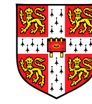
Technical Report

UCAM-CL-TR-831
ISSN 1476-2986

Number 831



UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Communication for programmability and performance on multi-core processors

Meredydd Luff

April 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2013 Meredydd Luff

This technical report is based on a dissertation submitted November 2012 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The transition to multi-core processors has yielded a fundamentally new sort of computer. Software can no longer benefit passively from improvements in processor technology, but must perform its computations in parallel if it is to take advantage of the continued increase in processing power. Software development has yet to catch up, and with good reason: parallel programming is hard, error-prone and often unrewarding.

In this dissertation, I consider the programmability challenges of the multi-core era, and examine three angles of attack.

I begin by reviewing alternative programming paradigms which aim to address these changes, and investigate two popular alternatives with a controlled pilot experiment. The results are inconclusive, and subsequent studies in that field have suffered from similar weakness. This leads me to conclude that empirical user studies are poor tools for designing parallel programming systems.

I then consider one such alternative paradigm, transactional memory, which has promising usability characteristics but suffers performance overheads so severe that they mask its benefits. By modelling an ideal inter-core communication mechanism, I propose using our embarrassment of parallel riches to mitigate these overheads. By pairing "helper" processors with application threads, I offload the overheads of software transactional memory, thereby greatly mitigating the problem of serial overhead.

Finally, I address the mechanics of inter-core communication. Due to the use of cache coherence to preserve the programming model of previous processors, explicitly communicating between the cores of any modern multi-core processor is painfully slow. The schemes proposed so far to alleviate this problem are complex, insufficiently general, and often introduce new resources which cannot be virtualised transparently by a time-sharing operating system. I propose and describe an asynchronous remote store instruction, which is issued by one core and completed asynchronously by another into its own local cache. I evaluate several patterns of parallel communication, and determine that the use of remote stores greatly increases the performance of common synchronisation kernels. I quantify the benefit to the feasibility of fine-grained parallelism. To finish, I use this mechanism to implement my parallel STM scheme, and demonstrate that it performs well, reducing overheads significantly.

For Sarah. We miss you.

Acknowledgements

A good supervisor is like good parents; one realises only gradually, and by looking around, how good one has it. I have been blessed with both.

Simon Moore has always taken care to allow me my freedom, yet freely offered his expertise and assistance. No matter where my research has taken me, he has always been there at a moment's notice with advice, encouragement or a crucial introduction. I could not have asked for a better supervisor.

My parents are my pillars of support, and my emotional bedrock. *Diolch yn fawr iawn i chi.* Geraint and Eleri have been reliably loving to their big brother, the occasional light-hearted mockery or moustache drawn on my face notwithstanding.

I am, of course, deeply indebted to each of the many colleagues, in the Computer Architecture group and beyond, who have discussed my work, pointed out where I was wrong, or just sat down for a chat. I owe particular thanks to Robert Mullins, whose conversations always leave me more knowledgeable than before, as well as for being one of the nicest people I know. Luke Church and Alan Blackwell provided expert advice for the user study in Chapter 2, and Ian Davies has kept me sane through the workday.

Outside the department, my friends are my lifeline. Maja Choma has been as reliable as I am scatty, despite an experimental workload that beggared belief, and I owe her for many a well-timed cup of tea. *Dziękuję.* The last eight years would not have been the same without Caians, past and present: Shashank Joshi, Rachel Newton, Jo Wensley, Carmen Rodriguez Gonzalvez, Alice Alphandary, and more. Helen Waller, Deepti Aswani and Rebecca Day have also, at various times, helped me out of sullen places and celebrated with me in joyful ones.

Working with Louie Oviedo, of Greetings Unlimited Inc, has allowed me to learn to fly, both metaphorically and literally.

Contents

1 Introduction **13**
1.1 Overview . . . 13
1.2 Outline . . . 14
1.2.1 Parallel Programming and Usability . . . 14
1.2.2 Parallelising Software Transactional Memory . . . 14
1.2.3 Remote Stores for Inter-Processor Communication . . . 14
1.3 Contributions . . . 16
1.4 Prior Publication . . . 16

2 Parallel Programming Practice and Usability **17**
2.1 Introduction . . . 17
2.2 Industry Practice in Parallel Programming . . . 17
2.2.1 Avoiding Concurrency . . . 17
2.2.2 Natural Parallelism . . . 18
2.2.3 Concurrency for Convenience . . . 18
2.2.4 Parallelism for Performance . . . 19
2.3 Alternative Paradigms . . . 19
2.3.1 Same Paradigm, Better Control . . . 19
2.3.2 Deterministic Parallelism . . . 20
2.3.3 Functional Programming . . . 20
2.3.4 Pure Message Passing . . . 20
2.3.5 Transactional Memory . . . 21
2.4 Empirically Investigating Paradigms of Parallel Programming . . . 22
2.4.1 High-Performance Computing . . . 22
2.4.2 Commodity Hardware . . . 23
2.5 A Pilot Study . . . 23
2.6 Materials and Methods . . . 23
2.6.1 Subjects . . . 24
2.6.2 Task . . . 24
2.6.3 Implementing Different Paradigms . . . 25
2.7 Results . . . 26
2.7.1 Objective Measures of Effort . . . 26
2.7.2 Subjective Reporting of Effort . . . 29
2.7.3 The Measurements Don't Even Agree . . . 29
2.7.4 Bifuricated Results: Does the Camel Have Two Humps? . . . 31
2.8 Discussion . . . 31
2.8.1 Possible explanations . . . 31
2.8.2 Additional threats to validity . . . 32
2.9 Conclusion . . . 34

3 Software Transactional Memory 35
3.1 Introduction 35
3.1.1 Contributions 35
3.2 Background 37
3.2.1 Beginnings 37
3.2.2 STM's overhead problem 37
3.2.3 Real applications often lack scalability 37
3.2.4 HTM 38
3.2.5 Helper Threading 39
3.2.6 Summary 39
3.3 Anatomy of a Parallel STM 40
3.3.1 Overview 40
3.3.2 Comparison model: Inline STM 41
3.3.3 STM Algorithm A: Validation-Based 41
3.3.4 STM Algorithm B: Invalidation-Based 41
3.3.5 Summary 44
3.4 Evaluation Framework 47
3.4.1 Choice of Platform 47
3.4.2 Hardware 47
3.4.3 Benchmarks 48
3.5 Results and Evaluation 49
3.5.1 Overview 49
3.5.2 Serial Performance 49
3.5.3 Performance at Scale, or Return on Investment 50
3.5.4 Breaking Down the Overhead 53
3.5.5 Investigating Helper Workload 54
3.5.6 The Effect of Communication Cost 55
3.5.7 Validating the Model 56
3.6 Conclusion 58

4 Asynchronous Remote Stores for Inter-Processor Communication 59
4.1 Introduction 59
4.1.1 Contributions 59
4.2 Background 60
4.2.1 Cache-aware Algorithms 60
4.2.2 Write Forwarding and Update Coherence 60
4.2.3 Abandoning Uniform Shared Memory 60
4.2.4 Dedicated Message-Passing Hardware 60
4.2.5 A Message Passing Coprocessor 61
4.2.6 Possible Applications 61
4.3 The StRemote Instruction 61
4.3.1 Ordering 62
4.3.2 Robustness 62
4.3.3 Applicability 63
4.4 Implementation 63
4.4.1 Architecture 63
4.4.2 Memory Consistency and Ordering 64
4.4.3 Interconnect 64
4.5 Evaluation 65
4.5.1 FIFO queues 65
4.5.2 Synchronisation Barriers 68
4.5.3 Quantifying Fine Grained Parallelism with Map-Reduce 70
4.5.4 Parallelising Dynamic Analysis 70

4.5.5 Parallel STM . . . 72
4.6 Conclusion . . . 75

5 Conclusion **77**
5.1 Introduction . . . 77
5.2 Usability and Parallel Programming . . . 77
5.3 Software Transactional Memory . . . 77
5.4 Hardware Support for Communication . . . 78
5.5 Further Work . . . 78
5.5.1 Software Transactional Memory . . . 78
5.5.2 Asynchronous Remote Stores . . . 78
5.6 Summary . . . 79

List of Figures

2.1 A worked example of the problem posed to experimental subjects. 24
2.2 Example code using the Deuce transactional memory system. 25
2.3 An example demonstrating my Actor API, which enforces isolation between actors. 26
2.4 Time taken by subjects to complete each condition. 27
2.5 The increase or decrease in time taken, compared with the time taken by the same subject under the SMTL condition. 27
2.6 Non-comment lines of code in each solution. 28
2.7 The increase or decrease in code size, compared with the same subject's code under the SMTL condition. 28
2.8 The average subjective ratings of subjects in each condition. Ratings are sign-normalised such that higher ratings are good, and presented as a difference from ratings in the SMTL condition. The sequential condition is omitted, as most of the survey questions were inapplicable to it. Subjects significantly preferred TM to SMTL ($p \approx 0.04$), and showed a suggestive preference for Actors over SMTL ($p \approx 0.07$). 29
2.9 The three metrics used to evaluate programmer performance do not correlate with each other: $r^2 < 0.08$ and $p > 0.19$, for all pairs. $p > 0.15$ for all pairs when using ANCOVA to control for the learning effect between first and second run. 30
2.10 Aggregate completion times of every study. With one exception, subjects either completed the task within two hours or did not complete it within the four-hour session. 31
2.11 An illustration of an alternative Actor API, which could have addressed some limitations of the model actually used in the experiment. 33

3.1 Communication between application and helper cores by message-passing within a multicore processor 40
3.2 Pseudocode for a simple validation-based STM algorithm. This code may be run inline, or spread between application and helper cores as described in Figure 3.3 42
3.3 Pseudocode for a parallel validation-based STM runtime system, which offloads bookkeeping overhead to a helper thread. 42
3.4 An worked example of instrumenting a simple transaction, using the algorithms in Figures 3.2 and 3.3. 43
3.5 Pseudocode for a simple invalidation-based STM algorithm. This code may be run inline, or spread between application and helper cores as described in Figure 3.8 44
3.6 The helper thread compares the current value of a memory location with the value read by the application thread. This allows it to detect any conflicting modifications that occurred before the read was recorded in that location's orec. After the read is recorded, any subsequent conflicts will mark our transaction as invalidated. 45
3.7 If the application thread reads and then writes a single memory location, the helper thread cannot detect a conflicting transaction until it processes the write event. 45
3.8 Pseudocode for a parallel invalidation-based STM runtime system, which offloads bookkeeping overhead to a helper thread. 46
3.9 Hardware prototype board running eight soft processors 47
3.10 Overview of the system hardware 48
3.11 Operation of time-dilation circuitry. When one of the Nios soft cores makes a memory request, the gated clock, from which all components except the DDR2 memory controller are run, is paused. The clock is restarted when the memory operation has completed. With a 50 MHz external clock, the system clock runs at an average of 18 MHz. 49
3.12 Workload parameters for the integer STAMP benchmarks 49

3.13 Parallelising the STM system using helper threads substantially decreases the overheads in each application thread. Runtimes for single application threads are plotted relative to the applications' uninstrumented performance. Inline algorithms use one core per application thread, and parallel algorithms use two. . . . 50
3.14 Parallelising the STM system using helper threads does not decrease the return on investment of using more cores, even compared core-for-core against an equivalent inline STM running twice as many application threads. Performance is plotted as throughput, the reciprocal of run-time. . . . 51
3.15 Parallelising the STM system using helper threads increases the return on investment of parallelising a program. Performance is plotted as throughput (the reciprocal of run-time), against available parallelism (the number of application threads). . . . 52
3.16 A breakdown of the serial overhead of a validation-based parallel STM, running one application thread and one helper thread. . . . 53
3.17 A breakdown of the serial overhead of an invalidation-based parallel STM, running one application thread and one helper thread. . . . 54
3.18 The helper thread spends a significant amount of time idle, in both STM algorithms. This graph represents runs with one application thread and one helper thread. . . . 55
3.19 Parallel STM performance is insensitive to delays in message transmission, but acutely sensitive to the cost of enqueue and dequeue operations. This graph represents runs with one application thread and one helper thread, and performance is plotted relative to uninstrumented sequential code. Dotted lines indicate regions where the parallel system is outperformed by the inline system, indicating that communication costs have overwhelmed the benefits of parallelism. . . . 56
3.20 Comparing benchmark performance on different models. The performance of the FPGA model is comparable to that of the same code running on the Gem5 simulator and a real 4-core x86 machine. Performance is plotted as throughput (reciprocal of runtime). . . . 57

4.1 A contention-free FIFO queue using remote stores. . . . 65
4.2 Latency and throughput of FIFO queues. Lower times are better, and measurements are normalised to the performance of the remote-store queue on the O3 (weakly-ordered superscalar) CPU model. . . . 66
4.3 Cost of communication operations in different FIFO states. Performance is normalised to the performance of the remote-store queue on the O3 (weakly-ordered superscalar) CPU model. I could not accurately measure the "Full" condition on the O3 model, due to the activity of the prefetcher – see text for details. . . . 67
4.4 Time spent in level-one cache misses, for each algorithm running on the *TimingSimple* model. Dark colours represent time spent on cache misses; light colours represent all other runtime. Results are normalised to the performance of the remote-store queue on the O3 model. . . . 68
4.5 Bus occupancy, for each algorithm running on the *TimingSimple* model. Dark colours represent time during which the shared (level three) bus is occupied; light colours represent all other runtime. Results are normalised to the performance of the remote-store queue on the O3 model. . . . 69
4.6 Bus occupancy, for remote-store and MCRB FIFOs in the optimal throughput condition on the *TimingSimple* model. Occupied time is plotted on the same numerical scale as in Figure 4.5. . . . 69
4.7 A contention-free synchronisation barrier using remote stores. . . . 70
4.8 Performance of a simple remote-store barrier, compared with counting and tournament barriers. Runtime is normalised to the performance of the remote-store barrier on the O3 (weakly-ordered superscalar) CPU model. . . . 71
4.9 Parallel overheads for fine-grained Map-Reduce. For each workload size, the parallel runtime is plotted as a proportion of the baseline single-threaded runtime. Parallel runs were conducted with four processors. . . . 71
4.10 Overheads for function-call profiling in benchmarks from the SPEC2000 benchmark suite. This graph compares inline profiling, running on a single core, with parallel profiling running on one application and one helper core. Runtimes are normalised to the runtime of the uninstrumented program. Due to the size of the benchmarks, measurements were made on the *TimingSimple* model only. . . . 72

4.11 Pseudocode for communication between application and helper threads in a parallel STM system using asynchronous remote stores. This pseudocode provides implementations of the FIFO `send()` and `receive()` functions in Figure 3.8. 72
4.12 Remote stores allow useful parallelisation of an invalidation-based STM system using helper threads. Run-time is plotted as a proportion of uninstrumented run-time. The parallel STM uses two CPUs to run one application thread. 74
4.13 A breakdown of the serial overhead of an invalidation-based parallel STM, using remote stores in a simulated system. 74

Chapter 1

Introduction

1.1 Overview

As feature sizes have decreased and thermal budgets become more limiting, the steady increase in single-core processor performance has petered out [121]. Processor manufacturers have instead combined increasing numbers of processor cores onto a single die. This near-universal transition to multi-core processors has brought a fundamentally new sort of computer onto every desk and into every pocket.

Software can no longer benefit passively from improvements in processor technology. Until the mid-2000s, each successive generation of chips promised greatly increased serial performance, and would run the same applications more quickly. Now, in order to receive any benefit from the continued increase in processing power, programs must perform their computations in parallel. In many ways, software development has yet to catch up. There are good reasons for this: parallel programming is hard, error-prone and often unrewarding.

This is the multi-core challenge. The usability of parallel programming is poor, and it is difficult to choose between alternatives which seek to improve it. Some of the most intuitively promising of these alternatives suffer from basic practical problems. And modern hardware, in its attempt to preserve the sequential programming model as far as possible, does not allow us to take charge of the communication within our programs explicitly.

In this dissertation, I consider the challenges of programming in the multi-core era as problems of communication. How should communication be expressed in a program? How can communication help us to implement more human-friendly parallel programming models such as STM? And how can we express this communication explicitly to the hardware, while preserving the benefits we gain from today's processor architectures?

I present one line of enquiry into each of these questions. I investigate the expression of communication within a parallel program with a controlled user study. I investigate the use of parallelism and explicit hardware communication to accelerate software transactional memory (STM). And finally, I investigate how explicit hardware communication can be supported with high-performance in a cache-coherent multi-core system.

I begin by reviewing a number of alternative paradigms which aim to address these challenges, and conduct a pilot user study to compare the usability of two popular alternatives empirically. The inconclusive results, and the trajectory of subsequent similar work, lead me to conclude that empirical user studies are poor tools for evaluating the design of parallel programming systems.

I then consider one such alternative paradigm, transactional memory, which has promising usability characteristics but suffers performance overheads so severe that they mask its benefits. By modelling an ideal inter-core communication mechanism, I propose using our embarrassment of parallel riches to mitigate these overheads. By pairing "helper" processors with application threads, I offload the overheads of software transactional memory, thereby greatly mitigating the problem of serial overhead.

Finally, I shift my attention to the mechanics of inter-core communication. Due to the use of cache coherence to preserve the programming model of previous processors, explicitly communicating between the cores of any modern multi-core processor is painfully slow. The schemes proposed so far to alleviate this problem are complex, insufficiently general, and often introduce new resources which cannot be virtualised transparently by a time-sharing operating system. I propose and describe an asynchronous remote store instruction, which is issued by one core and completed asynchronously by another into its own local cache. I evaluate several patterns of parallel communication, and determine that the use

of remote stores greatly increases the performance of common synchronisation kernels. I quantify the benefit to the feasibility of fine-grained parallelism. To finish, I use this mechanism to implement my parallel STM scheme, and demonstrate that it reduces overheads significantly.

1.2 Outline

1.2.1 Parallel Programming and Usability

In Chapter 2, I review the lukewarm industry response to the challenges of parallel programming, particularly the dominant model of multithreading, which shares mutable memory between threads and uses locks for concurrency control. I discuss several alternative paradigms which attempt to address those challenges: although there are many, they are all to some extent tentative, and adoption is uncertain.

Evaluating the merits of these alternative paradigms is difficult. I review some previous attempts to do so empirically, and present my own pilot study which evaluates the usability of two such models, transactional memory and the Actor model. Volunteer subjects solved a simple programming problem under controlled conditions with a variant of Java supporting one of these programming models, and under a control condition using the standard Java threading framework.

The results of this experiment were inconclusive. Not only did I see no differences in objective metrics between subjects' performance in the different parallel conditions, but the three common measures I observed did not even correlate with each other. Given this result, the many confounding factors in such experiments, and the similarly weak findings of many subsequent studies, I conclude that controlled quantitative studies are too weak a tool to guide the development of parallel programming systems. Our understanding of the cognition of programming is too primitive, and we should make peace with the fact that programming language design is like industrial design. Just because the difference between good and bad design is difficult to formalise does not mean that it does not exist, but good design must rely heavily on the designer's taste.

1.2.2 Parallelising Software Transactional Memory

In Chapter 3, I consider one programming paradigm which is generally regarded as superior in usability to classic multithreading: transactional memory. Until dedicated hardware for transactional memory (HTM) is available, transactional programs must rely on software transactional memory (STM) runtimes.

However, because it requires instrumenting every read and write during a transaction, STM suffers performance overheads so severe that they generally overwhelm its benefits. This is not often stated explicitly in the STM literature, but studies indicate serial overheads so large that they are difficult or impossible to "win back" by scaling up real-world applications – even if synthetic parallel benchmarks can just about pull it off.

I propose a scheme for using the resources we have (increasing numbers of cores) to preserve our critically limited resource (serial performance). I pair each application thread with a separate "helper" core, which performs the STM bookkeeping operations. This allows the application thread to execute with much lower overheads.

I evaluate this scheme by constructing a model of a multi-core processor with ideal on-chip communication, using off-the-shelf soft cores on an FPGA. I demonstrate that my scheme can substantially reduce STM overheads. I also investigate the effect of communication costs on the performance of this system. I conclude that the transmission latency is of little import, but the cost of transmitting and receiving messages is critical. Unfortunately, current cache-coherent multi-core processors transfer cache lines synchronously from one core to another, imposing communication costs this scheme could not withstand.

1.2.3 Remote Stores for Inter-Processor Communication

In Chapter 4, I address the problem of explicitly communicating between cores on a multi-core chip. Although these cores are connected directly, and capable of passing messages to one another on the hardware level, this capability is not exposed to software. Instead, threads must communicate by reading and writing shared memory locations. The cache coherence system will transfer data from the cache of the core that wrote it into that of a core reading it, but the process is synchronous and very slow. This makes explicit communication between cores on a multi-core chip very expensive, which in turn limits our ability to coordinate fine-grained parallel programs.

I review a number of proposed architectural modifications which aim to provide reasonable communication performance in multi-core processors. With one exception, no scheme proposed thus far can provide low-cost communication while remaining virtualisable between processes in a time-sharing operating system. The singular exception, HaQu, involves a highly complex co-processor and supports only a single communication pattern, the FIFO queue.

I describe the asynchronous remote store instruction, an architectural modification for high-performance, general-purpose, virtualisable communication. The extension comprises a store instruction which is issued by one processor, but completed asynchronously by another processor into its own cache. This prevents cache contention, allowing high-performance inter-core communication while remaining fully virtualisable. The remote store offers an intuitively comprehensible relaxation of memory ordering, while remaining safe to use with even existing, unmodified operating systems.

I evaluate the remote store extension by implementing several patterns of parallel communication. Remote stores yield substantially superior performance to state-of-the-art cache-coherence-based algorithms for FIFOs and barriers. I also quantify the increased feasibility of fine-grained parallelism with remote stores, using a variably-grained numerical benchmark. I also demonstrate the use of remote stores to parallelise a real-world application: concurrent dynamic analysis.

1.3 Contributions

- In Chapter 2, I present a pilot study into the usability of two alternative parallel programming paradigms, transactional memory and the Actor model. This was one of the first empirical studies of programming models for "commodity" parallelism, as opposed to the high-performance computing systems on which most previous work was focused.
- I present the (inconclusive) results of this study, and document the lack of consistency between popular measures of programming effort. Subsequent experiments by other researchers have found similarly weak results, and I conclude that such controlled quantitative user studies are too weak a tool usefully to inform the design of parallel programming systems.
- In Chapter 3, I propose a scheme for parallelising the instrumentation overhead of software transactional memory (STM). This instrumentation overhead is currently prohibitive, but I show that given sufficiently fast inter-core communication, STM overheads can be cut dramatically.
- I investigate the sensitivity of this parallel STM scheme to different costs of communication, separating send, receive and transmission delays. I find the scheme extremely sensitive to operation costs, but tolerant of large delivery delays – the very opposite of the trade-off provided by today's cache-coherent multi-core processors.
- In Chapter 4, I propose the asynchronous remote store instruction, a simple architectural extension to allow high-performance communication between cores on a cache-coherent multi-core processor. Unlike any other on-chip communication mechanism, it offers high performance, while remaining virtualisable by existing time-sharing operating systems, and without mandating a particular pattern of communication.
- I implement some popular patterns of parallel communication using remote stores, and demonstrate substantial improvements in performance. I quantify the improved feasibility of fine-grained parallelism with a variably-grained benchmark, and demonstrate effective concurrent dynamic analysis.
- I use remote stores to implement my parallel STM scheme on a simulated processor, and demonstrate that the scheme is indeed viable with the reduced communication costs offered by the remote store mechanism.

1.4 Prior Publication

- The pilot study described in Chapter 2 was published in 2009 as *Empirically Investigating Parallel Programming Paradigms: A Null Result* [86].
- A preliminary presentation of the remote store mechanism described in Chapter 4 was published in 2012 as *Asynchronous Remote Stores for Inter-Processor Communication* [87].

Chapter 2

Parallel Programming Practice and Usability

2.1 Introduction

The advent of multi-core processors has refocused attention on the cognitive challenges of parallel programming. As single-core performance reaches a plateau, programs do not automatically increase in performance when run on new hardware. Instead, programs must run in parallel if they are to make use of new hardware resources. The parallelism required to exploit new hardware fully is growing exponentially with each generation, as Moore's law gives us more cores rather than serial performance.

But there is a wide consensus that parallel programming is cognitively difficult, and no consensus on the correct response. In this chapter, I will review practices and paradigms of parallel programming, and present a pilot study into the relative usability of three of these paradigms. Although this study was ultimately inconclusive, it and related work illustrate the limitations of scientific analysis of such a complex cognitive task.

In Section 2.2, I discuss trends in industrial practice. In Section 2.3, I review some alternative paradigms which may be more suited for managing concurrency than the current lock-based model. In Section 2.4, I review some experiments which empirically investigate the usability of parallel programming models.

I introduce my own work in Section 2.5. In Section 2.6, I describe the experiment, and present the results in Section 2.7. The results are inconclusive, and in Section 2.8, I discuss the specific implications for the design of such experiments, and the general implications for the design of parallel programming systems. I conclude in Section 2.9.

2.2 Industry Practice in Parallel Programming

Most industrially widespread languages provide a multithreading system which allows multiple threads of control within the same memory space. The threading system typically provides mutual-exclusion locks and condition variables for concurrency control. Communication between threads occurs implicitly, through modification of shared memory locations. Examples include C's `pthreads` API, and Java and .NET's built-in threading support. When industrial programmers talk about "multithreading", they generally mean these APIs. These predate the transition to multi-core, and were already in use as an organisational tool.

This model is widely criticised for its usability problems [126, 81]. Implicit communication through shared memory is so awkward to reason about that Intel felt the need to strengthen the ordering guarantees on its multi-core processors, at the expense of performance, to relieve a little of the burden [101]. Some have gone so far as to say that most programmers cannot handle multi-core at all [104].

2.2.1 Avoiding Concurrency

Patient: Doctor, Doctor – it hurts when I do this.
Doctor: Then don't do that!

A first, and entirely reasonable, response is to avoid parallel programming entirely, or at least use no more multithreading than we were before multi-core arrived. We have no moral obligation to squeeze every drop of performance from our hardware. Indeed, the historical interaction between processing power and programming usability has been quite the opposite.

For the last ten to twenty years, we have spent increased microprocessor performance on making programmers' lives easier. Where previously most applications were written in fast but challenging languages such as C or C++, we now mostly use type-safe, garbage-collected, late-binding languages such as Java, C# and Python. The rise of the Android operating system shows that even power-constrained mobile devices can afford the luxury of a type-safe, garbage-collected virtual machine. An increasing number of applications are accessed through a web browser – itself effectively a large virtual machine, with a complex constraint-based graphical interface system and a dynamic language runtime.

We have paid a performance penalty for these powerful tools. In return, we enjoy easier, safer and more portable programming systems.

To expect programmers to expend extra effort to run their applications on multiple cores, then, is to put the cart before the horse. For most applications, speed is not the terminal goal, but is instead freely sacrificed in favour of easier or more rapid development. Where performance increases are required, partially reversing this tradeoff is often easier than parallelisation.

Even in applications heavily geared towards performance, the difficulty of parallel programming today makes it a less attractive target than other optimisations. For example, the Gecko web browser engine, which forms the core of the Firefox browser, is performance-critical. In an attempt to boost its speed, developers have rewritten its Javascript compiler three times since 2009 [42, 16, 88]. And yet, no serious attempt has been made to parallelise the Gecko engine, even coarsely – it would just be too much work. If we are to make use of multiple cores, even in performance-critical applications, parallel programming must first become easier and less dangerous.

2.2.2 Natural Parallelism

Google's Chrome browser, however, has found a way to exploit embarrassing parallelism, by placing the browser engine for each tab into a separate process. These processes communicate only sparsely, and with a single controller. This avoids the problems of complex multithreaded code, while effectively exploiting multi-core hardware.

This pattern is common in server applications. Some of the world's most popular websites are served by languages, such as Python and Ruby, whose canonical runtimes do not support parallel execution. Instead, like Chrome, server frameworks execute several instances of the program in parallel, isolated processes. These processes communicate – sparsely and explicitly – with a central coordinator which distributes HTTP requests, and with a central database which handles concurrent modification of shared data.

This approach is particularly well suited to server applications, which must already scale across multiple servers. A program which scales to multiple computers will have no problem exploiting a single, multi-core computer.

Where such natural parallelism is available, developers will take advantage of it. But there are many applications where such low-hanging fruit is not available, or incomplete. For example, the utility of Chrome's natural parallelism is limited: the most prominent criterion for browser performance is the responsiveness of the foreground page. And the Webkit renderer, like Gecko, is single-threaded.

2.2.3 Concurrency for Convenience

Of course, shared-memory multithreading has long been used for organisational rather than performance purposes, since well before the arrival of multi-core processors. Multithreading allows one thread to perform long-running tasks, such as blocking I/O or CPU-intensive computation, without pausing the rest of the program. While it is not free from controversy [100], multithreading is frequently used to divide up tasks within a program. For example, the windowing toolkits for Java and .NET confine user interface manipulation to a separate thread, allowing UIs to remain responsive during computations by the rest of the program.

Applications which exhibit this sort of "organisational multithreading" will benefit from multi-core processors, but only to a limited extent. A program's user interface thread, for example, is probably not a significant computational burden, and running it on its own processor core will not dramatically improve overall performance.

2.2.4 Parallelism for Performance

Relatively few applications use multithreaded parallelism for performance [15]. Video games are one example [5]. Game engines are complicated, heterogenous, performance-critical code, and their developers are running headlong into the usability problems of standard multithreading [127]. Image and video manipulation is another compute-bound example, but these algorithms' relatively small kernels lend themselves to more straightforward parallelisation, and/or the use of GPU processors.

In the late 1990s and early 2000s, it was hoped that standardised parallel skeletons [25, 48] would deal with the coordination of parallel algorithms, leaving programmers to write only the sequential kernels. The most popular example of this pattern is the MapReduce framework first championed by Google [34]. However, although MapReduce is a popular tool for distributed computing, neither it nor other standard skeletons are widely used for running parallel applications on a single machine.

High-Performance Computing

I have not yet discussed high-performance computing (HPC) systems. Although today's HPC machines are highly parallel, they are primarily distributed systems with highly non-uniform memory access. This contrasts with multi-core processors, whose parallelism is usually within a single chip. Even HPC systems with a single address space, such as Cray's XMT architecture [22], feature a substantially different programming model to commodity multi-core chips.

In addition, the HPC ecosystem differs substantially from the rest of the industry. It is typically more research-oriented, and deals almost exclusively with compute-bound numerical applications, with which the field has many decades' experience. Only recently are organisations such as DARPA beginning to address the usability issues of parallel programming, with initiatives like the High Productivity Computing Systems project [51], which funded some of the more recent work described in Section 2.4.

2.3 Alternative Paradigms

"Standard" multithreading, in which threads read and write a memory space shared between them, has come in for severe criticism [126, 81]. The central problem with this model is its huge amount of intrinsic nondeterminism. This stems from the large number of possible interleavings between threads [81]. Each time a thread reads or writes a shared memory location, it creates another point where its execution might depend on the ordering of that operation with respect to other threads. This combinatorial explosion of possible orderings makes it difficult for programmers to reason about, detect or eliminate ordering-dependent errors such as deadlock or race conditions.

Several alternatives have been proposed [120], many of them predating the multi-core transition. They range from small tweaks to a complete reimagining of the programming paradigm, but they have a common goal. Each seeks to reduce the amount of nondeterminism the programmer must reason about.

A thread's execution *between* points of possible interaction with other threads is deterministic, just like that of a sequential program. In the standard threading model, each read or write of mutable shared data is a possible interaction. In order to tame the resulting nondeterminism, alternative models must manage, reduce or eliminate either the sharing of memory locations or their mutability. This usually requires intrusive changes to the programming language, which may be responsible for the low uptake of these models.

2.3.1 Same Paradigm, Better Control

These goals can be pursued without changing the threading model. For example, immutable objects are increasingly popular in commercial practice and pedagogy of languages like Java and C#. An immutable object can safely be shared between threads without inducing nondeterminism. Likewise, the use of common components such as queues, futures and atomic data structures allows nondeterminism to be contained or eliminated.

As it does not affect the language or programming model, this approach can be supported as a library for a language supporting the standard model. Examples include Intel's Threading Building Blocks [135] for C++, Java's `java.util.concurrent` [80], and Microsoft's Parallel Extensions for .NET [84].

The concurrency support added to the C++ language in C++11 is partially of this form: it adds primitives such as futures, mutexes and threads themselves to the standard library. In addition, C++11 standardises the memory-access ordering model and atomic memory operations, which were previously a matter for negotiation between the programmer, compiler and CPU architecture.

One could even include OpenMP [28] in this category. Although it requires a modified C compiler, the OpenMP compiler provides syntactic sugar rather than fundamentally changing the programming model. A framework like OpenMP could be implemented with C macros and function calls, without changing its fundamental character.

However, reliance on such “best practices” can be fragile: they work well when followed, but they are all too easy to bypass. One need only accidentally share a pointer to a mutable object, and the Pandora’s box of unpredictable concurrency errors is reopened. In this respect, this approach is an incremental improvement upon careful use of locking: somewhat easier to get right, but just as dangerous when one makes a mistake.

2.3.2 Deterministic Parallelism

Another appealing idea is to confine existing shared-memory imperative languages to guarantee that their results are deterministic. This is known as “deterministic parallelism” or “determinism by default” [17] – not to be confused with deterministic program behaviour in general, which is a goal pursued by users of all parallel frameworks.

This can be as simple (from the programmer’s point of view) as using an automatic parallelising compiler or speculation to parallelise sequential code – or as complicated as using a statically verified subset of Java which can only express deterministic parallelism [17]. The former approach has seen a “lack of success” [132] of parallelising compilers due to “widely-acknowledged...limited effectiveness” [118], and software speculation has experienced the same problem of serial overhead as STM (see Chapter 3). The latter has not seen much uptake at all, possibly due to the strict limitations it imposes.

2.3.3 Functional Programming

Functional languages enforce the use of immutable data structures. This culls most of the nondeterminism experienced in the standard threading model. Most functional languages do provide facilities for mutable pointer cells, such as ML’s `ref` objects, but they are deliberately awkward and used sparingly. They frequently point to immutable data structures. For example, a mutable queue in ML might contain only one word of mutable storage, pointing to a pair of immutable linked lists representing the queue’s state. While this design would be possible in Java, the language encourages more mutation-heavy designs such as array-based queues or mutable linked lists.

Functional programming therefore constrains mutability to a small number of locations, accessed infrequently. Among other things, this means that functional languages can afford sophisticated contention control mechanisms which could not practically be applied to the standard model. For example, the overheads of software transactional memory (STM) are large, making current implementations infeasible for most imperative programs (see Chapter 3). However, because access to mutable locations is so infrequent in functional languages, the Haskell and Clojure STM implementations provide acceptable performance. Even Clojure’s closest equivalent to ML’s `ref`, the `atom`, uses an atomic compare-and-swap operation for every update in order to preserve consistency.

Advocates have long touted the benefits of functional programming, even for sequential code [9], but its uptake remains low. Concurrency, however, may make the case for functional programming significantly more compelling.

2.3.4 Pure Message Passing

In a pure message-passing system, each thread of control has its own private memory, and threads communicate by sending immutable messages to other threads. Threads process incoming messages one at a time, sequentially. This prevents accidental communication, and confines all nondeterminacy to the order in which messages are received. Between message-receive operations, each thread runs deterministically.

Two of the most popular formalisms for message-passing communication are CSP [65] and the Actor model [64]. CSP heavily influenced the design of the Occam language and the transputer parallel computer architecture [70], which has now been reborn as XMOS [90]. The Actor model is most famously embodied with syntactic support and enforcement in the Erlang language, but has been implemented in other languages as well.

Scala provides library support for actors, but does not enforce isolation. Actors may communicate via other means than messages, and there is no mechanism to ensure that messages are immutable. Relying on the programmer to provide isolation may reduce the benefit of this model.

By contrast, Kilim [124] implements an isolated Actor model in Java, by modifying the language. Isolation is enforced statically, by ensuring that message objects are either immutable or owned by a single actor at a time. Because Java does not provide facilities for enforcing such constraints, Kilim code must be precompiled by a "weaver" before it is run, using information from source-code annotations.

Message passing is a natural fit for distributed systems, which do not share a single memory. MPI [122] provides message-passing on top of C, and is used widely in high-performance computing. MPI messages must explicitly be marshalled and unmarshalled into buffers suitable for passing across a network, as C has no native facilities for doing so. This makes MPI somewhat cumbersome to use, and makes it much less attractive for fine-grained or within-system communication.

IBM's X10 language [115] augments message passing with syntactic support and a global address space, while still enforcing isolation. Each object resides in an single "place", of which there may be an arbitrary number. Immutable value types and object properties can be passed freely between places, but only code running in the same place as a mutable object can change it. Instead of marshalling and unmarshalling messages, closures can be executed asynchronously in any place. "Futures" and `finish` blocks allow the calling code to synchronise on the result of these computations. Within a place, nondeterminism is controlled with an `atomic` keyword, which has the semantics of a single mutual-exclusion lock for each place. X10 thus achieves many of the benefits of Actor-style message passing, while remaining very similar to shared-memory languages such as Java, on which its syntax is based. X10 is marketed to the HPC community, and has not seen much adoption in general-purpose computing.

2.3.5 Transactional Memory

Transactional memory (TM) retains the shared, mutable memory of the standard threading model. But whereas the standard model uses mutual-exclusion locks to ensure that code executes atomically, and relies upon the programmer to avoid deadlocking or breaking atomicity, transactional memory performs these functions automatically.

The programmer marks sections of code as atomic, and that code executes as a transaction. A runtime system ensures serialisability – that is, that the result of executing a series of transactions across multiple threads is the same as executing each of them sequentially, in some order or another. This corresponds to the first three of the ACID criteria: atomicity, consistency and isolation [57]. As the effects of a transaction are undecidable *a priori*, the runtime system must execute transactions optimistically, recording each read and write and detecting conflicts dynamically.

The semantics of transactional memory are the subject of some debate [30]. For most, the ideal is strong isolation, in which intermediate values in tentative transactions are not visible to non-transactional code, and conflicts between transactional and non-transactional code are detected correctly. While hardware transactional memory (HTM) solutions achieve this, the overheads of software transactional memory (STM) make it difficult to achieve. "Single Global Lock" semantics [92] are slightly more relaxed: they allow non-transactional code to observe intermediate values of successful transactions, and therefore do not require monitoring or isolation of non-transactional code.

Achieving single-global-lock semantics, however, requires correctly handling "privatisation", in which non-transactional code operates on data which is no longer shared thanks to a recently committed transaction [123]. Care must be taken to avoid interference between these non-transactional accesses and concurrent transactions which have not yet caught up with the fact that this data is no longer shared. Waiting for these transactions to catch up is costly, and so some systems perform safe privatisation only when explicitly requested by the programmer [40]. This leaves room for subtle errors.

Given acceptable semantics, though, transactional memory is an appealing paradigm, and requires very little change from the existing lock-based standard model. Although implementations are publicly available, the principal obstacle to TM adoption is that no hardware implementation is yet available, and software implementations suffer severe performance problems. Chapter 3 is dedicated to the latter problem.

2.4 Empirically Investigating Paradigms of Parallel Programming

With so many contenders to replace the current standard model of threading, each struggling for critical mass, we want to compare the merits of different approaches. The design of concurrency mechanisms for programming languages is a usability problem – a serious one, and painfully unresolved.

Like most programming language features, concurrency paradigms have historically been built on a hunch, and evaluated by anecdote and holy war. But since the 1990s, researchers have advocated the use of empirical techniques to evaluate objectively the usability of concurrent programming systems [128].

2.4.1 High-Performance Computing

Before the multi-core transition, such experiments targeted high-performance computing systems, as they exhibited the greatest demand for parallelism.

Szafron and Schaeffer [128] evaluated a proposed parallel programming system by posing a parallel transitive closure problem to fifteen novice parallel programmers. This study used several approaches to measure productivity: the authors measured time to correct solution, code size, and the performance of the resulting program. They also counted invocations of the editor, compiler and program itself.

Hochstein et al. [67] taught a class of students either MPI or a shared-memory C variant for a research computer architecture. They found MPI to involve significantly more effort. The same group compared MPI with another shared-memory C variant, OpenMP, and again found the shared-memory system easier to use [66]. As VanderWiel et al. [133] noted over a decade earlier, most of the extra effort is likely to do with manually packing and unpacking message buffers rather than message-passing *per se*. This makes MPI-based experiments much less relevant to the modern message-passing techniques reviewed in Section 2.3.

A perennial problem with such studies is the small number of subjects. High-performance computing, in particular, is a relatively niche field. Most research in this field is conducted on students, and class sizes limit the sample size. A multi-university collaboration to investigate HPC usability was launched in 2005 [68], but the results were fragmented.

In 2006, Ebcioglu et al. [41] evaluated the X10 language [115], which formed part of IBM's submission for the DARPA High Productivity Computing project [51]. They compared X10 with two C-based systems, MPI [122] and UPC [43]. The experiment was large and resource-intensive, involving 27 subjects and taking five days, and employed some very sophisticated observational methods [33]. Nevertheless, the results did not reach statistical significance, showing only indicative evidence in favour of X10 over the two other languages.

A number of other studies have evaluated the usability of parallel systems by implementing larger programs, often existing benchmark problems, and discussing the experience. The evaluation is qualitative, based on the subjective judgement of a very small number of programmers – typically just one. Quantitative measurements are commonly limited to source code metrics, which we will see to be problematic in Section 2.7.3. However, this approach can investigate larger and more diverse programs than a multi-subject controlled experiment.

In this vein, Cantonnet et al. [18] reimplemented some of the NAS benchmark suite to evaluate Unified Parallel C. Chamberlain et al. [21] compared Fortran variants, Single-Assignment C and ZPL on a single NAS benchmark. Vanderwiel et al. [133] compared several C-based languages and High-Performance Fortran over a variety of banchmarks.

While their experimental techniques are useful to observe, experiments in high-performance computing are not enormously useful in evaluating the pressing problems facing general-purpose computing today. Modern commodity systems do not have distributed memory with large inter-node communication costs, and the emerging paradigms presented in Section 2.3 make heavy use of language features not available in Fortran or C.

Another critical problem is that these studies evaluate complete – and quite different – programming systems, rather than the principles they embody. For example, IBM's evaluation suggested that X10 is largely easier to use than MPI or UPC. But is that a result of its memory safety and garbage collection, its more expressive types, or its innovative approach to distributed parallelism? We cannot say.

2.4.2 Commodity Hardware

As commodity machines have become multi-core, more recent work has addressed parallel programming for general-purpose hardware and programming systems. All of the work cited here post-dates the pilot study described later in this chapter [86], and most cites it.

Sadowski and Shewmaker [113] argue for more empirical testing of the usability of parallel programming systems, and identify the critical need for accurate metrics of programmer effort.

Pankratius et al. [102] compare the use of two shared-memory C systems, `pthreads` and OpenMP, to parallelise `bzip2` in a student programming competition. In another iteration of the same competition, the same group compares lock-based and transactional programming [103]. They find suggestive evidence that the transactional memory teams produced clearer code, faster and with fewer errors, although they struggled to optimise these programs.

Nanz et al. [98] compare Java and an entirely different concurrent language, SCOOP, with a 67-strong student cohort. They attempt to avoid instructor-induced bias with self-study material, then administer a written test. They evaluate program comprehension, error-finding by inspection, and correctness in a program-writing assignment. They find a statistically significant improvement in favour of SCOOP.

Rossbach et al. [111] evaluate transactional memory. They find that, although students subjectively rated coarse-grained locking as substantially easier than transactional memory, even coarse-grained lock-based programs contained more errors than transactional programs. They also found strong learning effects within each condition.

These studies gain statistical power from the widespread arrival of parallel programming. Like the HPC studies in the previous section, these experiments were all performed on students. As multi-core processors become more common, parallel programming is taught at a lower level, and so experimenters enjoy larger student cohorts for these classes. However, these experiments still struggle to reach statistical significance. Only one, Nanz et al. [98], succeeds – and they follow the pattern of comparing two entire programming systems, which makes it difficult to attribute the improvement to any particular difference in the model of parallelism.

2.5 A Pilot Study

In the rest of this chapter, I present a pilot study into the relative usability of parallel programming paradigms. I compare the Actor model, transactional memory (TM), and standard shared-memory threading and locking ("SMTL"). This experiment was published in 2009 [86], and was the first empirical user study of parallel programming paradigms for multi-core hardware.

In Section 2.6, I describe an experiment which directly measures and compares programmer performance when solving a problem using different parallel paradigms. In contrast with many of the experiments described above, subjects solve the same problem, in the same language, varying only the concurrency paradigm.

In Section 2.7, I present the results of this experiment, and show them to be inconclusive. I demonstrate the inconsistency between common measures of programmer effort which are used in much of the literature. I also observe a surprising bifurification in my subjects' performance.

In Section 2.8, I discuss this inconclusive result. I enumerate some possible causes, and in Section 2.9, I draw conclusions about the difficulty of using controlled empirical experiments to evaluate programmer productivity.

2.6 Materials and Methods

I tested three parallel paradigms: the Actor model, transactional memory, and standard shared-memory threading with locks (henceforth "SMTL"). I also tested subjects writing sequential code, as a positive control: it is generally agreed that sequential programming is substantially easier than SMTL, and any acceptably powerful study should show this effect clearly.

I provided all four programming models for the Java programming language. Java is a widely adopted language, taught in the Cambridge undergraduate curriculum, and provides a relatively uncontroversial baseline for this experiment.

Each subject solved the same problem twice: once with the standard (SMTL) Java threading model, and once with Actors, transactional memory, or no parallelism at all (the sequential condition). The

whole session took approximately 4 hours per subject. Scheduling was balanced, with half of the subjects assigned the SMTL condition first, and the other half assigned it second.

Each subject filled out a questionnaire before the experiment, to assess their level of experience and self-perception of skill. After each task, the subject filled out a questionnaire indicating the level of subjective difficulty and opinions of the concurrency model used.

During the experiment, subjects' screens were recorded, and a webcam recorded the subject to monitor off-computer events. A snapshot of the project directory was taken at 1-minute intervals, and instrumented tools logged each invocation of the compiler or run of the resulting program.

I measured two gross quantitative proxies of programmer effort: the time to complete the task (judged as the first correct solution of the provided dataset), and the number of non-comment lines of code in the final program.

2.6.1 Subjects

Seventeen subjects were recruited from the undergraduate (10) and graduate (7) student population of the University of Cambridge Computer Laboratory. Of these, eleven successfully completed both tasks.

2.6.2 Task

I chose to minimize variability, at the expense of significant learning effects, by using a single problem. This was an "unstructured grid" problem, in the classification of the View from Berkeley [8].

I presented a toy physics problem, modelling heat flow between identical blobs, connected by rods of varying conductivity. If the temperature of blob i at time t is represented as T_t^i, the temperature change due to a rod of conductivity k connecting blobs a and b is:

$$T_{n+1}^a = T_n^a + k(T_n^b - T_n^a)$$

A graphical example is shown in Figure 2.1.

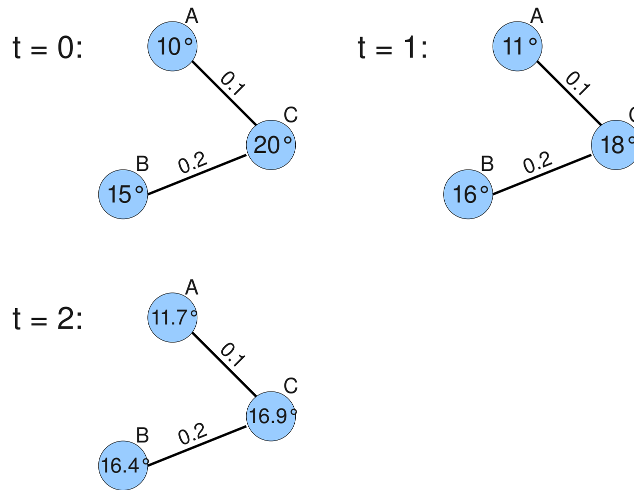


Figure 2.1: A worked example of the problem posed to experimental subjects.

To reduce the time subjects spent writing irrelevant I/O code, I provided them with code to load a sample data set into memory and pass it as arguments to a function call. I provided the inputs in a deliberately un-useful data structure, and the written instructions instructed each subject to translate the data into their own choice of structure. I instructed the subjects to translate the solution back to the original un-useful format and call a function to check its correctness.

2.6.3 Implementing Different Paradigms

I aimed to replicate the user experience of programming in each paradigm, while keeping the underlying language as close as possible to idiomatic Java.

This involved trade-offs which made performance or scalability comparisons between different conditions impractical. This is a disadvantage, as scalability is the ultimate goal of all such parallel programming, but other studies have examined the scalability characteristics of different concurrency mechanisms, and I considered usability the more important target for the present study.

SMTL and sequential

Java's native concurrency model is based upon threads and mutual-exclusion locks, so standard Java was used for the SMTL and sequential conditions.

Transactional Memory

For the transactional memory condition, I used an existing transactional memory system for Java, Deuce [76]. Deuce modifies Java bytecode during loading, transforming methods annotated with `@Atomic` into transactions. Figure 2.2 gives a simple example of the Deuce API.

```
class TransactionalCounter {
  private int x;

  public @Atomic increment() {
    x = x + 2;
  }
}
```

Figure 2.2: Example code using the Deuce transactional memory system.

However, at the time of this study, Deuce did not instrument classes in the Java standard library, so it could not be used with idiomatic Java. Instead, I enabled Deuce's "single global lock" mode, which makes all atomic methods mutually exclusive. This preserves the semantics of transactional memory, but prevents us from evaluating scalability.

Actor Model

Implementing the actor model for Java presented a challenge. Java assumes mutable shared memory throughout its design, whereas actors require disjoint memories and messages which the sender cannot change after they are sent.

One of the touted advantages of the actor model is that enforcing actor isolation may prevent the user from making certain sorts of mistake. Enforced isolation is therefore necessary realistically to model the desired user experience.

I considered Kilim [124], an implementation of the Actor model in Java with an annotation-based type system to enforce actor isolation. Kilim enforces a complete transfer of ownership of mutable objects sent in messages, so that only one actor can refer to a mutable object at any one time. However, this is a substantial departure from idiomatic Java. In addition, no version of Kilim including this type system is publicly available, and the author warned that his pre-release version was unreliable. I therefore concluded that Kilim was not suitable for this experiment.

I also considered using a run- or compile-time system to ensure that only immutable objects – objects whose fields are all `final`, and point to similarly immutable objects – could be passed in messages. However, Java's standard library is built with mutable objects, and I judged that forbidding the use of standard library objects would be a major departure from idiomatic Java. I therefore rejected this option as well.

Instead, I implemented a reflection-based runtime system, using deep copies for isolation. In my implementation, each actor is represented by an object, to which no reference is held by any other actor. Other actors interact only with a wrapper class, `Actor`, through which they can send messages to this hidden object. Messages are named with strings, and handled by methods of the same name, so an `"add"`

```java
// AddingActor.java
public class AddingActor {
  private int x = 0;

  public void incrementBy(int y) {
    x += y;

    System.out.println("x = " + x);
  }
}
```

```java
// Main.java
public class Main {
  public static void main(String[] args) {
    Actor a = new Actor("AddingActor");
    a.send("incrementBy", 2);
    a.send("incrementBy", 5);
  }
}
```

Figure 2.3: An example demonstrating my Actor API, which enforces isolation between actors.

message is handled asynchronously by the `add()` method. Figure 2.3 gives an example of this framework in use.

All arguments to messages and constructors are deep-copied, using the Java serialization mechanism. This enforces isolation, by preventing multiple actors from obtaining pointers to the same mutable object. (Isolation can still be violated, by use of `static` fields, but I decided that this could be forbidden verbally without seriously affecting coding style.)

Of course, this isolation comes at a significant performance cost, making scalability analysis impractical. This grafting of isolation onto a shared-memory language is not elegant, but aims to emulate the user experience of pure message passing.

2.7 Results

The results of this experiment were inconclusive. In Section 2.7.1, we see no significant difference in any objective measurement between the four test conditions. The results of the subjective surveys in Section 2.7.2 indicate a preference for the newer mechanisms, although there are good reasons to be suspicious of such results.

More alarmingly, in Section 2.7.3, we see that the three measures of effort used in this study – each of which has been used as the foundation for previous work in this field – were almost entirely uncorrelated.

Finally, in Section 2.7.4 I document an unexpected phenomenon in the completion data, suggesting a bimodal distribution: Subjects either completed the first task within two hours, or could not within the entire four-hour session.

2.7.1 Objective Measures of Effort

Completion Times

Figure 2.4 displays the completion times for every trial on the same graph. There is great inter-subject variability in performance, and the aggregate data shows no significant difference between conditions. The only visible effect is that of sequence: subjects are faster when solving the problem for the second time than the first time. This is a statistically significant result: a paired (within-subjects) t test finds an improvement between first and second runs at $p \approx 0.04$.

We control for inter-subject variation in Figure 2.5 by plotting the relative times: that is, how much faster or slower did each subject complete their test condition, compared with the SMTL baseline? We see clearly that the within-subject learning effect dominates all other variation, but neither the Actor model nor transactional memory shows a difference from SMTL performance. The sequential condition (our positive control) shows a suggestive decrease in time taken, but this fails to reach significance.

Code Size

Figure 2.6 displays the number of non-comment lines of code in each solution, and Figure 2.7 shows the within-subject differences in size between conditions. We do not see the same learning effect between trials as we do when examining completion times. Again, even our positive control – the sequential condition – fails to show more than a suggestive decrease in code size.

Figure 2.4: Time taken by subjects to complete each condition.

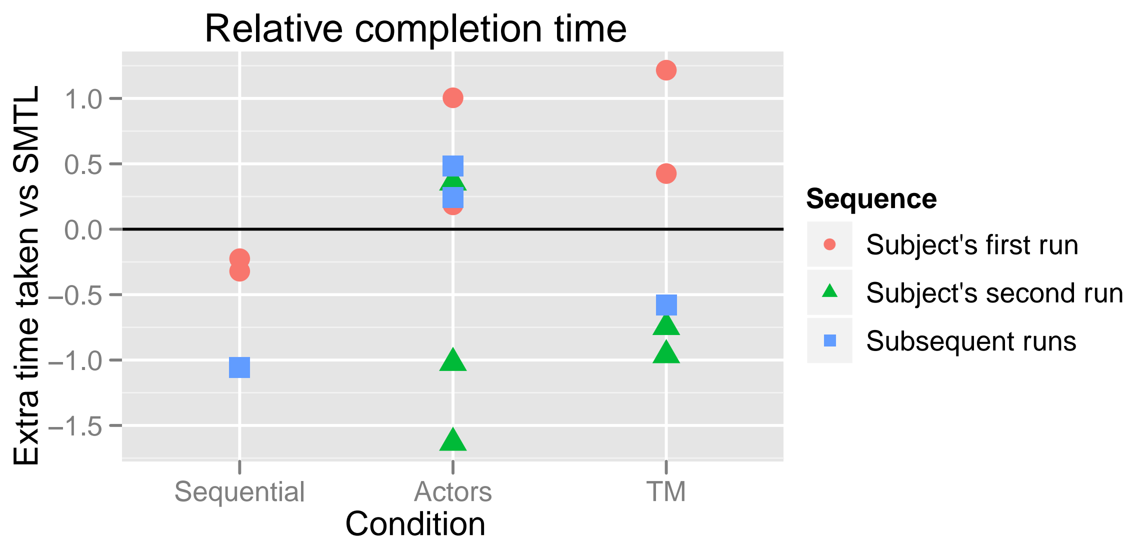


Figure 2.5: The increase or decrease in time taken, compared with the time taken by the same subject under the SMTL condition.

Figure 2.6: Non-comment lines of code in each solution.

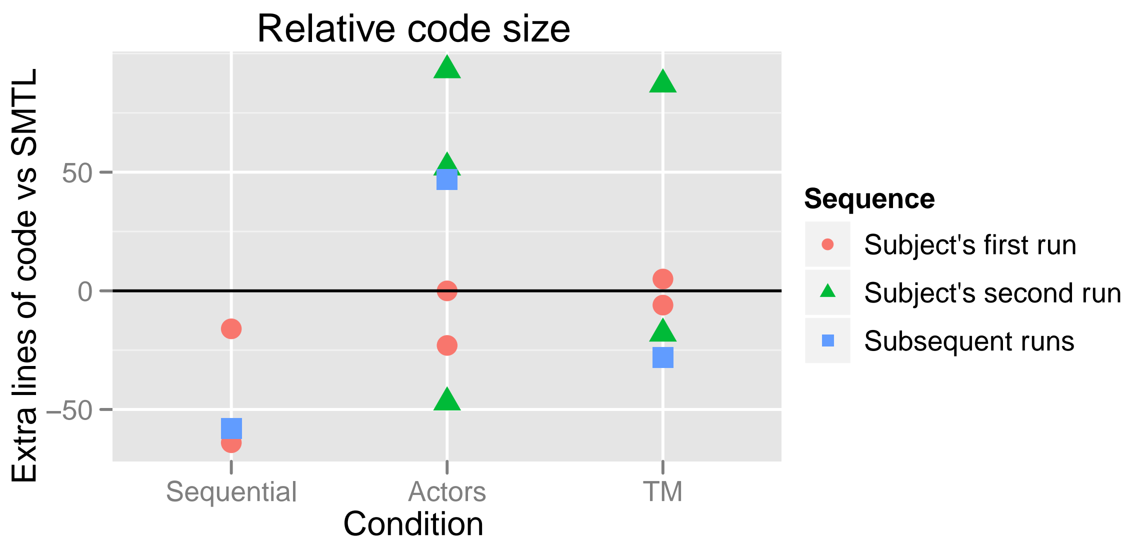


Figure 2.7: The increase or decrease in code size, compared with the same subject's code under the SMTL condition.

2.7.2 Subjective Reporting of Effort

After each task, subjects were given a questionnaire requesting their subjective impression of the task. As well as overall task difficulty, they were asked how the framework compared to writing sequential code and SMTL, and asked directly how easy they found using the framework.

Figure 2.8 presents the average of all these subjective ratings.

A within-subjects ANOVA finds that subjects tested on transactional memory significantly preferred it to SMTL ($p \approx 0.04$). Although subjects tested on the Actor model also appeared to prefer it to SMTL, this result did not reach the 5% significance level ($p \approx 0.07$).

However, we must approach these results with caution, because experimental subjects are liable to say what they think the experimenter wants to hear.

2.7.3 The Measurements Don't Even Agree

Each of these three metrics – time to completion, code size and subjective reporting – has been used in prior work as a proxy for programmer effort. Completion time is a common metric, and is sometimes asserted to be synonymous with effort [67, 68]. Other studies assert or imply that lines of code are equivalent to effort [21, 133]. Subjective impressions are of course the primary guidance used in when designing languages in the first place, but are also incorporated into empirical studies [110].

However, when we compare these measurements in the same experiment, we see that they do not even agree with each other. Figure 2.9 plots each measure against each of the others.

When analysed with a linear regression, we find that no measure is correlated with any other ($r^2 < 0.08$ and $p > 0.19$ in all cases).

This holds even if we control for the learning effect between first and second runs by adding sequence as a categorical variable to our linear model. (ANCOVA of first- and second-run data using sequence and measure A as independent variables, and measure B as the dependent variable; $p > 0.15$ for all pairs of measures).

Unless this lack of correlation is unique to this experiment, it presents a serious threat to the validity of any experiment which blindly uses one of these metrics.

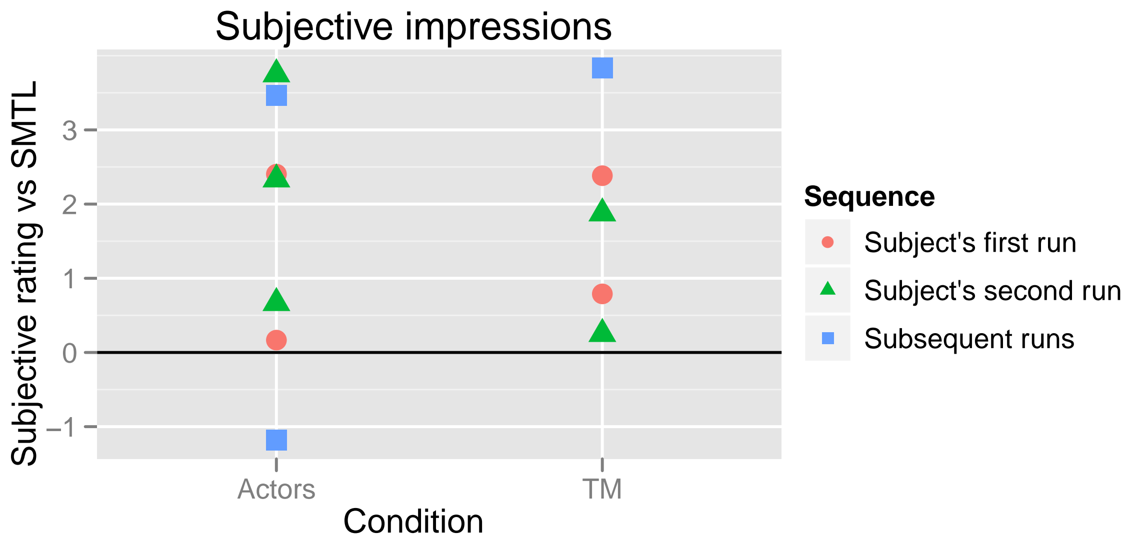


Figure 2.8: The average subjective ratings of subjects in each condition. Ratings are sign-normalised such that higher ratings are good, and presented as a difference from ratings in the SMTL condition. The sequential condition is omitted, as most of the survey questions were inapplicable to it. Subjects significantly preferred TM to SMTL ($p \approx 0.04$), and showed a suggestive preference for Actors over SMTL ($p \approx 0.07$).

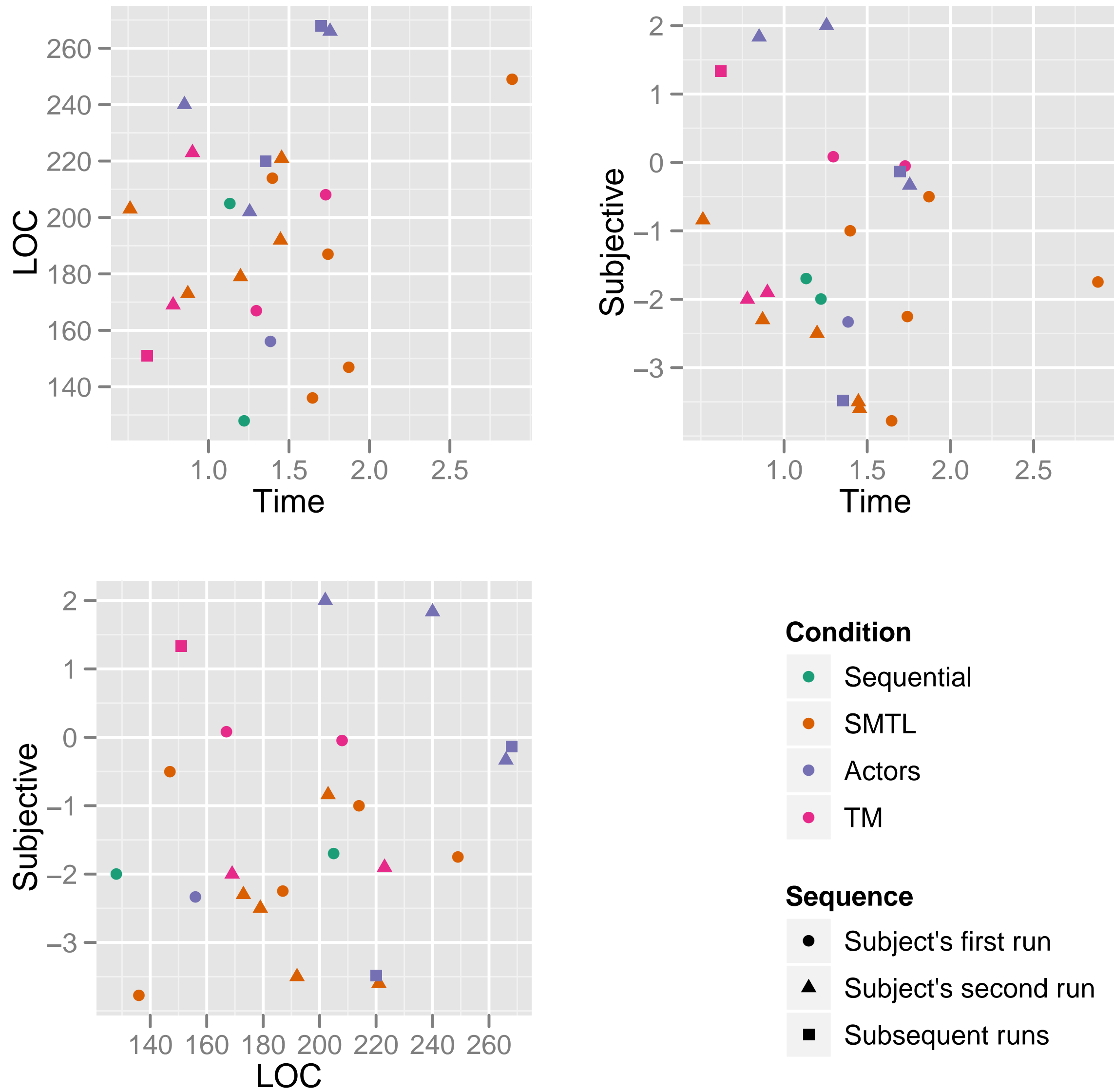


Figure 2.9: The three metrics used to evaluate programmer performance do not correlate with each other: $r^2 < 0.08$ and $p > 0.19$, for all pairs. $p > 0.15$ for all pairs when using ANCOVA to control for the learning effect between first and second run.

2.7.4 Bifuricated Results: Does the Camel Have Two Humps?

Unexpectedly, subjects either finished the first run within two hours, or could not finish it at all within the four-hour session. (There was only one exception, who completed the first task in just under three hours. He did not attempt the second.)

The distribution of completion times for all tasks is displayed in Figure 2.10. The solid bar represents subjects who did not complete a task at all. The yawning gap between these five subjects and the rest of their cohort suggests that the distribution is not continuous.

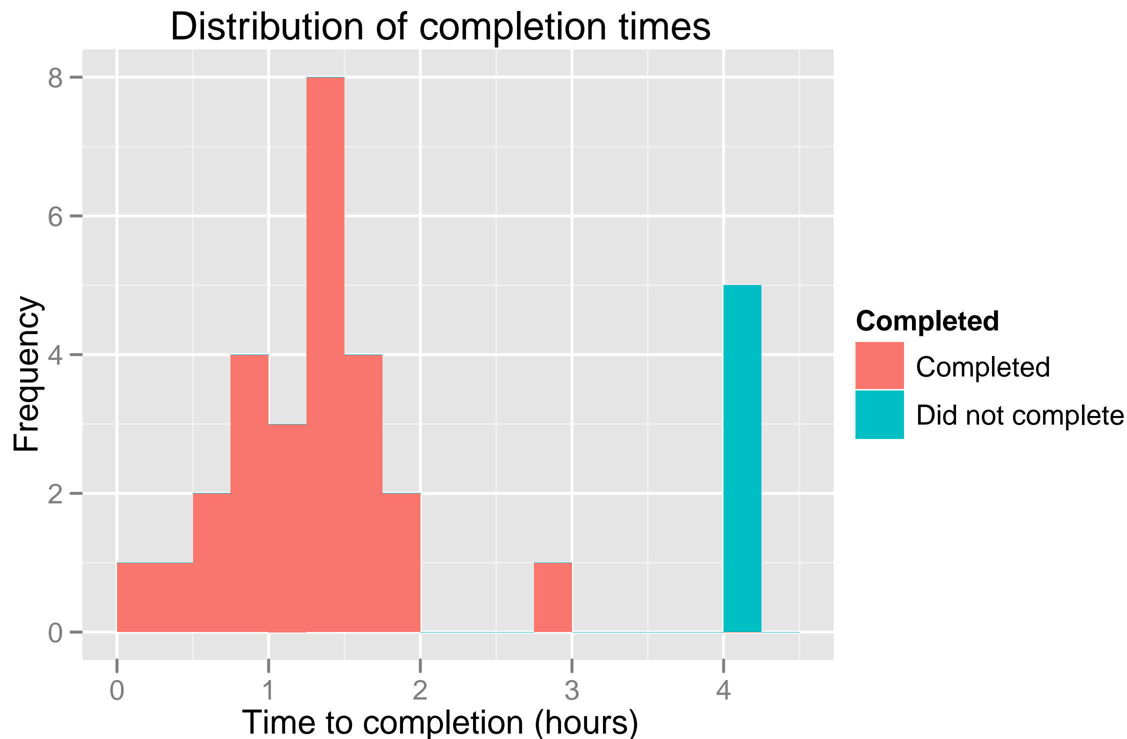


Figure 2.10: Aggregate completion times of every study. With one exception, subjects either completed the task within two hours or did not complete it within the four-hour session.

A bimodal distribution of programmer ability has been posited before [35], but this controversial suggestion distinguished between those who entirely "cannot learn" to program and those who can.

By contrast, these unsuccessful subjects wrote valid code, with control and data structures not obviously less complicated than their successful peers'. They did not give up, or stop debugging their programs, until stopped at the four-hour mark. Post-session interviews indicate that they correctly understood the problem. In short, the circumstantial evidence does not support the idea that these subjects were simply incompetent, gave up, or failed to understand the problem. This result, then, remains an intriguing mystery.

2.8 Discussion

This inconclusive result is disappointing, but also instructive.

Broadly, it illustrates the difficulties of empirical research into such a complicated phenomenon as programmer productivity. It also illustrates the power of many hard-to-avoid confounding factors, and the inconsistency of empirical measures of programmer productivity.

2.8.1 Possible explanations

I will now consider some confounding factors which might have caused this inconclusive result, even in the presence of large usability differences in the frameworks tested.

Weak metrics

Their dismal internal consistency suggests that the available metrics for programmer effort are not particularly powerful or reliable tools.

It is possible to construct more complicated measures, such as calculating the cyclomatic complexity of a program [91], or breaking down programmer time into meaningful subtasks [33]. However, it is difficult to see how these metrics have any more valid claim to represent programmer effort than the gross measures used here, especially when those gross metrics are so inconsistent.

Learning effects and subject variability

The learning effect between the two trials for each user greatly interfered with the results. However, designing a study such as this one inevitably puts the experimenter between a rock and a hard place.

If I had designed an experiment where each subject attempts only one task, under one condition, the results would have been swamped by inter-subject variation. It would take an impractical number of subjects to see any effect at all.

The design I actually chose controls for some subject variability – but far from all – at the expense of a short-term learning effect that ends up swamping the results.

Familiarity

Most programmers have experience with the standard threading model, and so come to this study with a substantial familiarity bias in favor of the SMTL condition. By contrast, few subjects had previous experience with transactional memory, and none with the Actor model. This study therefore captured the effort required to learn these models for the first time, as well as the effort of solving the problem.

This might be mitigated with a practice session before the study, in which users gained some experience with the unfamiliar model before attempting the task. That said, no length of practice session can eliminate the learning curve entirely, and familiarity is an ever-present confounding factor.

Toy Problem Syndrome

The task in this experiment was, necessarily, a small problem which could be solved in two hours and 200 lines of Java. Most solutions included a single, symmetrical concurrent kernel. By contrast, the usability problems of concurrency are often related to complexity, and the inability to hold the behavior of all threads in the programmer's head at once.

The proverbial "concurrency problem from hell" is an intermittent deadlock whose participants are distributed across 80,000 lines of code written by ten different people. Such situations are difficult to model experimentally.

It is a perennial problem that empirical studies of programming can only ever test toy problems, and it is difficult to imagine any experimental design which could sidestep it. Lengthening the task might have helped slightly, but recruiting volunteers for a four-hour study was difficult enough as it was. There is a reason that much of the work in this field is done by educators, at institutions where course credit may be awarded for participating in such experiments.

2.8.2 Additional threats to validity

Lest we think that these big problems are the only ones, a number of issues would have qualified even a significant positive result from this experiment.

Problem choice

The choice of problem greatly affects the suitability of different paradigms. There is unlikely to be One True Model which outperforms all others, all the time. Choice of a problem which can be more easily modelled with shared state, or message passing, could therefore have a significant effect on the results.

Controlling for this issue would require a large-scale study including representatives of, say, all the Berkeley Dwarfs [8], in an attempt to cover all common parallel computations.

Is creation representative?

Software spends almost all of its life being maintained, but this experiment observed only the initial creation of a program. Safe and easy modification – low *viscosity*, in the terms of the Cognitive Dimensions framework [52] – might even be more important than the ease with which a program is first written.

Limited frameworks

The frameworks used in this experiment lack features available in industrial implementations. For example, the Actor model implementation lacked multicast or scatter-gather support, which is available in frameworks such as MPI [122]. The SMTL condition explicitly prohibited the use of utilities such as synchronization barriers from the package `java.util.concurrent.*`. Subjects repeatedly re-implemented both barriers and scatter-gather patterns during the experiment.

As discussed in Section 2.3.1, the use of prefabricated patterns and libraries to perform communication and synchronisation can substantially improve usability and correctness.

Limitations of Actor simulation

Actor messages, unlike normal method calls, are only checked at run time, not at compile time. This may spuriously have reduced performance in that condition.

Indeed, the overall syntax of actor calls in my framework is a departure from idiomatic Java. In retrospect, it would have been possible to provide a more idiomatic experience. Figure 2.11 illustrates an alternative API, which could have been implemented with the use of a bytecode generation library. By expressing messages as method calls, this scheme allows the compiler to check message names and their argument types at compile time.

Unrealistic performance

The unrealistic performance of the experimental frameworks might have cause subjects to mis-optimize their programs. (This was not borne out by observations or discussions after experimental sessions; no subject so much as profiled their code for bottlenecks.)

Unrepresentative subjects

The students participating in this study may not be representative of professional programmers. This could go either way: they might be more flexible and open to new techniques, or less practiced at quickly getting the hang of unfamiliar tools.

Accidental misdirection

The awkward data structures used to provide inputs appear to have had an anchoring effect on the subjects. Several subjects largely copied the provided data structures instead of devising their own, despite emphatic instructions to the contrary.

```
// AddingActor.java
public class AddingActor extends Actor {
  private int x = 0;

  public void incrementBy(int y) {
    x += y;

    System.out.println("x = " + x);
  }
}
```

```
// Main.java
public class Main {
  public static void main(String[] args) {
    Actor<AddingActor> a =
        Actor.create(AddingActor.class);
    a.send.incrementBy(2);
    a.send.incrementBy(5);
  }
}
```

Figure 2.11: An illustration of an alternative Actor API, which could have addressed some limitations of the model actually used in the experiment.

2.9 Conclusion

I have considered the challenges posed to software development by the advent of multi-core processors. Our ability to exploit additional cores is limited by the cognitive difficulty of multithreaded programming. So far, mainstream software practice has relied on shared-memory threading and mutual-exclusion locks for concurrency management – or simply eschewed multithreading entirely.

Several alternative paradigms of parallel programming aim to change that. I have reviewed some of them, and then described a pilot study which attempted to investigate their relative merits in a controlled experiment. The study was inconclusive. Not only did large inter-subject variation and learning effects swamp the data, but the basic measures often used as proxies for programmer effort were almost entirely uncorrelated. This result, as well as the limited nature of similar work, underscores the relative unhelpfulness of empirical experimentation in the design of parallel programming systems.

Designing programming languages is comparable to visual or industrial design: it is a series of judgement calls about complex cognitive phenomena of which we have little formal understanding. There is good design and bad design, but it is very difficult to explain the difference rigorously. The best design work is done by individuals, using their own internal, informal, intuitive understanding of these cognitive phenomena – what we call "taste".

Our inability to communicate this understanding explicitly, and bad experiences of previous attempts, is why "designed by committee" is a pejorative term in the computer industry. Almost every popular programming language had only one or two designers, and many are still maintained by a single benevolent dictator.

This reliance on intuition contrasts with the design process in other engineering fields. When designing a bridge or aeroplane, whose constraints and operating environment are well understood, communication is easy enough to coordinate teams of dozens or hundreds of engineers.

This is not to say that empirical input is useless in informally-understood fields. Industrial designers create prototypes using their own design sense, but then show them to focus groups to gauge their reaction. Language designers often release prototypes to the internet seeking feedback, or even conduct qualitative user studies. But we cannot A/B test programming language features like Google does web design. The desired outcome is simply not that well defined, and testing is prohibitively large-scale and expensive.

Faced with the difficulty of testing individual features, the most convincing experiments in parallel programming usability have compared complete systems (see Section 2.4). But these experiments remain extraordinarily expensive and lack power, and entire systems are usually capable of competing for themselves in the marketplace of ideas.

One large-scale experiment of this type was done by IBM, to provide supporting evidence for the superior usability of the X10 HPC language [41]. They had a language which was convincingly superior to the C-based alternatives: it supported garbage collection, memory safety, and safe typing, each of which has been embraced as a boon to productivity by programmers across the industry. It offers sophisticated concurrency control, while supporting a strict superset of the concurrency features of competing C systems. And yet, when X10 was compared with these competing systems, it took a monumental effort to produce a purely indicative result which did not achieve statistical significance. It is hard to imagine that IBM would have invested such effort in large controlled experiments, if they had not been seeking to impress a single, large, grant-awarding customer. Had they been targeting industrial programmers, rather than high-performance computing in the public sector, they might have done better to emulate the release of Java, the language on which X10 was based. After James Gosling designed the language, with reference to his own design taste and informal feedback from his colleagues, Sun Microsystems put a working system into the marketplace and let the users decide.

It is true that, with a suitably composed hypothesis, even art may be subjected to quantitative study [39]. But for the most part, the design of programming systems, including parallel ones, must remain a matter of intuition, design and taste.

Chapter 3

Software Transactional Memory

3.1 Introduction

Transactional memory (TM) is an appealing programming model which aims to simplify the correct concurrent use of shared memory by multiple threads. Blocks of code which modify shared state execute as atomic transactions, and see an isolated view of the contents of memory. A runtime system detects conflicts between transactions, aborting and retrying transactions to maintain atomicity, consistency and isolation. TM offers the prospect of retaining the multi-threaded programming model, while automating the error-prone task of managing critical sections. TM provides an automatically managed, safe way to use implicit communication.

Hardware transactional memory (HTM) support is the most efficient way of implementing transactional memory. However, HTM is architecturally complex and not available in current commercial procesors. In any case, most practical HTM systems have limited capacity, and must be supplemented by a software system when they overflow [79, 32].

But software transactional memory (STM) runtimes have forbiddingly large overheads. Each potentially conflicting load or store operation must be instrumented to record it or check its validity. Even scalable multithreaded programs often require four or more processors in order to match the throughput of a single-threaded program without STM instrumentation. Programs with more scarce or irregular parallelism fare much worse, and may never exceed their uninstrumented sequential performance.

This problem appears unavoidable. Increasingly, STM research has avoided addressing it entirely, concentrating instead on scalability to large numbers of processors. Publications often supply little if any information about the absolute overhead of new STM algorithms. The implicit argument is that if scaling can be maintained, serial overheads do not matter, as we can use the Moore's-law bounty of more cores to "win back" poor serial performance.

Unfortunately, Amdahl's law doesn't work that way. Writing highly scalable programs is difficult, work is not always uniformly distributed between threads, and serial bottlenecks abound. Real programs often have limited parallelism – and as efforts to "transactify" real-world applications have shown [138, 45], this limited parallelism makes it difficult or impossible ever to overcome the overheads of STM. If STM demands that we write fine-grained parallel programs, and scale uniformly to at least eight threads before we gain any benefit from that effort, it forfeits its claim to usefully simplify parallel programming.

3.1.1 Contributions

I survey this research area, and suggest an alternative approach. Instead of attempting to reduce the overhead of STM instrumentation, I propose using a "helper threading" model to shift that overhead off the critical path, onto another core in a chip multi-processor. If the two cores can communicate at a sufficiently fine-grained level, an "application core" can offload its STM bookkeeping onto a "helper core", and execute the application thread at high speed with little serial overhead.

This trade-off uses an increased number of cores, of which Moore's law is giving us more every year, to preserve serial performance, which has now reached a plateau. Rather than use more processors in an attempt to outscale the STM overheads, I instead dedicate half of them to helping serial threads run faster. While both approaches use extra cores to overcome STM overheads, this helper-threading approach does not demand additional scalability from the underlying application. It therefore works with, rather than against, Amdahl's law.

In Section 3.2, I survey the history and challenges of software transactional memory.

In Section 3.3, I describe two simple STM algorithms, one validation-based and one invalidation-based. I discuss how they can be implemented by an application thread and a helper thread working in parallel.

In Section 3.4, I describe a model system constructed on an FPGA, which I use to evaluate this approach. I use off-the-shelf soft cores and memory-mapped FIFOs to simulate a chip multi-processor with high-speed inter-core communication.

In Section 3.5, I investigate the performance of these parallel STM systems on this hardware model. I find that parallelising the STM runtime significantly improves the serial performance of transactional code, without sacrificing scalability. I further investigate the remaining serial overhead, and utilisation of the helper thread.

Finally, I investigate the inter-core communication performance necessary to support this scheme. While demanding, its requirements are not unrealistic: it is relatively insensitive to transmission latency, but requires that communication operations themselves not cost dramatically more than a normal memory access. I conclude that the only reason this scheme is not yet viable on modern CMPs is that they require expensive and inefficient cache-coherence mechanisms to communicate between cores (see Chapter 4).

3.2 Background

3.2.1 Beginnings

The name "transactional memory", and its first hardware implementation, were proposed in 1993 by Herlihy and Moss [63]. The implementation principles described in that paper persist into most modern HTM systems: Transactional writes are buffered in a processor's local cache, and the cache coherence system is used to track conflicts.

Although a limited "software transactional memory" scheme was proposed in 1995 [116], it required the programmer to declare in advance the memory operations that would be performed by each transaction. The first real STM implementations were published in 2003 [62, 59], and a flood followed throughout the 2000s.

Transactional memory was greeted with great excitement. With the advent of commercial multicore processors in the mid-2000s, multi-threading became necessary to fully exploit new hardware. This fed a growing desire to mitigate the difficulties of lock-based multi-threaded programming. Transactional memory seemed like the perfect solution: it offered a familiar threading model, while automating the difficult housekeeping of locks and synchronisation. It would do for concurrency what garbage collection did for memory management [53]. STM would see wide adoption, as a natural addition to managed virtual-machine languages such as Java and C#, with hardware support to follow. Sun Microsystems started a prolific transactional memory research group [62, 38, 85]. Microsoft released a research STM system for .NET [60], followed by full developer preview integrated with Visual Studio [93].

3.2.2 STM's overhead problem

Early STM research, such as the papers presenting the DSTM [62], TL2 [38] and RSTM [89] systems, primarily investigated the effects of contention in concurrent data structures and in scalable, lightly-synchronised parallel benchmarks. Overheads, while large, were regarded as an implementation detail.

However, an awareness was growing that STM's overhead was a significant problem. In 2006, Microsoft's Bartok research system [60] tackled the problem of overhead explicitly. In the absence of realistic parallel benchmarks, the original Bartok paper dedicated much of its evaluation to an analysis of *sequential* applications. This made it the first to analyse the impact of STM instrumentation on more complicated, real-world code. Using the optimisation advantages of integrating STM with a virtual machine, they obtained overheads as low as 40% on some concurrent data structures. But the overheads for the more complex, realistic programs were prohibitive, ranging from 150% to 350%.

With the introduction of more realistic parallel workloads, such as the STAMP transactional benchmarks [19, 94], the burden of overheads became more widely understood. An influential 2008 article, entitled "Software Transactional Memory: Why is it Only a Research Toy?" [20], used some of these new benchmarks to present the problem baldly. They found that, even using four processors, leading STM runtimes fell short of uninstrumented performance as often as not.

This paper was controversial in the field, but even direct responses such as "Why Transactional Memory Can Be More Than a Research Toy" [40] accidentally supported its wider point. This benchmark marathon included data from five STM systems in multiple configurations on both x86 and SPARC. Of these, four out of five – including the authors' new SwissTM algorithm – required at least four processors before they could exceed sequential performance for more than half the examined benchmarks. Although the abstract quotes impressive best-case numbers, their strongest overall result is that, when given 16 or 64 processors to play with, STM "generally outperforms sequential code".

More recently, STM research has pursued scaling at the expense of serial performance. SkySTM [85] (2009) provides sophisticated contention management, at the expense of increased overheads relative to the TL2 system on which it is based [38]. Some publications, such as 2010's NOrec [31], do not even provide serial overhead numbers for most of their benchmarks – although the 300% overhead on a red-black tree suggests a magnitude for the other benchmarks.

3.2.3 Real applications often lack scalability

The research cited above measures parallel benchmarks that are designed to scale well. However, most consumer applications do not scale so well. While many modern applications are multi-threaded, few can make good use of more than two cores [15]. These applications would have no chance of "winning back" STM-induced overheads. Suggestions that transactional memory will make writing fine-grained

parallel programs so easy that limited or irregular parallelism will be a thing of the past are, at best, "magic bullet" thinking.

It is difficult to disprove such arguments directly, though, because large applications have almost universally avoided STM. While I consider this to be fairly strong indirect evidence, this could just be because STM is relatively new, and still regarded as a "research system".

The first real-world test of this assertion may be provided by the PyPy Python interpreter, whose developers have announced plans to use STM to elide Python's notorious Global Interpreter Lock [12]. They project serial overheads of 100-1000%, in exchange for the ability to use true multithreading in Python programs [109]. The response of users to this Faustian bargain will be informative.

Although no industrial applications are known to use STM, one large real-world example has been ported to STM for research purposes. The Atomic Quake project [138] ported a parallel Quake server to STM. It experienced 300-400% overheads. Another effort, QuakeTM [45], which re-parallelised Quake from the ground up with STM, incurred overheads of 250-500%. In both of these cases, there was insufficient parallelism in the application to overcome these overheads by adding more processors. Even with eight processors, neither port ever matched the performance of the original, sequential program.

3.2.4 HTM

Hardware TM offers much better performance than software implementations, but STM remains critical. Signals from chipmakers indicate that commercial HTM will be cache-based, and must fall back to STM for transactions which overflow the cache.

Most HTM proposals track transactions using cache metadata, and detect conflicts using coherence mechanisms. This approach cannot handle transactions which overflow a processor's private cache. TCC [58] addressed this problem with a single global lock token, which must be held during commit and during the execution of any transaction which overflows the cache. While correct, this solution does not support multiple concurrent large transactions. More complicated "unbounded HTM" schemes [4, 96] effectively implement an STM algorithm in hardware, and maintain transactional state in memory. However, the complexity of such schemes probably precludes commercial hardware implementation.

Hybrid approaches [79, 32], which fall back to STM when the HTM overflows, have been more popular with hardware manufacturers. Sun Microsystems, which had been heavily involved in STM research, dipped a toe in the water with the Rock processor [37], which supported very limited transactional speculation. This was suitable mainly for the small kernels of lock-free algorithms, or for speculatively eliding small critical sections, as its transactions could not span function-call boundaries. Hybrid TM systems were built and tested on preproduction Rock chips [32, 29, 129], but production was cancelled in 2009. AMD also proposed an HTM specification that year [36], but gave no indication that it was on their hardware roadmap.

Intel's announcement, in February 2012, that they would support HTM in their upcoming Haswell architecture [108], has ended this debate for now. Haswell will support a cache-based hybrid approach which will abort on overflow. While the research described in this chapter was conducted at a time when HTM support was not on the road-map for any commercial architecture, and on the assumption that message-passing hardware would be an easier sell than HTM for chip manufacturers, the arrival of commercial HTM implementations only strengthens the need for an STM system with acceptable serial performance to handle larger transactions.

Hybrid TM and general-purpose hardware

An alternative use of the term "hybrid transactional memory" refers to hardware which does not implement HTM, but provides facilities to accelerate an STM algorithm [94, 119, 61]. The reasoning is that hardware manufacturers would be more willing to accept small changes than a pervasive modification of the cache system.

This chapter's parallel STM proposal might also be considered in this category. I assume hardware facilities not yet present in commodity processors – namely high-speed inter-core communication – and use them to accelerate an STM implementation. However, as I review in Chapter 4, inter-core communication is a general-purpose facility which is useful in many different domains.

3.2.5 Helper Threading

The approach I propose divides the cores of a multi-core processor between "application threads", which run user code, and "helper threads", which handles STM overhead. This is not a new idea, and I shall briefly review previous work in this vein.

Confusingly, the term "helper threading" can also refer to a prefetcher thread that runs ahead, fetching the targets of frequently-missed ("delinquent") loads into the cache before the application thread requests them. These helpers normally run on the same core as the application thread, in a different SMT strand. "Helper threading", in this sense, is considered an alternative to speculative techniques such as thread-level speculation or transactional memory. [136]

Closer to the spirit of this work is Ha et al.'s work on concurrent dynamic analysis [55]. They present a framework for streaming dynamic instrumentation, such as function-call or memory traces, to a parallel processor for profiling. Their solution is implemented on current shared-memory hardware, using ingenious cache-tickling algorithms to minimise overhead in application threads. However, they still found that passing these messages between cores presents severe performance problems, underscoring our reasons for modelling hardware-accelerated communication.

Tiwari et al. [131] use a helper thread to perform the costly work of manual memory management in C programs. By batching requests for bulk transmission, they overcome the communication costs of cache coherency to achieve a 20% speed-up. They also use this technique to perform safety checks on memory allocation in the background thread, while still running faster than inline allocation in the application thread.

Shetty et al. [117] use a helper thread to perform dynamic checking of memory accesses. They propose dedicated hardware to stream memory request traces from an application thread, into a FIFO drained by a helper thread. By using dedicated hardware to reduce instrumentation and communication costs, they achieve overheads of below 20%, compared with over 500% for comparable inline software-based solutions. However, in a key difference from this work, Shetty et al. employ application-specific hardware as opposed to a general-purpose communication facility.

3.2.6 Summary

The serial overhead of software transactional memory is prohibitive. As Amdahl's law predicts, and empirical investigation of real-world programs confirms, adding more application threads to overcome this overhead is not a practical solution.

The work presented in the rest of this chapter brings extra processors to bear to compensate for STM overheads, without requiring more thread-level parallelism from the application. By dedicating processors to STM processing, we can improve serial performance for each individual application thread. While helper threading is an established technique, its application to STM is a new contribution.

The problems of serial overhead in STM are great, and I do not attempt to present a "silver bullet". However, I do aim to make inroads into an often overlooked problem.

3.3 Anatomy of a Parallel STM

This chapter asks, "If we had sufficiently fast inter-processor communication, could we offload the overhead of STM and improve serial runtime?"

To address this question, I construct two examples of a parallel STM runtime system, in which each application thread is paired with a helper thread to perform its bookkeeping activities. I then construct an FPGA-based model of a multicore system with fast inter-core communication. I compare the parallel STM runtimes with their inline equivalents, using standard STM benchmarks. I conclude that this approach substantially improves serial performance, without penalising parallel scaling.

3.3.1 Overview

Each thread started by the application is paired with a helper thread started by the STM runtime. The helper thread performs STM bookkeeping tasks, allowing the application thread to spend more time running application code and less time on STM overheads.

This approach requires high-performance asynchronous communication between application and helper threads. I use memory-mapped FIFOs on an FPGA to model a multicore system with high-performance inter-core communication.

When an application thread enters an atomic section, it streams records of its memory accesses to its associated helper thread. The helper thread processes these events asynchronously, while the application thread continues running.

The specifics of this communication depend on the algorithm used to detect conflicts between application threads. I evaluate two such algorithms. The validation-based algorithm uses a global clock to detect conflicts at commit time, and features lower total overheads. The invalidation-based algorithm detects conflicts at write time. This incurs higher total costs, but these costs can be decoupled completely from the application thread. The two algorithms are described in Sections 3.3.3 and 3.3.4, respectively.

In both cases, contention management follows a straightforward "first writer wins" policy, with exponential random backoff. Neither implementation guarantees safe privatisation, isolates non-transactional code from transactional code, or prevents dirty reads within doomed transactions. These properties are not the most practical for writing transactional code in an unsafe language such as C. But, as the Bartok

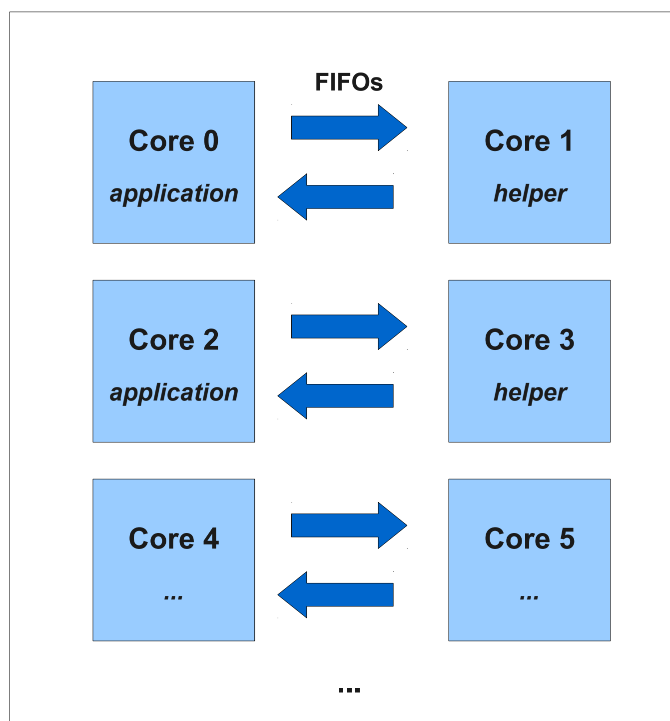


Figure 3.1: Communication between application and helper cores by message-passing within a multicore processor

authors confirmed [60, 1], these issues can efficiently be handled within a virtual machine such as the .NET CLR or the JVM, rather than by adding complexity and cost to the STM system.

3.3.2 Comparison model: Inline STM

I evaluate the impact of offloading STM bookkeeping by comparing each parallel runtime with an equivalent traditional, inline STM, as well as with uninstrumented sequential code. Each inline STM executes the same bookkeeping code as its equivalent parallel STM, but executes that code directly in the application thread rather than offloading it to a helper thread.

A parallel STM system does more total work than an inline system using the same algorithm. Although both inline and parallel versions of an STM algorithm execute the same bookkeeping code, the parallel STM also incurs the cost of sending, receiving and interpreting messages.

On a perfectly scalable program, we would therefore expect the inline version of any algorithm to be more efficient than the parallel version. For the reasons explained in Section 3.2, this is a worthwhile tradeoff for most real programs, which need acceptable serial performance more urgently than they need efficient parallel scaling. As it turns out, in Section 3.5.3, we see that increased serial performance almost entirely offsets the increase in total overheads.

3.3.3 STM Algorithm A: Validation-Based

My validation-based STM algorithm performs eager updates, eager write locking, and lazy read validation with a global clock for versioning. Its design is therefore similar to TinySTM [44] in "write-through" mode, or a word-based version of Bartok [60].

Each memory address is hashed to an ownership record (orec). An orec normally contains a version number: that is, the value of the shared global clock when a word matching that orec was last modified. An orec can also be locked, indicating that it has been written to by a currently-running transaction. When a transaction commits, it writes the current global timestamp to the orec of every location it modified. Before it commits, the transaction checks the orec of every address it has read, and checks whether it has been updated since the transaction began. Pseudocode for basic transactional operations can be found in Figure 3.2.

Parallelisation

During a transaction, the application thread streams the address of each transactional read and write to the helper processor, locking and validating each write location before transmitting its address. When the transaction commits, the application and helper threads cooperatively validate the read set before declaring the transaction a success. Pseudocode for this parallelisation can be found in Figure 3.3.

The only elements which remain on the critical path in the application thread are write locking (`txv_lock_write()`) and 50% of the read validation (`txv_handle_validate()`).

3.3.4 STM Algorithm B: Invalidation-Based

The principal problem with our validation-based STM is that it does a lot of its work at commit time. This is not a problem for traditional inline STM runtimes: they are sensitive to the amount of overhead, but not its timing. However, commit-time validation cannot be decoupled from the application thread by a parallel STM. Validation must occur after the transaction is complete, and it must be completed before we can proceed beyond the end of the transaction. This work therefore cannot be performed in the background by the helper thread, and incurs large costs on the critical path. The best we can do is to cut the overhead in half, by sharing the work between the two threads.

An invalidation-based algorithm [49] does not detect conflicts by checking each read location at commit time. Instead, transactions publish a record of their memory operations. A transaction can identify and invalidate conflicting transactions, and force them to abort. If a transaction reaches commit time and has not been invalidated, it is valid by definition.

I have implemented an invalidation-based algorithm which uses bitfields in a location's orec to identify transactions which have read that location. Before writing to a location, a transaction will lock its orec and invalidate any other transactions which have read from it. Pseudocode for basic transactional operations can be found in Figure 3.5.

```
// Globals:

counter GLOBAL_CLOCK;
struct {
  boolean isLocked;
  union {
    counter version;
    txn * owner;
  } data;
} ORecs[];

// Thread-local variables;
// one per application thread

FIFO app_to_helper;
FIFO helper_to_app;
struct txn {
  vector<WORD *> reads;
  vector<WORD *, WORD> writes;
  counter start_time;
  jmp_buf back_to_start;
} cur_txn;

// Instrumentation functions:

void txv_handle_read(
                 WORD * addr) {
  cur_txn.reads.add(addr);
}
```

```
WORD txv_lock_write(WORD * addr,
                     WORD new_val) {
  var orecp = &ORecs[HASH(addr)];
  var orec = *orecp;
  if(orec.isLocked &&
     orec.data.owner != cur_txn) {
    txv_abort();
    longjmp(cur_txn.back_to_start);
  }

  if(!orec.isLocked) {
    var locked = orec;
    locked.isLocked = TRUE;
    if(!CAS(orecp, orec, locked)) {
      txv_abort();
      longjmp(cur_txn.back_to_start);
    }
    orecp->data.owner = cur_txn;
  }

  WORD old_val = *addr;
  *addr = new_val;
  return old_val;
}

void txv_record_write(WORD * addr,
                     WORD old_val) {
  cur_txn.writes.add(addr, old_val);
}
```

```
boolean txv_handle_validate(
      vector<WORD *> readSet) {
  for(addr : readSet) {
    var orec = ORecs[HASH(addr)];
    if(orec.isLocked && orec.owner != cur_txn
     || !orec.isLocked
       && orec.version > cur_txn.start_time) {
      return FALSE;
    }
  }
  return TRUE;
}

void txv_abort() {
  for(addr, oldval : reverse(cur_txn.writes)) {
    *addr = oldval;
  }
  txv_unlock_writes();
}

void txv_unlock_writes() {
  counter new_time = ++GLOBAL_CLOCK;

  for(addr, _ : cur_txn.writes) {
    if(ORecs[HASH(addr)] == {TRUE, cur_txn}) {
      ORecs[HASH(addr)] = {FALSE, new_time};
    }
  }

  cur_txn.reads.clear();
  cur_txn.writes.clear();
}
```

Figure 3.2: Pseudocode for a simple validation-based STM algorithm. This code may be run inline, or spread between application and helper cores as described in Figure 3.3

.

```
// In application thread:

WORD txv_app_read(WORD * addr) {
  app_to_helper.send(CMD_READ, addr);
  return *addr;
}

void txv_app_write(WORD * addr,
                   WORD new_val) {
  WORD old_val =
      txv_lock_write(addr, new_val);
  app_to_helper.send(CMD_WRITE, addr,
                     old_val);
}

void txv_app_commit() {
  app_to_helper.send(CMD_COMMIT, addr);

  vector * my_check_range =
      helper_to_app.receive();
  app_to_helper.send(
      txv_verify(*my_check_range));

  if(helper_to_app.receive() == CMD_ABORT) {
    longjmp(cur_txn.back_to_start);
  }
}
```

```
// In helper thread:

void txv_help_transaction() {
  while(true) {
    switch(app_to_helper.receive()) {
      case (CMD_BEGIN, start_time):
        cur_txn.start_time = start_time;
      case (CMD_READ, addr):
        txv_handle_read(addr);
      case (CMD_WRITE, addr, old_val):
        txv_record_write(addr, old_val);
      case (CMD_COMMIT):
        vector [my_half, app_half] =
            cur_txn.reads.divideIntoHalves();
        helper_to_app.send(&app_half);
        boolean my_ok = txv_verify(my_half);
        boolean app_ok = app_to_helper.receive();
        if(!my_ok || !app_ok) {
          txv_abort();
          helper_to_app.send(CMD_ABORT);
        } else {
          txv_unlock_writes();
          helper_to_app.send(CMD_OK);
        }
    }
  }
```

Figure 3.3: Pseudocode for a parallel validation-based STM runtime system, which offloads bookkeeping overhead to a helper thread.

```
// Before instrumentation

int counter;

void increment_counter() {
  atomic {
    counter = counter + 1;
  }
}
```

```
// Inline instrumentation:

void increment_counter() {
  setjmp(&cur_txn.back_to_start);

  txv_handle_read(&counter);
  int x = counter;

  txv_record_write(&counter,
    txv_lock_write(&counter, x + 1));

  if(txv_handle_validate()) {
    txv_unlock_writes();
  } else {
    txv_abort();
  }
}
```

```
// Parallel instrumentation:

void increment_counter() {
  setjmp(&cur_txn.back_to_start);

  int x = txv_app_read(&counter);

  txv_app_write(&counter, x + 1);

  txv_app_commit();
}
```

Figure 3.4: An worked example of instrumenting a simple transaction, using the algorithms in Figures 3.2 and 3.3.

Parallelisation

Because conflict detection need not occur at commit time, invalidation-based algorithms are a good fit for our parallel architecture. I have implemented a parallel algorithm which streams read events from the application to the helper processor. The helper asynchronously publishes this information by setting bits in the corresponding orecs, while the application continues executing. When an application thread writes to a memory location, it can examine the orec to discover any transactions reading from that location, and invalidate them.

Commits are instantaneous, leaving the STM bookkeeping entirely decoupled from the application thread. Only `txi_lock_write()` remains in the application thread – and as we see later, in Section 3.5.4, its overall cost is small. Pseudocode for this parallel system can be found in Figure 3.8.

Validating asynchronous reads

A little extra complexity is required to ensure correctness when parallelising this algorithm, because read publication occurs asynchronously, some time after the actual read was performed. This leaves a window of vulnerability between the time when an application thread reads from a location, and the time when its helper publishes the read to that location's orec. If a second, conflicting transaction were to write to that location during this time, it would not know that it conflicts with the first transaction.

This problem cannot occur in the validation-based algorithm proposed in Section 3.3.3, as the version number in each orec allows us to determine whether a location has been modified since the beginning of the transaction. This invalidation-based algorithm has no global clock, and is thus vulnerable.

In order to handle this situation correctly, the application thread transmits both the address and the value of each location it reads (Figure 3.6). The helper thread compares this expected value with the actual value of the memory location at the time it processes the event. If the value has changed since the application read it, the helper will notice and invalidate the transaction. Once the read is published, subsequent writers will observe that read and invalidate the transaction if necessary, just as in the inline algorithm.

This, in turn, presents a second problem: What to do if an application thread reads a location, but then modifies it before its helper thread processes the read event? Consider the scenario in Figure 3.7. When the helper thread processes the read event, it knows that the application thread has since written to the same location. Because orecs are locked on write, there is no way another transaction could have modified the value since the write. However, there is no way of telling whether a conflicting transaction modified that location between the read and the write – and then committed, unlocking the orec in time for "our" application thread to write to it.

Due to this ambiguity, the helper cannot validate the read immediately. However, the ambiguity can be resolved by checking the subsequent write event which modified that location. In order to allow writes to be "rolled back" when a transaction aborts, each write event already carries both the address that was written and the original value of that location. When the write event arrives, the helper can use this value to verify that the location was not disturbed between the read and the write. The helper therefore maintains a set of such locations, referred to as `pending_upgrades` in Figure 3.8.

```
// Globals:

BITFIELD kill_flags;

struct {
  boolean isLocked;
  BITFIELD readers;

  // Only used in parallel mode
  boolean writeProcessed;
} ORecs[];

// Thread-local variables;
// one per application thread

FIFO app_to_helper;
FIFO helper_to_app;
struct txn {
  vector<WORD *> reads;
  vector<WORD *, WORD> writes;
  map<WORD * -> WORD>
    pending_upgrades;
  jmp_buf back_to_start;
} cur_txn;
```

```
WORD txi_handle_read(WORD * addr) {
  var orecp = &ORecs[HASH(addr)];
 retry:
  var orec = *orecp;
  if(orec.isLocked) {
    if(orec.readers != cur_txn.id)
      txi_abort();
  } else {
    var includingUs = orec;
    includingUs.readers |= cur_txn.id;
    if(!(orec.readers & cur_txn.id)
       && !CAS(orecp, orec, includingUs))
      goto retry;

    cur_txn.reads.add(addr);
  }
}

void txi_lock_write(WORD * addr,
                    WORD new_val) {
  var orecp = &ORecs[HASH(addr)];
 retry:
  var orec = *orecp;

  if(orec.isLocked) {
    if(orec.readers != cur_txn.id)
      txi_abort();
  } else {
    if(!CAS(orecp, orec, {TRUE, cur_txn.id}))
      goto retry;

    var to_kill = orec.readers & ~cur_txn.id;
    while(var kf = kill_flags,
          !CAS(&kill_flags, kf, kf | to_kill));
  }
}
```

```
void txi_record_write(WORD * addr,
                      WORD old_val) {
  cur_txn.writes.add(addr, old_val);
}

boolean txi_is_valid() {
  return !(kill_flags & cur_txn.id);
}

void txi_abort() {
  for(addr, oldval : reverse(cur_txn.writes)) {
    *addr = oldval;
  }
  txi_unlock_writes();
}

void txi_unlock_writes() {
  for(addr, _ : cur_txn.writes) {
    if(ORecs[HASH(addr)] == {TRUE, cur_txn.id})
      ORecs[HASH(addr)] = {FALSE, 0};
  }

  cur_txn.reads.clear();
  cur_txn.writes.clear();
}

void txi_unmark_reads() {
  for(addr : cur_txn.reads) {
    var orecp = &ORecs[HASH(addr)];
   retry:
    var orec = *orecp;
    if (!orec.isLocked) {
      var unlocked = orec;
      unlocked.readers &= ~cur_txn.id;
      if(!CAS(orecp, orec, unlocked))
        goto retry;
    }
  }
}
```

Figure 3.5: Pseudocode for a simple invalidation-based STM algorithm. This code may be run inline, or spread between application and helper cores as described in Figure 3.8

3.3.5 Summary

I have proposed a parallel STM scheme, using a dedicated helper thread for each application thread. The helper thread performs bookkeeping tasks, relieving the application thread of costly serial overheads. This scheme will be compared with an equivalent inline system, which runs the bookkeeping code in the same thread as the application code.

I have described two conflict-detection algorithms: one validation-based, one invalidation-based. I will be comparing the performance of both algorithms, in both parallel and inline STM runtimes.

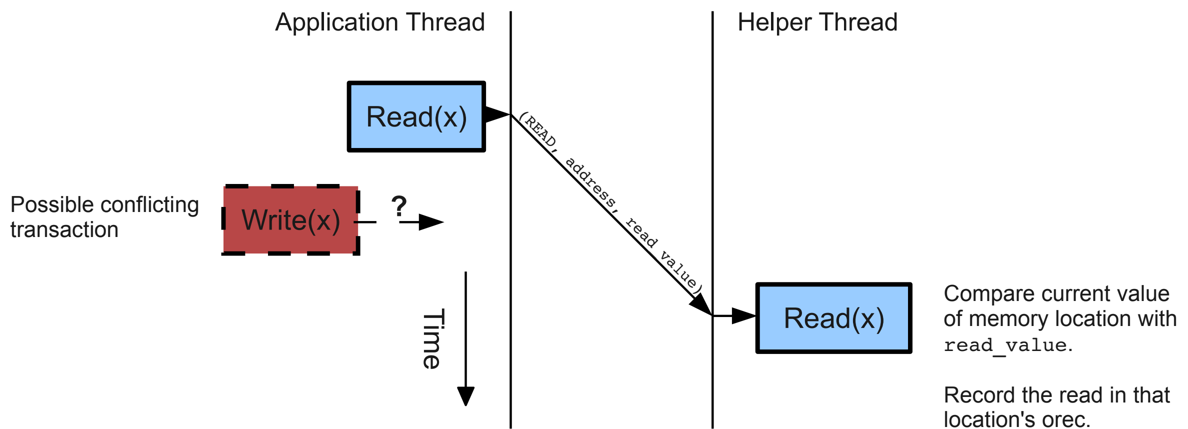


Figure 3.6: The helper thread compares the current value of a memory location with the value read by the application thread. This allows it to detect any conflicting modifications that occurred before the read was recorded in that location's orec. After the read is recorded, any subsequent conflicts will mark our transaction as invalidated.

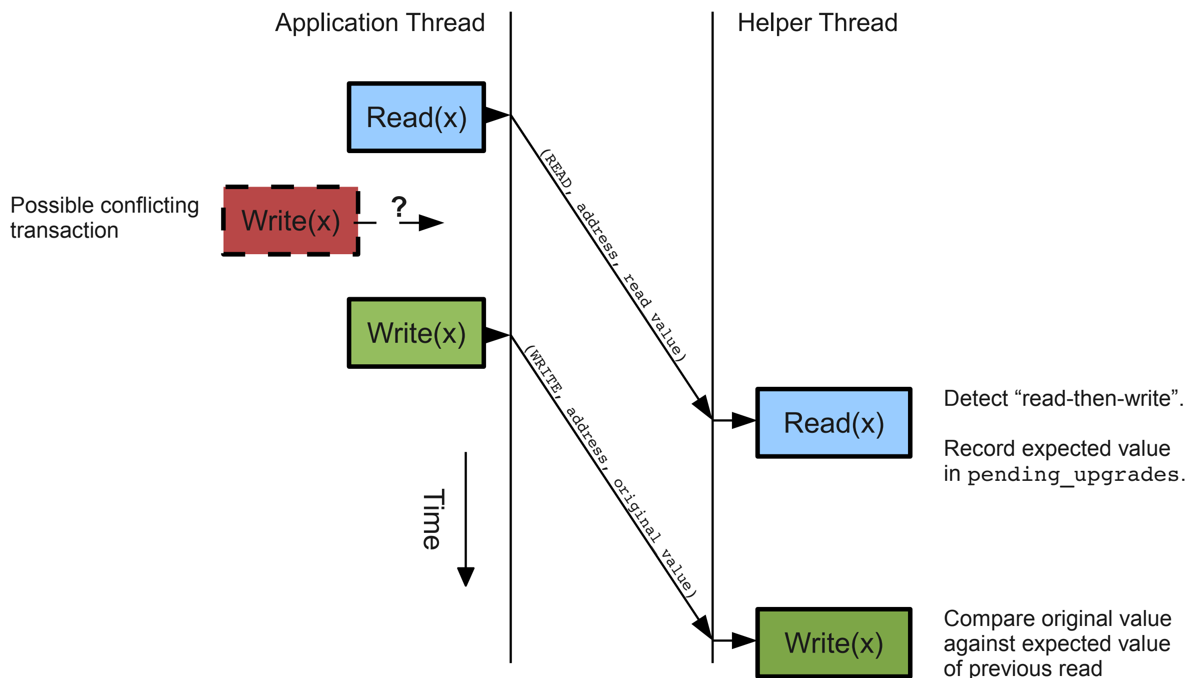


Figure 3.7: If the application thread reads and then writes a single memory location, the helper thread cannot detect a conflicting transaction until it processes the write event.

```
WORD txi_app_read(WORD * addr) {
  val result = *addr;
  app_to_helper.send(CMD_READ, addr, result);
  return result;
}

void txi_app_write(WORD * addr,
                   WORD * new_val) {
  txi_lock_write(addr, new_val);

  var old_val = *addr;

  app_to_helper.send(CMD_WRITE, addr, old_val);
}

void txi_app_commit() {
  app_to_helper_send(CMD_COMMIT);

  if(helper_to_app.receive() == CMD_ABORT) {
    longjmp(cur_txn.back_to_start);
  }
}

void txi_help_read(WORD * addr, WORD value) {
 retry:
  var orec = ORecs[HASH(addr)];

  if(!orec.isLocked) {
    if(!orec.readers & cur_txn.id) {
      if(!CAS(orec.readers |= cur_txn.id))
        goto retry;

      cur_txn.readers.add(addr);

      if(*addr != value) {
        txi_abort();
      }
    }
  } else { /* orec.isLocked */
    if(orec.readers != cur_txn.id)
      txi_abort();

    if(!orec.writeProcessed) {
      cur_txn.pending_upgrades
                 .put(addr, value);
    }
  }
}
```

```
void txi_help_transaction() {
  while(true) {
    switch(app_to_helper.receive()) {
      case (CMD_BEGIN) :
       retry:
        var kf = kill_flags;
        if (kf & cur_txn.id &&
             !CAS(&kill_flags, kf,
                      kf & ~cur_txn.id))
          goto retry;

      case (CMD_READ, addr, value) :
        txi_help_read(addr, value);

      case (CMD_WRITE, addr, old_val) :
        txi_record_write(addr, old_val);

        ORecs[HASH(addr)].writeProcessed = TRUE;

        var expected = pending_upgrades[addr];
        if(expected) {
          if(old_val != expected)
            txi_abort();

          pending_upgrades.remove(addr);
        }

      case (CMD_COMMIT) :
        if(txi_is_valid()) {
          txi_unlock_writes();
          helper_to_app.send(CMD_COMMIT);
          txi_unmark_reads();
        } else {
          txi_undo_writes();
          txi_unlock_writes();
          helper_to_app.send(CMD_ABORT);
          txi_unmark_reads();
        }
    }
  }
}
```

Figure 3.8: Pseudocode for a parallel invalidation-based STM runtime system, which offloads bookkeeping overhead to a helper thread.

3.4 Evaluation Framework

I have implemented this STM scheme on a simple parameterisable model of an eight-way chip multiprocessor with hardware support for message passing. This model was constructed on FPGA hardware, allowing it to run large benchmarks in reasonable time.

3.4.1 Choice of Platform

In order effectively to evaluate the use of helper processors for parallel STM, we required a system with low-cost communication between neighbouring cores. This facility is not provided by current commodity processors, so evaluating this proposal requires some sort of simulator.

Software simulation of such a system would be very slow, limiting the size of benchmarks that can be run. I therefore constructed an FPGA model using off-the-shelf components. This allows for rapid prototyping, and high-speed execution of large benchmarks, while allowing an exploration of architectural parameters which would be intractable on commercial processors.

3.4.2 Hardware

I used a an Altera Stratix III 3SL150 part with 142K logic elements on a Terasic DE3 evaluation board (Figure 3.9). Using Altera's SOPC Builder environment, I connected eight simple soft cores with an Avalon switched interconnect to a shared external 1GB of DDR2 memory (Figure 3.10). Each core is an Altera Nios II: a 32-bit single-issue in-order RISC processor with support for integer arithmetic.

Memory Latency

FPGA memory limits prohibit large on-chip caches, so I coupled the processors to custom time dilation circuitry. When one processor makes a memory request, this logic pauses the clock signal to the entire system while the request is fulfilled by the off-chip DDR2 memory. Overlapping requests from different processors cause the system to pause until all have been fulfilled. Each processor therefore experiences a uniform memory access latency typical of a first-level cache ($\sim$2 cycles). The operation of this circuitry is illustrated in Figure 3.11.

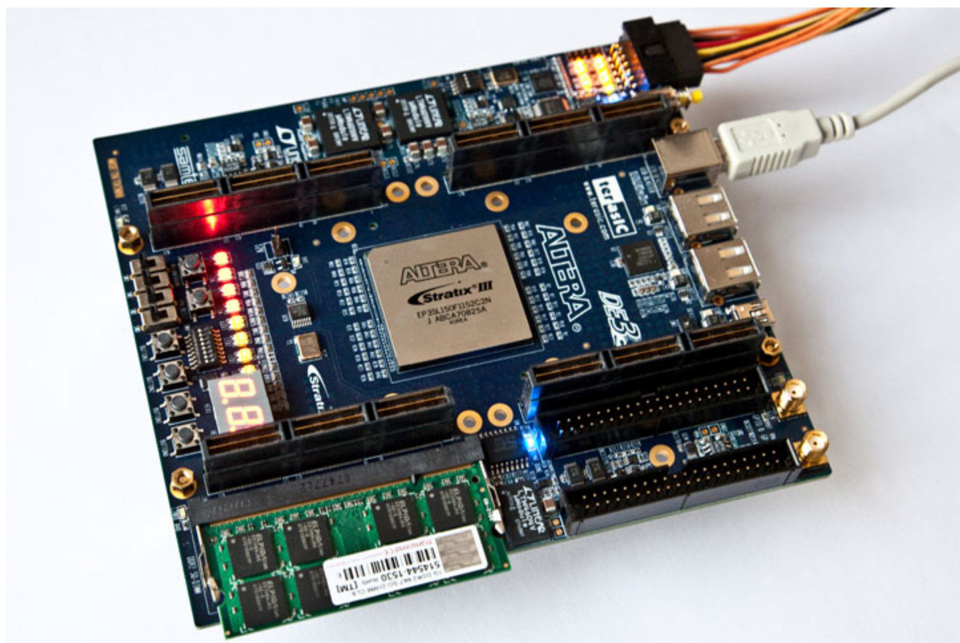


Figure 3.9: Hardware prototype board running eight soft processors

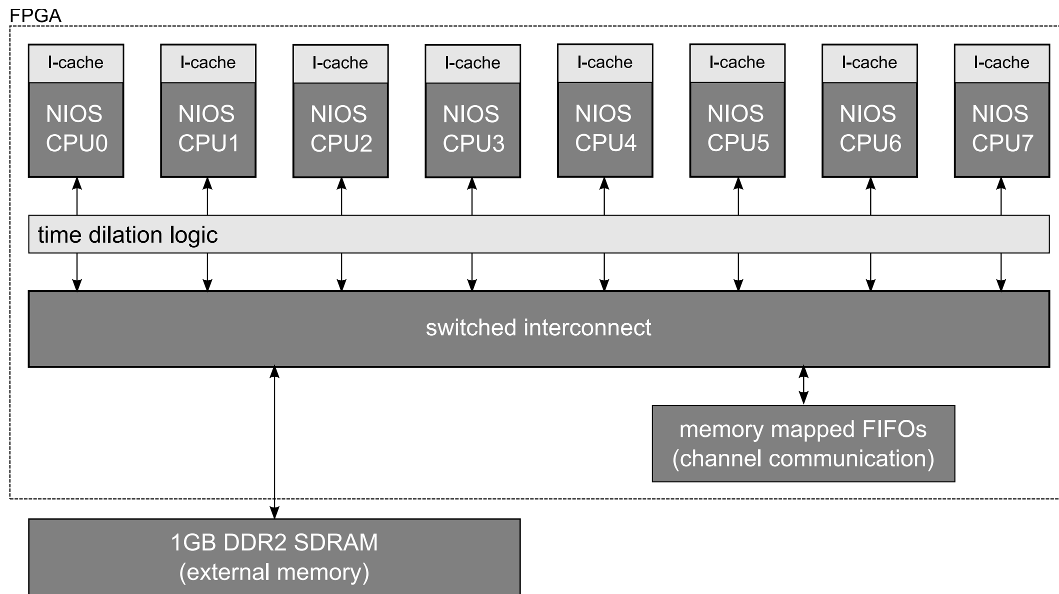


Figure 3.10: Overview of the system hardware

Hardware FIFOs

Low-cost asynchronous communication is provided via 8 memory mapped hardware FIFOs, each 256 words deep. Reading from an empty FIFO, or writing to a full one, stalls the processor. The simplicity of this scheme avoids confounding performance with the design complexity of more flexible message queues.

The custom FIFO circuitry provides an arbitrary transmission delay. This allows me to investigate the effect of transmission delays without affecting the cost of the enqueue or dequeue operations. Once a word is enqueued in a FIFO, it will not become available to dequeue for a number of cycles configurable at run time.

Performance

The whole design is clocked at 50MHz. Even after time dilation, it simulates virtual time at an average of 18MHz. This comfortably allows large benchmarks to be used.

3.4.3 Benchmarks

To evaluate the performance of this STM system, I used the three integer benchmarks from the STAMP suite [19]: `intruder`, `genome` and `vacation`. I also evaluated two microbenchmarks, exercising a hash table and a red-black tree. Each microbenchmark involved inserting (25%), looking up (50%) and deleting (25%) random 4-character strings from its respective data structure.

The remaining STAMP benchmarks rely upon floating point computations, and so could not be used. Although floating point support is available for Nios soft processors, its large logic footprint could not be accommodated in the FPGA device we used (see Section 3.4.2).

The benchmarks are written in C, with transactional memory accesses manually instrumented. They run on "bare metal", without an operating system. The STAMP benchmarks are parameterised, with workload sizes controlled from the command-line. Figure 3.12 describes the parameters I used, which are substantially larger than the normal "simulator" values.

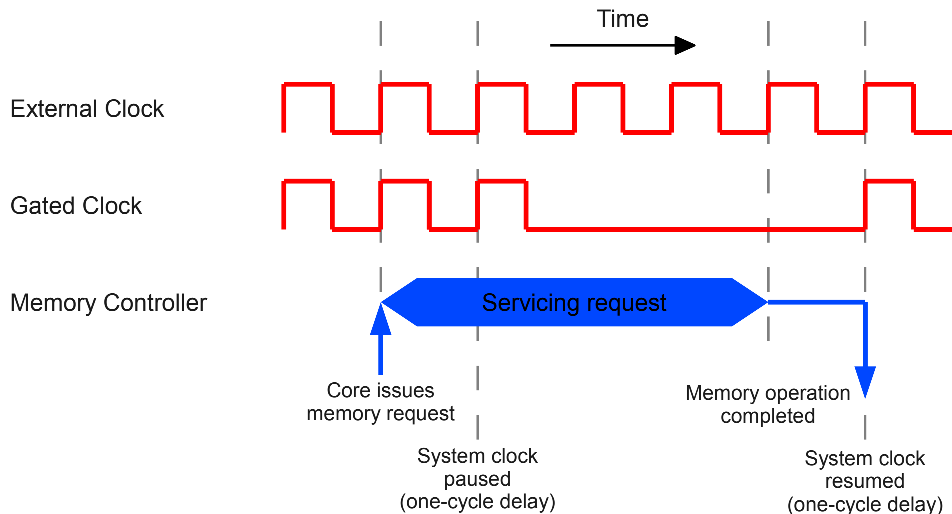


Figure 3.11: Operation of time-dilation circuitry. When one of the Nios soft cores makes a memory request, the gated clock, from which all components except the DDR2 memory controller are run, is paused. The clock is restarted when the memory operation has completed. With a 50 MHz external clock, the system clock runs at an average of 18 MHz.

Benchmark	Parameters
`intruder`	`-a10 -l64 -n8192 -s1`
`genome`	`-g8192 -s32 -n65536`
`vacation_hi`	`-n4 -q60 -u90 -r65535 -t131072`
`vacation_lo`	`-n2 -q90 -u98 -r65535 -t131072`

Figure 3.12: Workload parameters for the integer STAMP benchmarks

3.5 Results and Evaluation

3.5.1 Overview

The principal goal is to alleviate the crippling serial overheads of software transactional memory. In Section 3.5.2, I evaluate the serial performance of individual application threads, with and without helper threading. I show that parallel STM systems substantially decrease this serial overhead.

While serial overhead is a more critical obstacle to the feasibility of STM than efficiently scaling to many processors, it would be foolish to sacrifice scalability entirely. Thus, in Section 3.5.3, I verify that parallel STM algorithms scale comparably, core-for-core, with their inline equivalents.

Section 3.5.4 takes a closer look at the serial performance of the parallel algorithms, and determines where the remaining overhead lies. Section 3.5.5 examines utilisation of the helper processor.

Finally, in Section 3.5.6, I examine the impact of communication costs on parallel STM algorithms. For these experiments, I have modelled a more-or-less ideal high-performance message-passing system. In doing so, I have abstracted away the significant challenges of making such facilities work in a real system (more on these in Chapter 4). Now I ask, were such a real system to be constructed, what performance we would require of it in order to build a useful parallel STM.

3.5.2 Serial Performance

The primary objective of my parallel STM scheme is to reduce the serial overhead – that is, the amount by which using STM slows down the execution of each application thread.

In order to quantify this overhead, I ran each benchmark with a single application thread, using an on-chip counter to determine the number of simulated cycles required to complete the benchmark's main section. Running a single application thread requires one core for the inline STM systems, and two cores for the parallel systems.

I then normalised all run-times to the run-time of the uninstrumented single-threaded benchmark (without any STM). These results are plotted in Figure 3.13. The bare-metal model I used experiences no interference from other processes, so the maximum variation between runs was less than 0.5% of run-time for all benchmarks.

Parallelising the STM runtime yields a substantial reduction in serial overhead, compared with traditional inline systems. We see that by offloading STM bookkeeping onto a helper thread, we reduce the serial overhead of an individual application thread by a factor of three.

Over all benchmarks, the average overhead of the invalidation-based algorithm is reduced from 160% (inline) to 47% (parallel). The average overhead of the validation-based algorithm is reduced from 118% to 43%.

For comparison, I also ran each benchmark with the TL2 STM system distributed with the STAMP benchmark suite. The average serial overhead of the TL2 runtime ranged from 320% to 600% (not plotted).

3.5.3 Performance at Scale, or Return on Investment

Dedicating cores to helper threads may increase serial performance, but it halves the number of processors running application code. Although, even now, parallel performance is more often limited by parallelism in the application than by the number of cores available (see Section 3.2), it is worth maintaining scalable performance. We want to maintain our return on investment when adding more cores to a system.

In order to evaluate the return on investment of additional cores, I ran each benchmark and STM system using two, four and eight physical cores. I then calculated the reciprocal of each benchmark's runtime (an equivalent measure, "transactional throughput", is widespread in the STM literature). This provides an increasing ("up is good") value for performance. The maximum inter-run variation was less than 1.5% of runtime for all runs, so no error bars are plotted.

This data is plotted in Figure 3.14. We can see that parallel STM systems scale comparably with their inline equivalents. This indicates that we have not sacrificed our return on hardware investment in our quest to reduce serial overheads.

Return on Programmer Investment

Consider these results from the perspective of an application programmer. It is generally accepted that it takes effort to write a parallel program (see Chapters 1 and 2). The more finely the program is parallelised, the greater the effort.

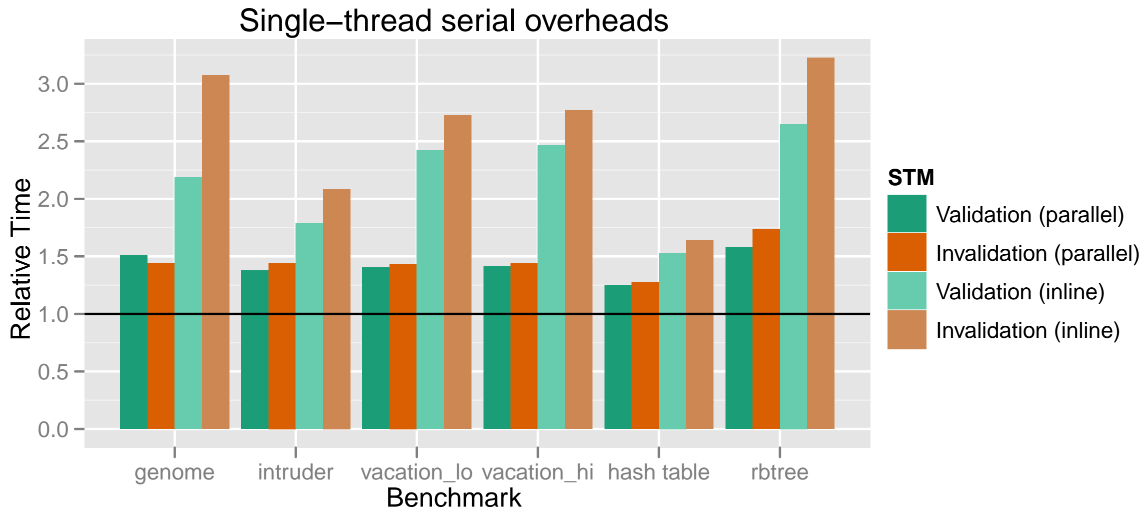


Figure 3.13: Parallelising the STM system using helper threads substantially decreases the overheads in each application thread. Runtimes for single application threads are plotted relative to the applications' uninstrumented performance. Inline algorithms use one core per application thread, and parallel algorithms use two.

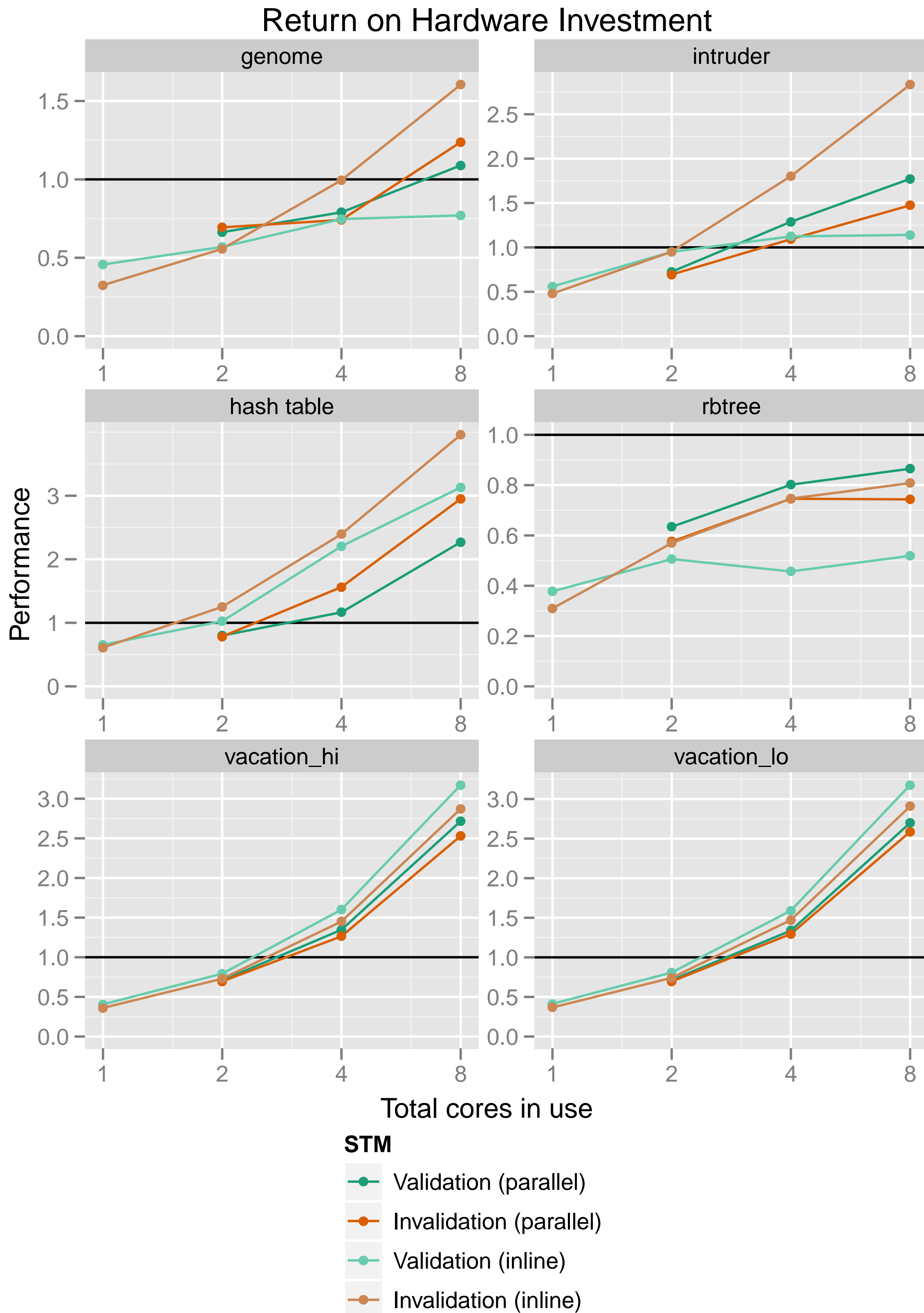


Figure 3.14: Parallelising the STM system using helper threads does not decrease the return on investment of using more cores, even compared core-for-core against an equivalent inline STM running twice as many application threads. Performance is plotted as throughput, the reciprocal of run-time.

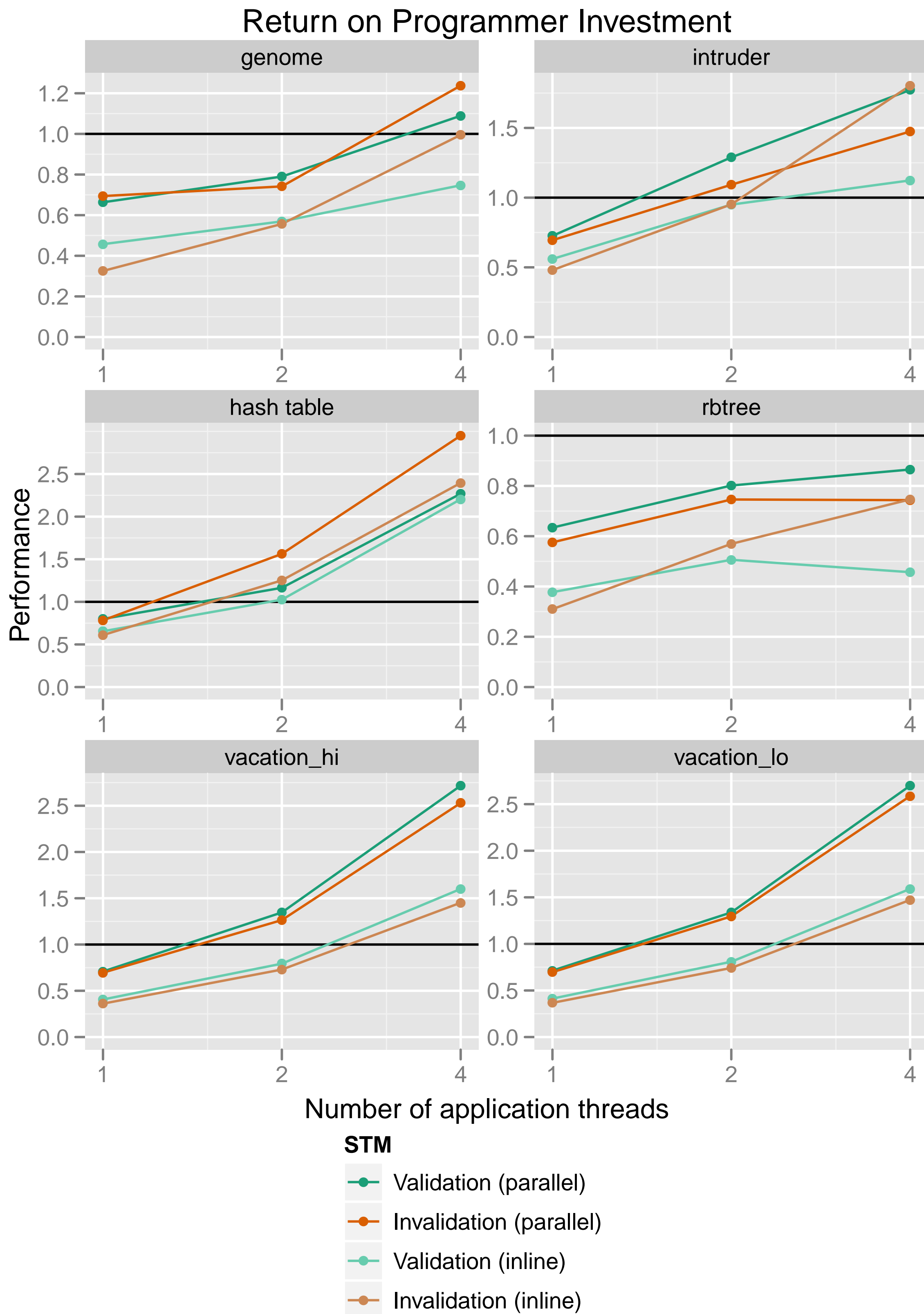


Figure 3.15: Parallelising the STM system using helper threads increases the return on investment of parallelising a program. Performance is plotted as throughput (the reciprocal of run-time), against available parallelism (the number of application threads).

Figure 3.15 plots parallel performance against *programmer* investment – the amount of parallelism that the application supports. This is the same data as Figure 3.14, but plotted by the number of application threads permitted by the program, rather than the number of cores provided by the hardware.

As I argue in Section 3.2.3, for many if not most applications, available parallelism rather than available cores is the limiting factor of performance. With hardware trends adding cores with Moore's law, this is only more likely to be the case.

It is good news, then, that when using the parallel STM systems, we see a distinctly better return in performance for our investment in parallelism. Four out of our six benchmarks exceed sequential performance with only two threads. Contrast the inline STM algorithms, under which most of the benchmarks require four or more application threads in order to exceed sequential performance.

3.5.4 Breaking Down the Overhead

If the STM bookkeeping is being performed by the helper thread, what is responsible for the remaining overhead in a parallel STM? To answer this question, I ran single application threads under a parallel STM, progressively disabling portions of the STM instrumentation to observe the effect on performance.

Given the differences between the two conflict-detection algorithms, I will consider each separately.

A. Validation-Based Algorithm

Figure 3.16 shows us a breakdown of application overhead in the validation-based parallel algorithm. About half (53%) of the overhead is caused by instrumentation in the application thread. The rest (47%) is incurred by read-set validation in the commit phase. As this must happen after the application is ready to commit the transaction, but before it is allowed to proceed, this bookkeeping overhead is incurred on the critical path.

Apart from commit-time read valdiation, our results indicate that all other helper computations have been entirely removed from the critical path. Disabling all helper computations gives no measurable speed-up, compared with disabling only read validation.

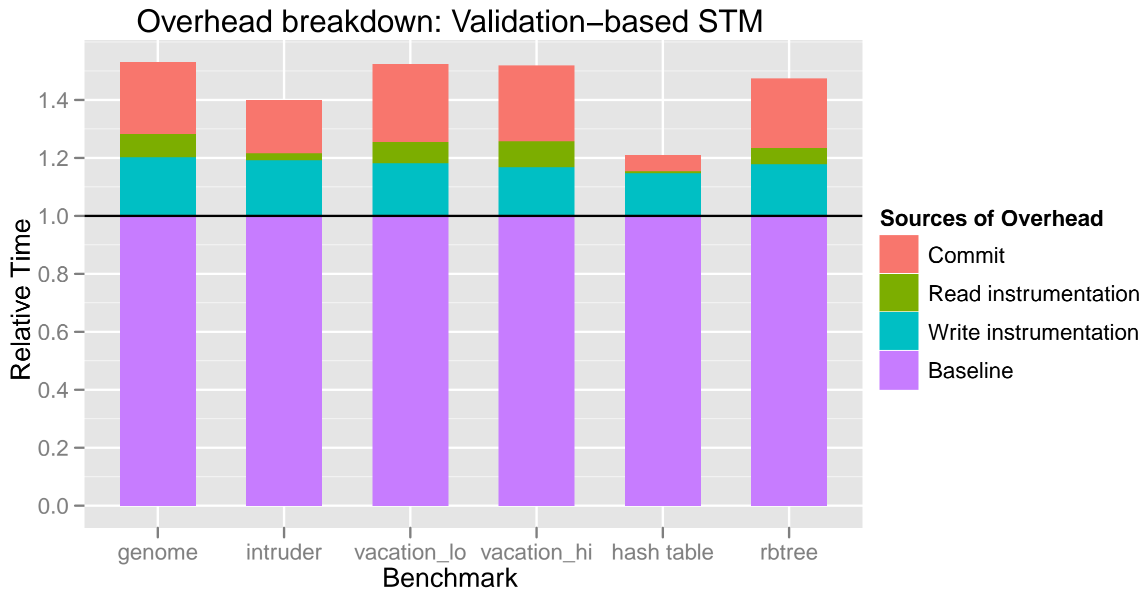


Figure 3.16: A breakdown of the serial overhead of a validation-based parallel STM, running one application thread and one helper thread.

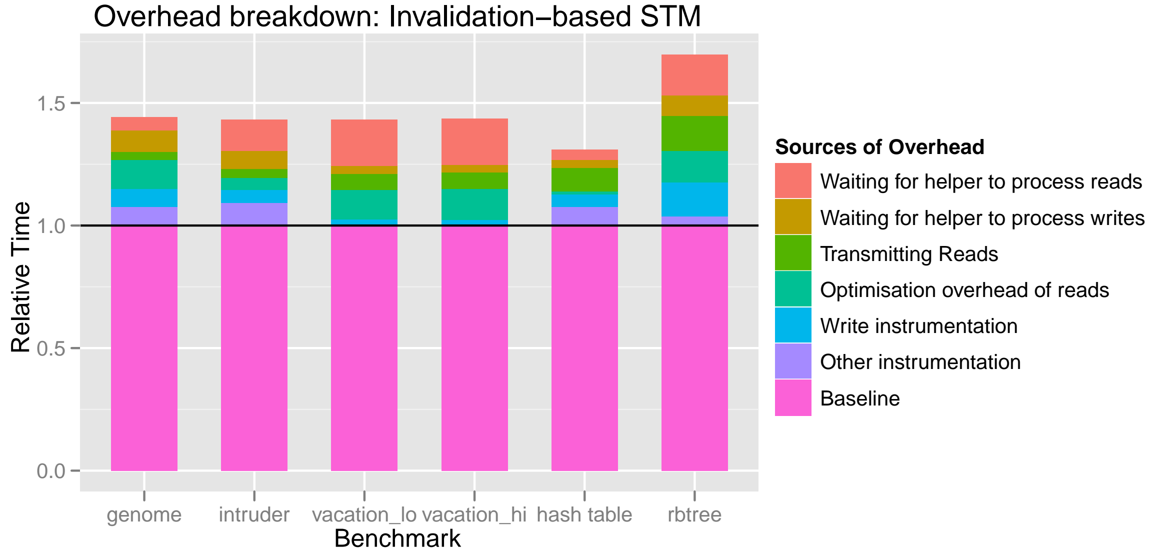


Figure 3.17: A breakdown of the serial overhead of an invalidation-based parallel STM, running one application thread and one helper thread.

B. Invalidation-Based Algorithm

Figure 3.17 shows us the factors responsible for application overhead in the invalidation-based algorithm.

Notably, all benchmarks spend a substantial amount of time idling, ready to commit, while the helper thread processes a backlog of memory operations. On average, about a third (34%) of the application thread's overhead time is spent waiting for the helper.

The conclusion is that the helper thread's computations are the limiting factor on this STM system's performance. This is not surprising: if we revisit Figure 3.13, we can see that the invalidation-based algorithm simply has more to do than the validation-based algorithm. Even though the bookkeeping is completely decoupled from the application thread, the application thread still ends up waiting for the helper to finish its processing

One response is to expand the number of helper processors. If cores are truly cheap, we can divide the STM bookkeeping over two helper threads. By eliminating the helper backlog, this could bring application-thread overheads as low as 30%.

This suggestion is not as farcical as it sounds. It is accepted orthodoxy in the STM world to implement runtime systems with overheads as high as 300% or more [31], and employ extra processor cores to "win it back" with parallel scaling. It is no less legitimate to tackle the overhead directly with additional helper threads.

3.5.5 Investigating Helper Workload

The helper thread in a parallel STM is reactive, and remains idle until it receives an event from the application thread. In Figure 3.18, I plot the proportion of time these threads spend idle.

This data was obtained by adding code to the helper thread which starts a hardware counter immediately before reading each command from the application thread. The helper then disables the counter as soon as a value has been read. The counter records both the number of times it has been enabled and the number of cycles it has spent enabled. I obtained the idle time by correcting the cycle count for the fixed overheads of reading from a FIFO, as well as enabling and disabling the timer. This is a fixed number of cycles per occurrence, as both the FIFO and timer are uncontended resources private to the core.

My first observation is that the invalidation-based algorithm does more bookkeeping work than the validation-based algorithm. This is unsurprising – we already know from the inline implementations evaluated in Section 3.5.2 that the invalidation-based algorithm has greater overheads. We also know, from Section 3.5.4, that a substantial fraction of the application thread's overhead comes from waiting for the helper thread to clear its backlog.

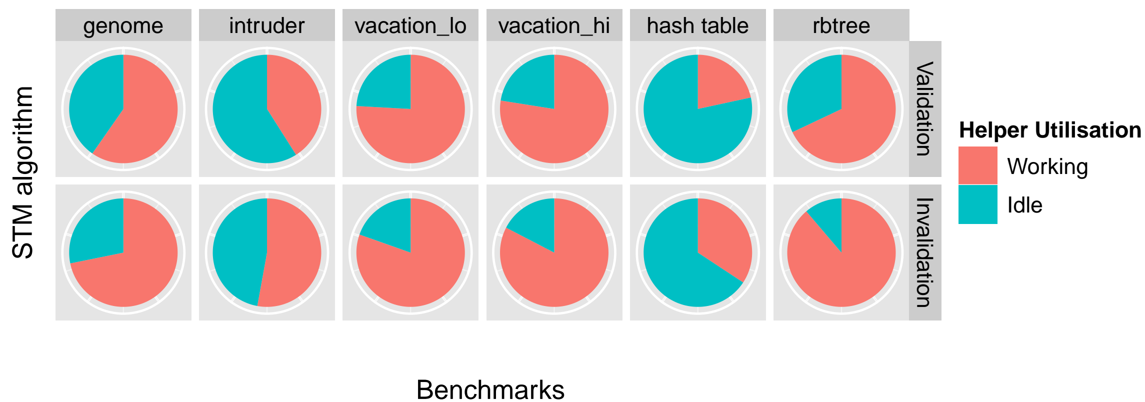


Figure 3.18: The helper thread spends a significant amount of time idle, in both STM algorithms. This graph represents runs with one application thread and one helper thread.

Likewise, despite the amount of time it spends idle during and between transactions, the validation-based helper is on the application's critical path during each commit operation.

One can conclude that in both algorithms, the helper thread alternates between lethargy and frantic activity. Ideally, we could use this slack to increase our computational efficiency. However, any diminished responsiveness would have a large impact on application performance, and other factors might justify or ameliorate the allocation of a whole core to each helper thread.

This "stop-go", latency-sensitive pattern makes it unlikely that two helper threads could usefully be multiplexed onto a single hardware context using standard preemptive multitasking. Cooperative multitasking might prove more fruitful, as it would allow one thread to divide its help finely and responsively between multiple application threads. Chip multi-threading is even more attractive, as it allows fine-grained sharing of the hardware without explicit coordination.

When running particularly scalable or low-overhead applications, such as `vacation` or the hash-table microbenchmark respectively, it would make sense to switch to an inline STM system. This decision could be made at run-time, and a just-in-time virtual machine (or double compilation) would eliminate the cost of conditional instrumentation.

In some cases, the power benefits of having some partially idle cores might justify leaving things as they are. Intel processors already feature a "turbo boost" [26], which increases the clock rate for a heavily loaded core if the whole chip is consuming sufficiently little power. A lightly or intermittently loaded helper core could therefore enable higher serial performance for the entire system.

Finally, to take a more exotic approach, this asymmetric workload could be a good fit for an asymmetric multiprocessor [78]. A processor containing a small number of complex cores with good sequential performance, and a larger number of simpler cores for highly parallel tasks, would map nicely onto application and helper threads respectively – or possibly even the reverse, for the invalidation-based algorithm.

3.5.6 The Effect of Communication Cost

If parallel STM is to be used in practice, it must run on a processor which provides low-cost communication. But what is a sufficiently low cost?

These experiments were conducted on a model chip multi-processor, with blocking hardware FIFOs which model almost perfect on-chip communication (see Section 3.4.2). However, such simple mechanisms are not practical for general purpose processors (Chapter 4 is devoted to this problem). Without assuming anything about the mechanism of on-chip communication, we can still evaluate the components of "communication cost", and assess their impact on these parallel STM algorithms.

The cost of communication is not a single scalar value. We can break it down into three parts: the cost to send a message (in this case, the delay incurred enqueueing to a FIFO); the cost to receive a message (dequeue delay); and the time taken to transfer a message from one processor to another (the delay before an enqueued message is available to dequeue).

I varied each of these independently: The transmission delay is varied by custom FIFO logic, and the cost of enqueue and dequeue operations can be modified with inline delay loops. I measured the the sensitivity of both STM algorithms to each of these variables.

As we can see in Figure 3.19, the transmission latency of on-chip communication is not critical in either case. This is good news, as the relative cost of transmitting a message across a chip multi-processor is trending upwards with successive hardware generations.

What *is* critical is the cost of the communication operations themselves, which are on the critical path. Both algorithms are acutely sensitive to the cost of enqueue operations, as these delay the application thread and contribute directly to serial overhead. The validation-based algorithm, which has more slack in its helper thread than the invalidation-based algorithm (Figure 3.18), is more tolerant of extra dequeue cost. Both algorithms are quite sensitive, however, as every cycle spent receiving messages increases the cost of processing a transactional memory operation.

These results show that parallel STM is a poor fit for current chip multi-processors, whose cores may only communicate through cache-coherent shared memory. Coherence-based polling stalls both sender and receiver for dozens or even hundreds of cycles, while the cache line containing the data is transferred between cores. This cost makes fine-grained helper threading such as parallel STM impractical. In Chapter 4, I review some proposals, including commercially-available silicon, which provide communication mechanisms with more acceptable performance for parallel STM.

3.5.7 Validating the Model

The FPGA model described in Section 3.4 has many advantages, but it differs substantially from most computers available today. Its processors are in-order, whereas most non-embedded systems are superscalar. It also has a uniform low memory-access latency, whereas most systems are subject to potentially large cache-induced delays.

In order to validate the results of this model, I ran two STM algorithms that do not require special hardware assistance: the inline version of the validation-based STM described in Section 3.3.3, and the TL2 STM [38], as distributed with the STAMP benchmarks [19].

Each benchmark and STM algorithm was run on the Gem5 simulator, using an in-order processor, the ARM instruction set and cache latencies based on those of Intel's Nehalem processor family. This is the same simulator configuration as used in Chapter 4, and more information about it can be found

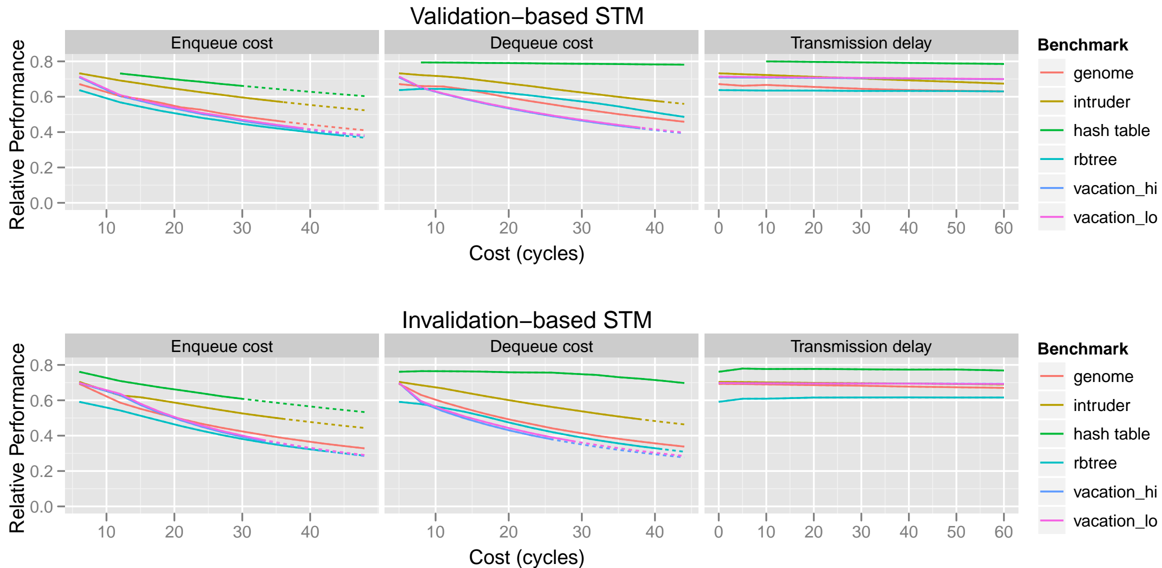


Figure 3.19: Parallel STM performance is insensitive to delays in message transmission, but acutely sensitive to the cost of enqueue and dequeue operations. This graph represents runs with one application thread and one helper thread, and performance is plotted relative to uninstrumented sequential code. Dotted lines indicate regions where the parallel system is outperformed by the inline system, indicating that communication costs have overwhelmed the benefits of parallelism.

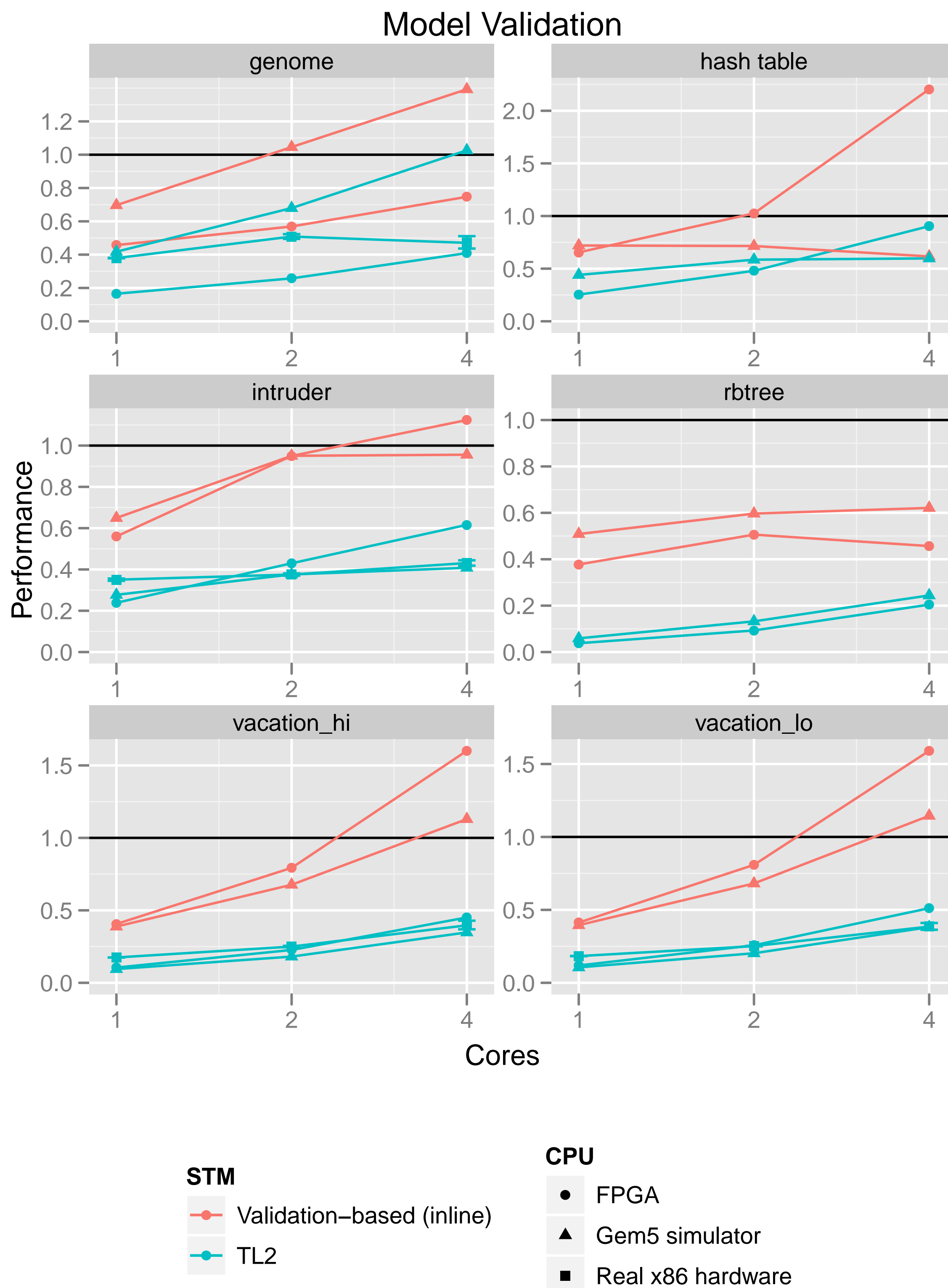


Figure 3.20: Comparing benchmark performance on different models. The performance of the FPGA model is comparable to that of the same code running on the Gem5 simulator and a real 4-core x86 machine. Performance is plotted as throughput (reciprocal of runtime).

there. Some porting was necessary to run my validation-based algorithm on the new platform, and I used the Gem5 timer (`m5_rpns`) to measure benchmark run-times. The Gem5 simulator is deterministic, so no repeats were necessary. The benchmarks used the recommended "simulator" workloads.

The simulator controls for the effect of cache latency, but not for the effect of an in-order processor. So, to provide an extra level of control, I ran the four STAMP-based benchmarks using the TL2 algorithm on a four-processor x86 machine (an Intel Core 2 Q6600 running Linux). They were compiled and run from fresh, unmodified STAMP source code, using the recommended "simulator" workloads for comparability with the Gem5 model. Run-times were quite variable (typical maximum variability was as much as 50% of the minimum value), so each measurement was taken nine times. I plot the median of these repeats, with error bars representing the standard error.

The results of the simulator and real-processor runs, along with the FPGA data from Figure 3.14, are plotted together in Figure 3.20. Performance is normalised to that of the uninstrumented sequential code on each platform.

We can see that for all benchmarks except `genome`, the TL2 results agree quite closely on all three platforms. (And for `genome`, it is the simulator – usually regarded as a reliable model – that is out of line with the FPGA and x86 platforms). The scaling of the validation-based STM algorithm does suffer a little from the addition of cache penalties, but only on the hash table microbenchmark – which appears to suffer from substantial cache aliasing – do the more realistic models tell a particularly different story from the FPGA model.

Overall, these results show that my FPGA model provides a reasonable approximation to modern multi-core hardware.

3.6 Conclusion

I have considered the impact of the prohibitively large serial overheads incurred by all software transactional memory systems. It is probably impossible to avoid these overheads without TM-specific hardware, and practically forseeable TM hardware has capacity limits beyond which it must overflow to STM. The large overheads of STM have hitherto presented a significant obstacle to the uptake of TM, and will become an even more pressing issue as hardware availability drives greater uptake of transactional programming.

We have seen, however, that this crippling overhead can be moved off the critical path, onto additional "helper" processors. In this way, we can spend the bounty of Moore's law on increasing STM performance, without demanding additional parallelism from the application.

I have implemented two such parallel STM runtimes, one with validation-based conflict detection, and one with invalidation-based conflict detection. Both were effective, reducing serial overheads from 118% to 43% and from 160% to 47% respectively.

These experiments were conducted on an FPGA-based hardware platform, which simulates ideal high-performance communication between processor cores. Without assuming anything about the mechanism of communication, I have measured the sensitivity of parallel STM to the various components of communication cost. While not compatible with today's inefficient cache-coherency based communication, these requirements are not unreasonable.

By boosting the serial performance of transactional code, I am hopeful that parallel STM techniques can help pave the way for more widespread adoption of transactional memory.

Chapter 4

Asynchronous Remote Stores for Inter-Processor Communication

4.1 Introduction

As multicore processors become the norm, mainstream systems have retained the shared-memory programming model, and look set to do so for the forseeable future. However, the cache-coherence mechanisms required to maintain this illusion make communication between cores very costly.

Threads on different cores may only communicate by reading and writing shared memory locations. When one core writes to an address which is cached near another core, it incurs a coherence miss, requiring several cross-chip messages to move the data into its own cache. With the relatively strong memory models used by today's most popular multiprocessors, this can cause the producing core to stall until the transfer is complete, if the store queue becomes full or the program requires a memory barrier. The consumer core must then incur another coherence miss, and stall until the data has completed its round-trip into its cache, also stalling for dozens or even hundreds of cycles [95].

The upshot is that explicit communication between processors on modern multicore systems is expensive. This makes fine-grained parallelism difficult to achieve, limiting our ability to exploit the resources our hardware provides.

Researchers have proposed a variety of architectural changes to allow cores to communicate directly and cheaply over on-chip interconnects. But these changes are forbiddingly heavyweight, restricted to a single pattern of communication (typically FIFO queues), and difficult to virtualise and therefore incompatible with time-sharing operating systems.

4.1.1 Contributions

I introduce an asynchronous remote store instruction: an efficient, multi-purpose, virtualisation-friendly hardware extension for low-cost inter-core communication. A remote store instruction is issued by the producer core, but is completed asynchronously by the consumer core, providing direct transfer of data. Remote stores are a straightforward architectural change, easily comprehensible, and binary-compatible with existing time-sharing operating systems. I describe their semantics in Section 4.3.

In Section 4.4, I describe the implementation of this instruction in a simulated multicore processor.

In Section 4.5, I investigate several common patterns of inter-thread communication and synchronisation. I evaluate two concurrent kernels and three complete applications.

The kernels are FIFO queues and synchronisation barriers. In each case, using remote stores yields significant performance advantages over state-of-the-art algorithms relying on cache coherency.

The first full application is a variably-grained numerical benchmark, parallelised using the MapReduce pattern. By varying the granularity of the parallelism, we can quantify the reduction in minimum feasible granularity. We see that remote stores allow many times more fine-grained parallelism than is currently available.

The second full application is a function-call profiler, parallelised using remote stores. I show that remote stores allow efficient offloading of computation to a helper thread, in a situation where cache-coherent communication is too expensive to be worthwhile.

Finally, I use remote stores to implement the parallel STM scheme from Chapter 3. It reduces the average overhead of the invalidation-based STM scheme in my simulated processor from 151% to 83%.

4.2 Background

The high cost of explicit communication in cache-coherent shared-memory systems is widely recognised, and researchers have proposed a number of responses. To address this problem in a general way, a communication mechanism must not only have high performance, but also be efficiently virtualisable, so that an arbitrary number of programs can share the hardware under the supervision of a time-sharing operating system.

4.2.1 Cache-aware Algorithms

A number of algorithms seek to mitigate the impact of cache coherence on communication, on existing hardware. These include FIFO queues such as MCRingBuffer [82], FastForward [47] and CAB [56] and a variety of barrier algorithms [7]. These algorithms still suffer painful coherence penalties, and often make substantial tradeoffs (for example, sacrificing latency in FIFO queues) in order to mitigate these communication costs.

4.2.2 Write Forwarding and Update Coherence

Write (or data) forwarding [105, 6] modifies the cache coherence system to transfer cache lines from producer to consumer processors before they are requested. As it does not affect memory semantics, write forwarding is trivially virtualisable. But to retain these semantics, it must perform the same cache transactions and synchronisation as a pure-software algorithm, which hampers performance.

Write forwarding can be performed speculatively by the cache system [24, 73], but this requires complex hardware predictors. What's more, the timing of this intervention must also be predicted, and misprediction can aggravate the problem of coherence misses [24].

Some early parallel machines supported write-update coherence schemes, which effectively forward a copy of each update to any processor which holds the affected data in its cache [27]. However, the interconnect traffic such schemes generate led to poor performance. Hybrid schemes were proposed, which adaptively switch modes [50] or invalidate unused copies [99] – essentially converging with speculative forwarding schemes. But commercial architectures generally abandoned update-based coherence protocols in the 1990s.

Alternatively, forwarding (or selective update) can be initiated by an explicit instruction [2, 77, 105, 46]. Although explicit write forwarding was shown to have some benefit for "synchronisation kernels" similar to the communication patterns evaluated in this paper [2], more recent investigation suggests it provides little or no improvement for communication-intensive fine-grained algorithms [107].

4.2.3 Abandoning Uniform Shared Memory

Some designs abandon the abstraction of a uniform shared memory, and make all communication explicit, by associating particular memories with physical locations. These memories can be entirely disjoint and communicate via DMA, as in IBM's Cell [54], or coexist with coherent caches [72]. Intel's SCC [71] combines non-coherent caches with uncached on-chip message-passing memories. Several architectures allow direct operations on remote private memories, ranging from embedded designs [69] to HPC machines [74].

All these architectures share the same obstacle to virtualisation. When a program is written for a finite per-processor resource, such as a fixed-size private memory, it is difficult to adapt to a change in availability of that resource – either through running on a different machine, or sharing it with other processors. This makes such resources difficult to virtualise for general-purpose computing.

4.2.4 Dedicated Message-Passing Hardware

Early parallel computers such as the Transputer [70] provided direct channel communication between cores. More recent work includes XMOS [90], proposals for a hardware synchronisation array [106], and the RAW [130] and Tilera [13] designs. The latter two incorporate a scalar operand network, which transmits words on a statically-routed hardware network between cores. This yields extremely high performance, and can be adapted to different algorithms. However, it is almost impossible to virtualise efficiently, because program state is stored not only in memory but also in the network. Such machines are carving out a thriving niche in embedded applications, where virtualisation is less of a concern.

As on-chip interconnect resources are already at a premium, direct communication may instead be passed over the existing memory interconnect, and buffered in memory [107]. However, the difficulties of virtualisation and sharing a finite number of hardware resources remain.

A more flexible machine was Alewife [3], whose message-passing unit contained a dedicated message queue, and could interrupt the processor to handle incoming "active messages". But the overhead of an interrupt for each message was found to be particularly expensive, even on a processor designed for fast interrupts, and polling mechanisms were preferred. On more modern CMPs, this cost might be mitigated by running active messages in SMT strands, rather than using interrupts [10]. However, the ability of long-running handlers to stall the on-chip interconnect, along with message queueing, still present difficulties for safe virtualisation.

4.2.5 A Message Passing Coprocessor

The HAQu [83] system augments a cache-coherent CMP with what is effectively a message-passing coprocessor. Given a FIFO queue with a specific memory layout, HAQu provides a hardware implementation of the enqueue and dequeue operations, based on MCRingBuffer [82].

Although the hardware performs the same calculations and memory accesses as the equivalent software queue, HAQu is significantly faster. The coprocessor caches queue head and tail pointers, and pipelines the index computation. A relaxed memory ordering allows subsequent instructions to overtake queue operations which would otherwise cause coherence stalls.

The complex HAQu instructions represent a departure from the RISC tradition of the past few decades, and the complexity of the queue coprocessor is significant. For example, the coprocessor requires direct access to the L1 cache, and contains within it a content-addressable cache of the head and tail pointers for each of the 16 most recently-acessed queues. A single queue instruction can involve two memory reads, two writes, and an arithmetic operation. Moreover, all this complexity is in the service of a single communication pattern: the single-reader, single-writer FIFO queue, with a hardware-defined memory layout.

For all that, HAQu is highly performant and fully virtualisable by existing operating systems. This makes it the closest approach I have yet found to our target criteria.

4.2.6 Possible Applications

A number of researchers have proposed fine-grained parallelisation schemes which are limited by the cost of cache-coherent communication, sometimes even describing ad-hoc hardware support for their designs. These proposals include Distributed Software Pipelining for automatic parallelisation [106], dynamic information flow tracking for security auditing [97, 112], dynamic program analysis [56, 137], MMT for offloading memory management overheads [131], and HeapMon for memory debugging [117].

Each of these proposals would likely benefit from the adoption of an efficient general-purpose communication system into mainstream architectures.

4.3 The StRemote Instruction

I propose a new unprivileged instruction, the asynchronous remote store. It takes three register operands:

```
StRemote [dest-addr], [src-data], [dest-core#]
```

When a processor executes this instruction, it causes the core named by [dest-core#] to write data from the [src-data] register into a virtual memory address provided by the [dest-addr] register. The destination core asynchronously performs a store operation, using data from the originating core's registers.

This operation combines *asynchrony*, a selective but intuitively comprehensible relaxation of memory ordering semantics, and *direct delivery*, transmitting messages directly – and once only – to their destination.

Asynchrony means that the results of a remote store need not be immediately visible. This allows the originating core to continue execution immediately, without securing access to the relevant cache line, or coordinating with any other core to maintain memory consistency.

Despite this relaxation of memory consistency, the programmer's model remains simple: The specified data will be written to the specified address, but at some point in the future.

Direct delivery means that the store is completed into the destination core's local cache. When the receiving thread reads from the target address, a cache hit occurs and execution can continue promptly.

Should the target address not already be held exclusively in the destination cache, the destination core need not stall. As I discuss in in Section 4.3.1, remote stores have few ordering requirements with respect to local memory accesses. The destination core therefore continues to execute while the miss is processed, the target line is fetched exclusively into the destination cache, and the remote store is completed.

Thus, data is transmitted from the originating to the destination core precisely once. We avoid wasting time, energy and memory bandwidth moving the cache line into and out of an originating core which is not concerned with its contents. Neither sending nor receiving thread need pause for longer than the latency of a cache hit.

4.3.1 Ordering

I make three guarantees about the ordering of remote-store fulfilment:

1. Two remote stores issued from the same origin to the same destination will become globally visible in program order. This allows the safe composition of remote stores – for example, to write new data into a queue and then update a pointer to indicate its arrival.

2. Remote stores will not become visible to the destination processor before any preceding writes by the originating processor. This is an intuitive fit with x86 write ordering semantics, which do not permit a later store to overtake one issued earlier. It also allows programs to publish data safely between threads, without incurring the cost of memory barrier operations. For example, a producer thread might update a shared data structure using ordinary memory-access instructions, then use a remote store to notify a consumer thread of this update. Under this rule, that operation will complete safely without memory barriers.

3. If a processor executes a remote store flush or TLB flush instruction (see Section 4.3.2), any remote stores previously issued by that processor will be globally visible before the flush instruction completes.

4.3.2 Robustness

Remote stores are suitable for use by untrusted software, because they are robust to misuse and changing circumstances, and do not require cooperation from other software components. This makes the remote store suitable as a user-mode instruction in a virtualised system.

Mis-Addressed Stores

Remote stores allow programs to specify the destination processor directly, avoiding the architectural complexity of a hardware "thread translation table" suggested by some other designs [23]. This is permissible, because a mis-addressed remote store has the same semantics as if it were correctly addressed. If a remote store is incorrectly addressed, the program incurs only a performance penalty – a coherence miss as the "correct" receiving core retrieves the data from whichever cache it has landed in.

Consider a receiving thread, polling for data on processor A. We use a remote store to supply new data, but mistakenly direct it to processor B instead. When processor B completes the remote store, it experiences a coherence miss because A is holding the relevant cache line. Thanks to the relaxed ordering described in Section 4.3.1, this does not affect local code running on processor B. When A next polls for data, it incurs another coherence miss, and returns the line to its own cache. Although the receiving thread is running slower than it might like, none of its expectations regarding remote store completion have been broken.

Because remote stores are robust to misaddressed messages, they are suitable for use in an environment in which threads may be migrated without warning from one processor to another. The user-mode software need only make a "best-effort" attempt to direct remote stores to the correct destination processor, using system calls such as Linux's `sched_getcpu()`.

Virtual Memory and Protection

Because all communication occurs through memory, we can use virtual memory to isolate processes from each other. A remote store to an unmapped or read-only page will incur a page fault, like any other illegal write.

Because a remote store cannot affect code running on the destination processor, address translation must be performed on the originating processor. That is, if the originating core executes a remote store instruction without incurring a fault, the write is guaranteed to complete eventually. The hardware message sent to the destination processor must therefore contain a physical address. (If the destination cache is virtually indexed, a few bits of the virtual address will also be required.)

This presents a challenge when unmapping or remapping pages. The situation to avoid is one in which the operating system unmaps a page while a remote store to that page is "in flight". The write could then be lost, or corrupt data belonging to another process.

We prevent this by flushing remote stores whenever we flush the TLB. Whenever a remote write completes, the destination core signals this to the originating core. Each core maintains a counter of outstanding remote store requests, incrementing each time it executes a remote store instruction, and decrementing each time it receives an acknowledgement. This allows us to implement a "remote-store flush" operation, in which a processor stalls until all its outstanding remote stores have completed. We provide a user-mode flush instruction, for code such as garbage collectors which needs to reason about remote stores.

We then modify the semantics of the TLB flush instruction to imply a remote-store flush. That is, each time the TLB is flushed, we wait for the completion of any remote stores that might have been based on entries in that TLB. This ensures that the operating system will flush all remote writes to a page before it is unmapped, using the existing TLB shootdown mechanisms. What's more, because the flush incurs no penalty if there are no remote stores outstanding, we accomplish this with no impact on the performance of existing applications.

Even for applications which make heavy use of remote stores, this is unlikely to damage performance. Exact figures are hard to find, but a TLB flush on a modern desktop processor takes most of a microsecond [134]. If the round-trip time of a remote store and its acknowledgement is equivalent to a coherence read miss, it will take a few dozen nanoseconds [95]. A remote store issued directly before a TLB flush will therefore normally be acknowledged long before the TLB flush completes.

This approach ensures binary backward compatibility. Existing operating systems with no awareness of remote stores will correctly manage virtual memory for userspace applications which use this new instruction. Remote stores are a compatible, incremental extension to existing architectures.

4.3.3 Applicability

The remote store operation is not specific to any particular data structure or pattern of inter-core communication, and affords great freedom to algorithm designers. In Section 4.5, we suggest algorithms for some common patterns of communication and synchronisation, but these are just examples.

4.4 Implementation

4.4.1 Architecture

I have implemented the remote store operation in the Gem5 simulator [14]. I simulate a four-core cache-coherent system with a three-level cache heirarchy, with cache bandwidth and latency based on a modern CMP architecture, Intel's Nehalem [95].

Remote stores are transmitted into the memory system by the originating processor's memory access logic, in the same way as normal stores. The packet header contains a flag to indicate remote stores, and the number of the destination processor. Routing logic in the caches forwards these packets to the destination processor, which performs a standard store operation to complete the remote store. The interconnect in this model preserves packet ordering, and the destination processor contains a remote-store queue, which buffers remote-store requests while previous stores are being completed. This ensures that remote stores complete according to the ordering guarantees set out in Section 4.3.1.

Limitations

I use the Gem5 "classic" cache model, which models a bus-based MESI coherent system. (Although Gem5 includes more sophisticated network-on-chip models, they are not yet compatible with the superscalar processor model.) I have modified the snoop logic to provide an extra latency penalty for coherence misses, to match the empirical results on Nehalem systems. Although I am interested in the implications

of the x86 architecture's strong memory model, I simulate the ARM instruction set in order to work around bugs in Gem5's x86 support.

4.4.2 Memory Consistency and Ordering

The x86 architecture, which accounts for the lion's share of the world's chip multi-processors, provides quite a strongly ordered memory model [101]. In particular, stores must always become visible in program order. When a processor makes a communicating store, subsequent stores will therefore back up in the write buffer, stalling the processor until the coherence miss is complete.

I wished to evaluate the interaction between the relaxed ordering of remote stores and the strong store ordering of x86. Unfortunately, Gem5 cannot model x86 ordering semantics in a superscalar processor. It does, however, offer models spanning the design spectrum from very weak to very strong memory ordering. We compare an aggressively superscalar, out-of-order CPU with weak memory ordering (*O3CPU*) against a simple in-order model with sequentially consistent memory (*TimingSimpleCPU*). As both CPU models use the same core clock frequency and the same memory infrastructure, we can usefully compare their performance in absolute terms. I therefore plot both models on the same scale throughout.

With the *TimingSimple* model, I hope to shed some light on the benefits of asynchrony in more strongly ordered architectures such as x86. With the *O3* model, I provide a fair test of our proposal on an aggressive superscalar architecture, whose features are designed to mitigate some of the problems I am attempting to address.

4.4.3 Interconnect

I do not add extra networks or interconnect bandwidth to our system. Instead, my implementation transmits remote store requests as packets across the existing interconnect. Remote store requests occupy the same cache ports and buses as memory requests, and their minimum end-to-end latency is therefore slightly more than half that of a coherence miss.

When a remote store request reaches the destination processor, it completes as if it were a local write by that processor. Remote store completion occupies the same cache and interconnect resources as any other write. When the write completes, an acknowledgement is sent back to the originating processor, also over the same interconnect.

A remote store therefore occupies the destination CPU's L1 cache port four times: receiving the remote store request, issuing the store request to complete the remote store, receiving the response to the completing store, and sending the acknowledgement to the originating core. A more realistic implementation would involve modifications to the first-level cache, allowing it to complete remote stores directly. This would yield higher performance than my model.

As with all memory requests in the Gem5 classic cache model, flow control for remote stores is enforced by interconnect occupancy. The interconnect does not permit remote store messages to overtake one another.

Deadlock avoidance

This scheme presents the possibility of deadlock. If the interconnect is filled with remote store requests, they might block the writes which would fulfil these requests, preventing forward progress. We avoid this scenario by allowing normal memory packets to overtake remote stores in caches. This does not affect the semantics of remote stores: by the rules in Section 4.3.1, any memory request may overtake a remote store.

A more realistic CMP implementation would have a more sophisticated interconnect than our bus-based model, and could solve this scheduling problem more fairly by separating traffic across virtual channels in an on-chip network. This could also be used to ensure liveness of remote stores: under the current model, a flood of conventional memory accesses could starve remote stores indefinitely.

4.5 Evaluation

I evaluate remote stores against my design goals of efficiency, wide applicability to different patterns of communication, and suitability for fine-grained parallelism.

In this section, I present algorithms using remote stores for two patterns of inter-thread communication: FIFO queues and synchronisation barriers. I compare the performance of these algorithms with unaccelerated equivalents, and find that remote stores provide substantial speedups.

To quantify the benefits to fine-grained parallelism, I evaluate a numerical computation which uses a popular pattern of inter-core communication, MapReduce [34]. I find that by using remote stores, we can divide work much more finely between threads than is possible with cache-coherent communication.

Finally, I use remote stores to parallelise a systems task: function-call profiling. I find that remote stores allow us to parallelise applications for which the overhead of cache-coherent communication would otherwise be forbidding.

4.5.1 FIFO queues

FIFO queues are a common abstraction for communication in parallel programs. Particularly demanding applications include high-frequency streaming techniques such as decoupled software pipelining [106], dynamic program analysis for debugging [56, 137] and security [97, 112], and distributed operating systems [11]. Research into all these applications has found them to be limited by cache-coherence overheads, or has resorted to heavyweight hardware acceleration to avoid them.

In Figure 4.1, I present an algorithm for a single-reader, single-writer FIFO queue using remote stores. The producer and consumer maintain private copies of the read and write pointers in their local caches. When an enqueue or dequeue operation occurs, they use a remote store to update the pointer at the other end. In addition, the enqueue operation uses a remote store to transmit each item to the receiving processor. Thus, this algorithm does not use cache-coherency to transfer data.

I compare this against a high-performance software queue for cache-coherent systems, MCRingBuffer [82] ("MCRB"). In this algorithm, the producer and consumer maintain conservative private copies of the read and write pointers, respectively. Only when these conservative limits are reached are they updated, thus minimising contention over these pointers.

Throughput and Latency

First, I measure the standard parameters of any communication link: the minimum latency of an item passing through the FIFO, and the maximum throughput under optimal conditions. These are plotted in Figure 4.2.

I measure latency by passing a message back and forth between two cores on the same chip 2,000 times, and throughput by conducting a unidirectional transfer of 10,000 words from one core to another. Each experiment was timed with the Gem5 timer (`m5_rpns` instruction). The simulator is deterministic, so there is no variation between multiple runs of a benchmark.

In order to obtain optimal results from MCRingBuffer, I start the throughput benchmark with the buffer already half-full. This minimises cache contention on the data buffer and, more importantly, the head and tail pointers.

As we see in Figure 4.2, the remote store algorithm provides an order-of-magnitude improvement in latency, because it suffers no cache contention. When polling for new data, both software queues suffer

```
struct FIFO {
  int enq_write;
  int enq_read;
  /* Cache padding */
  int deq_write;
  int deq_read;
  /* Cache padding */
  Item data[SIZE];
}
```

```
void enqueue(FIFO f, Item i) {
  while(f.enq_write % SIZE
        == f.enq_read - 1)
  { /* block */ }

  StRemote(f.data[f.enq_write++] = i,
           CONSUMER_CORE);

  f.enq_write %= SIZE;

  StRemote(f.deq_write = f.enq_write,
           CONSUMER_CORE);
}
```

```
Item dequeue(FIFO f) {
  while(f.deq_write == f.deq_read)
  { /* block */ }

  Item value = f.data[f.deq_read++];

  f.deq_read %= SIZE;

  StRemote(f.enq_read = f.deq_read,
           PRODUCER_CORE);

  return value;
}
```

Figure 4.1: A contention-free FIFO queue using remote stores.

heavy contention over the head pointer and data buffer.

Remote stores provide no improvement over MCRingBuffer in the optimal throughput tests. This is unsurprising: under the optimal conditions I constructed, MCRingBuffer does not contend over queue pointers. It issues a long string of consecutive reads of the data buffer, which amortise the cost of coherence misses.

Communication Operation Costs

Latency and throughput are not the only constraints on inter-thread communication. In general, communicating threads will perform other work as well as the communication itself. It is the time required to execute a communication operation before the thread can return to its work, or COMM-OP delay [107], which often limits the usefulness of fine-grained parallelism.

In Figure 4.3, I measure COMM-OP delays in three different regimes: when the FIFO is empty, because the consumer is processing items faster than the producer; when the FIFO is full, because the producer is generating items faster than the consumer can remove them; and in a perfectly balanced state, where the FIFO is partially full and neither producer nor consumer is faster than the other. These measurements were obtained by measuring the simulated time taken to perform each operation 4,000 times, in the presence of a second thread which maintained the FIFO in the target state.

Consistent with the results from Figure 4.2, the cost of enqueueing and dequeueing from an MCRingBuffer queue is small under optimal conditions. However, a perfectly balanced FIFO is unstable; in real life, one component or the other will run faster, and the queue will be frequently full or empty. In these situations, any software queue must contend over some cache line containing synchronisation state – in this case, a head or tail pointer. This produces large slowdowns in both software queues. Remote stores, of course, perform consistently well throughout.

Thanks to the speculative prefetcher in the O3 superscalar model, repeated MCRingBuffer dequeue operations are nearly twice as fast as enqueues when the data buffer is partially full. This can be seen in the "balanced" condition in Figure 4.2. This prevented me from evaluating the operation costs in the "full" condition, as it was impossible for the producer thread to keep the FIFO full for long enough. Under a more realistic workload, in which the consumer thread does some work between FIFO dequeues, this "blocking full" scenario is much more likely.

One might expect that, on a weakly-ordered superscalar processor, enqueuing to an empty list would be faster than we see in Figure 4.3. The processor performs blind writes to the data buffer and then to the consumer-visible copy of the head pointer. Execution could then continue in parallel with the cache miss. However, a memory barrier is required between the write to the data buffer and the pointer update, to ensure that the data is actually present in the buffer before the consumer attempts to read it. This

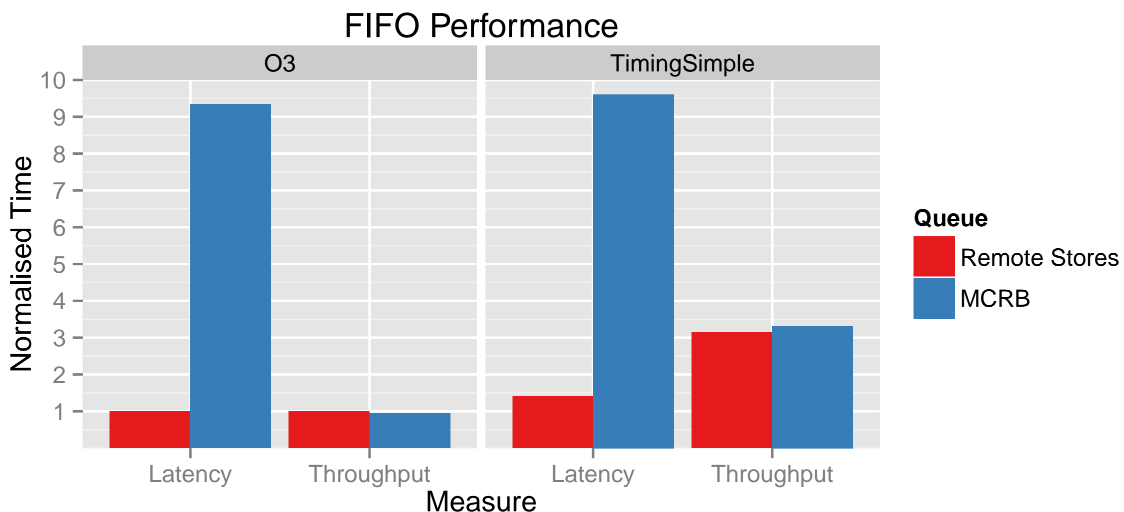


Figure 4.2: Latency and throughput of FIFO queues. Lower times are better, and measurements are normalised to the performance of the remote-store queue on the O3 (weakly-ordered superscalar) CPU model.

forces the update to stall (and, because ARM lacks a store-store barrier of the sort that is always implicit in x86, stalls subsequent reads as well) until the cache miss on the data buffer is complete. This goes to show that even very under very weak memory ordering, store misses can still degrade communication performance.

Identifying Efficiency

The advantages of remote stores stem from avoiding cache misses. In Figure 4.4, I plot the time spent by each FIFO algorithm in misses of the Level-1 cache on the *TimingSimple* model.

We can see that cache-miss penalties dominate the cost for the unaccelerated software algorithms. The remaining overhead represents the indirect cost of cache misses on the other core – a hypothesis supported by the high performance achieved by MCRingBuffer when *not* experiencing cache contention, in the perfectly balanced throughput case. Because remote stores allow communicating memory locations to remain exclusive to a single processor's cache at all times, the remote-store-based algorithm experiences a negligible number of cache misses.

We should also consider these algorithms' use of on-chip interconnect bandwidth. In Figure 4.5, I plot the shared (level three) bus utilisation of each algorithm. This is necessarily an approximate measure, which will depend on the interconnect mechanism. I model remote stores as a two-flit packet (header and data), and cache-to-cache transfers as a seventeen-flit packet (header and sixteen words of data for a 64-byte cache line). Thus, the absolute bandwidth cost of a remote store is substantially less than that of transferring an entire cache line, but the overhead per word of data is nearly 100%.

As we see in Figure 4.5, however, the bus spends most of the time idle even in high-throughput tests. We conclude that latency, as manifest in cache-miss delays, is much more important to the performance of these algorithms than bandwidth.

I compare the bus activity in the optimal throughput condition, where remote stores and MCRing-Buffer perform comparably, in Figure 4.6. Although the absolute bus utilisation of the remote-store algorithm is significantly higher than that of MCRingBuffer under optimal conditions, this does not limit performance: in both cases, the bus remains idle over 90% of the time.

This is a specific example of a more general trend in chip design. As feature sizes shrink and power

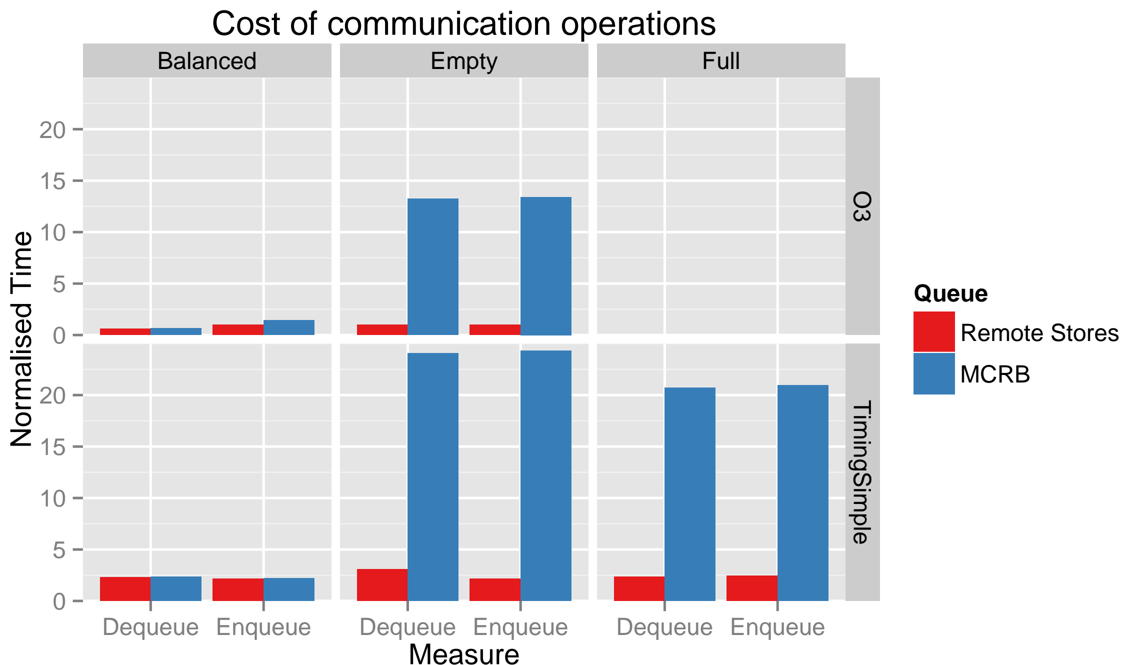


Figure 4.3: Cost of communication operations in different FIFO states. Performance is normalised to the performance of the remote-store queue on the O3 (weakly-ordered superscalar) CPU model. I could not accurately measure the "Full" condition on the O3 model, due to the activity of the prefetcher – see text for details.

becomes more crucial, bandwidth continues to scale, but achieving low latency becomes increasingly difficult. It would, of course, be possible to reduce the overhead of remote stores, for example by using a write-combining buffer to merge successive remote stores into a single packet. However, these results suggest that this would be an incremental optimisation, rather than a necessity in order for the scheme to be viable.

Conclusions:

Remote stores offer an order-of-magnitude improvement in latency over even cache-aware software FIFO queues. While the maximum throughput of a software queue matches that of a remote-store queue, that optimum is unstable and difficult to achieve in practice, and departures from that optimum cause cripplingly large coherence delays in enqueue and dequeue operations. Remote stores allow the implementation of a low-latency, high-throughput queue with consistently good performance.

4.5.2 Synchronisation Barriers

Barriers are a standard synchronisation mechanism for phased programs, in which each of a group of threads must have completed one phase before proceeding to the next. Barriers are so common in certain numerical applications, and current software implementations so limiting, that dedicated hardware to accelerate them is standard in high performance computing (HPC) systems, and has been proposed for CMPs [114]. The barrier pattern also occurs in the operating system "TLB shootdown" procedure, where it is considered a serious bottleneck [11, 134].

In Figure 4.7, I provide a simple algorithm for a synchronisation barrier using remote stores. We compare this algorithm with a straightforward counting barrier, and a more sophisticated cache-aware algorithm, the tournament barrier [7].

For the purposes of this experiment, I extended the experimental setup to model up to sixteen cores per chip, with a proportional increase in shared bus bandwidth for runs using more than four cores. Figure 4.8 shows the performance of these three barrier algorithms with increasing numbers of threads.

We can see that the remote-store-based algorithm outperforms even the tournament barrier by an order of magnitude.

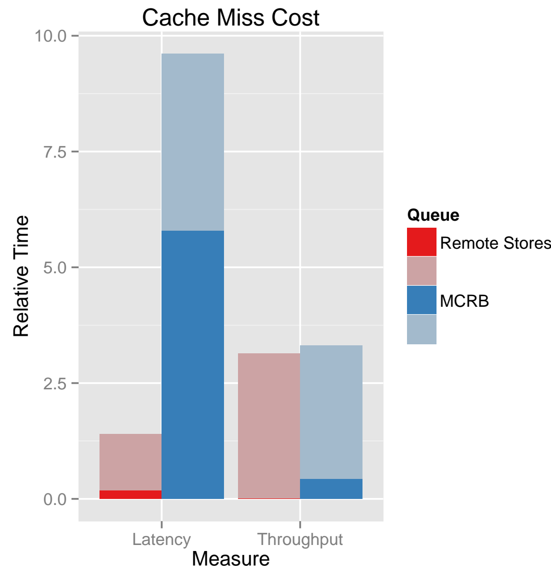


Figure 4.4: Time spent in level-one cache misses, for each algorithm running on the *TimingSimple* model. Dark colours represent time spent on cache misses; light colours represent all other runtime. Results are normalised to the performance of the remote-store queue on the O3 model.

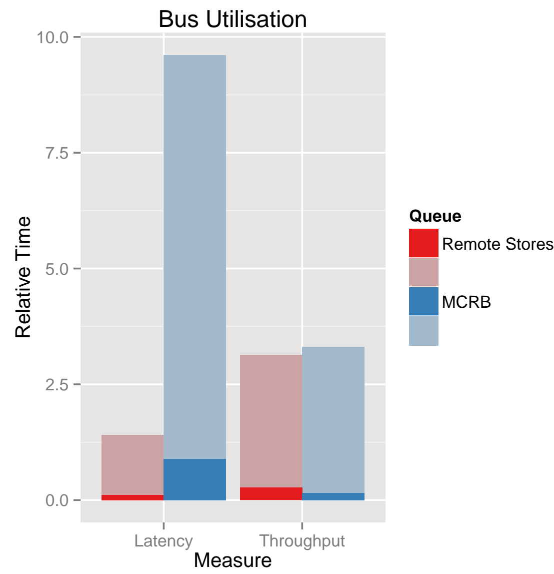


Figure 4.5: Bus occupancy, for each algorithm running on the *TimingSimple* model. Dark colours represent time during which the shared (level three) bus is occupied; light colours represent all other runtime. Results are normalised to the performance of the remote-store queue on the O3 model.

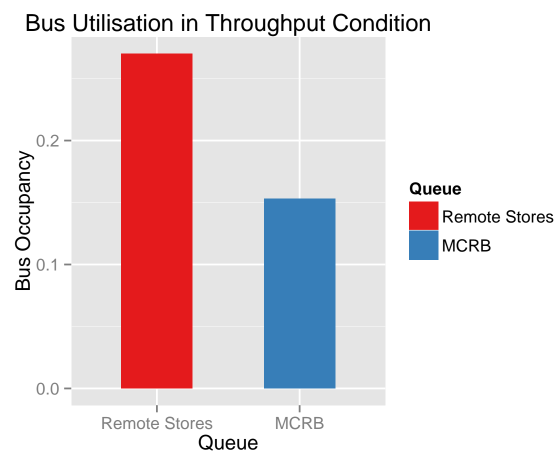


Figure 4.6: Bus occupancy, for remote-store and MCRB FIFOs in the optimal throughput condition on the *TimingSimple* model. Occupied time is plotted on the same numerical scale as in Figure 4.5.

4.5.3 Quantifying Fine Grained Parallelism with Map-Reduce

One common complaint about cache-coherent shared-memory systems is that they do not reward fine-grained parallelism. If a problem can be divided up coarsely between processors, it is easy to overcome the overhead of inter-thread coordination. If the problem is too small or too dynamic, the overhead of work distribution can overwhelm the benefits of parallelism.

To quantify the feasibility of fine-grained parallelism, I evaluate a numerical application with variable granularity. By changing this granularity, I can systematically vary the ratio between computation and communication. We are looking for the "break-even point": the smallest ratio for which the parallel speed-up outweighs the coordination overhead, making parallelisation worthwhile. The sooner we break even, the more fine-grained parallelism we can support.

I constructed a numerical benchmark using the MapReduce pattern of parallel calculation [34]. In this pattern, a controller thread divides up a workload between several workers, communicates assigned work to each, collects the results from each worker, and then combines them. Both computation and communication are therefore on the critical path. We vary the granularity by adjusting the workload size. Although absolute workload sizes are meaningful only for this algorithm, an improvement in the break-even granularity thanks to cheaper communication is a general result.

This benchmark calculates the dot-product of a vector with itself, squaring each element in the "map" phase and summing the results in the "reduce" phase. The granularity of parallelism can therefore be varied by changing the length of the vector. A large vector exhibits coarse-grained parallelism, with each processor assigned a large chunk of work, whereas a small vector exhibits fine-grained parallelism, with each core performing only a few arithmetic operations before synchronising with the other threads. The calculation is repeated 600 times, measuring the total time with the M5 timer instruction.

The two versions of my MapReduce code differ only in whether they use remote stores or a per-core "hot-box" location to distribute work and collect results. For each size of vector, the two parallel systems' performance is compared with a sequential computation. That is to say, I plot the ratio between parallel and sequential run-times for each granularity.

I plot the results in Figure 4.9. We see that by using remote stores, we can usefully parallelise at a much finer grain than allowed by cache-coherent communication. Using our strongly-ordered CPU model, we see useful speedups on vectors as small as 20 elements when using remote stores. Without them, we do not see speedups with fewer than 128 elements. This represents a sixfold improvement in the minimum computation-communication ratio, and therefore the minimum granularity of useful parallelism.

Remote stores continue to allow superior performance, even on workloads much larger than the break-even point for cache-coherent communication. We can say, then, that remote stores not only enable more widespread application of fine-grained parallelism, but make existing applications more compelling.

4.5.4 Parallelising Dynamic Analysis

Dynamic analyses such as function-call profiling are useful but costly. Parallelising the collection and computation of profiling data has the potential to reduce these overheads, but the communication cost has proven challenging or prohibitive on cache-coherent systems [56, 137].

I have implemented a profiler which measures the amount of time a program spends in each function. Using GCC's `-finstrument-functions` feature, I trigger a function call to the profiler each time execution enters or exits a function in the profiled code.

The profiling code uses a time-stamp instruction to measure the time spent in each function, using a hash table on the function's entry address to store cumulative time per function. A push-down stack

```
typedef
  int * [NTHREADS]
 barrier;
```

```
void barrier_master(barrier b) {
  for(t=1; t<NTHREADS; t++) {
    while(b[t]==NULL);
  }
  // everyone is here
  for(t=1; t<NTHREADS; t++) {
    int * notify = b[t];
    b[t] = NULL;
    StRemote(*notify = 1, t);
  }
}
```

```
void enter_barrier(barrier b) {
  if(MY_THREAD==0) {
    barrier_master(b);
  } else {
    int notify = 0;
    StRemote(b[MY_THREAD] = &notify, 0);

    while(!notify);
  }
}
```

Figure 4.7: A contention-free synchronisation barrier using remote stores.

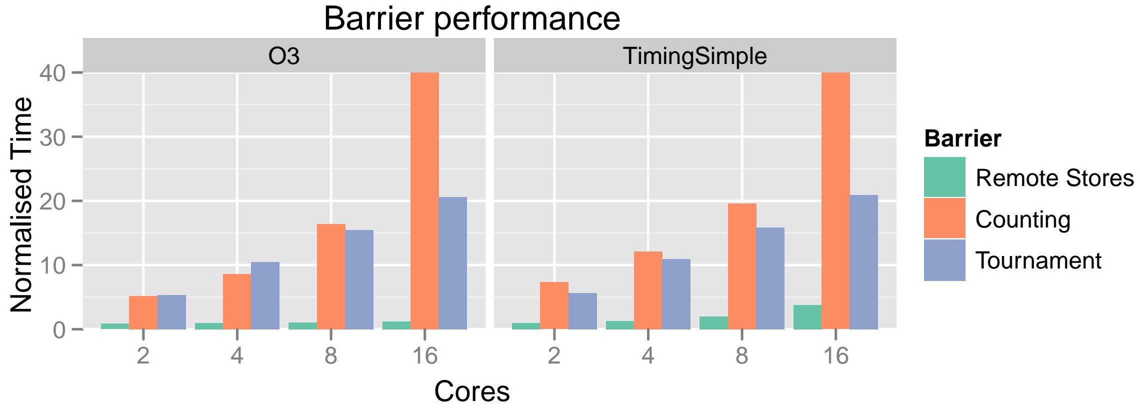


Figure 4.8: Performance of a simple remote-store barrier, compared with counting and tournament barriers. Runtime is normalised to the performance of the remote-store barrier on the O3 (weakly-ordered superscalar) CPU model.

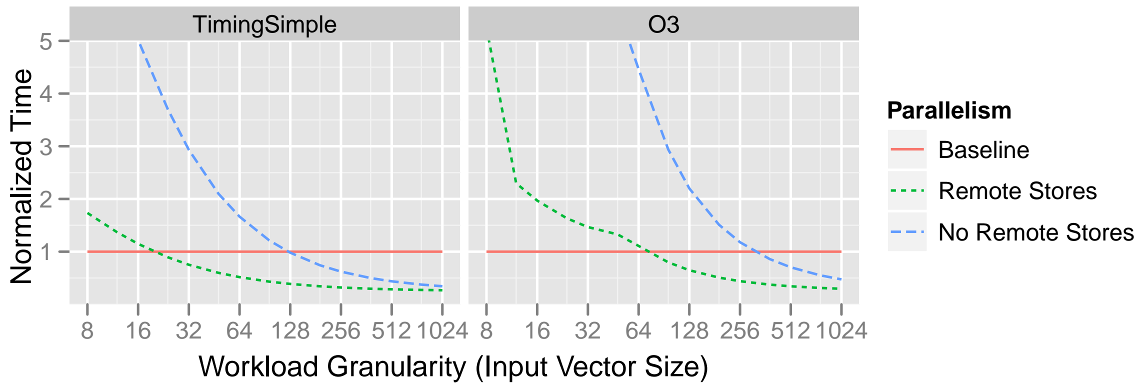


Figure 4.9: Parallel overheads for fine-grained Map-Reduce. For each workload size, the parallel runtime is plotted as a proportion of the baseline single-threaded runtime. Parallel runs were conducted with four processors.

models the program stack, so that further runtime is correctly assigned to the calling function after a callee returns. This is a straightforward, inline profiler: it performs all its work in the same thread as the profiled workload, during the instrumentation function call.

I then parallelised this profiler, using a helper thread to perform the analysis. The main thread streams events (function entries and exits) to the helper thread, using a FIFO queue based on those described in Section 4.5.1.

I measured the runtime overhead of this profiling system, evaluating the performance of the parallel profiler when using either MCRingBuffer or the FIFO queue described in Section 4.5.1 to transfer the data. I compared these to the overhead of the inline implementation.

I profiled seven single-threaded integer benchmarks from the SPEC CPU2000 suite [125]: `bzip2`, `crafty`, `gzip`, `mcf`, `parser`, `twolf` and `vpr`. Execution time was measured as the simulated execution time of each benchmark program from launch to exit. As they were running in simulation, I used the smallest MinneSPEC [75] workload for each benchmark. I present the results in Figure 4.10.

We see that when we use remote stores for communication, we can successfully parallelise this analysis. Using remote stores to parallelise profiling achieved overheads 38-42% lower than when profiling inline. By contrast, the overheads of communicating such fine-grained events over cache-coherent shared memory overwhelmed the benefits of parallelism: the overhead of MCRingBuffer-based parallel profiling is significantly greater than that of inline code.

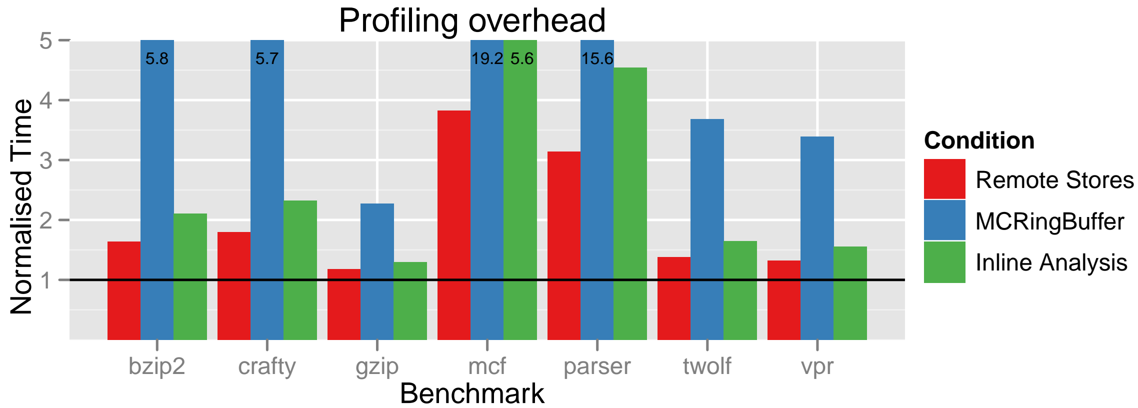


Figure 4.10: Overheads for function-call profiling in benchmarks from the SPEC2000 benchmark suite. This graph compares inline profiling, running on a single core, with parallel profiling running on one application and one helper core. Runtimes are normalised to the runtime of the uninstrumented program. Due to the size of the benchmarks, measurements were made on the *TimingSimple* model only.

4.5.5 Parallel STM

Background and Porting

Software transactional memory (STM) is a parallel programming model with promising usability characteristics (Chapter 2). However, it suffers from large serial overheads which limit its applicability to real-world programs. In Chapter 3, I proposed a scheme for offloading the overhead of STM instrumentation onto a helper processor. I empirically assessed its sensitivity to the costs of communication, which I modelled with an ideal core-to-core blocking FIFO in hardware. As I discuss in Section 4.2, this approach is not practical outside embedded designs.

In this final section, I evaluate the feasibility of using remote stores to implement a parallel STM algorithm, and reduce the serial overhead of STM.

This is essentially a repeat of the overhead measurements conducted in Chapter 3. I performed a direct port the invalidation-based STM system described in Section 3.3 to the platform described in Section 4.4. The only modifications were platform-specific changes to memory layout and booting of helper processes, and the replacement of memory-mapped FIFOs (as described in Section 3.4.2) with remote stores.

```
// Thread-local variables

WORD to_helper_buffer[BUFSZ];
WORD * receive_ptr;

/* cache-line padding */

WORD * send_ptr;
WORD from_helper_status;

void txn_init() {
  buf_ptr = to_helper_buffer;
}
```

```
// Application -> Helper

void app_to_helper.send(WORD[] msg) {
  for(WORD w: msg) {
    StRemote(*(send_ptr++) = w,
             HELPER_CORE);
  }
}

WORD[N] app_to_helper.receive() {
  while(! *receive_ptr)
  { /* block */ }

  var result = receive_ptr[0:N];
  receive_ptr[0:N] = 0x0;
  receive_ptr += N;
  return result;
}
```

```
// Helper -> Application

void helper_to_app.send(WORD status) {
  StRemote(from_helper_status = status,
           APP_CORE);
}

WORD helper_to_app.receive() {
  while (!from_helper_status)
  { /* block */ }

  var result = from_helper_status;
  from_helper_status = 0;
  return result;
}
```

Figure 4.11: Pseudocode for communication between application and helper threads in a parallel STM system using asynchronous remote stores. This pseudocode provides implementations of the FIFO `send()` and `receive()` functions in Figure 3.8.

Communication

As the application thread performs a transaction, it creates a transaction read and write log in memory, through remote writes to the helper CPU. Each application thread has a buffer allocated for this purpose. At the end of this buffer is a page which is marked non-readable and non-writable, so that any buffer overflow is caught by the MMU. This allows for minimal instrumentation overhead: a transmission from application to helper thread consists of a single remote store operation, followed by incrementing a pointer. Should a page fault occur as a result of a transaction overflowing this buffer, the buffer can be resized and the transaction restarted. In practice, with the benchmarks described here and a starting buffer size of 16kb per thread, this never happened.

Return communication from helper to application thread consists only of status responses (`COMMIT` and `ABORT`), and requires no FIFO queueing. It is accomplished with a remote store into a status word in the application's thread's state structure.

Pseudocode for the communication between helper and application threads can be found in Figure 4.11.

Evaluation

I measured the single-threaded overhead of the benchmarks described in Section 3.4.3, comparing the performance of both STM systems to the uninstrumented program. The workloads were the same, except for `vacation`, where I used the recommended "simulation" workload to achieve a tractable run-time in the simulator. Run-time was calculated by comparing simulator timestamps (`m5_rpns`) before and after the main section of each benchmark. All runs were performed in the faster *TimingSimple* simulator, due to the size of the workloads.

Figure 4.12 depicts the single-threaded overhead of using the system in inline mode, and in parallel mode using an extra "helper" thread to offload the overhead of STM instrumentation.

We see that parallelising the STM system with remote stores reduces the average serial overhead from 151% to 83%. This improvement is less spectacular than what we have seen on the hardware platform in Chapter 3, but is nonetheless significant.

Breaking Down the Overhead

I break down these overheads by measuring how much of the overhead remains when I omit various components of the parallel STM system. This is the same analysis performed in Section 3.5.4, and the results are shown in Figure 4.13.

We see that a large portion of the overheads are taken up with read and write instumentation. Write instrumentation is more costly in a heavily cached system, as the application thread can incur cache misses on ownership records as well as data. What is more, application and helper cores contend over exclusive access to ownership records: the helper thread modifies orecs on read, and the application thread modifies orecs on write. These costs are aggravated by the *TimingSimple* model's sensitivity to cache misses, as it cannot execute any other instructions while waiting for a memory access to complete.

I conclude that remote stores are a viable means for implementing parallel STM, allowing us to substantially reduce the overhead of transactional memory on a virtualisable, general-purpose processor.

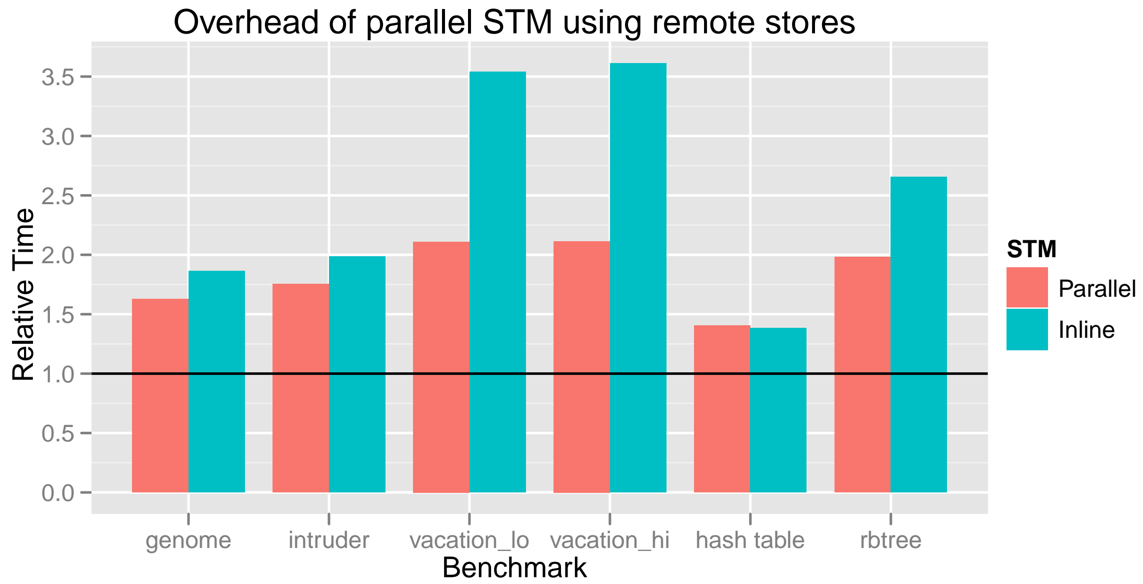


Figure 4.12: Remote stores allow useful parallelisation of an invalidation-based STM system using helper threads. Run-time is plotted as a proportion of uninstrumented run-time. The parallel STM uses two CPUs to run one application thread.

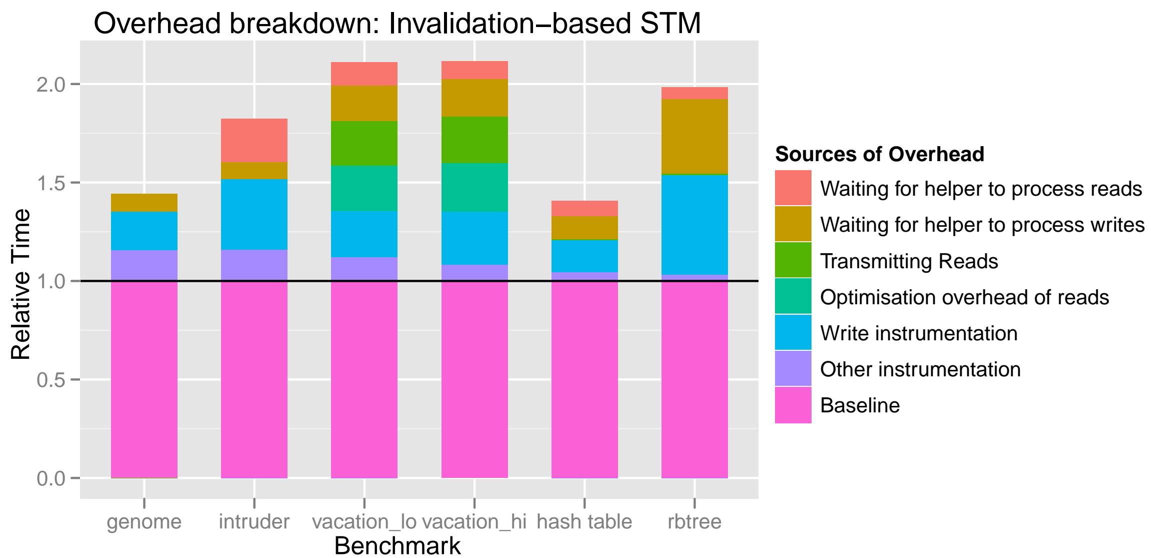


Figure 4.13: A breakdown of the serial overhead of an invalidation-based parallel STM, using remote stores in a simulated system.

4.6 Conclusion

I have described the asynchronous remote store, an elegantly simple but capable general-purpose mechanism for communication. This mechanism complements cache coherency, allowing communicated data to remain exclusive to a single processor at all times. Remote store semantics are straightforward, and lend themselves readily to hardware implementation. What is more, programs using remote stores can be virtualised safely by existing unmodified operating systems.

I have investigated diverse patterns of inter-thread communication, showing that remote stores are applicable to a wide variety of algorithms. By reducing the cost of communication relative to computation, remote stores allow many times more fine-grained parallelism than is available on current commodity machines. I have quantified this improvement in granularity with a variably-grained Map-Reduce benchmark.

Finally, I have demonstrated that remote stores can be used to achieve significant speedups in real-world systems applications such as parallel dynamic analysis, where the cost of communication can otherwise be prohibitive. I have demonstrated that remote stores have sufficiently good performance to support the parallel STM scheme described in Chapter 3, providing substantial decreases in serial overhead. I conclude that remote stores can open up new classes of parallel algorithms, allowing us to more widely exploit the new resources of the multicore world.

Chapter 5

Conclusion

5.1 Introduction

In this report, I have explored the characteristics and challenges of communication between threads in a multi-core processor. I address the challenges arising from the implicit model of communication represented by classic shared-memory parallel programming, and towards which today's cache-coherent processors are aligned.

Such implicit communication is difficult to reason about in the abstract. Keeping track of it computationally, as software transactional memory does, is easier on the programmer but very hard on the computer. Normally, this is a trade-off worth making, but in STM's case the serial overheads drown the parallel benefits. If the costs could be shifted to other processors, the traditional trade of processing power for usability might be salvaged.

As industrial practice increasingly recognises, *explicit* communication is much easier to use correctly, as well as being useful for efficient, structured parallel computation. However, cache-coherent hardware assumes implicit communication, and imposes serious performance penalties upon programs which do communicate explicitly. I have investigated how to support and reward efficient explicit communication in a cache-coherent multiprocessor.

5.2 Usability and Parallel Programming

I have discussed evidence indicating that implicit communication using shared memory, managed by the programmer using mutual-exclusion locks, is difficult to reason about and leads to subtle errors. I have conducted a pilot study into the relative usability of lock-based programming and two proposed alternatives.

The results of this study were inconclusive, and subsequent similar studies with longer tasks, larger cohorts and more diverse objects of comparison also struggle to produce significant results. This demonstrates some of the serious problems with attempting to use quantitative user studies of programming to guide the design of parallel systems. I conclude instead that the design of programming systems, like visual or industrial design, targets a cognitive process too complex to understand formally. This is not to say that the quality of programming languages does not matter – indeed, well-designed programming systems remain critical if we are to cope with the demands of parallel computation. But, as with other informally understood design disciplines, we must rely heavily upon our intuitive understanding of the cognition of programming – in other words, our taste.

5.3 Software Transactional Memory

I have investigated one framework for parallelism that aims to simplify implicit communication: transactional memory. Transactional memory is similar to lock-based programming, but detects and safely manages the communication implied by concurrent modification of shared state.

However, transactional memory requires a significant amount of run-time management, and performing these computations in software (STM) incurs prohibitive overheads. Each thread is slowed down so significantly that it is often not worth parallelising the program in the first place. While hardware acceleration almost eliminates this overhead, it is not currently available in commercial processor architectures, and upcoming implementations will have limited capacity and must fall back to STM for larger transactions.

In the absence of TM-specific hardware, I ask whether high-performance explicit communciation could help us implement STM. I hypothesise the existence of a high-performance interprocessor communication scheme, which I can model using off-the-shelf FPGA tools. I have parallelised a simple STM scheme, by pairing each application thread with a helper processor which performs STM bookkeeping tasks. I have showed that this can reduce the average serial overhead of STM instrumentation by more than half: from 118% or 160% with a standard inline system to 43% or 47% with a helper thread.

5.4 Hardware Support for Communication

I then ask how we would realise this assumption: How *should* one provide high-performance explicit communication for a general-purpose multi-core processor?

The memory-mapped FIFOs in my FPGA system are all very well for an embedded system, but they cannot be shared between programs by a multitasking operating system. I have reviewed a number of hardware-based communication schemes, and they suffer from the same problem. In addition, to justify inclusion in a commercial micro-architecture, a communication scheme must be as general as possible. Hardware should not dictate a communication pattern to software, but should be exploitable in as many different contexts as possible. Existing proposals for hardware-accelerated communication often prescribe a single pattern, usually FIFO channel communication.

I have instead proposed a new, more general scheme for explicit inter-core communication: the asynchronous remote store instruction. Remote stores are initiated by the producer core, but completed by the consumer core into the consumer core's cache. Because remote stores target a memory address, they create no new resource that must be managed by the operating system. They can even be virtualised by existing unmodified operating systems.

I have demonstrated the flexibility of remote stores by implementing several common patterns of inter-thread communication, and shown that they provide efficient communication in many contexts. I have shown speed-ups on a real-world fine-grained parallel application, concurrent dynamic analysis, where cache-coherent communication can be prohibitively slow. I have also used remote stores to implement my parallel STM scheme. It decreased average serial overhead from 151% to 83%.

5.5 Further Work

5.5.1 Software Transactional Memory

Intel's announcement [108] that hardware transactional memory will be available in forthcoming commodity processors has rearranged the landscape of TM research. It is said that no battle plan survives contact with the enemy, and the research agenda of STM will be realigned if it gains traction in the real world. When real-world applications exceed hardware capacity and encounter the limitations of current STM systems, I predict that the STM research community will be forced out of denial about the magnitude of the serial overhead problem.

It remains to be seen whether this will engender a return to lower-overhead, direct-update techniques, or whether even these will be too costly. In that case, we may abandon parallelism between overflowing transactions, and HTM will become no more than a performance optimisation for coarse-grained locking strategies. In any case, there is much implementation work to be done: transactional memory systems must be transformed from research frameworks into robust industrial tools with full virtual-machine, language and tool support.

5.5.2 Asynchronous Remote Stores

The work in Chapter 4 only scratches the surface of the possible applications of asynchronous remote stores. Remote stores are designed to be applicable to a wide variety of synchronisation patterns, and I have only explored a few.

For example, intelligent locking strategies such as queue-based locks could benefit from fast, direct communication. Applications with structured sharing patterns could use remote stores to place data explicitly near its next consumer. Remote stores also allow for new communication patterns which were simply not feasible beforehand. For example, one could use byte-sized remote writes to implement an efficient shared Bloom filter to which many threads could write simultaneously.

As for hardware implementations of remote stores, the most significant remaining challenge is addressing machine-level virtualisation. Although the scheme as proposed is compatible with existing operating systems, it cannot perform correctly in the presence of a hypervisor which prevents its guest operating

systems from accurately enumerating the physical processors. Even if that problem were solved, remote stores would allow one guest to degrade the performance of another by displacing entries from its cache. While it is commonly accepted that one process within an operating system may degrade the whole machine's performance – for example, with a "fork bomb" or similar techniques – hypervisors are currently able to contain such damage to a single guest instance.

Possible solutions include a hypervisor-maintained translation table, which maps processor numbers in remote store instructions to physical processors. While this would require modifications to the hypervisor, it would retain the important property of support for unmodified operating systems.

The concept of remote stores could also be generalised. The concept of issuing instructions on one core and completing them on another is a powerful one. New remote operations could be created: for example, a remote atomic-increment instruction for semaphores and counters, or a remote compare-and-swap.

One could generalise the concept of remote operations still further, by creating a mechanism to cause a remote processor to execute arbitrary instructions. One key advantage of remote stores is that faults and exceptions are trapped on the originating processor, and cannot disrupt code running on the destination processor. The store is therefore guaranteed to complete without error and in a timely manner. This principle might offer a solution to the problems of active messages, whose ability to disrupt or occupy the destination processor indefinitely is the principal barrier to their safe virtualisation. If remote operations were constructed in a limited microcode which was guaranteed to terminate and to execute without exceptions, they could prove a powerful tool for distributed computation.

5.6 Summary

Cache-coherent multi-core processors are geared to support implicit communication through shared memory. Not only is implicit communication difficult to reason about, in the abstract or computationally, but hardware tailored to it imposes performance penalties on programs which do perform explicit communication.

This is a shame, because explicit communication is easier to use for correct parallel programming. It is also required for high-performance fine-grained parallelism, of the sort we need if we are to exploit our new multi-core resources to the full. I have shown that, given hardware with high-speed explicit communication, we can parallelise even the prohibitive overheads of software transactional memory.

I have proposed a high-performance, flexible mechanism for explicit communication on a cache-coherent processor, which is compatible with existing operating systems. I have illustrated the power of high-performance explicit communication, by using it to accelerate even implicitly-communicating frameworks such as software transactional memory, as well as several common patterns of explicit inter-thread communication. I have made a case for the remote store instruction: a powerful, incremental extension to support explicit communication in modern cache-coherent multi-processors.

Bibliography

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. *SIGPLAN Not.*, 44(4):185–196, 2009.

[2] H. Abdel-Shafi, J. Hall, S.V. Adve, and V.S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 204 –215, Feb 1997.

[3] A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, D. Kranz, J.D. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine. *Proceedings of the IEEE*, 87(3):430 –444, Mar 1999.

[4] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Johan Andersson. Parallel futures of a game engine. Presentation at STHLM Game Developer Forum, May 2010.

[6] John B. Andrews, Carl J. Beckmann, and David K. Poulsen. Notification and multicast networks for synchronization and coherence. *Journal of Parallel and Distributed Computing*, 15(4):332 – 350, 1992.

[7] Norbert S Arenstorf and Harry F Jordan. Comparing barrier algorithms. *Parallel Computing*, 12(2):157 – 170, 1989.

[8] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[9] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[10] James M. Baker, Brian Gold, Mark Bucciero, Sidney Bennett, Rajneesh Mahajan, Priyadarshini Ramachandran, and Jignesh Shah. SCMP: A single-chip message-passing parallel computer. *The Journal of Supercomputing*, 30:133–149, 2004. 10.1023/B:SUPE.0000040612.33760.8a.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[12] David Beazley. Understanding the Python GIL. Presentation at PyCon 2010. `http://www.dabeaz.com/python/UnderstandingGIL.pdf`.

[13] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 processor: A 64-core SoC with mesh interconnect. In *ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, August 2011.

[15] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

[16] Christopher Blizzard. Improving Javascript performance with JaegerMonkey. `http://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/`.

[17] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[18] François Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the UPC language. *Parallel and Distributed Processing Symposium, International*, 15:254a, 2004.

[19] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[20] 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(5):46–58, 2008.

[21] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[22] D. Chavarria-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer. Early experience with out-of-core applications on the cray xmt. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, april 2008.

[23] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. In *SIGARCH Computer Architectire News*, 2005.

[24] Liqun Cheng, John B. Carter, and Donglai Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 328–339, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[26] Intel Corporation. Intel Turbo Boost technology in Intel Core microarchitecture (Nehalem) based processors (white paper), 2008.

[27] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, San Francisco, California, 1999.

[28] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[29] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 39–52, New York, NY, USA, 2011. ACM.

[30] Luke Dalessandro and Michael L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, Feb 2009.

[31] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proc. 15th ACM Symp. on Principles and Practice of Parallel Programming*, Jan 2010.

[32] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[33] C. Danis and C. Halverson. The value derived from the observational component in integrated methodology for the study of HPC programmer productivity. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[35] Saeed Dehnadi and Richard Bornat. The camel has two humps (working title). 2006.

[36] Advanced Micro Devices. Advanced Synchronization Facility - Proposed architectural specification, 2009.

[37] 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*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[38] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[39] Neil A. Dodgson. Balancing the expected and the surprising in geometric patterns. *Computers and Graphics*, 33(4):475 – 483, 2009.

[40] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 2010.

[41] Kemal Ebcioglu, Vivek Sarkar, Tarek El-Ghazawi, and John Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

[42] Brendan Eich. TraceMonkey: Javascript lightspeed. `http://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/`.

[43] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC*. Wiley, 2005.

[44] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[45] 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 23rd international conference on Supercomputing*, ICS '09, pages 126–135, New York, NY, USA, 2009. ACM.

[46] Umakishore Ramachandran Gautam, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of Supercomputing '95*, page 62. ACM Press, 1995.

[47] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, page 407, Sep 2007.

[48] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[49] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 101–110, New York, NY, USA, 2010. ACM.

[50] Håkan Grahn, Per Stenström, and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247 – 271, 1995.

[51] Robert Graybill. High productivity computing systems. Presentation at DARPATech Symposium, 2002.

[52] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[53] Dan Grossman. The transactional memory / garbage collection analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 695–706, New York, NY, USA, 2007. ACM.

[54] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 1–8, New York, NY, USA, 2006. ACM.

[55] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 155–174, New York, NY, USA, 2009. ACM.

[56] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 155–174, New York, NY, USA, 2009. ACM.

[57] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, December 1983.

[58] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24:92–103, November 2004.

[59] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM.

[60] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[61] Tim Harris, Saša Tomic, Adrián Cristal, and Osman Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 39–52, New York, NY, USA, 2010. ACM.

[62] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[63] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[64] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[65] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[66] Lorin Hochstein and Victor R. Basili. A preliminary empirical study to compare MPI and OpenMP. Technical Report ISI-TR-676, December 2011.

[67] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81:1920–1930, November 2008.

[68] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 35+, Washington, DC, USA, 2005. IEEE Computer Society.

[69] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. Remote store programming. In Yale Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 3–17. Springer Berlin / Heidelberg, 2010.

[70] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 transputer. *Micro, IEEE*, 7(5):10 –26, Oct 1987.

[71] Intel Corporation. SCC external architecture specification (EAS). July 2010.

[72] Stamatis G. Kavadias, Manolis G.H. Katevenis, Michail Zampetakis, and Dimitrios S. Nikolopoulos. On-chip communication and synchronization mechanisms with cache-integrated network interfaces. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 217–226, New York, NY, USA, 2010. ACM.

[73] Abdullah Kayi and Tarek El-Ghazawi. An adaptive cache coherence protocol for chip multiprocessors. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, IFMT '10, New York, NY, USA, 2010. ACM.

[74] R.E. Kessler and J.L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *Compcon Spring '93, Digest of Papers.*, pages 176 –182, Feb 1993.

[75] A.J. KleinOsowski and D.J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7, Jan-Dec 2002.

[76] Guy Korland, Nir Shavit, and Pascal Felber. Poster: Noninvasive Java concurrency with Deuce STM. In *Systor '09, Haifa, Israel*, 2009.

[77] D.A. Koufaty, Xiangfeng Chen, D.K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(12):1250 –1264, Dec 1996.

[78] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[79] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[80] Doug Lea. Jsr 166. Java Specification Request, 2004.

[81] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[82] P.P.C. Lee, Tian Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, Apr 2010.

[83] Sanghoon Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99–110, Feb 2011.

[84] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[85] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, Feb 2009.

[86] Meredydd Luff. Empirically investigating parallel programming paradigms: A null result. In *Proceedings of the 1st Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '09, 2009.

[87] Meredydd Luff and Simon Moore. Asynchronous remote stores for inter-processor communication. In *Future Architectural Support for Parallel Programming*, FASPP'12, Jun 2009.

[88] David Mandelin. Mozilla Javascript 2011. `http://blog.mozilla.com/dmandelin/2011/04/22/mozilla-javascript-2011/`.

[89] 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, Mar 2006. Condensed version submitted for publication.

[90] D. May. *The XMOS XS1 Architecture*. XMOS Ltd., UK, `http://www.xmos.com/`, 2009.

[91] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[92] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. *SIGPLAN Not.*, 43(5):15–26, May 2008.

[93] Microsoft. .NET framework 4 beta 1 enabled to use software transactional memory, Jan 2009.

[94] 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 34th annual international symposium on Computer architecture*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[95] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261 –270, Sep 2009.

[96] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, Feb 2006.

[97] Vijay Nagarajan, Ho-Seop Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *12th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, 2008.

[98] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM'11)*. IEEE Computer Society, 2011.

[99] Håkan Nilsson and Per Stenström. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*, pages 363–374. Springer Berlin / Heidelberg, 1994.

[100] John Ousterhout. Why threads are a bad idea (for most purposes). Presentation at the USENIX Annual Technical Conference, January 1996.

[101] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[102] V. Pankratius, A. Jannesari, and W.F. Tichy. Parallelizing bzip2: A case study in multicore software engineering. *Software, IEEE*, 26(6):70 –77, Nov-Dec 2009.

[103] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 43–52, New York, NY, USA, 2011. ACM.

[104] D. Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7):28 –32, 53, Jul 2010.

[105] D.K. Poulsen and Pen-Chung Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Parallel Processing, 1994. ICPP 1994. International Conference on*, volume 2, page 280, Aug 1994.

[106] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. Decoupled software pipelining with the synchronization array. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 177 – 188, Sep-Oct 2004.

[107] Ram Rangan, Neil Vachharajani, Adam Stoler, Guilherme Ottoni, David I. August, and George Z. N. Cai. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 259–272, Washington, DC, USA, 2006. IEEE Computer Society.

[108] James Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[109] Armin Rigo. We need software transactional memory `http://morepypy.blogspot.com/2011/08/we-need-software-transactional-memory.html`, 2011.

[110] Andreas Rodman and Mats Brorsson. Programming effort vs. performance with a hybrid programming model for distributed memory parallel architectures. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 888–898, London, UK, UK, 1999. Springer-Verlag.

[111] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[112] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 35–45, New York, NY, USA, 2008. ACM.

[113] Caitlin Sadowski and Andrew Shewmaker. The last mile: parallel programming and usability. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 309–314, New York, NY, USA, 2010. ACM.

[114] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 235–246, Washington, DC, USA, 2006. IEEE Computer Society.

[115] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 271–271, New York, NY, USA, 2007. ACM.

[116] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[117] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. In *IBM Journal of Research and Development*, volume 50, pages 261–275, 2006.

[118] Jun Shirako, Hironori Kasahara, and Vivek Sarkar. Languages and compilers for parallel computing. chapter Language Extensions in Support of Compiler Parallelization, pages 78–94. Springer-Verlag, Berlin, Heidelberg, 2008.

[119] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer, Iii Michael, and F. Spear. Hardware acceleration of software transactional memory. Technical report, Dept. of Computer Science, Univ. of Rochester, 2006.

[120] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30:123–169, 1996.

[121] K.C. Smith, A. Wang, and L.C. Fujino. Through the looking glass: Trend tracking for isscc 2012. *Solid-State Circuits Magazine, IEEE*, 4(1):4 –20, march 2012.

[122] Marc Snir and Steve Otto. *MPI - The Complete Reference*. MIT Press, Cambridge, MA, USA, 1998.

[123] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM.

[124] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

[125] Standard Performance Evaluation Corporation. CPU2000. `http://www.spec.org/cpu2000`.

[126] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3:54–62, September 2005.

[127] Timothy Sweeney. The next mainstream programming language: a game developer's perspective. Presentation at POPL'06, February 2006.

[128] Duane Szafron and Jonathan Schaeffer. An experiment to measure the usability of parallel programming systems, 1996.

[129] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong Wang. NZTM: nonblocking zero-indirection transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 204–213, New York, NY, USA, 2009. ACM.

[130] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW microprocessor: a computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25 – 35, Mar/Apr 2002.

[131] D. Tiwari, Sanghoon Lee, J. Tuck, and Yan Solihin. MMT: Exploiting fine-grained parallelism in dynamic memory management. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, Apr 2010.

[132] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[133] Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. Complexity and performance in parallel programming languages. *International Workshop on High-Level Programming Models and Supportive Environments*, 0:3, 1997.

[134] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova1, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *20th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2011.

[135] Thomas Willhalm and Nicolae Popovici. Putting Intel threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 3–4, New York, NY, USA, 2008. ACM.

[136] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation helper threads and runahead execution. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 410–420, New York, NY, USA, 2009. ACM.

[137] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 185–194, New York, NY, USA, 2008. ACM.

[138] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 25–34, New York, NY, USA, 2009. ACM.