Contents

1.1 Memory Transactions

Concurrent programs that deliver scalable speed-up on Chip Multi-Processors are difficult to develop. As the number of processing cores in a Chip Multi-Processor increases so does the potential speed-up from concurrent execution but there are few programs that actually achieve scalable concurrent speed-up when executing on a Chip Multi-Processor.

Transactional Memory is a programming methodology that promises to make concurrent programming easier and concurrent programs more scalable.

The main contribution of this section is to examine the challenges faced by concurrent programming methodologies. This section focuses on whether Transactional Memory systems really deliver on these promises.

1.1.1 Concurrent Programming

This thesis examines the problem of getting the processors of a Chip Multi-Processor to work together on a single program and complete the program in less time than it would take a single processor working alone. The program can be divided into tasks which are simultaneously executed by the processors and these tasks may or may not be interdependent.

The most commonly used concurrent programming technique is mutual exclusion. Unfortunately, mutual exclusion limits the scalability of a concurrent program and concurrent programming using mutual exclusion is regarded as difficult.

Section 1.1.3 introduces mutual exclusion.

Transactional Memory is a technique to support speculative execution that can be used as an alternative to mutual exclusion. It facilitates scalable concurrent execution by allowing the simultaneous execution of tasks that may be interdependent.

Section 1.1.4 introduces speculative execution and section 1.1.5 introduces Transactional Memory.

Software Transactional Memory systems provide a software framework for programmers to construct concurrent programs that can be executed speculatively. However, the overheads of supporting speculative execution entirely in software often exceed the benefits of concurrent execution.

Section 1.1.6 discusses the claim that Software Transactional Memory makes concurrent programming easier.

Hardware Transactional Memory systems support concurrent execution by providing a hardware environment in which concurrent programs can be executed speculatively. The engineering challenges that must be overcome by Hardware Transactional Memory are significant and the commercial barriers to adoption are high.

Section 1.1.7 discusses the claim that Hardware Transactional Memory makes concurrent programming easier.

Transactional Memory can make programming easier by freeing the programmer from having to reason about locks, but concurrent programming using Memory Transactions is not necessarily easier than concurrent programming using mutual exclusion.

Section 1.1.8 discusses the claim that Transactional Memory makes concurrent

programming easier.

Research carried out in both the private and public sectors has yet to produce convincing evidence that Transactional Memory systems are making progress towards delivering on their promise of scalability, because the overheads of supporting concurrent execution exceed the benefits of concurrent execution. They also fail to deliver on their promise of improved programmer productivity, because concurrent programming using Memory Transactions is no easier than concurrent programming using mutual exclusion.

1.1.2 Design Priorities

Transactional Memory research is founded on the premise that speculative execution is necessary to support scalable concurrent execution on Chip Multi-Processors and it has the goal of making concurrent programming easier. This thesis does not doubt this premise nor question this laudable goal, but it does question the priorities that motivate the design of Transactional Memory systems.

Transactional Memory proposals prioritise some aspects of system design at the expense of others:

- They focus on the speculative execution of programs, at the expense of the interaction with external systems.
- They choose to buffer speculative state, at the expense of increased memory bandwidth.
- They sacrifice strict transactional isolation, at the expense of semantic simplicity.
- They centralise the responsibility for transaction management, at the expense of scalability.
- They focus on ease of programming per se, at the expense of total productivity across the software development cycle.

Given the disappointing progress of Transactional Memory systems to date it is reasonable to suggest that some of the priorities can be re-assessed and that designs based on a different set of priorities should be considered.

Some aspects of the design of a concurrent system that need to be considered are:

- How to interact with entities outside the concurrent system?
- How to maintain shared state and support speculative execution?
- How to provide access to shared state with intuitive concurrent semantics?
- How to implement concurrency control to guarantee correct concurrent execution?
- How to implement contention management to eliminate progress pathologies?
- How to marshall work and schedule concurrent execution?
- How to integrate a concurrent programming solution into the software development cycle?

This thesis is divided into chapters, each of which considers one aspect of the design of a concurrent system. The first section of each chapter examines how the priorities have influenced the design of concurrent systems. Subsequent sections develop an alternative approach based on a different interpretation of the priorities.

1.1.3 Mutual Exclusion

The most commonly used mechanism to support concurrent execution is mutual exclusion. Mutual exclusion does not permit tasks with possible dependencies to execute simultaneously. Instead, it permits those sections of a program in which there are known to be no conflicting operations to execute in parallel and ensures that the critical sections of the program, that may have dependencies, execute serially.

If there is the slightest possibility of a dependency between tasks then they must always be executed serially. Mutual exclusion ensures the serial execution of critical sections regardless of how often dependencies between tasks actually arise. As the number of processes is scaled up, the execution time of the program code within the critical sections dominates and the benefit of parallel execution is diminished. This effect is a consequence of Amdahl's law [Amd67] [HM08].

1.1.4 Speculative execution

An alternative to mutual exclusion should be capable of executing tasks optimistically. A concurrent program can safely speculate that a task is not affected by tasks running on other processors, provided it has a mechanism to re-execute the task should that speculation prove incorrect.

To discover the dependencies between tasks, information must pass between the processors performing them, while the program executes. Processors cannot pass information to each other instantaneously, so each task has a slightly delayed view of the progress that tasks on other processors are making. This delay necessitates speculative execution because if a processor were to wait for a task, on which it is possibly dependent, to complete then there would be little benefit from executing on multiple processors.

1.1.5 The development of Transactional Memory

Research into Transactional Memory is comprehensively described in a book entitled 'Transactional Memory' [HLR10].

The following are some of the significant developments in the history of Transactional Memory:

Lomet proposed the use of transactions within programs [Lom77].

Weihl proposed the use of transactions to support concurrent programming [WL83].

Stone recognised that Memory Transactions are dis-contiguous multi-word atomic operations [SSHT93]. Computer hardware typically provides atomic operations that act on a single word or a contiguous double-word in memory.

Herlihy and Moss proposed Hardware Transactional Memory [HM93]. A Hardware Transactional Memory implements Memory Transactions by using modified hardware to support speculative execution.

Shavit proposed the conventional model of Software Transactional Memory [ST95]. A Software Transactional Memory implements Memory Transactions entirely in software by buffering speculative state in core or in a log.

Lie proposed Hybrid Transactional Memory [LA04]. Recent Hardware Transactional Memory systems are typically hybrids involving both compiler and runtime support for Memory Transactions executing on modified hardware.

1.1.6 Software Transactional Memory

Software Transactional Memory systems provide a framework and a run-time system to support the speculative execution of Memory Transactions. Dependencies between tasks are checked at run-time. If conflicting operations are found then the tasks containing them are re-executed resulting in wasted work.

The influential paper "Software Transactional Memory: Why is it only a research toy?" was written by the team responsible for IBM's Software Transactional Memory system [CBM+08]. They compared the performance of their Software Transactional Memory with comparable systems from Intel and Sun [ART08] [DDS06]. They examined the performance of programs from the STAMP benchmark suite [CMCKO08]. This benchmark suite contains programs that are good candidates for concurrent execution. The team does not discuss the interaction with the Network or Operating System because the benchmark programs are monolithic.

The team found that none of the Software Transactional Memory systems they examined overcame the overhead of supporting Memory Transactions. They also found that, as more processors were added to the concurrent system, the overheads of concurrent execution increased faster than the benefits, so the Software Transactional Memory systems they studied did not exhibit scalable concurrent execution.

The team concluded that complex concurrent semantics, weak atomicity, transactional pathologies, the interaction with serial code, memory reclamation and the support for legacy binaries are all major barriers to the development of Software Transactional Memory.

The conclusion of the IBM paper is that Software Transactional Memory does not achieve its goal of supporting scalable concurrent execution. Soon after publication of the IBM paper Microsoft cancelled its Software Transactional Memory research project without publishing any results [Duf10].

This thesis examines why Software Transactional Memory fails to achieve this goal and explores alternatives that might make the goal achievable in the future.

1.1.7 Hardware Transactional Memory

Hardware Transactional Memory systems use a combination of techniques including run-time systems, modified programs and modified hardware to support

speculative execution. The original goal of Hardware Transactional Memory was to facilitate the concurrent execution of critical sections in programs written for mutual exclusion without program modification. Unfortunately, programs written for mutual exclusion rarely contain enough information about dependent variables for this to be achievable.

The influential paper "Early experience with a commercial Hardware Transactional Memory implementation" was written by the team responsible for SUN's ROCK processor [DLMN09]. The ROCK processor is a Chip Multi-Processor which aims to support concurrent processing. The processor contained hardware support for speculative execution and explicit support for Memory Transactions. The team evaluated the system using programs from the STAMP benchmark suite [CMCKO08].

Prior to the ROCK processor Hardware Transactional Memory research was restricted to architectural simulation. A hardware architecture can be simulated in software by a virtual machine on which the target program runs. Typically, the proposed hardware support for Hardware Transactional Memory is included in the virtual machine and the execution time for benchmark applications is determined by simulation. The main problem with architectural simulation is that one cannot be sure that the results will be similar to those that would be obtained were the proposed modifications to be implemented in physical hardware. This is especially true of simulated Chip Multi-Processors because cycle accurate simulation of the shared memory subsystem is extremely difficult to achieve [Jac09]. The ROCK processor was seen, by the Transactional Memory research community, as the most significant Hardware Transactional Memory design to be implemented in real hardware.

The team reported some speed-up on the benchmarks they tested. They also demonstrated some scalability. However, the speed-ups were not very impressive and a great deal of program adaptation was required to obtain them. The team concluded that Hardware Transactional Memory is a promising area of research.

The ROCK project was cancelled a few months after publication of the paper which might suggest that support for Hardware Transactional Memory in commercial Chip Multi-Processors is not considered economically viable [Van09]. It might also suggest that the benchmark results obtained from the hardware implementation were disappointing when compared to those of architectural simulation [And09].

Many Transactional Memory research papers are able to demonstrate scalable speed-up from the concurrent execution of benchmark applications. There is no doubt that Hardware Transactional Memory implementations can speed-up the concurrent execution of Transactional Memory benchmark programs, such as those in the STAMP or DaCapo benchmark suites [CMCKO08] [BGH+06]. However, these benchmark applications are not general applications. Performance results obtained using them are not indicative of the performance one might expect from the concurrent execution of an operating system or a game.

The conclusion of the ROCK paper is that Hardware Transactional Memory does not achieve its goal of supporting scalable concurrent execution.

This thesis examines why Hardware Transactional Memory fails to achieve this goal and explores alternatives that might make the goal achievable in the future.

1.1.8 Ease of programming

Concurrent programs are difficult to write because using mutual exclusion to serialise access to shared data is error prone. Concurrent programs must implement mutual exclusion, correctly, to avoid the run-time problem of data races and the pathology of deadlock. Concurrent programming must be done at a very low level of abstraction, because locks are not composable, so the fundamental interaction with the program cannot be hidden by abstraction [HMPJH05].

Ease of programming is a subjective criterion for determining the efficacy of Transactional Memory systems. Writing a concurrent program using Memory Transactions can be just as difficult as writing one using mutual exclusion as it is often very difficult to break an algorithm into transactions of sufficient size to be worth scheduling. Transactional Memory systems are also prone to run-time pathologies such as live-lock and priority inversion.

Research publications often claim that programming concurrent systems using Transactional Memory is somehow easier than writing the same algorithm using locks. In a sample of 25 papers chosen at random from the on-line transactional memory bibliography we found that 17 asserted that Transactional Memory made concurrent programming easier [JBR10]. However, none of the papers in our sample contained any explicit justification for this claim and most referred to it only in the introductory section. Indeed, many of the papers contained descriptions of syntax and semantics that would indicate precisely the opposite to be the case.

Many Transactional Memory research papers claim that concurrent programming using Memory Transactions is easier than using mutual exclusion. However, very few papers support this assertion with quantitative analysis or empirical results. In an exceptional paper Rossbach describes experiments that showed that students found programming a concurrent algorithm using Software Transactional Memory was just as difficult as constructing the same algorithm using mutual exclusion [RHW09]. Rossbach was not able to show that Memory Transactions were easier to use than mutual exclusion.

The claim that transactional programming is easier than using mutual exclusion is largely based on experience of programmers using transactions to program Relational Database systems which is undoubtedly easier than accessing a database using mutual exclusion. Relational Database systems that support serialisable transactional execution have largely replaced earlier systems, such as CICS, in which the programmer is responsible for enforcing mutual exclusion [GR93]. However, Relational Database systems and Transactional Memory systems have very little in common as a Memory Transaction coded using an atomic section is very different from a database transaction specified as a Structured Query Language (SQL) statement. Transactional programming is easier in the context of a database system but this does not necessarily mean that using Memory Transactions make concurrent programming easier in the context of a Chip Multi-Processor.

If we compare the specification of TCC [HCW+04], an early Transactional Memory system, with that of openTM [BMT+07], which is a more recent system from the same institution, then we find the more recent specification to be more complex. So, the claim that Transactional Memory systems have the *potential* to make concurrent programming easier does not appear to be based on an extrapolation of the current trend.

Bibliography

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[And09] Mark Anderson. Sun can kill rock, but not its memory tech. *IEEE Spectr.*, June 2009.

[ART08] Adl-Tabatabai Ali-Reza and Xinmin Tian. The intel software transactional memory compiler. http://software.intel.com/file/8097, November 2008.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BMT+07] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The openTM Transactional Application Programming Interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, November 2008.

[CMCKO08] 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.

[DDS06] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168. ACM, March 2009.

[Duf10] Joe Duffy. A (brief) retrospective on tranasactional memory. http://www.bluebytesoftware.com, January 2010.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HCW+04] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13. ACM Press, October 2004.

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of*

the 20th Annual International Symposium on Computer Architecture, pages 289–300, May 1993.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[JBR10] Tim Harris Jayaram Bobba, Mark Hill and Ravi Rajwar. The transactional memory bibliography. http://www.cs.wisc.edu/trans-memory/biblio/index.html, June 2010.

[LA04] Sean Lie and Krste Asanovic. Hardware support for unbounded transactional memory. Technical report, Masters thesis, MIT, 2004.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[RHW09] Christopher Rossbach, Owen Hofmann, and Emmett Witchel. Is transactional memory programming actually easier? In *WDDD '09: Proc. 8th Workshop on Duplicating, Deconstructing, and Debunking*, jun 2009.

[SSHT93] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[Van09] Ashlee Vance. Sun is said to cancel big chip project. *The New York Times*, June 2009.

[WL83] William Weihl and Barbara Liskov. Specification and implementation of resilient, atomic data types. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 53–64, jun 1983.