Transactions Everywhere

Bradley C. Kuszmaul and Charles E. Leiserson, *SMA Fellow*
MIT Laboratory for Computer Science

***Abstract*— Arguably, one of the biggest deterrants for software developers who might otherwise choose to write parallel code is that parallelism makes their lives more complicated. Perhaps the most basic problem inherent in the coordination of concurrent tasks is the enforcing of *atomicity* so that the partial results of one task do not inadvertently corrupt another task. Atomicity is typically enforced through locking protocols, but these protocols can introduce other complications, such as deadlock, unless restrictive methodologies in their use are adopted. We have recently begun a research project focusing on *transactional memory* [18] as an alternative mechanism for enforcing atomicity, since it allows the user to avoid many of the complications inherent in locking protocols.**

Rather than viewing transactions as infrequent occurrences in a program, as has generally been done in the past, we have adopted the point of view that all user code should execute in the context of some transaction. To make this viewpoint viable requires the development of two key technologies: effective hardware support for scalable transactional memory, and linguistic and compiler support. This paper describes our preliminary research results on making "transactions everywhere" a practical reality.

I. Introduction

In a seminal paper ten years ago, Herlihy and Moss [18] proposed ***transactional memory*** as a way to ease the writing of concurrent programs. Transactional memory is sometimes described as an extension of Load-Linked/Store-Conditional [24] and other complex instructions. The idea of transactional memory is to allow a program to read and modify multiple, disparate memory locations as a single atomic operation, much as occurs within a database transaction [14], [15]. With transactional memory, they argued, programmers can avoid such concurrency anomalies such as ***priority inversion***, ***convoying***, and ***deadlock***. Herlihy and Moss proposed an extension to hardware cache-consistency mechanisms that can provide hardware transactional memory (***HTM***) efficiently.

The traditional way to specify atomicity using transactions is by textually marking the start and end of the transaction. Within the transaction, all results are "sandboxed," which means that the values written by the transaction are held in abeyance until the end of the transaction, at which point the transaction ***commits***, and all these values are atomically made globally available. If the transaction is unable to commit for any reason, the commit action returns an ***abort*** code, and no global changes are made. Otherwise, it returns a ***committed*** code, and all global changes are viewable. If the transaction aborts, it is usually the responsibility of the programmer to retry the transaction.

This traditional view of transactional memory simply replaces a locking protocol with a transactional protocol. Moreover, the common perception among researchers seems to be that transactions occur infrequently (as evidenced implicitly by the designs and codes that have been proposed [18], [28]) and have high overhead. Many researchers have focused their efforts on software transactional memory (***STM***) [30], [17], [16], leading to a dearth in the exploration of HTM.

We believe that the real advantage of transactional memory is that programmers can largely be freed from writing and debugging synchronization protocols. Rather than viewing transactions as infrequent, our research explores the notion of transactions everywhere: user code is *always* executing within some transaction. This extreme point of view requires both linguistic and hardware support. In this paper, we describe our research progress towards justifying a transactions-everywhere approach, making HTM as familiar a computer subsystem as cached memory.

The remainder of this paper outlines our research progress to date. Section II describes our proposal for adapting software to exploit HTM, focusing on linguistic issues and compiler technology. Section III describes our modification to Herlihy and Moss's scheme to make it scalable and how we plan to design a Bluespec hardware specification and simulator for HTM to measure overheads. Finally, we offer some conclusions in Section IV.

II. Linguistic and compiler support for HTM

Our preliminary investigations of language support for HTM focus on providing "implicit" atomicity within Cilk [2], [3], [25], [34], [12], a multithreaded programming language developed by our research group which features a provably efficient, randomized, "work-stealing" scheduler. The current Cilk-5.3.2 release uses locking to achieve atomicity, but we wish to investigate how HTM can be employed to eliminate the use of locks and other concurrency protocols. In particular, we are studying the degree to which transactions can be specified ***implicitly***, rather than ***explicitly***.

In this section, we first offer a brief introduction to Cilk.[1] We then describe our proposal to extend Cilk to specify atomicity. Finally, we report on some preliminary studies of the implications of this language extension.

The Cilk multithreaded language extends the C programming language with five keywords:

- `cilk` before a function definition indicates that this function can operate in parallel;
- `spawn` before the call to a Cilk function indicates that this call can execute in parallel with the caller;
- `sync` waits until all spawned functions have finished;
- `inlet` allows the result from a spawned function to be incorporated into the caller in a user-defined fashion;
- `abort` stops the execution of a spawned function.

Cilk is a ***faithful*** extension of the C programming language: if the Cilk keywords for parallel control are elided from a Cilk program, a syntactically and semantically correct C program results. Consequently, just by `#define`'ing all Cilk keywords to nil, a Cilk program becomes an ordinary C program.

The current Cilk-5.3.2 release uses locks to implement mutual exclusion when data structures are operated on concurrently by Cilk threads. Locking introduces many concurrency anomalies, such as deadlock, into "pure" Cilk programming. In previous work, we have developed a tool, called the "Nondeterminator" [10], [7], [8], [32], which helps with the debugging of locking protocols, but specifying locks remains an error-prone exercise.

Replacing locking protocols with transactions may mitigate some of the problems with concurrency control, but specifying the start and end of each transaction seems to be no simpler than specifying where to grab a lock and where to release it. Although locking protocols must be wary of deadlock, transactions can have livelock problems. When a transaction aborts, recovery code must roll back and restart the transaction. Having the programmer worry about livelock instead of deadlock seems tantamount to replacing one protocol poison with another.

Although specifying locations in code where atomicity needs to be enforced seems to be a popular option (see, for example, [18], [17], [16], [9], [27]), our transactions-everywhere philosophy encourages a different tack. Rather than specifying where atomicity should be enforced, we take the view that atomicity is always enforced, but it is implicitly "cut" at certain reasonable points in the code. Of course, what is linguistically "reasonable" is a matter of debate and personal taste. Our initial studies for Cilk assume that atomicity is broken implicitly at the following ***cutpoints***:

- at backward branches (such as the end of a loop);
- when a C function is called or returns;
- when a Cilk function is spawned or returns;
- at a `sync`.

Intuitively, these cutpoints partition the program into atomic sections. In other programming environments, such as pthreads [23], similar cutpoints could be defined.

The compiler needs not only to partition the code into atomic sections, it must generate code to recover and restart transactions in case they abort. Intuitively, this job is not hard. The transaction variables are rolled back to their states from before the transaction began, and the transaction is reexecuted. Of course, the programmer need not be aware that transactions are being aborted during execution. The mechanisms to implement transactions are beneath the layer of abstraction provided by the programming language. The programmer need only understand where the cutpoints are that divide the program into atomic sections.

We have modified the Cilk compiler to ***atomize*** C and Cilk code by inserting cutpoints. We then compiled our Cilk benchmark programs to determine which would operate correctly without locks. Although many run correctly with this implicit atomization, several do not. For example, one benchmark uses linked lists. In the benchmark's implementation with locks, each list is locked in its entirety while it is operated on. In another, a call to a simple arithmetic function breaks the atomicity in the middle of what one would want to be an atomic section.

We did not want to introduce explicit locking or a transactional protocol into the Cilk language to cope with situations where implicit atomicity seems to be inadequate. Our goal is to determine the extent to which

[1] More detailed information on Cilk can be found at `http://supertech.lcs.mit.edu/cilk`.

an HTM computing environment can rid concurrent computing of error-prone protocols altogether.

Consequently, we decided to add a new keyword `atomic` to the Cilk language, which can be used as a type qualifier in function declarations or as a statement qualifier. When `atomic` is applied to a function declaration, calling the function does not break atomicity. When `atomic` occurs before a statement, it forces the statement to be atomic. The compiler signals an error if a function declaration or statement is declared atomic and it contains, for example, a `spawn` or a function call to a nonatomic function.

As an example, labeling a `while` loop as `atomic` causes all iterations of the loop to form a single transaction, rather than each individual iteration, as would normally be the case. In our benchmarks, few loops need to be labeled `atomic`. Many built-in library functions, such as those in `math.h`, need to be declared `atomic`, but only a few functions in user code need the type qualifier.

After modifying the Cilk compiler to C and Cilk code, we ran it on many of our Cilk benchmarks. We then investigated the character of the resulting atomic sections in order to determine whether a hardware implementation of transactional memory was reasonable under our linguistic model. Figure 1 shows the results, which although preliminary, are nevertheless encouraging. These statistics are conservative and do not exploit the fact that many variables cannot escape their lexical context and need not be included in the general HTM mechanism.

As can be seen from the table in the figure, the average number of variables per transaction is about 4, which indicates that hardware support can reasonably expect to handle the common case. The FFT code has 380 variables in its largest transaction, which suggests that a scalable HTM mechanism, rather than a fixed-size HTM mechanism, will probably be necessary for some codes. Of course, all these benchmarks are small, and we must do a more complete and scientific study to validate any proposed strategy for providing atomicity in a largely automatic fashion.

The compiler modifications we have just described simply break code into atomic sections. They do not actually produce running code with embedded transactions. In the future, we plan to modify the Cilk compiler to execute on the HTM system described in Section III. We shall measure the efficiency of the software/hardware support for HTM on the HTM simulator. These measurements should allow us, among other things, to explore compiler optimizations. For example, the compiler can identify some variables in an atomic section as being unsharable. Although these variables may need to be rolled back if the transaction is aborted, the mechanism is much simpler than with shared variables. A working compiler and a hardware simulator will allow us to make reasonable tradeoffs between what the compiler should implement and what should be put into hardware.

Program name	Max # var in any transaction	avg # var per transaction	# atomic blocks	# lines
`blockedmul.cilk`	68	12.0	7	352
`bucket.cilk`	13	3.7	12	295
`cholesky.cilk`	14	3.6	28	908
`cilksort.cilk`	2	1.5	22	510
`ck.cilk`	37	5.0	25	542
`fft.cilk`	380	37.9	31	3242
`fib-benchmark.cilk`	2	2.0	3	107
`game.cilk`	3	1.9	9	234
`heat.cilk`	14	4.8	13	414
`kalah.cilk`	26	4.3	46	911
`knapsack.cilk`	2	1.8	4	205
`knary.cilk`	5	3.0	3	158
`lu.cilk`	25	5.7	16	560
`magic.cilk`	5	1.8	81	993
`matmul.cilk`	2	1.6	9	177
`notempmul.cilk`	68	11.0	7	352
`plu.cilk`	7	2.7	25	432
`queens.cilk`	1	1.0	3	126
`rand.cilk`	1	1.0	2	40
`rectmul.cilk`	68	19.6	8	493
`spacemul.cilk`	68	19.0	8	468
`testall.cilk`	8	1.5	47	1334
`test-locks.cilk`	1	1.0	2	42

Fig. 1. Statistics on transaction sizes for Cilk benchmarks.

We also plan to make the compiler produce output that can execute on a software transactional memory (STM) system. Although compiling for STM will generally produce low-performance codes, it will allow us to experiment with real applications to learn about the strengths and inadequacies of our linguistic framework for HTM. In addition, a running implementation will allow us to experiment real programs with algorithms for contention resolution.

III. Architectural support for an HTM computing environment

We want an HTM computing environment to be implementable for single processors; for bus-based multiprocessors using, for example, snoopy caches [13]; and for

scalable multiprocessors using, for example, directory-based cache coherence [4], [6], [5]. Herlihy and Moss [19] showed how to implement transactional memory for these various architectures, but their implementation is not scalable in that it would not work if the transaction size exceeds the hardware resources. This section presents the outline of a design for a scalable HTM mechanism.

Since we wish to put transactions everywhere in the code, rather than in just a few critical sections, almost all memory operations will take place within a transaction. Achieving high performance will require hardware support, since in a software-only scheme, nearly every load and store instruction would incur significant overhead.

To support an HTM programming environment, the common case must be fast and correct, and the uncommon cases must interact correctly with the common case. Our compiler studies (described in Section II) indicate that when transactions are everywhere, the common case is a small transaction that fits within the on-processor cache. Herlihy and Moss showed how to implement small transactions efficiently by extending the MESI protocol [13]. We would like small transactions to run just as efficiently in a scalable system.

The uncommon case, in which a transaction is too big to fit in cache, must run correctly. Limiting transactions to the size of on-processor caches, or any other fixed size, makes the compiler's job difficult, because it is unwise for executable binaries to depend on implementation-specific, as opposed to architectural, parameters. One might propose that the compiler could, for large transactions, simply generate code to obtain a global lock, but locking protocols and transactions do not interact seamlessly. Specifically, every small transaction may now need to check the lock, resulting in increased overhead for the common case, or increased system complexity to mitigate the overhead.

We want to allow transactions to be huge, perhaps requiring many gigabytes of memory for the transaction's "undo log." If an application is willing to devote the memory, then the hardware should interact smoothly with software to support that application. A scalable HTM computing environment must support large transactions whether they access a large number of memory locations or run for a long time.

To solve the scalability problem, we plan to add memory to the computer architecture in three places: a commit record, a transaction log, and extra status for every cache line of main memory. The commit record is simply a location in memory that contains one of three values, indicating whether a transaction is pending, committed, or aborted. The idea is that a single write to the commit record will have the effect of committing (or aborting) all of the writes of a whole transaction. The transaction log contains, for each memory operation, the information needed to abort or commit a transaction. For each memory store, the transaction log contains the new value of memory. For each memory location that is read, the transaction log also contains some bookkeeping information. Thus, the extra status for each cache line indicates whether the cache line has been operated on as part of a transaction, in which case it points to a transaction log entry.

Surprisingly, reads are trickier than writes, because our protocol requires writers to gain exclusive access to each memory location. But, there can be many readers of a memory location, and the hardware must be able to find them all when a conflict is discovered. To make the protocol work, we allocate one entry in the transaction log for every read operation. The entry makes up one cell of a doubly linked list of all the transactions that read that location. The doubly linked list is constructed from the transaction logs of the various transactions that have performed reads. The extra status simply points to the head of the list.

Whenever a processor performs a load, the extra status must be checked. If the extra status indicates that the cache line is not part of a write in a transaction, then the read may proceed. If the extra status points to a write in the log of a transaction whose commit record is pending, then there is a conflict. If the extra status points to a write in the log of a transaction whose commit record indicates that the transaction has aborted, then the read may proceed (since the old value is stored in the memory location). If the extra status points to a write transaction and the commit record indicates that the transaction has committed, then the data must be read out of the transaction log. A similar set of rules applies when writing to a memory location.

The commit record and transaction log are only needed for memory locations that spill from the cache during the transaction. If indeed the common case is that most transactions do not spill from the cache, as we are hypothesizing, then transactions will usually incur little or no overhead.

Implementing our protocol requires additional memory. In our current design, the commit record and the transaction log are provided from ordinary program memory by the language runtime system. If the memory provided for the transaction log turns out to be too small, then the transaction can abort, and the runtime system can retry with a bigger transaction log. The memory for

extra status on a cache line, in contrast, must be added to the hardware main-memory system. In a directory-based cache system, it may be possible to conscript unused bits in the directory entry to keep track of the status. Thus, part of the memory is architecturally visible (the commit record and transaction log) and part is architecturally invisible (the extra status).

Thus, our implementation of scalable HTM extends the transactional memory instruction-set architecture (ISA) of Herlihy and Moss [18] to support scalable transactions, even those that do not fit within an on-processor cache. The ISA our scheme will use is essentially identical to that given by [18], except that the language runtime system provides memory to the transaction. We have not yet determined the policy and mechanism by which the HTM system decides which transaction to abort in the case of a conflict.

We are implementing a Bluespec [29] model of our HTM design to show that we have not missed any important details. Bluespec is a high-level hardware description language developed at Sandburst Corporation which makes it easier to write term-rewriting descriptions of hardware. Bluespec developed out of work by others in our Laboratory on using term-rewriting systems to design and verify cache-coherence protocols [1], [20], [21], [26], [22], [31], [33], [29]. Bluespec can be compiled into circuits or to a cycle-accurate C-language simulator. Bluespec produces Verilog output, which can then be used to program an FPGA, as well as a cycle-accurate C-language backend.

IV. Conclusion

The overall goal of this research is to make parallel computing easier for ordinary programmers, not just for expert computer scientists. Today, it makes little sense for most programmers to take on the complexities of parallel programming. Consequently, high-performance programming is a niche business, codes are expensive to develop, and they often underperform expectations. For parallel programming to advance significantly into the mainstream, it must become simpler. Although it will be intellectually challenging to address all the problems inherent in developing an HTM computing environment, we believe that HTM represents a path towards overall programming simplicity. Our thesis is that the ability of programmers to exploit the high-level structure of their applications (such as exploiting sparsity in a matrix) will outweigh the complexities of the low-level implemention, leading to a large net improvement in overall performance.

V. Acknowledgments

Special thanks to Clement Menier of ENS Lyon. As a summer intern in the Laboratory for Computer Science, Clement implemented the changes to the Cilk compiler so that we could study the tradeoffs between implicit and explicit atomicity. Thanks to Victor Luchangco of Sun Labs and Larry Rudolph of MIT for helpful discussions. Thanks to the many people at Silicon Graphics for their interest in our ideas.

References

[1] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, May–June 1999. `http://csg.lcs.mit.edu/pubs/memos/Memo-419/memo-419.pdf`.

[2] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, Sept. 1995. `ftp://theory.lcs.mit.edu/pub/cilk/rdb-phdthesis.ps.Z`.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.), `ftp://theory.lcs.mit.edu/pub/cilk/cilkjpdc96.ps.gz`.

[4] L. M. Censier and P. Feautrier. A new solution to cache coherence problems in multicache systems. *IEEE Transactions on Computers*, pages 1112–1118, Dec. 1978.

[5] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherence scheme. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, Santa Clara, California, Apr. 8–11 1991. Proceedings published as SIGARCH Computer Architecture News, Volume 19, Number 2, April 1991. SIGOPS Operating Systems Review, Volume 25, Special Issue, April 1991. SIGPLAN Sigplan Notices, Volume 26, Number 4, April 1991, `ftp://ftp.cag.lcs.mit.edu/pub/papers/pdf/asplos4.pdf`.

[6] David Lars Chaiken. Cache coherence protocols for large-scale multiprocessors. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, Sept. 1990. Also available as MIT/LCS Technical Report 489., `ftp://ftp.cag.lcs.mit.edu/pub/papers/pdf/chaiken-thesis.pdf`.

[7] Guang-Ien Cheng. Algorithms for data-race detection in multithreaded programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 1998. `ftp://theory.lcs.mit.edu/pub/cilk/cheng-thesis.ps.gz`.

[8] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2 1998. `ftp://theory.lcs.mit.edu/pub/cilk/brelly.ps.gz`.

[9] Digital Equipment Corporation, Maynard, Massachusetts. *DIGITAL Fortran 90—User Manual for DIGITAL UNIX Systems*, Mar. 1998. http://www.helsinki.fi/atk/unix/dec_manuals/df90au52/dfum.htm.

[10] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 199. A preliminary version appeared as [11].

[11] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 1–11, Newport, Rhode Island, June 22–25 1997. ftp://theory.lcs.mit.edu/pub/cilk/spbags.ps.gz.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998., ftp://theory.lcs.mit.edu/pub/cilk/cilk5.ps.gz.

[13] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Conference Proceedings of the tenth International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, June 1983. ftp://ftp.cs.wisc.edu/galileo/papers/retro-goodman.pdf.

[14] Jim Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference of Very Large Data Bases*, pages 144–154, Sept. 1981. Also published as [15].

[15] Jim Gray. The transaction concept: Virtues and limitations. Technical Report 81.3, Tandem Computers, June 1981. http://www.hpl.hp.com/techreports/tandem/TR-81.3.html.

[16] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free software transactional memory for supporting dynamic data structures. Unpublished manuscript received from Victor Luchangco, Oct. 2002.

[17] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. Unpublished manuscript received from Victor Luchangco, Oct. 2002.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Conference on Computer Architecture. (Also published as ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, May 1993.)*, pages 289–300, San Diego, California, 1993. http://www.cs.brown.edu/people/mph/isca2.ps.

[19] M. P. Herlihy and J.E.B. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, One Kendal Square, Cambridge, MA 02139, Dec. 1992. ftp://ftp.cs.umass.edu/pub/osl/papers/crl-92-07.ps.Z.

[20] James C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 2000. http://www.ece.cmu.edu/~jhoe/distribution/thesis.ps.

[21] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of X IFIP International Confernce on VLSI (VLSI 99)*, Lisbon, Portugal, Dec. 1–4 1999. http://www.ece.cmu.edu/~jhoe/distribution/csgmemo/memo-421a.ps.

[22] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of 2000 International Conference on Computer-Aided Design*, pages 511–518, San Jose, California, Nov. 5–9 2000. http://www.ece.cmu.edu/~jhoe/distribution/csgmemo/memo-426a.ps.

[23] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — part 1: System application program interface (API) [C language]. IEEE Std 1003.1, 1996 Edition.

[24] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, California, Nov. 1987.

[25] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, Jan. 1996. ftp://theory.lcs.mit.edu/pub/cilk/joerg-phd-thesis.ps.gz.

[26] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the Java memory model using CRF. Computation Structures Group Memo 428, MIT Laboratory for Computer Science, Oct. 6 2000. http://csg.lcs.mit.edu/pubs/memos/Memo-428/memo-428.pdf.

[27] OpenMP: A proposed industry standard API for shared memory programming. OpenMP white paper, Oct. 1997. http://www.openmp.org/specs/mp-documents/paper/paper.ps.

[28] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, San Jose, California, Oct. 5–9 2002. http://www.cs.wisc.edu/~rajwar/papers/asplos02.pdf.

[29] Sandburst Corporation. Bluespec. Web page, Oct. 2002. http://bluespec.org.

[30] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed*, pages 204–213, Ottawa, Ontario, Canada, 1995.

[31] Xiaowei Shen. *Design and Verification of Adaptive Cache Coherence Protocols*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, Jan. 2000.

[32] Andrew F. Stark. Debugging multithreaded programs that incorporate user-level locking. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, May 1998. ftp://theory.lcs.mit.edu/pub/cilk/astark-thesis.ps.gz.

[33] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In J. N. Oliveira and Pamela Save, editors, *Proceedings of the Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe (FME 2001)*, volume 2021 of *Lecture Notes in Computer Science*, pages 43–71, Berlin, Germany, Mar. 12–16 2001. Springer-Verlag. http://csg.lcs.mit.edu/pubs/memos/Memo-432/memo-432.pdf.

[34] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, Nov. 2001. http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf.