider a set of tasks T_p that can potentially run in parallel. The data dependence of a task $t \in T_p$ on the rest of the tasks in the set is defined as:

$$\text{dep\_dens}(t) = \frac{\displaystyle\sum_{s \in (T_p - t) \wedge (\text{flow\_dep}(s,t) \vee \text{flow\_dep}(t,s))} \text{len}(s)}{\displaystyle\sum_{s \in T_p - t} \text{len}(s)}$$

The dependence density represents the probability that the task t would see a conflict if executed in parallel with another task in T_p. Note that the dependence density takes into account the length of a task. This means that a short running task is unlikely to create a conflict with a longer running task even if the longer running task is flow dependent on it, since it may well finish before the flow dependence manifests itself.

Von Praun used manual instrumentation of the source to discover which tasks could potentially run in parallel. That work included looking at various sources of parallelism such as executing loop iterations in parallel (e.g. thread level speculation). Von Praun used data from a *single threaded* run of the binary.

In contrast I am concerned only with parallelism from contending instances of critical sections executing in parallel and this admits a more automatic approach. I use an actual run of the binary and place a total order on critical sections executed. I then use a sliding window on this total order that includes one critical section from each thread. This approximates critical sections that are likely to run together. It also takes into account that critical sections on the same thread might be causally dependent on each other (such as if they were executed in a loop) and thus should not be considered for parallelism. Any instance of the sliding window thus represents the set T_p in the equation above. To compute the dependence density I also trace all the memory accesses in a critical section.

Tasks represent dynamic instances of static critical sections in the binary. It is thus useful to also roll-up the dependence density computed for each task into a dependence density for the static critical section that originates it. I do this using a simple average: If T_{cs} represents all the dynamic tasks for a static critical section: cs, then its rolled up dependence density is:

$$\text{dep\_dens}(\text{cs}) = \frac{\sum\limits_{t \in T_{\text{cs}}} \text{dep\_dens}(t)}{|T_{\text{cs}}|} \tag{5.1}$$

Finally, I also compute the dependence density for the application as a whole. For every window of tasks T_p I compute the data dependence density for the window (Von Praun used the same notion of dependence density for a set as a whole):

$$\text{dep\_dens}(T_p) = \frac{\sum\limits_{t \in T_p} \text{dep\_dens(t)}}{|T_p|}$$

I then average this across all windows:

$$\text{overall\_dep\_dens} = \frac{\sum\limits_{T_p \in \text{Windows}} \text{dep\_dens}(T_p)}{|\text{Windows}|} \tag{5.2}$$

The overall dependence density represents how much disjoint access parallelism is present in the application as a whole. If one assumes a scheduler that picks disjoint (non-data-flow dependent) critical sections for execution, then the average number of threads that can be scheduled, given n available threads is:

$$\text{Average\_Threads} = \sum_{i=0}^{n-1} (1 - \text{overall\_dep\_density})^i \tag{5.3}$$

The sum reflects the fact that with increasing threads the probability of encountering a dependence increases. Note that the equation assumes for simplicity that the dependence density is independent and thus adding threads does not increase the pairwise dependence density.

5.2 Lock contention

The two most direct metrics for lock contention are the amount of time spent waiting for a lock and the number of waiting threads (waiters) seen on the average when a thread attempts to acquire a lock. I measure both of these at runtime. The benefit of measuring both is that for applications with transient lock contention (such as the Quake benchmark I examine in

Chapter 7), the waiter count reveals lock contention rather than the fraction of time spent waiting for the lock.

There are three data structures that I use to measure lock contention. The first is `Counters`, a per-thread array, each element of which is a set of four unsigned integer counters used to accumulate the amount of time spent waiting for and executing in a critical section; the average number of waiters seen (count of threads either waiting for or holding the lock); and the number of times this critical section has been sampled. The second is the globally shared `WantCS` array each element of which is an integer counter. The third is the globally shared `GotCS` array each element of which is also an integer counter. Critical sections are mapped (many-to-one) to a numerical index used to look up the first array, this is accomplished by hashing the program address of the first instruction in the critical section. The address of the lock protecting the critical section is mapped to a numerical index used to look up the second and third array, by hashing the lock address.

These arrays are thus essentially hash tables and this means that collisions can have an impact on accuracy. If there are collisions in the per-thread hash table then data for multiple critical sections gets coalesced into one. I use a hash table size of 1024 for experiments in the dissertation and I did not observe any collisions (there are far less than 1024 critical sections in the binaries I studied). On the other hand if there are collisions in the global hash table then the number of waiters is incorrectly estimated: in essence multiple locks are treated as one. This can happen in practise, particularly if the binary is using locks embedded in dynamically allocated data structures in which case the number of active locks can be extremely large. Note that even in this case, it is only when two simultaneously acquired locks map to the same hash bucket that inaccurate measurement results, since I depend on the difference between two quantities that are updated within the lifetime of a dynamic instance of the critical section.

Instrumentation needs to be inserted before and after a lock acquire and before and after a lock release. The instrumentation in pseudocode form is shown in Algorithms 12, 13, 14 and 15.

Algorithm 12 PreAcquireLock(cs, L)

```
Counters[cs].Waiters += (-GotCS[L] + WantCS[L])
AtomicIncrement(WantCS[L])
Counters[cs].WaitTime -= Clock()
```

Algorithm 13 PostAcquireLock(cs, L)

```
Counters[cs].WaitTime += Clock()
Counters[cs].CSTime -= Clock()
```

An interesting subtlety is present in the algorithms due to the racing access to two quantities, `WantCS[L]` and `GotCS[L]` in Algorithm 12. Both of these quantities are being updated by racing acquisitions of the lock. Allowing the race ensures that I avoid any additional costly

Algorithm 14 PreReleaseLock(cs, L)

```
Counters[cs].CSTime += Clock()
Counters[cs].WaitTime -= Clock()
```

Algorithm 15 PostReleaseLock(cs, L)

```
Counters[cs].WaitTime += Clock()
AtomicIncrement(GotCS[L])
Counters[cs].samples++
```

locking protocols or atomic operations that would add instrumentation overhead. It is important to note the ordering of reads to those two variables in line 1 of Algorithm 12. `GotCS[L]` is read before reading `WantCS[L]`. Due to the order of updates to those two variables we always have `GotCS[L]` ≤ `WantCS[L]`. This ensures that the added quantity to the accumulation variable is always positive in line 1 of Algorithm 12. This however means that I can slightly overestimate the number of waiters due to a racing update to `GotCS[L]` that I miss. This error is minimal since the window between the two reads is small; and is in any case far preferable to reading them in the opposite order, which might result in a negative quantity leading to overflow errors in the unsigned numbers used to represent counts.

In order to accurately measure timing, I use the CPU timestamp counter (accessed using the `rdtsc` instruction) represented as the call to `Clock()`. This provides an accurate fine-grained time source. However it is entirely possible that there is a drift in the timestamp counters across different processors. I bind threads to processors in order to ensure that this does not affect timing measurements due to threads migrating across processors. Finally, for the experiments using the profiler in this dissertation, I never use more threads than processors. This ensures that multiprogramming does not play any significant role in the measured contention.

5.3 Profiler operation

The profiler operates in two steps. First, the binary is run with passive instrumentation. I use a variant of the null backend that continues to acquire locks and perform memory accesses directly. However, in addition, it dumps a trace of memory accesses in critical sections and also measures lock contention using the algorithms in the previous section. The hash tables containing lock contention measurements are dumped at the end of the run. In the second step a post-profiling tool is run that analyses the data generated by the profiling run and generates the final profile. I describe these two steps next.

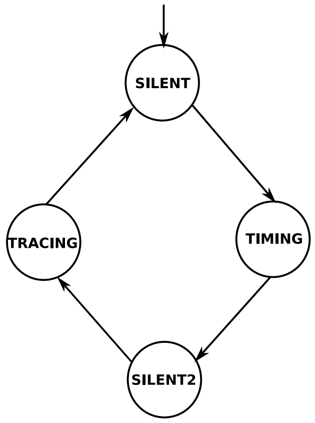


Figure 5.1: Profiler phases

5.4 Profiling

The profiler generates information from a *single* execution of the binary by sampling a subset of the executed critical sections. The sampling is done by using a state machine. Each state has a threshold associated with it. If the total count (across all threads) of critical sections executed while in a particular state reaches its associated threshold, a state change occurs. Critical sections are profiled only in particular states and thus the state machine picks (samples) critical sections. The need for this state machine is explained below.

For the sampled critical sections, the profiler needs to generate lock contention information and memory access traces. This straightforward, since lock and unlock calls are already instrumented in passive instrumentation mode and all shared memory accesses within a critical section are instrumented due to execution out of the PIC. The difficult part is doing the two together. Measuring lock contention requires accurate timing information and minimum instrumentation overhead. On the other hand, tracing memory accesses in a critical section requires recording every memory access to file. This adds significant overhead to critical section execution.

In order to reconcile these conflicting requirements, the profiler uses a 4 state machine during execution, shown in Figure 5.1.

In the `TIMING` phase critical section execution is done uninstrumented out of the native binary. The dispatcher after executing the `Elide` instrumentation call switches execution to the native binary instead of the PIC. This is possible due to the flexibility of passive instrumentation that allows direct execution of native code. The critical section thus executes uninstrumented and imposes no extra overhead to lock contention measurements.

In the `TRACING` phase no lock contention measurement is done but critical section execution is redirected by the dispatcher to execute out of the PIC. The backend writes accessed addresses together with information about the access out to file. In addition to tracing individual critical

```
CS            cs_frac wait_frac avg_q_length rd_ops   rd_locs  wr_ops  wr_locs dep_dens
client.c:247  0.001   0.008     6.87         447.585  203.204  20.510  14.076  0.440
client.c:267  0.000   0.009     6.87         126.768  74.950   4.363   4.307   0.016
client.c:196  0.080   0.828     6.87         447.412  127.811  12.094  11.601  0.007

Overall_waiting Overall_dep_density Pred_AvgThreads
0.845409        0.011508            7.685098
```

Figure 5.2: Example profiling output

sections, this phase also imposes a total order on all critical sections (regardless of the protecting lock) by incrementing a globally shared counter.

The two `SILENT` phases serve to separate the timing and tracing phases (ensuring that timing does not suffer from errors due to simultaneous tracing). In addition they also serve to impose a sampling rate, since no instrumentation is performed during the silent phases. The number of executed critical sections (measured as lock acquisitions) spent in each phase is configurable as a runtime parameter to the profiler. For the runs in this dissertation, I used settings of 700 for the timing phase, 100 for the tracing phase and 100 for each of the silent phases. This meant timing was measured for the majority of the execution while 10% of the critical sections were traced.

5.5 Post processing

The post processing tool takes as input the lock contention related hash tables and the memory access traces, as well as debug information from the binary being profiled. It then produces a profile for the binary, as shown in Figure 5.2 (which was generated from the Vacation benchmark in the STAMP suite). Each critical section profiled is identified by the source file and line number. For each critical section, the fraction of total execution time (counting time on each thread separately) spent in the critical section and waiting for the critical section is printed. The profile also includes the average number of other threads, executing in or waiting for, the critical section when an attempt is made to acquire the lock (the `avg_q_length` field).

Next memory access related statistics are printed for the critical section; these include the number of read and write operations seen (read-modify-writes are counted as both reads and writes) as well as the unique locations that are read or written to. Finally the dependence density of that static critical section across all the sample windows is computed (using equation 5.1) and printed. The last two lines display application-wide summaries. The overall amount of time (as a fraction of time counted across threads) spent waiting for a lock is displayed. Also displayed is the overall dependence density (computed using equation 5.2). This is converted to the number of threads that could be scheduled in parallel (assuming as many threads as were profiled is available) using equation 5.3.

Computing the exact number of unique locations touched or determining whether any data-flow exists between two accesses requires determining the exact set of bytes touched by the access.

One way to do this is to record the size of every access as additional information in the trace files. Instead, I take as input to the post-processing tool a size parameter (which is a power of 2) and assume that every access touches a set of bytes of that size with the starting address rounded down to a multiple of the size. This saves space in the trace files and reflects the assumption that most modern compilers allocate data in multiples of four or eight bytes. It also allows exploring the effect on dependence density of STMs that track conflict at coarser granularities, such as 16, 32 or 64 bytes.

Rolling up the lock contention related information from the dumped hash tables is straightforward, as shown in Algorithm 16. It involves rolling up data in hash buckets on file into appropriate static critical sections. The per-thread hash table buckets on file include their key (the starting program address of the originating critical section). This is mapped to the static critical section through debug information available from the binary.

Algorithm 16 ProcessTiming

```
for all thread t in profiled_threads do
    for all bucket b in per-thread hash table for thread t do
        Locate static_cs for bucket b
        static_cs.WaitTime += b.WaitTime
        static_cs.CSTime += b.CSTime
        static_cs.Waiters += b.Waiters
        static_cs.samples += b.samples
for all static critical section static_cs do
    static_cs.WaitFrac = static_cs.WaitTime/TotalTimeAcrossThreads
    static_cs.CSFrac = static_cs.WaitTime/TotalTimeAcrossThreads
    static_cs.AvgQLength = static_cs.Waiters/static_cs.samples
```

The next phase of the profiler computes the dependence density using the memory access traces generated by the profiler. An abstraction of the steps to do this is shown in Algorithm 17. The outermost loop walks all the dynamic critical sections traced (tasks in the terminology of Section 5.1) in the total order generated. It updates the current window in line 3, as part of which it also rolls up memory access statistics such as locations accessed and operations done. It then does a pairwise comparison of tasks to determine whether a flow exists (this is done by generating bloom filters from the accessed addresses for quick comparison against each other). The dependence densities for individual tasks are then rolled up into the corresponding static critical sections and the overall dependence density for the window is also computed. This bookkeeping is straightforward and has been abstracted into line 7 of the pseudocode for clarity.

Line 7 can clearly be an expensive step since it has $O(n^2)$ cost with n threads. Traces can be large, often ranging to hundreds of thousands of reads and writes (Section 5.7). One way to reduce this cost is to observe that all instances of it can be executed in parallel. In the implementation of the post processing tool I spawn a number of threads to do exactly this.

Since I generally run post processing on the same system as on which I took the traces I spawn as many threads as were actually profiled, bringing the sequential cost down to $O(n)$. This helps keep post processing costs down to a reasonable level. Even for the largest traces discussed in Section 5.7, post processing did not take more than five minutes.

Algorithm 17 ProcessDependences

```
for all dynamic task d in master trace file do
    t := profiled thread that generated d
    window[t]:=d
    Update memory access related statistics for critical section that generated d
    for all thread t1 in profiled_threads do
        for all thread t2 in profiled_threads such that t1 $\neq$ t2 do
            consider flow from window[t1] to window[t2]
```

5.6 Characterising a Microbenchmark

In this section, I illustrate the application of the profiling tool by characterising a microbenchmark: the red-black tree long studied by the STM community [Fra03, MSH$^+$06, DSS06]. The red-black tree is used to hold a set of key-value pairs, supporting *lookup*, *update* and *delete* operations. As is well known, red-black trees have $O(log\ n)$ costs for lookup, insert and delete for a tree with n nodes. This fact can be verified by using the profiler on an x86 binary containing the implementation of such a tree.

I used a simple red-black tree implementation, placing a coarse Pthreads reader-writer lock around single-threaded implementations of the tree access functions: update, delete and lookup. Running this benchmark with eight threads, I obtained critical section memory footprints for varying depths. The results shown in Figure 5.3 agree well with the analytically known properties of red-black trees. The lookup function does no writes but does perform reads that in number are twice the depth of the tree plus a constant factor. The scale of two comes in because both the key and the next child pointer are read while moving down the tree. The constant factor is due to the initial access to obtain the root of the tree. The number of reads for updates and deletes follows a similar scaling (but incorporating additional constants to check keys and so on). For updates and deletes, on the average, a constant number of writes are made to rebalance the tree (regardless of depth) and this is reflected in the graph.

The experiment also confirms that tracing memory accesses through the PIC and the post-profiling tool works properly. This is important since it forms a critical but complex part of the profiler unlike the relatively simple measurement of lock contention.

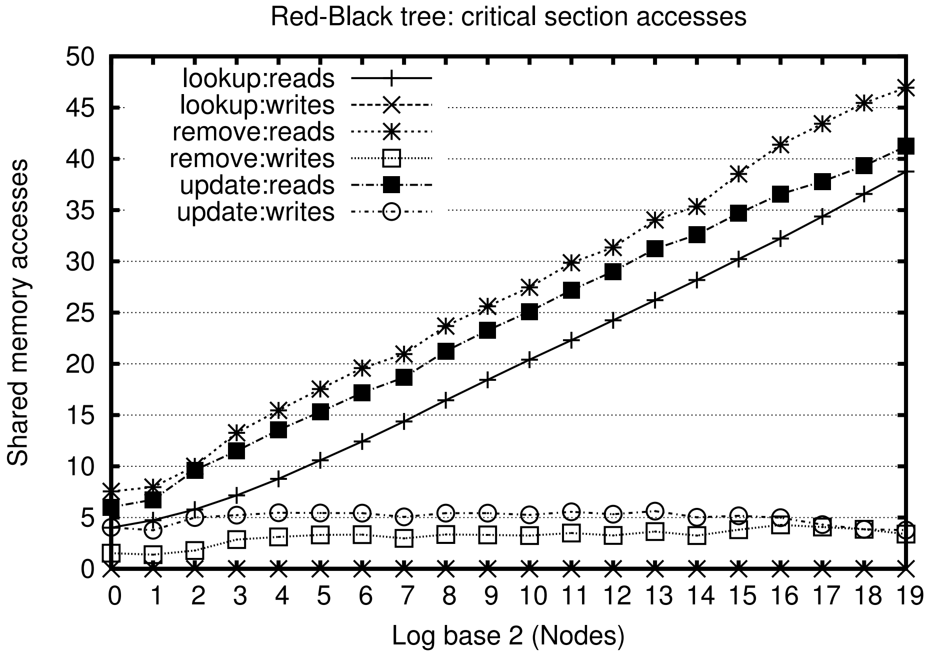


Figure 5.3: Red-Black tree memory access characterisation

5.7 Characterising STAMP

I now apply the profiler to the STAMP benchmarks that I have used thus far in the dissertation. I profile a run with 8 threads for each benchmark, using the same PIC that was built for SLE_x86 in Chapter 4.

I first present per-critical section results from the STAMP benchmarks in Figure 5.4. Considering the largest of the read or write sets (indicated by read or write locations), in decreasing order we have labyrinth (31427), bayes (1859), yada (729), genome (502), vacation (203), intruder (95), kmeans (68) and ssca2 (20). Another important point that is brought out in the profile is that the actual number of read or write operations is also of the same or larger order of magnitude and places the STAMP benchmarks in the same relative order: labyrinth (1376788), bayes (747345), yada (7310), genome (4162), vacation (447), intruder (167), kmeans (100), ssca2 (38).

An important observation that can be made here is that the benchmarks reporting poor performance with SLE_x86 are the ones that seem have the largest number of shared memory accesses. I confirmed using TL2 instrumentation that the number of memory accesses indirected into the STM with manual source level instrumentation is far lower.

This anomaly can be explained by considering the way in which STM-related barriers[1] are inserted in STAMP benchmarks. Consider the transaction generating the large number of reads in Labyrinth (file router.c line 396, as identified by the profiler). The manually instrumented

[1] The instrumented read and write accesses when using an STM are often referred to as barriers.

Source	CS frac	Wait frac	avg_q_len	Read Ops	Read Locs	Write Ops	Write Locs
bayes							
learner.c:1189	0.000	0.086	4.709	21.548	8.857	4.905	3.929
learner.c:1202	0.000	0.043	4.797	336.525	79.300	74.275	22.525
learner.c:1414	0.051	0.091	4.770	747345.295	1859.068	97683.182	70.727
learner.c:1385	0.000	0.006	4.837	3.000	3.000	1.000	1.000
learner.c:1425	0.014	0.000	4.764	30144.953	119.116	3933.163	24.419
learner.c:1267	0.002	0.043	5.030	19121.933	178.267	2496.200	32.133
learner.c:1326	0.000	0.090	4.822	8825.474	150.737	1147.632	31.263
learner.c:1296	0.000	0.000	3.667	575.750	52.750	75.000	13.000
learner.c:1437	0.022	0.003	4.759	46043.571	272.524	6032.143	46.833
learner.c:1348	0.000	0.077	4.739	2141.200	75.600	278.650	17.950
learner.c:1317	0.000	0.000	3.667	2.000	2.000	1.000	1.000
learner.c:1288	0.000	0.000	5.091	2.000	2.000	1.000	1.000
learner.c:1451	0.000	0.000	4.907	41.447	25.026	4.000	4.000
genome							
sequencer.c:395	0.010	0.171	6.767	81.706	56.289	4.000	4.000
sequencer.c:290	0.068	0.706	6.821	4162.933	502.357	0.003	0.003
sequencer.c:369	0.000	0.003	6.743	2.003	2.003	1.000	1.000
sequencer.c:408	0.000	0.003	6.768	8.140	7.378	4.000	4.000
sequencer.c:476	0.000	0.002	5.343	71.096	21.784	3.667	3.667
intruder							
intruder.c:199	0.003	0.321	6.936	7.000	7.000	1.000	1.000
intruder.c:210	0.056	0.328	6.933	167.043	95.856	9.441	5.836
intruder.c:226	0.002	0.274	6.952	5.153	5.153	1.038	1.038
kmeans							
normal.c:168	0.020	0.647	5.669	100.000	68.999	33.000	33.000
normal.c:182	0.002	0.177	5.728	2.000	2.000	1.000	1.000
normal.c:190	0.000	0.000	3.627	2.000	2.000	1.000	1.000
labyrinth							
router.c:396	0.371	0.629	1.746	1376788.750	31427.667	172873.333	28511.500
router.c:379	0.000	0.000	1.748	6.667	5.067	0.533	0.533
ssca2							
computeGraph.c:475	0.007	0.956	6.770	4.000	4.000	2.000	2.000
vacation							
client.c:247	0.001	0.008	6.870	447.585	203.204	20.510	14.076
client.c:267	0.000	0.009	6.868	126.768	74.950	4.363	4.307
client.c:196	0.080	0.828	6.871	447.412	127.811	12.094	11.601
yada							
yada.c:207	0.002	0.236	0.119	16.312	11.922	2.468	2.234
yada.c:215	0.000	0.212	0.119	2.000	2.000	0.000	0.000
yada.c:228	0.266	0.206	0.126	7310.245	729.703	1064.870	481.507
yada.c:246	0.001	0.046	0.127	11.827	8.874	1.701	1.283
yada.c:233	0.000	0.029	0.127	2.000	2.000	1.000	1.000

Figure 5.4: STAMP critical section memory operations

```
TM_BEGIN();
    /* ok if not most up-to-date */
    grid_copy(myGridPtr, gridPtr);
    if (PdoExpansion(routerPtr, myGridPtr,
                     myExpansionQueuePtr,
                     srcPtr, dstPtr)) {
        pointVectorPtr = PdoTraceback(gridPtr, myGridPtr,
                                      dstPtr, bendCost);
        ...
        if (pointVectorPtr) {
            TMGRID_ADDPATH(gridPtr, pointVectorPtr);
            TM_LOCAL_WRITE(success, TRUE);
        }
    }
TM_END();
```

Figure 5.5: A fragment of code from the labyrinth benchmark

code for that transaction is shown in Figure 5.5. The transaction in question is delimited by the calls TM_BEGIN and TM_END. As the TM prefixes and lack of them suggest, only the functions TMGRID_ADDPATH and TM_LOCAL_WRITE are instrumented with STM barriers while the grid_copy and PdoExpansion functions do not. The grid_copy function accesses the global shared grid (containing the maze to be routed through) contained in the gridPtr array. The grid_copy function in fact, copies the whole array into a private copy accessed by PdoExpansion and thus generates the large number of read and write accesses (since the array is large, sized at 512x512x7 elements). None of this is known to the instrumentation infrastructure, which generates instrumentation for all the accesses, including those to update the private copy.

This is a case where the programmer has used their knowledge of the program to appropriately remove unnecessary barriers. This leads to a race condition in the case of access to the global grid in grid_copy. As the comment in the code indicates, not receiving the most up-to-date copy is not a problem from the perspective of correctness. A similar problem occurs with Genome and Yada. This naturally leads to large gaps in performance when using automatic instrumentation. In Chapter 6 I discuss a technique that uses existing x86 memory management hardware (the paging unit) to automatically identify thread private regions in order to reduce this gap.

Finally, I classify the STAMP benchmarks from the perspective of potential benefits from the use of transactional memory in Figure 5.6, which is a plot of the overall dependence density for each benchmark. The lower right portion of the graph is the TM friendly region, which includes almost all the benchmarks. This is not surprising given than STAMP is written with transactional memory in mind. Yada shows the lowest disjoint access parallelism (and hence

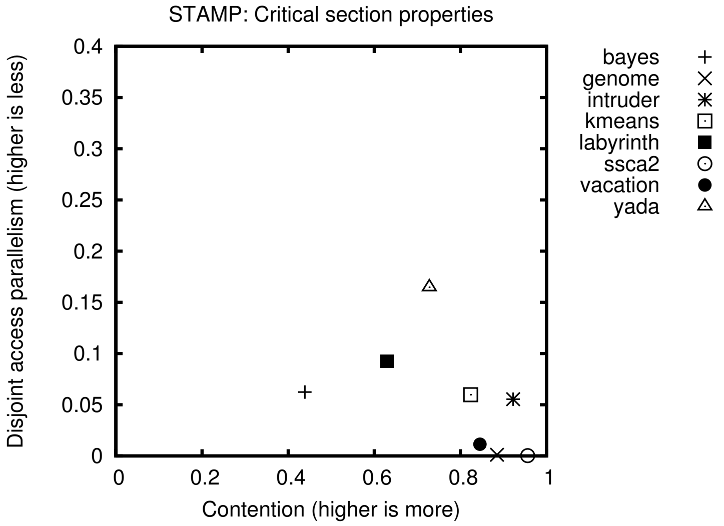


Figure 5.6: STAMP critical section characteristics

the highest conflict rate as has been pointed out previously). Bayes has the lowest contention and hence should show the least improvement over the lock-based version, a fact that is borne out by the experiments in the previous chapters.

The plot in Figure 5.6 is useful to demonstrate the performance spectrum of x86 binaries with respect to ideal transactional memory, such as that promised by hardware TM. Performance with software transactional memory such as in this dissertation also needs consideration of the overheads associated with software transactional memory. A sweet spot for an STM benchmark is one that has sufficient disjoint access parallelism, high contention for the critical section and a sufficient number of accesses within the critical section to offset setup overheads. Vacation falls within such a sweet spot and thus benefits more from using an STM compared to SSCA2, which has the smallest critical sections among the STAMP benchmarks and is dominated by transaction setup overheads. On the other hand Kmeans and Intruder have roughly similar read and write set sizes and thus as predicted by Figure 5.6, Intruder outperforms Kmeans when using an STM, since SLE_x86 manages to beat the performance of the lock-based version in the case of the former but not the latter.

5.8 Discussion

This chapter deals with the design and construction of a profiler for critical sections in x86 binaries. It measures lock contention and memory access related metrics of critical sections such as the number of locations read and written. It also incorporates a metric determining the amount of disjoint-access parallelism between critical sections. Transactional memory related measurements are made without actually using it, which means that programs can be profiled for TM suitability without rewriting them to use TM first. Furthermore, unlike other tools [PHW07] that measure similar TM-related metrics for lock-based programs using simulators, this tool runs programs at full speed on native hardware, thus making it practical for real-world programs (such as the Quake benchmark I study in Chapter 7).

The profiler is useful to explain the performance with SLE seen thus far in the dissertation. It is also decoupled from any specific TM implementation and thus is a useful performance profiling tool for *any* transactional memory implementation. It is difficult to make specific speedup predictions for a particular transactional memory system (that would require a detailed model of the effects of an HTM or STM or both for a hybrid TM). Nevertheless, even in its current form I believe that the profiler is useful for performance debugging when using transactional memory.

The most immediate use of the profiler has been to identify deficiencies in automatic instrumentation vis a vis instrumentation inserted and optimised by a programmer with knowledge of the benchmark. In the next chapter I discuss a set of techniques by which this gap can be closed.

Chapter 6

Thread-private data tags

The profiler described in the previous chapter identified a deficiency in the automatic instrumentation system: the inability to distinguish data that is private to a thread from that which is not. Indiscriminate instrumentation can slow down the STM substantially or even overflow its internal logs. Unlike STMs applied at a source level there is not enough information available through static analysis to eliminate some of these overheads. In this chapter I present a dynamic technique that at runtime can *safely* eliminate much of the STM overhead for such thread-private data.

This chapter is organised into two parts. The first part deals with a set of generic modifications to the algorithms of Chapter 3. These modifications support the dynamic assignment of tags to memory locations that indicate whether the location is thread-private. Threads avoid unnecessary STM overheads for accesses to such locations.

The second part of the chapter discusses a practical implementation of tagging. The implementation focuses on three key applications. The first is heap data that is known to be thread-private. An example of this is OpenMP thread-private data that is allocated on the heap. The second application is locations on the stack, that are *usually* thread-private, but might occasionally be shared between threads. The third application is heap data that *may* be thread-private, as exhibited by some of the STAMP benchmarks. I discuss a dynamic means for detecting such thread-private data at runtime.

6.1 Generic capabilities

In this section I discuss modifications to the logging algorithms of Chapter 3. There are four location tags (the state of a location is its tag):

1. **Private(t)**: The location is private to thread t
2. **SharedRO**: The location is shared read-only
3. **SharedRW**: The location is shared read-write
4. **Unavailable**: The location in currently inaccessible pending a change of state

This section assumes that outside a transaction (critical section in lock-based programs):

- Locations marked `Private(t)` are not accessed by any thread other than `t`
- Locations marked `SharedRO` are not written to by any thread

The tag `SharedRW` is the default tag and is the only one that the logging algorithms in Chapter 3 support. I use the notation `Tag(loc)` to identify the tag for a location `loc`.

Another addition to the STM is the `UndoLog` maintained per-thread. This is an ordered list of (location, value) pairs. It is used to record the old values of thread-private locations before they are overwritten in order that they may be restored on a transaction abort. The enhanced logging algorithms are Algorithms 18, 19 and 20.

The enhancements change the behaviour of the logging algorithm based on the tag. The enhanced algorithms can write to locations directly if owned by the thread and read from locations directly if either owned by the thread or shared in read-only mode. If an access is not possible the thread changes the state of the location to permit access (this involves first aborting the currently running transaction and hence the call does not return). Note that on encountering an unavailable location, the transaction is aborted. The abort algorithm is enhanced to undo the effects of updates to thread-private memory.

The most important new algorithm from the perspective of supporting thread-private data is Algorithm 22, for changing the tag of a location. It first marks the location as unavailable. This ensures that any transaction that attempts to access the location is aborted. Next, it blacklists a “dummy” lock using Algorithm 7 in Chapter 3. This ensures that all executing threads that might have a reference to the location in their STM logs have finished (and thus no longer use the old tag for the location). Finally it sets the new tag for the location. It is important to note that the algorithms enforce a monotonic increase in sharing for the location across the range: [`Private(t)`, `SharedRO`, `SharedRW`]. To enforce this in the case of a `ChangeTag` to `SharedRO` racing with a `ChangeTag` to `SharedRW` for the same location, a check is made (line 9 of `ChangeTag`) to detect the case where the location already permits read-write access.

Algorithm 18 SpeculationBegin(t)

```
Epoch(t) := Epoch(t) + 1
Memory Fence
SnapshotSeqNo(t) := Stable
Initialise WriteLog(t) to empty
Initialise ReadLog(t) to empty
Initialise DirtyList(t) to empty
Initialise UndoLog(t) to empty
```

Algorithm 19 SpeculativeWrite(t, loc, Value)

```
if Tag(loc) = Unavailable then
    abort
if Tag(loc) = SharedRW then
    DirtyList(t) = DirtyList(t) $\cup$ Metadata(loc)
    Append (loc, Value) to WriteLog(t)
else if Tag(loc) = Private(t) then
    oldvalue := contents of memory at loc
    Prepend (loc, oldvalue) to UndoLog(t)
    set contents of loc := Value
else
    ChangeTag(t, loc, SharedRW)
```

Algorithm 20 SpeculativeRead(t, loc)

```
if Tag(loc) = Unavailable then
    abort
if Tag(loc) = SharedRW then
    if $\exists$ (loc, value) $\in$ WriteLog(t) then
        result := most recent write to loc in WriteLog(t)
    else
        result := contents of memory at loc
        if Metadata(loc) is odd then
            abort
        if Metadata(loc) $>$ Snapshot(t) then
            abort
        Append (loc, result) to ReadLog(t)
else if Tag(loc) = Private(t) or Tag(loc) = SharedRO then
    result := contents of memory at loc
else
    ChangeTag(t, loc, SharedRO)
return result
```

Algorithm 21 Abort(t)

```
for all (loc, Value) $\in$ UndoLog(t) (in order) do
    set contents of loc := Value
Epoch(t) := Epoch(t) + 1
```

Algorithm 22 ChangeTag(t, loc, newtag)

```
if Epoch(t) is odd then
    Abort(t) // Configured to return here
repeat
    repeat
        OldTag := Tag(loc)
    until OldTag $\neq$ Unavailable
until x86CAS(Tag(loc), OldTag, Unavailable) = OldTag
Blacklist(t, DummyLock)
if OldTag = SharedRW then
    NewTag = SharedRW
x86CAS(Tag(loc), Unavailable, NewTag)
```

6.2 Associating tags with locations

The foundation for any practical implementation of these capabilities is a way to associate tags with locations. There are various design choices available.

For example, one could follow the same technique as used for assigning metadata to locations: a hash function into a tag table. The problem with such an approach is that the many-to-one mapping would cause location tagging to quickly drop to the lowest common denominator: SharedRW, which is the dominant tag for the benchmarks I studied.

Using a one-to-one mapping at a low granularity like several bytes can lead to an overwhelming amount of tag metadata. Instead I made the observation that in most cases thread-private data either already has spatial locality (such as thread stacks) or can be forced to be so by using special memory allocations for heap data (as I show later). Hence I chose to associate tags with an entire page at a time ($/$ is used to represent 'C-style' integer division):

$$\mathrm{Tag}(\mathrm{loc}) = \mathrm{TagTable}[\mathrm{loc}/\mathrm{SystemPageSize}]$$

The benchmarks in this dissertation were done on a system with a 4KB page size. The mapping to tag metadata is shown in Figure 6.1. Rather than a single tag word, the entry in the TagTable itself points to a tag metadata structure, one of whose fields is the tag for the page. The remaining fields contain other useful information that I introduce later in this chapter. The tag metadata structure itself is 40 bytes long in the implementation and thus the space overhead of tagging (including the pointer) is 44 bytes per *physical* page used in the program which is slightly under 1.08% overhead. In the implementation, tag metadata is usually shared by multiple entries in the TagTable and thus that estimate is the worst case overhead. Finally, the pointer is set to NULL if the page has tag SharedRW, this imposes minimum indirection overhead for the common case.

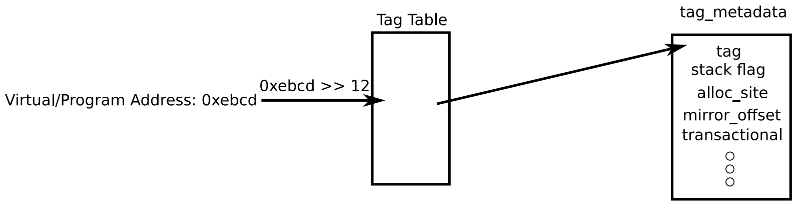


Figure 6.1: Association of tag metadata with pages

```
#pragma omp threadprivate(host_frametime)
...
#pragma omp parallel shared(realtime ..)\
 copyin(host_frametime ..)
-----------------------------
80693df:      push   $0x80bc6c8
80693e4:      pushl  0x8098ff8
80693ea:      push   $0x80bf010
80693ef:      push   %ebx
80693f0:      push   $0x8098f9c
80693f5:      call   8049748 <__kmpc_threadprivate_cached@plt>
```

Figure 6.2: Example of OpenMP ThreadPrivate from Quake

6.3 Applications

I now discuss how extensions to the tagging infrastructure were used to solve three thread-private data related problems in this dissertation. I extend the tagging infrastructure accordingly. The first is the simplest: certain allocations from the heap can be statically identified to contain thread-private data (Section 6.3.1). The next is slightly more complicated: locations on the stack are assumed to be thread-private and this is the basis for not instrumenting accesses made through the stack pointer. However I would like to try and detect the case where they are not and execute the program correctly (Section 6.3.2). The third (discussed separately in Section 6.4) is the most complicated and ambitious: to try to automatically replicate the optimisations done manually by the programmer in the STAMP benchmarks.

6.3.1 OpenMP thread private data

The OpenMP specification allows the declaration of thread private data, an individual copy of which is created for each OpenMP thread. If so specified, each thread's copy is initialised from the master copy every time an OpenMP parallel region is entered. An example from the Quake benchmark that I examine in Chapter 7 is given in Figure 6.2. The `host_frametime` variable is declared as thread-private. It is initialised at the beginning of an OpenMP parallel region through a compiler pragma that ensures every spawned thread receives a private copy initialised from the master one. In the x86 binary, every reference to host_frametime is indirected through __kmpc_threadprivate_cached. One of the key tasks performed by this function is to allocate a copy for the calling thread (whose number in the example is held in the ebx register before being passed to the function as a stack parameter). In an interception wrapper over this function I ensure, that during passive execution, allocations are done out of specially allocated pages from the heap whose tag has been set to Private(t) if the calling thread is t. The OpenMP specification forbids sharing between threads and the only accesses to these locations outside transactions can be from the owning thread.

The effect of tagging OpenMP thread-private data in this manner is discussed in detail in the benchmark results for Quake in Chapter 7. In summary, however, it leads to a performance improvement of around 40%.

6.3.2 Stack data

The instrumentation system of Chapter 4 makes the assumption that accesses through the stack pointer to locations on the stack are thread-private and hence does not instrument them. The thread-stack is treated as thread-private and Section 4.6.2 provided a brief description of how accesses to the stack are specially handled. In this section I provide a more detailed description in terms of extensions to the generic tagging algorithms.

First, stack pages[1] are tagged as thread private: Private(t). This is done by checking the value of the stack pointer at thread creation (this indicates the base of the stack), at transaction begin time and at any instruction within a critical section that changes the value of the stack pointer. These checks identify new stack pages as the stack grows and tags them appropriately. I also use an additional flag in the tag metadata structure to indicate that this page belongs to a thread stack.

Next, Algorithm 19 is modified to take into account that the stack is already checkpointed. It does so by checking for the stack flag in the tag metadata structure. If set, it only undo-logs updates if the location updated is *above* (in address terms) the checkpointed region of stack (recall that the region below is dead, as discussed in Section 4.6.2). Finally, some calls to SpeculativeRead and SpeculativeWrite are statically filtered out by the instrumentation system if they use the stack pointer for address generation.

I now turn my attention to what happens when a thread accesses another's stack. Imagine that a thread accesses another thread's stack in a transaction. Clearly this cannot happen through the stack pointer on that thread and hence the access must be instrumented and call one of SpeculativeRead or SpeculativeWrite, which finally calls ChangeTag. The ChangeTag algorithm is modified to check whether the location whose tag is being changed

[1] The stack is page aligned.

lies on a stack page (through the tag metadata structure). If so, it *blacklists* the lock that has been speculated past in the transaction that has generated this access.

Finally, I turn my attention to a basic assumption in the generic tagging algorithms: for `Private(t)` tagged locations, no thread other than `t` should access it outside a transaction (critical section). Unfortunately, there is no efficient way to implement this without instrumenting all memory accesses, including those outside a critical section. This would break one of the basic principles of the instrumentation system: no overhead outside critical sections.

The leads to the fourth and final restriction on programs used with SLE:

Restriction 4: A location on a thread stack that is shared between threads can only be accessed in critical sections protected by the same lock or always outside any critical section.

Any un-handled sharing would thus be detected and lead to the blacklisting of the lock. This safely handles all accesses including those which are not instrumented by virtue of being made through the stack pointer. Unfortunately there is no easy way to detect programs that violate the restriction on sharing stack locations (although creative tools could be constructed). It is interesting to note that STM compilers would also need to add a similar restriction to prevent threads sharing (uninstrumented) locations on stack, thereby leaking evidence of transactional behaviour (such as speculative writes that are rolled back).

One reason why this restriction is not too onerous is that sharing locations on stack is rare behaviour. Even in the case where they are shared, they would usually be synchronised through locks (as required by the restriction). The only example that I have encountered of threads sharing stacks in this dissertation is with the Bodytrack benchmark in the PARSEC suite (Chapter 7). Code fragments from that benchmark are shown in Figure 6.3 illustrating how sharing of locations on the thread stack happens. The `ParticleFilterPthread` object is instantiated in the stack frame of function `mainPthreads`. Methods of that object are executed concurrently by worker threads, all of which access the `TicketDispenser` attribute, which is itself an object whose attributes are also on the stack frame for function `mainPthreads`. However all `TicketDispenser` methods such as the `getTicket` method shown in the example are properly synchronised using the same lock and thus satisfy the conditions of Restriction 4.

6.4 Adaptive tagging for STAMP

In this section, I discuss how tagging can be applied to allocations on the heap. Unlike the case of thread-private OpenMP data I deal with a more complex problem here: there is no *a priori* static way to classify an allocation as thread private. Instead I discuss how an adaptive algorithm can automatically classify locations on the heap as thread private at runtime. The motivation for this set of extensions to the tagging algorithms is the STAMP benchmarks. As I have shown in the previous chapter, there is a lot of potential in the instrumentation infrastructure to improve

```
int mainPthreads(...)
{
   ...
   ParticleFilterPthread<TrackingModel> pf(workers);
   ...
   // Create threads and pass pointer to pf
   ...
   for(int i = 0; i < frames; i++){
       ...
       pf.Update((float)i);
       pf.Estimate(estimate);
   }
   ...
}


//Generic particle filter class templated on model object
template<class T>
class ParticleFilterPthread ... {
   ...
   threads::TicketDispenser<int> particleTickets;
   ...
};

//get a ticket and increment counter (in that order)
template <typename T>
T TicketDispenser<T>::getTicket() {
   T rv;

   l->Lock();
       rv = value;
       value += inc;
   l->Unlock();

   return rv;
}
```

Figure 6.3: Sharing stack locations in the Bodytrack benchmark from PARSEC

performance by eliminating STM related overheads for such thread-private data. One example has already been provided in the previous chapter, where a large chunk of memory is updated exclusively by one thread. Another interesting example is illustrated in Figure 6.4 from the Genome benchmark.

This shows a fragment of code from Genome. The code has been instrumented for use with TL2 and consists of a loop that iterates over a large string and inserts hashes of substrings into a shared hash table, using the `TMHASHTABLE_INSERT` call. The details of the hash

```
TM_BEGIN(); { // stamp/genome/sequencer.c:290
long ii_stop = MIN(i_stop, (i+CHUNK_STEP1));
for (long ii = i; ii < ii_stop; ii++) {
 void* segment = vector_at(segmentsContentsPtr, ii);
 TMHASHTABLE_INSERT(uniqueSegmentsPtr, segment,
                    segment);
} /* ii */
} TM_END();

ulong_t hash_sdbm (char* str) {
 ulong_t hash = 0;
 ulong_t c;
 while ((c = *str++) != ’\0’) {
   hash = c + (hash << 6) + (hash << 16) - hash;
 }
 return hash;
}
```

Figure 6.4: A fragment of code from the Genome benchmark

implementation are unimportant but it essentially involves iterating over the substring (acting as a key) to compute a hash function, shown as the function `hash_sdbm` in the example. The hash function accesses the substring using *uninstrumented* accesses. This reflects the knowledge of the programmer that the string iterated over by any hashing calls is immutable.

6.4.1 Tag metadata and allocation sites

The general idea is to use an adaptive runtime technique. In addition to detecting when locations become shared, we need some way to efficiently remember and propagate this information. A key insight that helps make adaptive tagging efficient is that this history can be maintained on a per-allocation site basis. An allocation site is the instruction pointer value at the call to the memory allocator for allocating data from the heap. Allocation sites for thread-private data are normally distinct from those for shared data and this is certainly true for the STAMP benchmarks. Other researchers have used similar techniques for separating allocations [Akr10], albeit for different purposes.

Adaptive tagging intercepts all memory allocation calls and handles them using a custom memory allocator. The memory allocator uses a different pool of pages for each allocation site. Adaptive tagging thus extends the tagging metadata as shown in Figure 6.5. That figure extends the previous example by showing the originating allocation site to whose pool the page belongs. Memory pools in the implementation are extended in ‘chunks’ of pages. All pages in a chunk share the same tagging metadata structure to further reduce space usage and enable quick switching of state for whole chunks of pages at a time. Once sharing is detected the entire

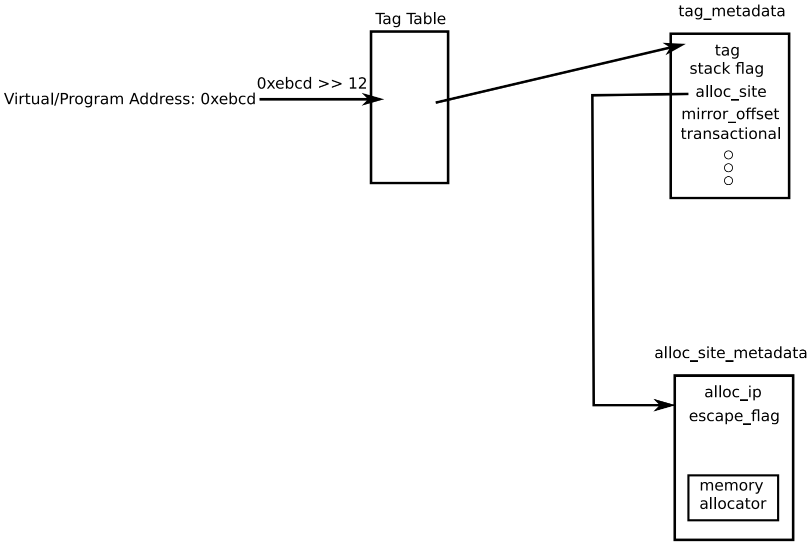


Figure 6.5: Extended tag metadata for adaptivity

chunk moves over to the new state thereby minimising calls to change tags. If memory is freed from any of these pages it returns to the originating allocation site.

The allocation site includes metadata used by the memory allocator (my design uses a memory allocator with out-of-band metadata) and linked lists of available chunks. There are also additional fields in the tag and allocation site metadata used by adaptive tagging that I describe shortly.

6.4.2 The non-transactional access problem

The generic location tagging algorithms make the assumption that in the `Private(t)` (`SharedRO`) state no non-transactional access (write) is made from a thread other than the owning thread `t` (any thread). For the case of stack pages I sidestep this problem by explicitly forbidding such accesses. This is reasonable since such stacks are rarely shared. Locations on the heap however are commonly shared across threads and can be accessed non-transactionally (the privatisation example in Chapter 3 illustrates such a construct). I therefore need to provide an explicit solution to this problem. There are two aspects to this: detecting such accesses from other threads and triggering tag changes on such an access to ensure that it can be safely executed. My solution uses the existing page-level memory protection that is provided by the hardware paging unit on the x86.

The `SharedRO` state

In the `SharedRO` state the only accesses allowed to the location, transactional or otherwise are reads. I thus set the protection for all pages in the chunk (that share the tag state) to read-only.

This is easily accomplished through the `mprotect` call.

The `Private(t)` state

This is considerably more complex. Any other thread is forbidden from making *any* access. However the owning thread can legitimately read or write the location. Unfortunately the implementation of threading using a shared address space on Unix (and most operating systems) means that different access permissions cannot be provided to the same page for different threads.

The first step towards a solution is to split up the state into two sub-states based on the value of the `transactional` flag (Figure 6.5). If `true` the state can only be accessed in a transaction by the owning thread. If `false` the state can be accessed outside a transaction by the owning thread.

The `Private(t), transactional=false` state is handled by simply setting the page access permissions to the most permissive, allowing all access by all threads. This might seem counterintuitive but results in the system being unable to track sharing outside transactions. However this is perfectly *safe* since SLE is built on top of a weakly atomic STM and does not care about simultaneous accesses to the same location if both are made outside transactions.

The `Private(t), transactional=true` state is handled by simply setting the page access permissions to the least permissive: no access at all. This ensures that no other thread can make a non-transactional access to the location satisfying the safety requirement of the tagging-capable read and write algorithms. This however leads to a different problem, how do `SpeculativeRead` and `SpeculativeWrite` themselves access the location if made from the owning thread `t` ?

To solve this problem I use a technique that has been used elsewhere to implement strong atomicity [AHM09]: multiple mappings for the same page. Figure 6.6 shows how this is accomplished. The figure takes the example of a page with virtual address (`0xe000`) in state `Private(t), transactional=true`. It is mapped into the virtual address space twice, once at virtual address (`0xe000`) and once more at virtual address (`0xe000 + mirror_offset`). Both the mappings in the page table point to the same physical page (physical address `0xa000`). Crucially, the second mapping has liberal access permissions and is used by `SpeculativeRead` and `SpeculativeWrite`. The double mapping is achieved by using `mmap` twice from the same backing file on disk (ramdisk in experiments for this dissertation in order to reduce overhead).

Finally there are (not infrequent) examples where thread-private objects are created and frequently accessed both within and outside transactions. The corresponding movement between states leads to a large number of system calls to change page protections and becomes a severe

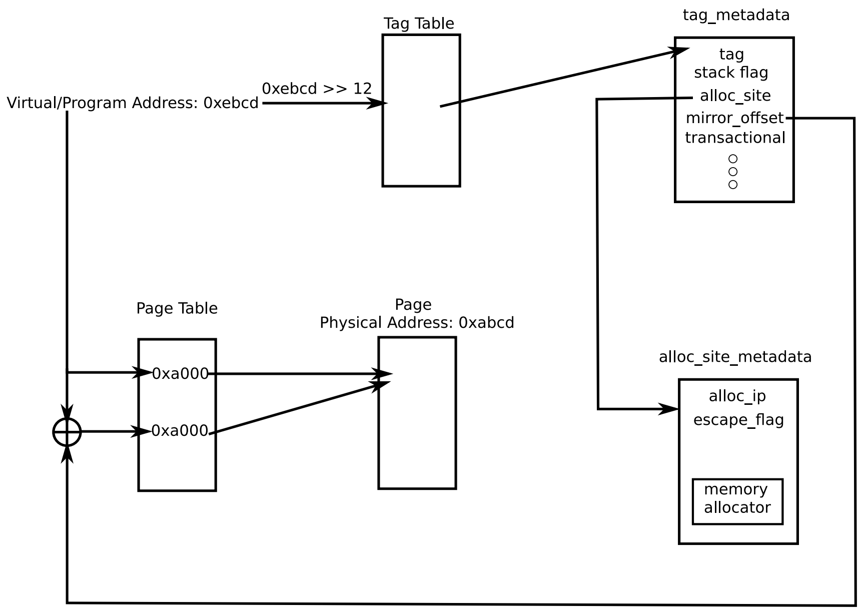


Figure 6.6: Extended tag metadata with mirror mappings

bottleneck to performance. To avoid this, I place a threshold of 10 state changes before such locations become irrevocably tagged as `SharedRW`.

ChangeTag modifications

There are two modifications that need to be made to ChangeTag in order to support these enhancements. The first is the capability to change memory protection accordingly. The second enhancement is the capability to handle memory protection faults that arise due to accesses that are not permitted from outside transactions. These are handled by hooking the fault (Unix SIGSEGV) handler, performing the appropriate state change (and memory protection changes), and restarting the faulting instruction.

Another important task done by ChangeTag is to release the virtual address space occupied by the mirror map of pages when the originating "chunk" moves to a tag other than `Private(t)`. This preserves virtual address space on 32-bit machines (where it is relatively scarce), and is particularly helpful for some of the STAMP benchmarks which can use as much as 1.5 GB of physical memory.

6.4.3 Reducing undo-logging overheads

Within transactions, stores form a minority of accesses compared to loads. Nevertheless undo logging for stores can be a significant source of overhead and a simple observation can serve to eliminate much of this. Dragojevic et al. observed that many transactions in STAMP perform a significant fraction of their accesses to captured memory [DNAT09]: memory allocated and

```
patch: jmp <next_instruction>
<Inlined tagging check>
<If success jump to access>
movl $patch, %eax
call fixup
<logging call>
access:
```

Figure 6.7: Inlined tag check

released in the same transaction. Such accesses need not be undo-logged since their initial contents are irrelevant to execution. To take advantage of this I track whether a chunk escapes from a transaction: (i.e. allocated within it but not released by the end) in the `escape_flag` field of the allocation site metadata. If the `escape` flag is not set then `SpeculativeWrite` does not perform any undo-logging for the access.

6.4.4 Inlining checks

Even if a location is identified as thread-private by tagging, I still need to pay the cost of the call into the logging functions. Ideally one would like to inline that check in the PIC to avoid that overhead for thread-private locations. At the same time one would not want to pay the cost of that inline check for locations that are not thread-private, since it is bound to fail and represents an unnecessary overhead.

An important observation here is that most accesses in the PIC are either always to thread-private locations or always to shared locations. I exploit this by enhancing the instrumentation engine to inline the check as shown in the x86 level pseudocode (for simplicity) in Figure 6.7 in the form of "self-modifying" instrumentation. If the inlined tagging check succeeds, then control jumps past the call to the logging function. If the check fails then the fixup procedure changes the initial jump to the next instruction to jump past the inlined tagging checks (reflecting the fact that the location that was accessed is now tagged as shared).

6.5 Evaluation

The evaluation in this chapter focuses on the effectiveness of tagging in reducing instrumentation overhead for the STAMP benchmarks. The evaluation therefore compares SLE_x86 to SLE_x86 with private data tagging on (labeled `SLE_x86 + PDT`). The objective is to close the gap with manual instrumentation and hence I also include the performance of STM_x86 (from Chapter 3) as reference. I use the 48-core system (Appendix B: Tigger) for all the experiments.

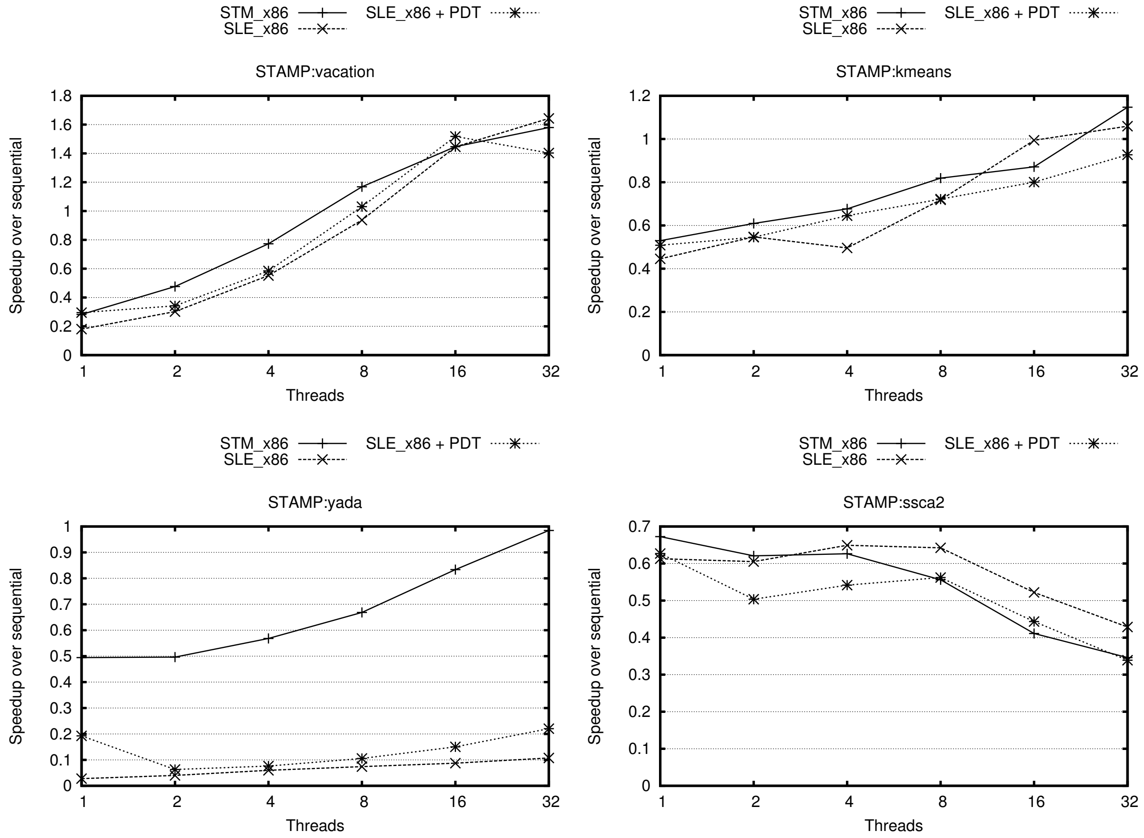


Figure 6.8: SLE (with private data tagging) on the STAMP benchmarks(1)

The results are shown Figure 6.8 and Figure 6.9. For the four benchmarks where automatic instrumentation is close to the performance of the manual one: Vacation, SSCA2, Intruder and Kmeans, private data tagging makes little difference. This is because private data tagging is *adaptive* and does not change the logging behaviour once a location is identified as *shared*.

In the case of Bayes, automatic instrumentation no longer adds an overwhelming amount of instrumentation, meaning that the STM no longer falls back to pessimistic locking. However the cost of checking the data tags still adds a large amount of overhead. In the case of Genome, private data tagging significantly improves performance. For Yada, private data tagging improves performance but only marginally and in the case of Labyrinth execution continues to fall back to the lock due to an overflow of the STM logs.

In the case of Yada, the effectiveness of private data tagging is limited by the organisation of a key data structure shown in Figure 6.10 (top half). In it the `isGarbage` and `isReferenced` fields can be updated after the object is created while the remaining fields are shared read-only. This breaks the tagging algorithms, which are limited to tracking at an allocation granularity and hence can't look inside the allocated object. One solution to this is to break up the object into a read-write and a read-only part as shown in the bottom half of the figure. This leads to a change of only 26 lines in 4600 lines of code. On the other hand it has a dramatic impact on performance, as shown in Figure 6.11. The remaining gap in performance from manual

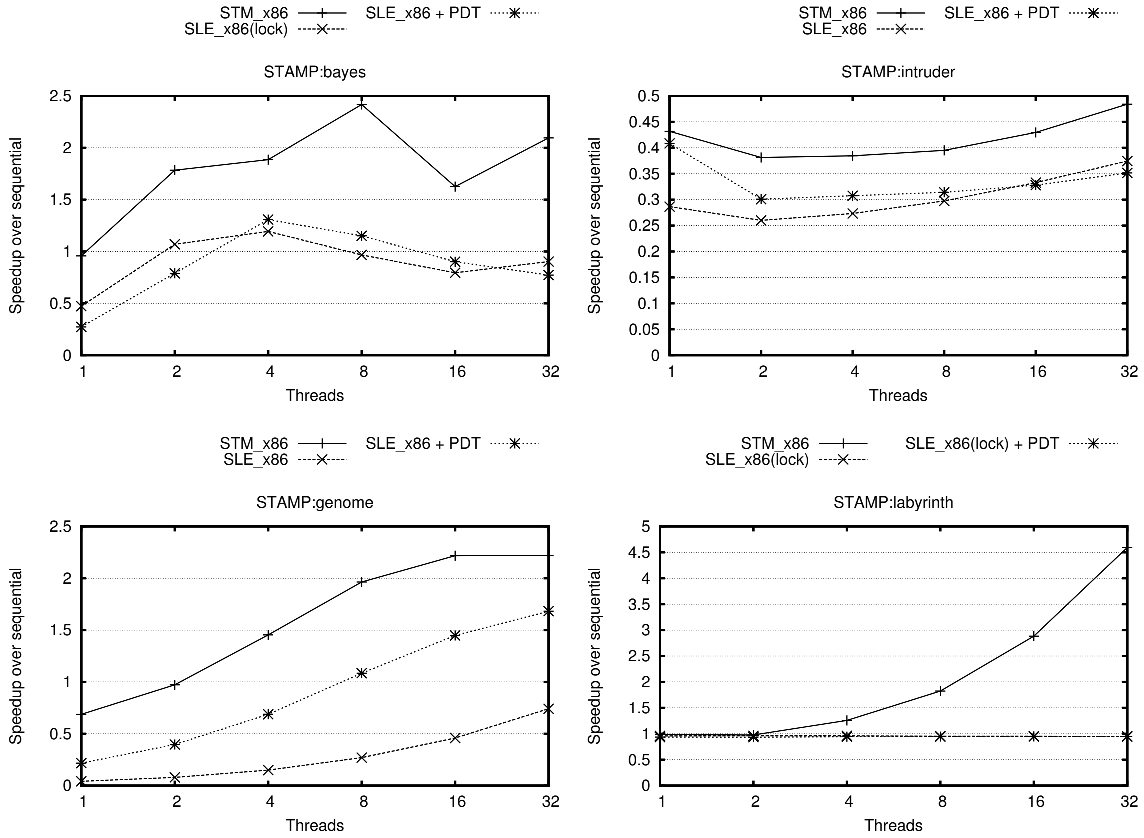


Figure 6.9: SLE (with private data tagging) on the STAMP benchmarks(2)

instrumentation is due to objects that are thread private but are frequently accessed both within and outside transactions. The changes of the `transactional` flag for them hits the imposed rate limit and leads to them being classified as shared.

In the case of the Labyrinth benchmark on the other hand, data tagging cannot by itself guess that a large data structure can be accessed in a racy manner (recall that the design must be *safe* and thus cannot introduce races). This is interesting, since the programmer has identified an algorithm specific opportunity for improving performance with an STM. The design goal of SLE_x86 is transparency. However, this does *not* mean that the programmer cannot communicate such an optimisation to it. To demonstrate this with Labyrinth, I implemented a simple mechanism for the programmer to provide a *hint* to the SLE runtime system that a region of memory should be treated as tagged thread-private for a subset of the execution. Figure 6.12 shows how this is done by adding two lines to the source code of Labyrinth (consisting of a total of 3113 lines). The instrumentation system is configured to look for calls to a special "trapdoor" functions that pass information to the private data tagging system. In the Labyrinth example, the system is first informed that a region of memory should be temporarily tagged `PRIVATE` and later informed that the tag should be withdrawn. This communicates the (racy) optimisation in Labyrinth to the tagging algorithms.

Figure 6.13 shows the results of this source code annotation. Execution no longer falls back to

```
struct element {
    coordinate_t coordinates[3];
    long numCoordinate;
    coordinate_t circumCenter;
    double circumRadius;
    double minAngle;
    edge_t edges[3];
    long numEdge;
    coordinate_t midpoints[3]; /* midpoint of each edge */
    double radii[3];           /* half of edge length */
    edge_t* encroachedEdgePtr; /* opposite obtuse angle */
    bool_t isSkinny;
    list_t* neighborListPtr;
    bool_t isGarbage;
    bool_t isReferenced;
};

------------------------------------------------------
typedef struct {
    bool_t isGarbage;
    bool_t isReferenced;
} shared_part;

struct element {
    coordinate_t coordinates[3];
    long numCoordinate;
    coordinate_t circumCenter;
    double circumRadius;
    double minAngle;
    edge_t edges[3];
    long numEdge;
    coordinate_t midpoints[3]; /* midpoint of each edge */
    double radii[3];           /* half of edge length */
    edge_t* encroachedEdgePtr; /* opposite obtuse angle */
    bool_t isSkinny;
    list_t* neighborListPtr;
    shared_part *shared;
};
```

Figure 6.10: Yada data structure: original (top) and separated (bottom)

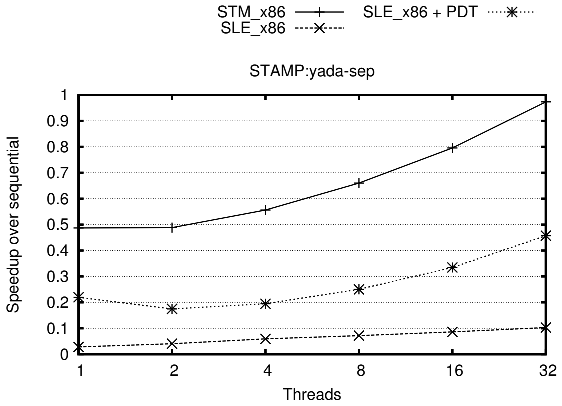


Figure 6.11: Performance of Yada after data structure decomposition

```
TM_BEGIN();
    /* ok if not most up-to-date */
    grid_copy(myGridPtr, gridPtr);

---------------------------------------------
```

```
void dbr_trapdoor(unsigned long a,
                  unsigned long b,
                  unsigned long c)
{
    /* nothing .. will be intercepted
     * by instrumentation system
     */
}
```

```
TM_BEGIN()
    /* mark direct access to shared memory */
    dbr_trapdoor(TAG_PRIVATE, gridPtr->points,
                 myGridPtr->width *
                 myGridPtr->height *
                 myGridPtr->depth);
    /* ok if not most up-to-date */
    grid_copy(myGridPtr, gridPtr);
    dbr_trapdoor(UNTAG_PRIVATE, 0, 0);
```

Figure 6.12: Labyrinth source fragment: original (top) and annotated(bottom)

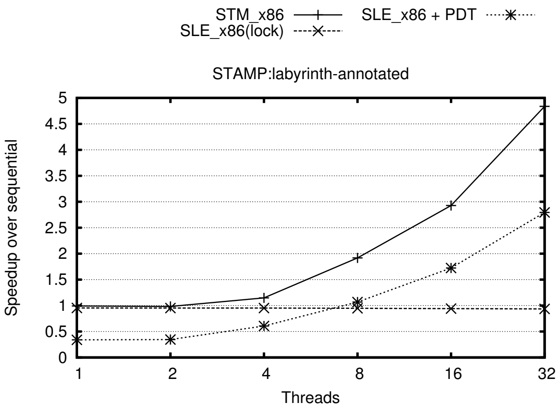


Figure 6.13: Performance of Labyrinth with annotation

	Threads					
Benchmark	1	2	4	8	16	32
labyrinth	NO	NO	NO	NO	NO	NO
labyrinth-hint	NO	NO	NO	_YES*_	_YES*_	_YES*_
yada	NO	NO	NO	NO	NO	NO
yada-decomposed	NO	NO	NO	NO	NO	NO
bayes	NO	NO	NO	NO	NO	NO
vacation	NO	YES	YES	YES*	YES*	YES*
genome	NO	NO	_YES_	_YES*_	YES*	YES*
kmeans	NO	NO	NO	NO	NO	NO
ssca2	NO	NO	YES	YES	NO	NO
intruder	NO	NO	NO	YES	YES	YES

Figure 6.14: Is using Best Of(SLE_x86, SLE_x86 with PDT) better than using the lock ? ('*' means also better than sequential)

pessimistic locking and scales well, capturing the algorithm specific optimisation in SLE_x86.

Finally, private data tagging also improves the absolute performance of SLE_x86 when compared to locking. Figure 6.14 shows the cases where the absolute best performance of SLE_x86 alone and SLE_x86 with private data tagging can outperform the lock. The entries with emphasis show where SLE_x86 with private data tagging outperforms the lock, while SLE_x86 alone does not.

6.6 Discussion

In this chapter I have discussed how memory locations can be tagged as thread-private and how such tagging can be used to safely reduce STM logging overheads. I have discussed three means of automatic tagging: 1) static identification of heap allocations that are thread-private due to the specification of the allocator such as OpenMP thread-private data 2) data on the stack that is usually thread private and 3) adaptively tagging allocations from the heap as thread-private.

These mechanisms include as a subset the thread-private data patterns identified thus far by STM researchers: stack locations [WCW$^+$07] and captured memory [DNAT09]. They also capture various other patterns of thread-private data usage such as the read-only shared string in the Genome benchmark of STAMP. Private data tagging is effective at closing the gap with manual instrumentation. One way to further reduce this gap is static analysis to identify thread-private objects. This is still an evolving field. Usui et al [USB09] for example, chose to expose the manual annotations in the STAMP benchmarks to their compiler, instead of depending on static analysis to identify accesses to thread private data. Nevertheless, when STM compiler designers choose to tackle this problem with static analysis it should be possible to bring some of those techniques over into the SLE system.

The chapters in this dissertation thus far have focused on building up the SLE system as a whole with the STAMP benchmarks on an 48-core machine as the driving motivator. In the next chapter I examine SLE_x86 along three different dimensions: larger systems, comparison to an automatic compiler-based STM and finally larger benchmarks that present interesting issues such as condition variables.

Chapter 7

Applicability

SLE_x86 is a mechanism to apply software transactional memory at the level of abstraction of x86 machine code. Thus far, I have focused on the STAMP benchmarks on a specific system to motivate the construction of STM_x86 and the binary instrumentation system. The objective has been to consider SLE_x86 as an automatic and safe means to apply transactional memory to these programs and evaluate the impact of the stricter x86 memory consistency model and automatic instrumentation.

In this chapter I move out to a more general setting and examine the following scenarios:

1. Scalability: Using SLE_x86 in a machine with a larger number of threads, I examine whether it imposes any hard limits to scalability due to the strictness of the x86 memory model. I also compare with an STM that only supports the much weaker C++ memory model and thereby provides more scalability.

2. Impact on software development: I consider a large program implementing the Quake multiplayer game server. I compare a version with a coarse-grained lock that is elided at runtime using SLE_x86 with a version that uses language level software transactional memory.

3. Condition Variables and Fine-grained locks: I consider a large benchmark suite written and optimised to use locks and condition variables. Condition variables have historically been a source of difficulty for STM designs, which usually forbid them within source-level transactions. SLE_x86 should seamlessly handle condition variables by falling back to pessimistic locking. In addition, I consider the impact of applying SLE_x86 to programs using fine-grained locking, uncovering interesting implications on the use of atomic blocks at a language level.

In this chapter I use SLE configured without support for private data tagging unless otherwise specified. The reason is that none of the benchmarks (other than Quake) perform a significant number of accesses to thread-private locations in their critical sections.

7.1 Scalability

SLE_x86 introduces significant synchronisation between concurrently executing transactions due to the strict nature of the x86 memory consistency model. This section examines the impact of this on scalability.

I use two simple microbenchmarks in this section: Skiplists and AVLTrees. Skiplist and tree benchmarks have long been used in the STM community because they offer enough disjoint-access-parallelism at large data structure sizes to support scaling to large number of accessing threads without any bottleneck in the data structure itself. The microbenchmarks perform a mix of 75% lookups, 12.5% updates and 12.5% deletes to the data structure holding 2^{19} keys (similar to Fraser's dissertation [Fra03]). That there is sufficient disjoint access parallelism here is illustrated by their profiles at 8 threads (generated by the profiler described in Chapter 5), shown in Figure 7.1. Threads spend a majority of their time waiting to enter a critical section with lots of disjoint-access parallelism (little dependence density).

I compare three different alternative implementations of the data structures. The three implementations are identical in terms of the concurrency oblivious algorithms used to implement the data structure. They differ however, in their choice of synchronisation mechanism.

The first implementation uses a simple Pthreads reader-writer lock to protect the tree. The second implementation is identical to the first except that at execution time, SLE_x86 is used the elide the lock. The third version uses transactions instead of a lock to protect the concurrent accesses. I use the Intel 3.0 STM compiler [WCW$^+$07, int] that provides language level atomic blocks and automatic instrumentation of memory accesses. Of the three implementations the lock is expected to be the least scalable. SLE_x86 should provide better performance than the lock. The STM compiler implements software transactional memory for the C++ memory model that attaches no semantics to data races. This admits very efficient STM designs, since properly ordering accesses across transactions is sufficient without needing to worry about interleaving non-transactional accesses that would be races. It is thus expected to provide the best performance. All three versions are compiled using the same (STM capable) compiler to eliminate any differences due to compiler optimisations.

Finally, I use a system supporting a larger number of hardware threads for this section (Appendix B: COSMOS): an SGI Altix with 96 NehalemEX 2.67GHz (Core i7) cores (six cores on a single multicore socket), connected by a cache coherent NUMA interconnect.

Figure 7.2 shows the results of running the benchmark on this system. The vertical axis in the graph has been truncated for better visibility. The implementation with the worst scalability is, as expected, the lock. The implementation with the best scalability is the C++ STM. SLE_x86 lies in between. I report the median of 5 runs and all the three benchmarks show a roughly equal amount of variation with a majority (more than 80%) of points for each benchmark having a variation of under 10% around the median. The variations primarily result from traffic due to

```
CS             cs_frac wait_frac avg_q_length rd_ops  rd_locs wr_ops wr_locs dep_dens
avltree.c:346 0.023   0.576     6.656        37.795  36.933  0.000  0.000   0.000
avltree.c:268 0.005   0.187     6.647        47.843  39.083  3.410  1.765   0.000
avltree.c:209 0.004   0.187     6.661        39.892  37.038  2.855  1.611   0.000
Overall_waiting Overall_dep_density Pred_AvgThreads
0.950407        0.000062            7.997756
-----------------------------------------------------------------------------------------
CS              cs_frac  wait_frac avg_q_length rd_ops  rd_locs  wr_ops  wr_locs dep_dens
skiplist.c:209 0.054    0.529906  6.698        76.961  52.607   0.000   0.000   0.000
skiplist.c:176 0.009    0.190715  6.691        84.459  53.819   41.824  11.418  0.000
skiplist.c:136 0.009    0.186149  6.693        78.840  53.186   41.977  12.479  0.000
Overall_waiting Overall_dep_density Pred_AvgThreads
0.906770        0.000035            7.998725
```

Figure 7.1: Microbenchmark profiles(8 threads)

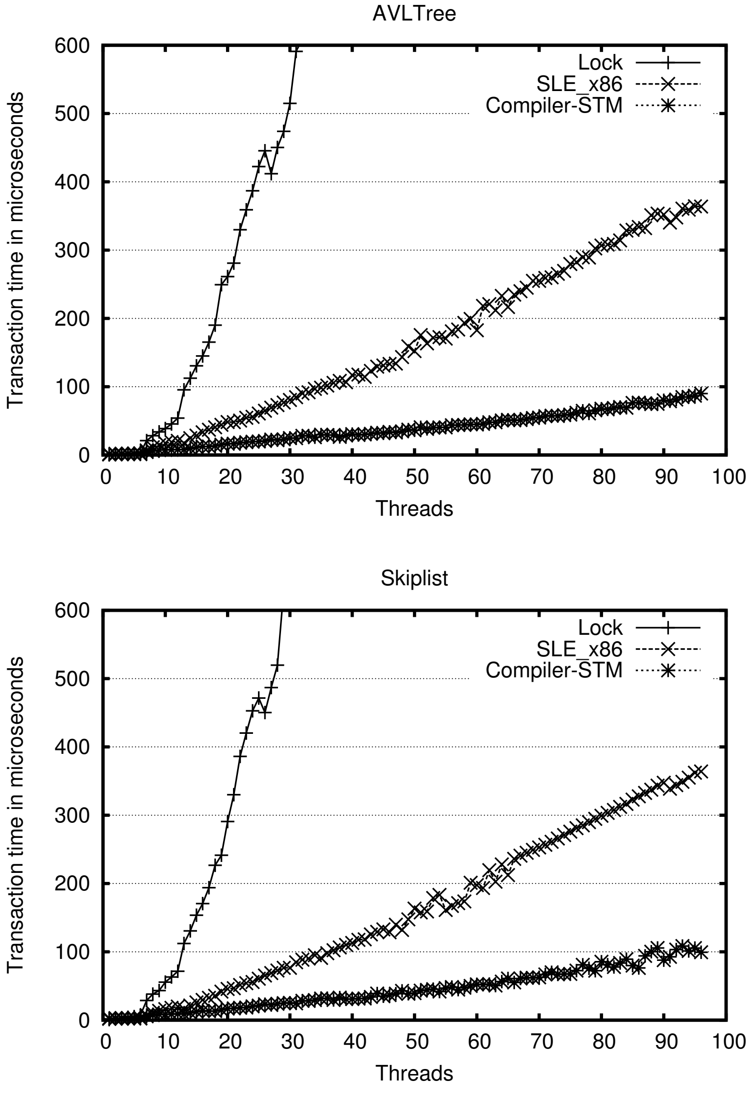


Figure 7.2: Scaling on a Corei7 NUMA system

AVLTree		
Threads	x86 STM	C++ STM
6	1.30	2.44
12	2.60	6.67
18	4.69	14.15
24	6.92	19.83
30	6.39	20.69
36	8.15	28.31
42	9.12	34.53
48	9.69	40.14
54	11.14	47.45
60	12.63	51.72
66	11.27	52.17
72	13.87	62.60
78	14.06	66.75
84	14.32	67.92
90	15.37	67.52
96	16.78	67.82

Skiplist		
Threads	x86 STM	C++ STM
6	2.13	2.93
12	3.60	7.71
18	6.35	16.97
24	8.11	22.42
30	8.40	25.48
36	9.73	29.70
42	11.14	41.45
48	11.67	36.05
54	12.46	54.60
60	14.02	53.00
66	11.46	49.07
72	16.33	60.98
78	17.62	67.28
84	17.96	63.68
90	19.74	78.07
96	20.93	76.65

Figure 7.3: Scalability at CPU socket intervals

other programs running on the (shared batch-processing) system, on cores other than the 96 that I reserved exclusively for this experiment.

A more fine-grained comparison of performance is given in the table of Figure 7.3 that provides the ratio of the time required to run a data structure access using a lock to that using a transaction (language level and SLE_x86) at intervals of 6 threads: representing each addition of a multicore (6 core) socket to the system. At the maximum of 96 threads, SLE_x86 is about 16 to 20 times faster than a lock, while the language level STM is about 4 times faster than SLE_x86.

7.2 Impact on software development

In this section I consider the impact that different approaches to software transactional memory (including SLE_x86) have on software development. There are three ways to go about using transactional memory in a program. The first is to use a library based STM and manually insert instrumentation in source code. The second is to use a compiler-based STM that requires annotation to delimit transactions and mark functions that can be called within a transaction. The third is to write the program using a lock and use SLE_x86 *optionally* at runtime. Developers on large scale software projects are usually concerned with more than just performance. They also care about ease of use, the ability to use stable toolchains and being able to easily debug multithreaded code. Library-based STMs thus are usually a non-option for large programs.

To illustrate these considerations, I use a reasonably large program: a server for the Quake multiplayer game [GZU^{+}09]. This is a multithreaded version of the Quake game server that can

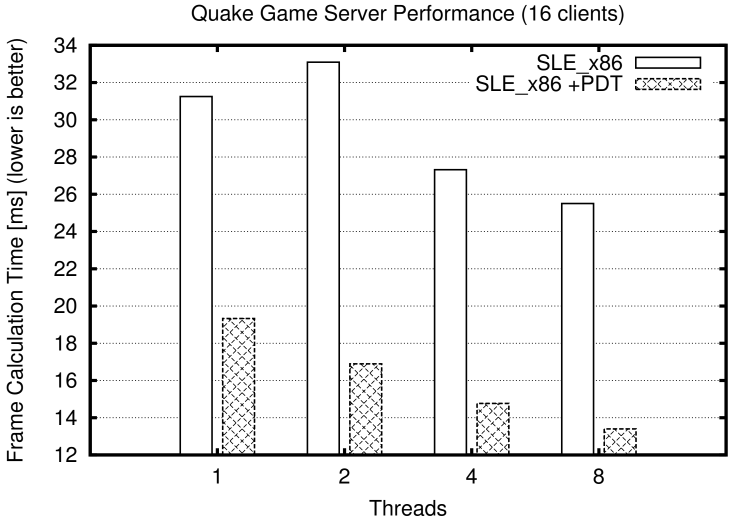


Figure 7.4: Quake using SLE

be reconfigured (through source level macros) to either use a coarse-grained lock; or software transactions through `atomic` blocks. The gameserver spans 49 files containing 27600 lines of C code and is written to use the OpenMP threading model. Both versions are compiled using the Intel 3.0 STM compiler. For repeatability, the server was driven by client traces (simulating gameplay) from another machine. The available version of Quake from the benchmark authors supports a maximum of 8 threads and 16 clients. I use results at the maximum load of 16 clients, where neither the client nor the network are the bottleneck. A portion of the server code that processes client requests and updates the game world is parallelised. As in Gajinov et al.'s original work, I report on the "frame calculation time", which is the average time to execute the parallel portion of the benchmark. I use the 48-core AMD system (Appendix B: Tigger).

First, I focus on the performance of SLE_x86 alone. The Intel compiler generates a number of accesses to OpenMP thread-private locations. These can be filtered out using the technique described in the previous chapter (Section 6.3.1). Figure 7.4 shows how this benefits SLE_x86. The benefits (reduction in frame calculation time) range from 38% to 49%.

Next, I focus on comparing SLE_x86 (running with PDT) to the compiler-based STM solution in Figure 7.5. SLE_x86 scales *better* than the STM and adds only 11% overhead to the STM at 8 threads. Neither however manages to beat the lock due to the intrinsic overheads of software transactional memory.

It is interesting to note that SLE_x86 is able to scale while the compiler-based STM fails to scale altogether, something that is contrary to expectation from the results of the previous section. This can be explained by looking at the profile for the Quake server running at 8 threads, shown in Figure 7.6. There is a significant amount of lock contention that is revealed by the average queue length. The fraction of time spent waiting for a lock is low since threads spend a large fraction of their time in packet IO. Although locks are contended there is very little

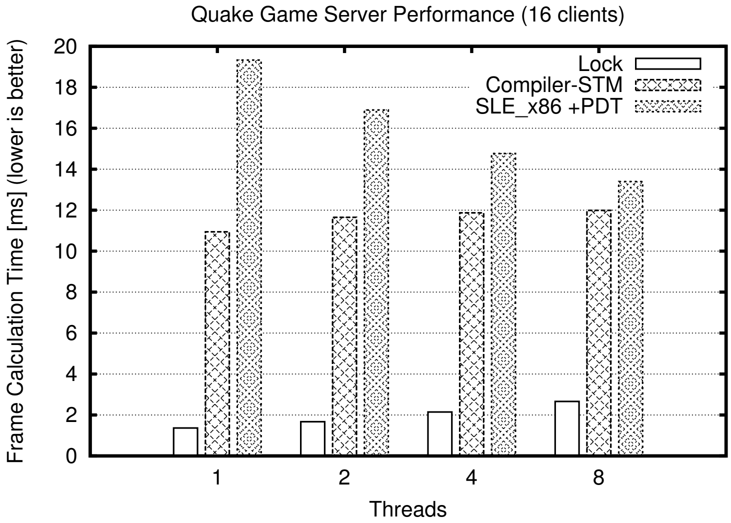


Figure 7.5: Quake using different implementations of critical sections

```
CS              cs_frac  wait_frac avg_q_len rd_ops    rd_locs  wr_ops    wr_locs  dep_dens
sv_user.c:1954 0.000    0.001     5.340     2063.683  403.678  1230.270  918.078  0.795
sv_user.c:1641 0.001    0.003     5.236     1335.027  422.335  1473.582  936.901  0.742
sv_user.c:1718 0.000    0.003     5.670     268.454   92.301   18.596    5.876    0.048
sv_user.c:1728 0.001    0.003     5.605     3683.126  170.150  493.964   52.405   0.026
sv_user.c:1761 0.000    0.002     5.515     484.420   188.304  54.822    25.882   0.075
sv_user.c:1966 0.000    0.000     4.343     23621.500 2355.750 4623.750  2117.000 0.989
Overall_waiting Overall_dep_density Pred_AvgThreads
0.010843        0.316659            3.055371
```

Figure 7.6: Quake profile

disjoint-access parallelism behind them: on the average only 3 threads out of 8 can execute critical sections in parallel. This has a larger impact on the STM used in the Intel C++ compiler [WCW$^+$07] since it uses eager locking and in-place updates making aborts expensive when compared to the write buffering used in SLE_x86. The results are also at odds with Gajinov et al. who reported positive scaling when using the STM version at 8 threads. This is likely a consequence of the system in use. I repeated the experiments on an 8-core (older Intel Xeon, Appendix B: Lander) system with four sockets on a frontside bus. The results are shown in Figure 7.7. Once again SLE_x86 scales better than the STM. However unlike the other system, the STM shows positive scaling at 8 threads.

I now turn my attention to software engineering considerations. Compared to a simple coarse-grained lock, using software transactional memory at a language level required about 700 source level annotations and in addition required porting some of the standard C library functions into the source code. This is necessary because language-level STM compilers cannot deal well with calls to legacy libraries (not instrumented to use STMs), something that is not a problem when working at x86 machine code level. The effort to use a language-level STM with Quake starting from a lock-based version is clearly non-trivial. In addition, debugging support for STM is currently limited to research projects and is complicated by speculating transactions. Gajinov et al. depended only on strategically placed print statements; a commendable effort but

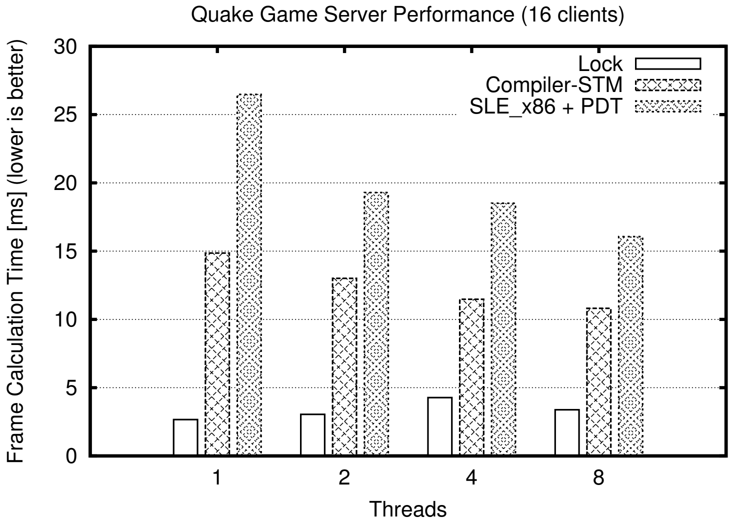


Figure 7.7: Quake using different implementations of critical sections (older Xeon system)

difficult to convince mainstream programmers on large software projects to use.

In contrast SLE_x86 was added on to the lock-based version at zero effort. It requires no new annotations and, being a runtime option, leaves the original program free to be debugged with standard tools. Since software transactional memory does not lead to better absolute performance compared to the lock for the examined thread counts, it is worth thinking whether the effort to port the application to use STM was worth it in this case. This serves to illustrate why operating at the level of machine code in a language, compiler and debugger agnostic manner can be extremely useful, given that the value of porting a large program to use an STM may not be evident.

7.3 Condition variables and fine-grained locks

Condition variables are a widely used mechanism for specifying synchronisation in multithreaded programs. Condition variables are usually manipulated within critical sections (such as the pthread_cond related primitives in the Pthreads library). The semantics for condition variables is that a thread holds a lock before initiating a wait on a condition variable. It *atomically* releases the lock when initiating the wait (usually to avoid lost signal problems). On being woken up, the thread continues with the lock held (re-acquired on a wakeup). In general condition variables are difficult to reconcile with transactions. Attempts to integrate condition variables with language level transactions either involve specially designed transactional condition variables [DS09], splitting transactions into a before and after transaction with respect to the wait [SKBY07] or by allowing communication between memory transactions [LP11]. SLE_x86 on the other hand has been designed to safely handle programs with condition variables. The reason is that the underlying STM_x86 of Chapter 3 falls back to pessimistic locking on encountering locked instructions or systems calls that are usually a part of condition variable implementations (this is true for the Pthreads condition variables). I verify this behaviour in this section.

Another interesting aspect of multithreaded programs today is the usage of fine-grained locks. Since transactional memory is not a mainstream programming option, developers carefully tune their applications to avoid bottlenecks on coarse-grained locks. The do this through the use of fine-grained locking, where threads a likely to require different locks and thus do not needlessly wait on each other.

I use the PARSEC [BKSL08] benchmark suite to examine the behaviour of SLE_x86 when presented with programs that display usage of well tuned contention-free locks and condition variables. PARSEC is a set of benchmarks that aim to represent the next generation of workloads that will run on Chip Multi-Processing (CMP) systems. Another more immediate aim is to replace the dated SPLASH [WOT$^+$95] benchmark suite that is still in use for research in general, including for software transactional memory. SPLASH benchmarks are too small to evaluate real machines today (or those in the future) and some of the benchmarks in it use algorithms that have been replaced by more modern versions.

PARSEC consists of 13 benchmarks all of which have been built to scale well and take advantage of threads. In addition to locking, PARSEC benchmarks also make use of ad-hoc synchronisation [JT10] such as the construct that was discussed in Figure 3.10 where a thread spins on a flag that indicates when another thread has published a value for consumption. When using an STM at the x86 machine level, the STM must thus provide the x86 memory consistency model, something that STM_x86 does do.

I begin with a profile of the benchmarks using a slightly modified form of the profiler in Chapter 5. The modification aims to detect the usage of locked instructions or systems calls in critical sections that would necessitate pessimistic locking. This is easily done by emitting in the traces a flag for when a locked instruction or memory fence is executed. Any such critical section is marked with an asterix (“*”) in the profile. Further the post-processing tool simulates the blacklist used by STM_x86. The lock corresponding to execution of such critical sections is blacklisted and any critical section acquiring the same lock is also marked similarly. The profiler thus provides an accurate picture of which critical sections need pessimistic locking with SLE_x86. In the case of the PARSEC benchmarks all these critical sections (after manual examination of the source code) were found to use condition variables.

Three of the 13 benchmarks in PARSEC: Blackscholes, Freqmine and Swaptions do not use locks at all for synchronisation and thus I do not consider them here. The Vips benchmark uses the Glib threading model, that is not as yet supported by SLE_x86. Of the remaining 9 benchmarks Canneal acquires exactly one lock at startup for a short period and thus is beyond the scope of the profiler that depends on sampling (and also is uninteresting from the perspective of lock elision). The profile for remaining 8 benchmarks (including lines of code) is shown in the table of Figure 7.8. As is evident from the profile, except for Facesim and Fluidanimate all the other benchmarks use critical sections only to protect condition variables. All of them are executed purely using pessimistic locking when executing with SLE_x86 and their performance is identical to that using locks. Crucially, SLE_x86 handles the condition variable seamlessly. For the rest of this section I focus on the two benchmarks that include conventional critical sections unrelated to condition variables. I use the 48-core AMD system (Appendix B: Tigger) for the experiments.

7.3.1 Facesim

At least some of the critical sections in Facesim are amenable to software lock elision. However, the fraction of time spent waiting for or executing these critical sections is negligible. Since SLE_x86 is built to execute code outside critical sections directly from the native binary, it should have little impact on performance. This fact is confirmed by Figure 7.9 that shows the performance of lock elision with SLE_x86 relative to running natively with the lock.

7.3.2 Fluidanimate

Fluidanimate represents an interesting benchmark since its critical sections are extremely short but represent a non-negligible fraction of execution time. An examination of the source code line number pointed to by the profiler reveals that it uses fine-grained locking and this explains why lock contention is not an issue with Fluidanimate.

Running Fluidanimate with SLE_x86 however reveals an interesting performance anomaly, shown in Figure 7.10. Scalability is reversed: increasing the thread count decreases performance relative to running the native binary without lock elision.

Fluidanimate has been written to use fine-grained locking and scales well without SLE_x86 in the picture. Using SLE_x86 on the other hand effectively *increases* lock contention since it executes critical sections using Single Lock Atomicity semantics. Critical sections that were previously unrelated (executed with different locks) now execute with SLA semantics requiring system wide serialisation for some portions of the commit phase. On the other hand the baseline lock-based version scales well since critical sections use different locks and proceed in parallel. This leads to the inversion in performance.

To confirm this, I used a specially constructed preloaded library that replaces all the different Pthread locks with a single lock. Running Fluidanimate with this library approximates the behaviour of using atomic blocks instead of critical sections with the atomic blocks implemented using a single global lock. Figure 7.11 shows that relative to this implementation of single lock atomicity SLE_x86 does indeed improve scalability. The initial drop in performance is due to the overhead added to each critical section by SLE_x86 which is balanced by the reduction

Source	CS frac	Wait frac	avg_q_len	Read Ops	Read Locs	Write Ops	Write Locs
Bodytrack (10279 LOC)							
Mutex.cpp:96*	0.297	0.002	0.141	10.342	7.871	4.384	3.287
Dedup (3689 LOC)							
queue.c:65*	0.001	0.001	0.029	130.800	10.128	61.900	8.920
queue.c:35*	0.141	0.001	0.006	130.736	10.048	55.457	8.005
encoder.c:838*	0.000	0.000	0.000	501.301	191.863	0.644	0.644
encoder.c:889*	0.011	0.000	0.000	600.559	233.432	0.457	0.457
encoder.c:224*	0.032	0.236	3.021	426.692	160.231	26.433	10.257
queue.c:24*	0.000	0.000	0.000	7.000	3.000	3.000	2.000
encoder.c:118*	0.000	0.000	0.004	176.663	54.343	8.000	4.000
Facesim (29310 LOC)							
taskQDistFixed.c:29	0.000	0.000	0.009	10.518	4.185	5.857	3.918
taskQDistCommon.c:81*	0.000	0.000	6.353	15.933	6.982	10.952	4.982
taskQDistFixed.c:10	0.000	0.000	0.000	5.215	2.405	0.937	0.937
taskQDistCommon.c:92*	0.018	0.000	0.586	18.401	9.189	9.382	5.311
taskQDistFixed.c:63	0.000	0.000	0.000	6.000	2.000	6.000	4.000
taskQDistCommon.c:70*	0.166	0.001	3.898	19.184	10.197	10.239	5.611
taskQDistFixed.c:30	0.000	0.000	0.011	7.590	3.368	4.055	2.875
Ferret (9735 LOC)							
semaphore.c:288*	0.000	0.000	0.000	990.274	6.651	495.017	8.887
semaphore.c:123*	0.000	0.000	0.000	377.069	4.228	183.556	3.843
semaphore.c:373*	0.010	0.000	0.000	438.077	4.455	215.030	4.319
ferret-parallel.c:176*	0.000	0.000	0.248	2109.749	10.008	1065.827	18.318
ferret-parallel.c:244*	0.237	0.000	0.332	20.000	5.000	10.000	3.000
ferret-parallel.c:258*	0.000	0.000	0.359	13.000	4.000	8.000	3.000
ferret-parallel.c:271*	0.239	0.000	0.339	20.000	6.000	10.000	3.000
ferret-parallel.c:283*	0.000	0.000	0.429	13.000	3.000	8.000	3.000
ferret-parallel.c:295*	0.191	0.000	0.425	19.967	4.995	9.985	2.997
ferret-parallel.c:320*	0.000	0.000	0.241	12.918	3.959	7.959	3.004
ferret-parallel.c:333*	0.105	0.000	0.425	19.695	5.946	9.848	2.992
ferret-parallel.c:364*	0.000	0.000	0.280	12.885	3.951	7.944	3.009
ferret-parallel.c:374*	0.045	0.000	0.000	19.728	5.946	9.864	2.996
Fluidanimate (1391 LOC)							
pthreads.cpp:500	0.002	0.007	0.000	4.000	4.000	1.000	1.000
pthreads.cpp:685	0.014	0.054	0.000	5.000	3.522	3.000	1.522
pthreads.cpp:694	0.015	0.056	0.000	5.000	3.490	3.000	1.490
pthreads.cpp:603	0.019	0.075	0.000	3.000	3.000	1.000	1.000
pthreads.cpp:612	0.021	0.076	0.000	3.000	3.000	1.000	1.000
Raytrace (13302 LOC)							
RTThread.hxx:167*	0.111	0.000	5.651	7.570	4.487	4.349	2.898
Streamcluster (1255 LOC)							
streamcluster.cpp:893*	0.000	0.000	6.333	0.000	0.000	1.000	1.000
streamcluster.cpp:869*	0.000	0.000	5.765	0.000	0.000	1.000	1.000
streamcluster.cpp:843*	0.001	0.000	2.693	9.201	6.480	5.011	2.615
x264 (40393 LOC)							
frame.c:880*	0.000	0.000	0.630	8.109	3.582	6.163	3.054
frame.c:888*	0.000	0.000	0.151	3.809	2.183	1.478	0.739

Figure 7.8: PARSEC critical section memory operations

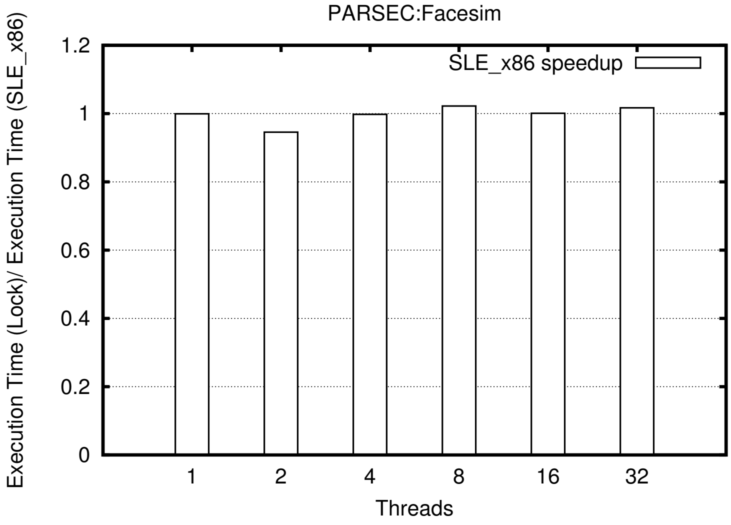


Figure 7.9: Using SLE on Facesim

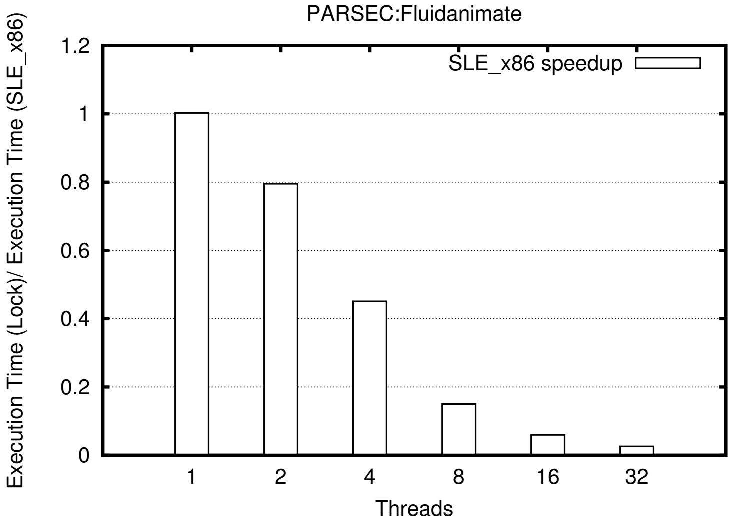


Figure 7.10: Using SLE on Fluidanimate

in time spent waiting at the lock. Increasing the number of threads increases the overhead of SLE_x86 (due to the epoch). This is significant given the very short critical sections. Ultimately however the time spent waiting for a lock becomes a larger factor leading to the improvement in performance.

These results pose an interesting question from the perspective of language level atomic blocks. Language level atomic blocks are essentially *anonymous* meaning that they they do not reflect that fact that some set of atomic blocks can be *statically* determined as non-conflicting. On the other hand locks allow the programmer to easily express what they already know about necessary synchronisation in the program. Even an efficiently implemented language level STM would lead to slowdowns in Fluidanimate over the lock-based version due to the need to preserve the language level memory model. An effort to port Fluidanimate to use language level atomic blocks would clearly be wasted. SLE_x86 is a runtime option and reveals this without

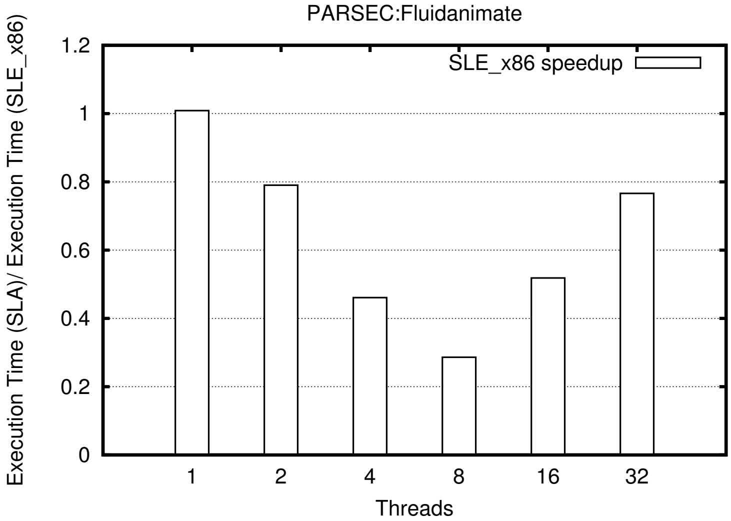


Figure 7.11: Using SLE on Fluidanimate with Single Lock Atomicity

the need to expend this effort. An interesting option worth exploring with language level STMs as well as with SLE_x86 is the ability to statically determine which transactions cannot conflict and exploit this information in the STM.

7.4 Discussion

This chapter examined the performance of SLE_x86 in a variety of application settings. The first is on hardware (and benchmarks) that offer unlimited scalability (disjoint-access parallelism). There is no fundamental bottleneck in the scalability of SLE_x86 and it performs better than a lock. The second is a real-world multiplayer game server written to use the OpenMP threading primitives. In this case SLE_x86 scales better than the C++ level STM. More importantly, it illustrates how SLE_x86 demands no extra annotations in software and leaves the developer free to use standard debuggers on their programs. The third application looked at programs that make heavy use of condition variables and are otherwise tuned to perform well with locks. SLE_x86 has no problem handling condition variables.

An interesting observation that comes out from the Fluidanimate benchmark in PARSEC is that language level transactions may not allow adequate expression of opportunities for safe parallel execution. Critical sections protected by different locks need not incur the cost of being serialised by the same lock. One way to exploit this – and a possible direction for future work – would be to look at forms of “multiple” lock atomicity, where critical sections can be mapped to a (small) set of logical locks in the STM runtime system. This can be achieved, for example, by replicating the STM data structures and algorithm for each lock, thus exploiting the fact that fine-grained locking already specifies the available parallelism in such applications.

Chapter 8

Conclusion

This dissertation presented SLE_x86: a safe and automatic way to apply software transactional memory to x86 binaries that synchronise using locks. SLE_x86 leads to better scalability than using a coarse-grained lock, when the lock is a bottleneck in the program. The application of optimistic concurrency control allows threads to speculate past locks leading to absolute better performance than the lock-based version for many of the STAMP benchmarks and the AVLtree and Skiplist microbenchmarks. In the case of the more real-world Quake multiplayer game it leads to better scalability than the lock and is competitive to a compiler-based STM. Finally, SLE_x86 is widely applicable, even to critical sections with condition variables. In the rest of this chapter I summarise my contributions and directions for future research.

8.1 Summary

In Chapter 1 I motivated the need to apply transactional memory at the level of abstraction of x86 machine code. I presented my thesis that it is possible to do this automatically, transparently and correctly to binaries not written to have any awareness of transactional memory.

In Chapter 2 I outlined research in the field of transactional memory in general and software transactional memory in particular. I taxonomised the different kinds of STM designs possible and in particular discussed why it is difficult to build an STM that correctly handles the interaction between transactional and non-transactional accesses. I also discussed the different ways to apply software transactional memory: manually using an STM library; automatically using an STM compiler and source code annotations; and automatically using dynamic binary rewriting alone.

In Chapter 3 I presented the first primary contribution of this dissertation: a software transactional memory design (STM_x86) that provides single lock atomicity and preserves the x86 memory consistency model. I show that, given the strictness of the x86 memory model, the only way to build STM_x86 is to exclude programs that contain a certain type of data race. I argue that in the context of lock-based programs this race likely corresponds to buggy synchronisation. The evaluation using the STAMP benchmarks in Chapter 3 examines the performance of STM_x86 in comparison to the more scalable TL2 STM that provides a weaker memory consistency model.

In Chapter 4 I presented a means to apply this STM automatically to x86 machine code synchronising with locks. I use a *hybrid* form of binary rewriting that uses dynamic execution to discover code to be instrumented but places instrumentation in a static persistent instrumentation cache for reuse. In general, the instrumentation system presented borrows some of the best ideas from static and dynamic binary rewriting engines in order to reduce instrumentation overhead to acceptable levels. This leads to the complete SLE_x86 system that is competitive to manual instrumentation for four of the STAMP benchmarks (Vacation, Kmeans, SSCA2 and Intruder). For the remaining benchmarks, there is a large gap between the performance achieved through manual instrumentation and that achieved through automatic instrumentation.

In Chapter 5 I presented a profiler that allows analysis of critical section characteristics that are important for transactional memory. It uses the same instrumentation mechanism as in the previous chapter but instead of software lock elision it simply measures lock contention and memory access related characteristics of critical sections. It allows the determination of whether a binary contains a critical section that is in a “sweet spot” for transactional memory: lots of lock contention and lots of disjoint-access parallelism (no data flow between simultaneously executing critical sections). In addition, the memory access profiles pinpoint the precise reason for the performance gap in four of the STAMP benchmarks: the programmer has deliberately omitted STM instrumentation in the source code by exploiting an understanding of which program locations are thread private.

In Chapter 6 I presented a generic mechanism for tagging program locations as thread-private and then exploiting such tagging in the STM logging algorithms to eliminate STM overheads for them. I explore three specific applications of this mechanism: the first is for locations on the heap that are known statically to be thread-private; the second is for locations on the stack that are *usually* thread-private but it is necessary to detect when they are shared; and the third is adaptive tagging that can classify thread-private heap data dynamically from that which is not. *I demonstrate that with adaptive tagging SLE_x86 provides better performance than the lock in many of the STAMP benchmarks.*

In Chapter 7 I explored the application of Software Lock Elision in more general settings. I show that there is no fundamental scalability limit to SLE_x86 and it is able to beat the lock on AVLTree and Skiplist micro-benchmarks on a large multicore system (up to 96 cores). Next, I show how SLE_x86 is much easier to apply to the large Quake multiplayer game server and is only 11% slower than a much harder to use STM compiler. Finally, I show using the PARSEC benchmarks that SLE_x86, by virtue of being designed for safety and transparency has no problems with handling condition variables.

8.2 Future research

SLE_x86 allows the wide and transparent applicability of software transactional memory to *all* programs. Mainstream adoption of transactional memory is thus no longer dependent on efforts to either change the software development environment or rewrite legacy code.

The next step then is to build a complete hybrid transactional memory solution for x86 binaries. There is already a proposal for best-effort hardware transactional memory support on the x86 (Advanced Synchronisation Facility [CCD+10]) for which simulators are already available. ASF can be easily integrated with SLE_x86 to maintain the read and write logs in hardware. The common case (of short transactions) can be executed in hardware while the uncommon case (longer transactions) can be executed *correctly* on SLE_x86, leading to low overhead transactions by Amdahl's law. Evaluating large multithreaded programs on such a hybrid solution should provide enough impetus for actually adding ASF to mainstream x86 microprocessors (and for other architectures). This will effectively break the deadlock that is currently impeding progress in transactional memory.

Another interesting direction is to use the profiler described in Chapter 5 on a large representative set of real-world programs. This can help resolve ongoing debate in the transactional memory community about whether critical sections in real-world code can truly benefit from optimistic concurrency control.

Appendix A: SLE_x86 restrictions

This dissertation includes a careful examination of the problems caused when applying software transactional memory to the unmanaged environment of x86 machine code. Some these problems necessitate restrictions on the programs that can be used with SLE_x86. The complete set of restrictions is listed below.

1. **Restriction 1:** The program must not admit an execution with a memory fence or a locked instruction in a critical section (Section 3.6).
2. **Restriction 2:** Execution cannot depend on stores in a critical section being made visible to other threads before the critical section completes (Section 3.6).
3. **Restriction 3:** The execution must not contain a race between a read in a critical section and a write outside any critical section (Section 3.6).
4. **Restriction 4:** A location on a thread stack that is shared between threads can only be accessed in critical sections protected by the same lock or always outside any critical section (Section 6.3.2).

Of these, the first three are either detected and handled by avoiding the use of the STM for some critical sections or map to program behaviour that is likely buggy (Section 3.6). The last one is necessary for efficient instrumentation and cannot be practically avoided without (as far as I can see) undue addition of complexity and overhead.

Appendix B: Experiment configurations

System: There are three systems that I have used in this dissertation.

1. Tigger (Medium-sized multicore NUMA): Consisting of 48 AMD cores and used for the vast majority of experiments in this dissertation. The system consists of four AMD 6168 processors running at 1.9 Ghz connected by a Hypertransport interconnect. Each processor consists of two CMP dies connected by a faster inter-chip interconnect. Each die consists of 6 processors sharing a 6MB L3 cache. The system has 64 GB main memory distributed among the processors.

2. COSMOS (Large-scale multicore NUMA): An SGI Altix UV consisting of 768 cores used for the experiments in Section 7.1. It is built around 128 Intel NehalemEX 7460 2.67GHz CPUs connected by the NUMAlink interconnect. Each CPU is a CMP die with six cores sharing a 16MB L3 cache. The system has 2TB of main memory. I was able to reserve 96 cores for my experiments but had no control over jobs running on other cores on the shared system.

3. Lander (Small-scale SMP) : Four dual-core Intel 7130M Xeon processors (8 cores of the old P4 microarchitecture) running at 3.2 Ghz. This system only features in Figure 7.7 in this dissertation although much of the development and test for SLE_x86 happened on this system. Each CPU has two cores sharing an 8MB L3 cache. The CPUs are connected by a frontside bus. Although each core is capable of supporting two hyperthreads the experiments were done with hyperthreading off. The system has 8GB of main memory.

Compiler: gcc 4.4 except for Sections 7.1 and 7.2, which used the Intel 3.0 STM capable prototype compiler. I use optimisation level 3 for all benchmarks.

PIN: I used PIN version 27887 for all experiments in this dissertation.

Benchmarks:

1. STAMP version 0.9.10 `http://stamp.stanford.edu/`.

2. QuakeTM version 1.0 `http://www.bscmsrc.eu/software/quaketm`

3. PARSEC version 2.0 `http://parsec.cs.princeton.edu/`

Bibliography

[ABH$^+$08] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Dynamic separation for transactional memory. Technical Report TR-2008-43, Microsoft Research, 2008.

[AH98] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *25 years of the International Symposia on Computer Architecture (selected papers)*, pages 363–375, 1998.

[AHM09] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009.

[Akr10] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the USENIX Security Symposium*, pages 177–192, 2010.

[ATLM$^+$06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 68–78, 2008.

[BDA00] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, 2008.

[CCD+10] Dave Christie, JaeWoong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the European Conference on Computer Systems*, pages 27–40, 2010.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[DM05] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for Linux, 2005. RedHat technical white paper.

[DMS10] David Dice, Alexander Matveev, and Nir Shavit. Implicit privatisation using private transactions. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2010.

[DNAT09] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. Optimizing transactions for captured memory. In *Proceedings of the Annual Symposium on Parallelism in Algorithms and Architectures*, pages 214–222, 2009.

[DS07] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

[DS09] Polina Dudnik and Michael M. Swift. Condition variables and transactional memory: problem or opportunity ? In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2009.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the International Symposium on Distributed Computing*, pages 194–208, 2006.

[Enn05] Robert Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge, 2005.

[FFM$^+$07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2007.

[Fra03] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[GZU$^+$09] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the International Conference on Supercomputing*, pages 126–135, 2009.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[HF05] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 72–82, 2005.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceed-*

ings of the Symposium on Principles of Distributed Computing, pages 92–101, 2003.

[HLR10] Tim Harris, Jim Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, pages 289–300, 1993.

[HOSS97] Sabine Hanke, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed balanced red-black trees. In *Proceedings of the Italian Conference on Algorithms and Complexity*, pages 193–204, 1997.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[HWC$^+$04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture*, pages 102–113, 2004.

[int] `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.

[IR94] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[JT10] Ali Jannesari and Walter F. Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–10, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[LP11] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 157–168, 2011.

[MBL06] Milo M. K. Martin, Colin Blundell, and E. Christopher Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17–20, 2006.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 254–265. 2006.

[MBS+08a] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2008.

[MBS+08b] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java stm. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 314–325, 2008.

[Mic03] Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Proceedings of the Euro-Par Conference on Parallel Processing*, pages 651–660, 2003.

[MSH+06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, 2006.

[MTC+07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees.

In *Proceedings of the International Symposium on Computer Architecture*, pages 69–80, 2007.

[NM10] Takuya Nakaike and Maged M. Michael. Lock elision for read-only critical sections in Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 269–278, 2010.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[OCS07] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375, 2007.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

[PG10] Mathias Payer and Thomas R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the Haifa Experimental Systems Conference*, pages 1–14, 2010.

[PHW07] Donald E. Porter, Owen S. Hofmann, and Emmett Witchel. Is the optimism in optimistic concurrency warranted? In *Proceedings of the USENIX Workshop on Hot topics in Operating Systems*, pages 1–6, 2007.

[PW10] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *Proceedings of the International Symposium on Performance Analysis of Software Systems*, pages 97–108. 2010.

[Raj02] Ravi Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin-Madison, 2002. ISBN:0-493-92677-1.

[RBdB08] Torvald Riegel and Diogo Becker de Brum. Making object-based STM practical in unmanaged environments. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2008.

[RBK+09] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 173–184, 2009.

[RCCS07] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 74–88, 2007.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the International Symposium on Microarchitecture*, pages 294–305, 2001.

[RHH09a] Amitabha Roy, Steven Hand, and Tim Harris. Exploring the Limits of Disjoint Access Parallelism. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2009.

[RHH09b] Amitabha Roy, Steven Hand, and Tim Harris. A runtime system for software lock elision. In *Proceedings of the European Conference on Computer Systems*, pages 261–274, 2009.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 494–505, 2005.

[RHP$^+$07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 87–102, 2007.

[SATH$^+$06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[SDAL01] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *In Proceedings of the Workshop on Binary Translation*, 2001.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SKBY07] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the Conference on Object-oriented Programming Systems and Applications*, pages 191–210, 2007.

[SMAT$^+$07] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[SMDS07] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report TR 915, Computer Science Department, University of Rochester, 2007.

[SN05] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of USENIX Annual Technical Conference*, pages 2–15, 2005.

[SS05] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 240–248, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[TWGM07] Fuad Tabba, Cong Wang, James R. Goodman, and Mark Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2007.

[USB09] Takayuki Usui, Yannis Smaragdakis, and Reimer Behrends. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[vPBC08] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2008.

[VPCDB$^+$05] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the International Symposium On Signal Processing And Information Technology*, pages 7–12, 2005.

[WCW+07] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, 2007.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.

[WSAT08] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, June 2008.

[WYW08] Cheng Wang, Victor Ying, and Youfeng Wu. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *Proceedings of the International Conference on Compiler Construction*, pages 291–306, 2008.

[x8609] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3a*, November 2009.

[ZBS08] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Commit phase in timestamp-based STM. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 326–335, 2008.

[ZWAT+08] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 129–154, 2008.