Chapter 1

Introduction

Embedded systems sense physical processes and control their behavior, typically through feedback loops. Since physical processes are concurrent, computations that control them must also be concurrent, enabling them to process multiple streams of sensor input and control multiple actuators, all concurrently. Often, such computations need to concurrently read/write shared data objects. Typically, they must also process sensor input and react, satisfying application-level time constraints.

The de facto standard for programming concurrency is the threads abstraction, and the de facto synchronization abstraction is locks. Lock-based concurrency control has significant programmability, scalability, and composability challenges [24]. Coarse-grained locking (e.g., a single lock guarding a critical section) is simple to use, but permits no concurrency: the single lock forces concurrent threads to execute the critical section sequentially, in a one-at-a-time order. This is a significant limitation, especially with the emergence of multicore architectures, on which improved software performance must be achieved by exposing greater concurrency.

With fine-grained locking, a single critical section is broken down into several critical sections – e.g., each bucket of a hash table is guarded by a unique lock. Thus, threads that need to access different buckets can do so concurrently, permitting greater parallelism. However, this approach has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion.

Perhaps, the most significant limitation of lock-based code is its non-composability. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is not possible in a straightforward manner: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety.

Lock-free synchronization [23], which uses atomic hardware synchronization primitives (e.g., Compare And Swap [30,31], Load-Linked/Store-Conditional [51]), also permits greater concurrency, but has even lower programmability: lock-free algorithms must be custom-designed for each situation (e.g., a data structure [8,19,22,26,38]). Additionally, it is not clear how to program nested critical sections using lock-free synchronization. Most importantly, reasoning about the correctness of lock-free algorithms is significantly difficult [23].

1.1 Transactional Memory

Transactional memory (TM) is an alternative synchronization model for shared memory data objects that promises to alleviate these difficulties. With TM, programmers write concurrent code using threads, but organize code that read/write shared memory objects as *memory transactions*, which speculatively execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. Objects read and written by transactions are also monitored, in read sets and write sets, respectively. Two transactions conflict if they access the same object and one access is a write. (Conflicts are usually detected by detecting non-empty read and write set intersections.) When that happens, a contention manager (CM) resolves the conflict by aborting one and committing the other, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffers (lazy).

In addition to a simple programming model (locks are excluded from the programming interface), TM provides performance comparable to lock-based synchronization [45], especially for high contention and read-dominated workloads, and is composable. TM's first implementation was proposed in hardware, called hardware transactional memory (or HTM) [25]. HTM has the lowest overhead, but HTM transactions are usually limited in space and time. Examples of HTMs include TCC [21], UTM [1], Oklahoma [52], ASF [13], and Bulk [11]. TM implementation in software, called software transactional memory (or STM) was proposed later [49]. STM transactions do not need any special hardware, are not limited in size or time, and are more flexible. However, STM has a higher overhead, and thus lower performance, than HTM. Examples of STMs include RSTM [53], TinySTM [44], Deuce [33], and AtomJava [27].

Listing 1.1: STM example

```
BEGIN_TRANSACTION;
        stm::wr_ptr<Counter> wr(m_counter);
        wr->set_value(wr->get_value(wr) + 1, wr);
END_TRANSACTION;
```

Listing 1.1 shows an example STM code written by RSTM [50]'s interface. RSTM's `BEGIN_TRANSACTION` and `END_TRANSACTION` keywords are used to enclose a critical section, which creates a transaction for the enclosed code block and guarantees its atomic execution. First line inside the transaction makes a write pointer to a variable "m_counter" of type "Counter". The second line reads the current value of the counter variable through "wr−>get_value". The counter value is incremented through "wr−>set_value" operation.

Hybrid TM (or HyTM) was subsequently proposed in [36], which combines HTM with STM, and avoids their limitations. Examples of HyTMs include SpHT [35], VTM [43], HyTM [14], LogTM [39], and LogTM-SE [55].

1.2 STM for Real-Time Software

Given the hardware-independence of STM, which is a significant advantage, we focus on STM. STM's programmability, scalability, and composability advantages are also compelling for concurrency control in multicore embedded real-time software. However, this will require bounding transactional retries, as real-time threads, which subsume transactions, must satisfy application-level time constraints. Transactional retry bounds in STM are dependent on the CM policy at hand (analogous to the way thread response time bounds are OS scheduler-dependent).

Despite the large body of work on STM contention managers, relatively few results are known on real-time contention management. STM concurrency control for real-time systems has been previously studied, but in a limited way. For example, [37] proposes a restricted version of STM for uniprocessors. Uniprocessors do not need contention management. [18] bounds response times in distributed multicore systems with STM synchronization. They consider Pfair scheduling [29], which is largely only of theoretical interest[1], limit to small atomic regions with fixed size, and limit transaction execution to span at most two quanta. [46] presents real-time scheduling of transactions and serializes transactions based on transactional deadlines. However, the work does not bound transactional retries and response times.

[47] proposes real-time HTM, which of course, requires hardware with TM support. The retry bound developed in [47] assumes that the worst case conflict between atomic sections of different tasks occurs when the sections are released at the same time. We show that, this assumption does not cover the worst case scenario (see Chapter 4). [17] presents a contention manager that resolves conflicts using task deadlines. The work also establishes upper bounds on transactional retries and task response times. However, similar to [47], [17] also assumes that the worst case conflict between atomic sections occurs when the sections are released simultaneously. Besides, [17] assumes that all transactions have equal lengths. The ideas in [17] are extended in [3], which presents three real-time CM designs. But no retry bounds or schedulability analysis techniques are presented for those CMs.

Thus, past efforts on real-time STM are limited, and do not answer important fundamental

[1]This is due to Pfair class of algorithm's time quantum-driven nature of scheduling and consequent high run-time overheads.

questions:

(1) How to design "general purpose" real-time STM contention managers for multicore architectures? By general purpose, we mean those that do not impose any restrictions on transactional properties (e.g., transaction lengths, number of transactional objects, levels of transactional nestings), which are key limitations of past work.

(2) What tight upper bounds exist for transactional retries and task response times under such real-time CMs?

(3) How does the schedulability of real-time CMs compare with that of lock-free synchronization? i.e., are there upper bounds or lower bounds for transaction lengths below or above which is STM superior to lock-free synchronization?

(4) How does transactional retry costs and task response times of real-time CMs compare with that of lock-free synchronization in practice (i.e., on average)?

1.3 Research Contributions

In this dissertation proposal, we answer these questions. We present a suite of real-time STM contention managers, called RCM, ECM, LCM, and PNF. The contention managers progressively improve transactional retry and task response time upper bounds (and consequently improve STM's schedulability advantages) and also relax the underlying task models. RCM and ECM resolve conflicts using fixed and dynamic priorities of real-time tasks, respectively, and are naturally intended to be used with the fixed priority (e.g., G-RMA [9]) and dynamic priority (e.g., G-EDF [9]) multicore real-time schedulers, respectively. LCM resolves conflicts based on task priorities as well as atomic section lengths, and can be used with G-EDF or G-RMA. Transactions under ECM, RCM and LCM can restart because of other transactions that share no objects with them. This is called transitive retry. PNF solves this problem. PNF also optimizes processor usage through reducing priority of aborted transactions. So, other tasks can proceed.

We establish upper bounds on transactional retry costs and task response times under all the contention managers through schedulability analysis. Since ECM and RCM conserve the semantics of the underlying real-time scheduler, their maximum transactional retry cost is double the maximum atomic section length. This is improved in the design of LCM, which achieves shorter retry costs. However, ECM, RCM, and LCM are affected by transitive retries when transactions access multiple objects. Transitive retry causes a transaction to abort and retry due to another non-conflicting transaction. PNG avoids transitive retry, and also optimizes processor usage by lowering the priority of retrying transactions, enabling other non-conflicting transactions to proceed.

We formally compare the schedulability of the proposed contention managers with lock-free synchronization. Our comparison reveals that, for most cases, ECM and RCM achieve higher schedulability than lock-free synchronization only when the atomic section length does not exceed half of lock-free synchronization's retry loop length. However, in some cases, the atomic section length can reach the lock-free retry loop length for ECM and it can even be larger than the lock-free retry loop-length for RCM, and yet higher schedulability can be achieved with STM. This means that, STM is more advantageous with G-RMA than with G-EDF.

LCM achieves shorter retry costs and response times than ECM and RCM. Importantly, the atomic section length range for which STM's schedulability advantage holds is significantly expanded with LCM (over that under ECM and RCM): Under ECM, RCM and LCM, transactional length should not exceed half of lock-free retry loop length to achieve better schedulability. However, with low contention, transactional length can increase to retry loop length under ECM. Under RCM, transactional length can be of many orders of magnitude of retry loop length with low contention. With suitable LCM parameters, transactional length under G-EDF/LCM can be twice as retry loop length. While in G-RMA/LCM, transactional length can be of many orders of magnitude as retry loop length. PNF achieves better schedulability than lock-free as long as transactional length does not exceed length of retry loop.

Why are we concerned about expanding STM's schedulability advantage? When STM's schedulability advantage holds, programmers can reap STM's significant programmability and composability benefits in multicore real-time software. Thus, by expanding STM's schedulability advantage, we increase the range of real-time software for which those benefits can be tapped. Our results, for the first time, thus provides a fundamental understanding of when to use, and not use, STM concurrency control in multicore real-time software.

We also implement the contention managers in the RSTM framework [50] and conduct experimental studies using the ChronOS multicore real-time Linux kernel [15]. Our studies confirm that, the contention managers achieve shorter retry costs than lock-free synchronization by as much as 95% improvement (on average). Among the contention managers, PNF performs the best in case of high transitive retry. PNF achieves shorter retry costs than ECM, RCM and LCM by as much as 53% improvement (on average).

1.4 Summary of Proposed Post Preliminary Research

Based on our current research results, we propose the following work:

Supporting nested transactions. Transactions can be nested *linearly*, where each transaction has at most one pending transaction [40]. Nesting can also be done in *parallel* where transactions execute concurrently within the same parent [54]. Linear nesting can be *1) flat:* If a child transaction aborts, then the parent transaction also aborts. If a child commits, no effect is taken until the parent commits. Modifications made by the child transaction are only visible to the parent until the parent commits, after which they are externally visible. *2) Closed:* Similar to *flat nesting*, except that if a child transaction conflicts, it is aborted and retried, without aborting the parent, potentially improving concurrency over flat nesting. *3) Open:* If a child transaction commits, its modifications are immediately externally visible, releasing memory isolation of objects used by the child, thereby potentially improving concurrency over closed nesting. However, if the parent conflicts after the child commits, then compensating actions are executed to undo the actions of the child, before retrying the parent and the child. We propose to develop real-time contention managers that allow these different nesting models and establish their retry and response time upper bounds. Additionally, we propose to formally compare their schedulability with nested critical sections under lock-based synchronization. Note that, nesting is not viable under lock-free synchronization.

Combinations and optimizations of LCM and PNF contention managers. LCM is designed to reduce the retry cost of a transaction when it is interfered close to the end of its execution. In contrast, PNF is designed to avoid transitive retry when transactions access multiple objects. An interesting direction is to combine the two contention managers to obtain the benefits of both algorithms. Further design optimizations may also be possible to reduce retry costs and response times, by considering additional criteria for resolving transactional conflicts. Importantly, we must also understand what are the schedulability advantages of such a combined/optimized CM over that of LCM and PNF, and how such a combined/optimized CM behaves in practice. This will be our second research direction.

Formal and experimental comparison with real-time locking protocols. Lock-free synchronization offers numerous advantages over locking protocols, but (coarse-grain) locking protocols have had significant traction in real-time systems due to their good programmability (even though their concurrency is low). Example such real-time locking protocols include PCP and its variants [12, 32, 42, 48], multicore PCP (MPCP) [34, 41], SRP [2, 10], multicore SRP (MSRP) [20], PIP [16], FMLP [5, 6, 28], and OMLP [4]. OMLP and FMLP are similar, and FMLP has been established to be superior to other protocols [7]. How does their schedulability compare with that of the proposed contention managers? How do they compare in practice? These questions constitute our third research direction.

1.5 Proposal Organization

The rest of this dissertation proposal is organized as follows. Chapter 2 overviews past and related work on real-time concurrency control. Chapter 3 describes our task/system model and assumptions. Chapter 4 describes the ECM and RCM contention managers, derives upper bounds for their retry costs and response times, and compares their schedulability between themselves and with lock-free synchronization. Chapters 5 and 6 similarly describe the LCM and PNF contention managers, respectively. Chapter 7 describes our implementation and reports our experimental studies. We conclude in Chapter 8.

Chapter 2

Past and Related Work

2.1 Real-Time Locking Protocols

2.2 Real-Time Lock-Free Synchronization

2.3 Real-Time TM Concurrency Control

Chapter 3

Models and Assumptions

Chapter 4

The ECM and RCM Contention Managers

4.1 Algorithm Description

4.2 Illustrative Example

4.3 Properties

4.4 Schedulability Comparisons

Chapter 5

The LCM Contention Manager

Chapter 6

The PNF Contention Manager

Chapter 7

Implementation and Experimental Evaluations

7.1 Implementation

7.2 Experimental Settings

7.3 Retry Cost Measurements

7.4 Response Time Measurements

Chapter 8

Conclusions and Proposed Post Preliminary Research

8.1 Conclusions

8.2 Proposed Post Preliminary Research

Bibliography

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 316 – 327, feb. 2005.

[2] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3:67–99, 1991.

[3] A. Barros and L.M. Pinho. Managing contention of software transactional memory in real-time systems. In *IEEE RTSS, Work-In-Progress*, 2011.

[4] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, pages 119–128, 2007.

[5] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47 –56, aug. 2007.

[6] B.B. Brandenburg and J.H. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS-RT. In *RTCSA*, pages 185–194, 2008.

[7] Bjrn Brandenburg and James Anderson. A comparison of the m-pcp, d-pcp, and fmlp on litmus rt. In Theodore Baker, Alain Bui, and Sbastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 105–124, 2008.

[8] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In Antonio Fernndez Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 207–221. Springer Berlin-Heidelberg, 2011.

[9] G.C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer-Verlag New York Inc, 2005.

[10] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[11] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990.

[13] D. Christie, J.W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[14] 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.

[15] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. Chronos linux: a best-effort real-time multiprocessor linux kernel. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 474–479, New York, NY, USA, 2011. ACM.

[16] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377 –386, dec. 2009.

[17] S. Fahmy and B. Ravindran. On stm concurrency control for multicore embedded real-time software. In *International Conference on Embedded Computer Systems*, SAMOS, pages 1 –8, July 2011.

[18] S.F. Fahmy, B. Ravindran, and E. D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *DATE*, pages 688–693, 2009.

[19] K. Fraser. *Practical lock-freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[20] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189 – 198, may 2003.

[21] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] M. Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. In *Unpublished Manuscript*. Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[23] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[24] Maurice Herlihy. The art of multiprocessor programming. In *PODC*, pages 1–2, 2006.

[25] 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.

[26] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In Gadi Taubenfeld, editor, *Distributed Computing*, volume 5218 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin / Heidelberg, 2008.

[27] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 82–91, New York, NY, USA, 2006. ACM.

[28] P. Holman and J.H. Anderson. Locking under pfair scheduling. *TOCS*, 24(2):140–174, 2006.

[29] Philip L. Holman. *On the implementation of pfair-scheduled multiprocessor systems*. PhD thesis, University of North Carolina, Chapel Hill, 2004.

[30] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf`, 2007.

[31] Intel Corporation. Intel Itanium Architecture Software Developers Manual Volume 3: Instruction Set Reference. `http://download.intel.com/design/Itanium/manuals/24531905.pdf`, 2007.

[32] D.K. Kiss. Intelligent priority ceiling protocol for scheduling. In *2011 3rd IEEE International Symposium on Logistics and Industrial Informatics*, LINDI, pages 105 –110, aug. 2011.

[33] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[34] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 469 –478, dec. 2009.

[35] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 197–206, New York, NY, USA, 2008. ACM.

[36] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[37] J. Manson, J. Baker, et al. Preemptible atomic regions for real-time Java. In *RTSS*, pages 10–71, 2006.

[38] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[39] 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.

[40] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186 – 201, 2006.

[41] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *ICDCS*, pages 116–123, 2002.

[42] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[43] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of 32nd International Symposium on Computer Architecture (ISCA)*, pages 494 – 505, june 2005.

[44] T. Riegel, P. Felber, and C. Fetzer. TinySTM. `http://tmware.org/tinystm`, 2010.

[45] Bratin Saha, Ali-Reza Adl-Tabatabai, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[46] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *RTCSA*, pages 477–485, 2009.

[47] M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *ACM SAC*, pages 326–333, 2010.

[48] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175 –1185, sep 1990.

[49] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[50] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 104–115, New York, NY, USA, 2007. ACM.

[51] Richard L. Sites. Alpha AXP architecture. *Commun. ACM*, 36:33–44, February 1993.

[52] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58 –71, nov 1993.

[53] University of Rochester. Rochester Software Transactional Memory. `http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml`, `http://code.google.com/p/rstm`, 2006.

[54] H. Volos, A. Welc, A.R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. *ECOOP 2009–Object-Oriented Programming*, pages 123–147, 2009.

[55] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261 –272, feb. 2007.