What Kinds of Applications Can Benefit From Transactional Memory?

Mark Moir† Dan Nussbaum†
Sun Labs at Oracle
mark.moir@oracle.com dan.nussbaum@oracle.com

Abstract

We discuss the current state of research on transactional memory, with a focus on what characteristics of applications may make them more or less likely to be able to benefit from using transactional memory, and how this issue can and should influence ongoing and future research.

1. Introduction

We discuss characteristics of applications that might benefit from transactional memory (TM). We do not aim to identify specific applications or domains, but rather to clarify the nature of transactional memory, its benefits in various contexts, tradeoffs and challenges in various contexts, and how TM research should proceed in order to have the best chance of being useful in as many contexts as possible.

Part of clarifying these issues is to attempt to dispel various myths and misconceptions that are often stated but rarely justified. We should first establish *what* it is that we are discussing. The phrase "transactional memory" can refer to a variety of technologies, some that propose fundamentally new programming paradigms, some that aim to augment and improve existing programming paradigms, and some that are transparent or nearly transparent. Furthermore, TM can be implemented in hardware [14], in software [20], or in a variety of combinations of the two [5, 17]. Therefore, in following various informal debates and discussions about TM, we have come to the following conclusion: All short sentences about transactional memory are wrong. Except that one!

Despite the many interfaces, contexts, and implementation approaches, there is a common theme: The *essence* of transactional memory is the ability for a programmer to specify that a set of memory accesses occur *atomically*[1], without specifying *how* this is achieved. We believe that this common theme captures the power of TM to simplify concurrent code, making it easier to write, read, and maintain.

Whether and how this power can be effectively exploited is the subject of ongoing research. Substantial progress and numerous encouraging results have been achieved. Nonetheless, challenges remain and more research is needed before TM could be seriously considered for widespread production use. We believe that there is plenty of room for improvement over today's TM prototypes, and that the power of TM can be effectively exploited in a range of contexts.

Paul McKenney [19] quotes an unnamed researcher as saying "[TM will] soon be the only synchronization mechanism, supplanting all the complex and error-prone synchronization mechanisms currently in use", and then launches into a series of arguments to the contrary. We don't think any reasonable TM researcher thinks this. Furthermore, we are not aware of *anyone* (reasonable or otherwise!) putting their name to such a claim. But many arguments put forward by TM detractors amount to negating this claim. This is easy, but does not make a useful contribution to the discussion of whether and how TM can make programmers' lives easier.

Based on our research and experience, we are confident that TM can be extremely valuable in some contexts. In other cases, we are more skeptical. Nonetheless, we certainly think it is reasonable to explore how far the boundaries can be pushed. Furthermore, we do not find arguments against TM based on predictions that it will fail to achieve the (alleged) original goals of some researchers convincing. The fruits of research are frequently different from those stated and anticipated when the research began.

In this paper, we discuss whether and how we think various kinds of TM can be useful in various contexts, and how these issues should affect ongoing research. In Section 2, we describe various approaches to supporting mechanisms that

[1] Following the terminology of the distributed computing theory community, e.g. [18], we use the word "atomic" to mean not only "all or nothing", but also "indivisible", so that a transaction does not observe effects of concurrent activity by other threads or processes, and other threads or processes do not observe partial effects of a transaction. Those from a database background would view this as the combination of atomicity and isolation.

fall under the TM umbrella, and contexts in which they may be useful. Then, in Section 3 we address some myths and misconceptions we have found to be stated without justification by some and accepted at face value by others. Section 4 discusses advantages and disadvantages of benchmarks and applications that have been used to evaluate TM designs to date, and how they should change to best support ongoing research. Concluding remarks appear in Section 5.

2. Types of TM and how they can be useful

TM eliminates the need to define and obey conventions dictating which data is protected by which lock[2]. Establishing and enforcing such locking conventions can be challenging, and many bugs result from programmers applying the convention incorrectly. Furthermore, trivial locking conventions (such as all data being protected by a single lock) inhibit scalability, especially as more cores become available, while more complex, fine-grained ones are more difficult to follow, and importantly, more difficult to modify.

The best choice of locking convention often depends on the target architecture. TM establishes an abstraction layer, so that the same application code can run on different runtime platforms on different systems, as appropriate for a given target architecture. Another benefit of this abstraction layer is that improvements to the TM infrastructure—whether in hardware or software—can improve performance with no changes to the application code.

In concurrent programming, including with TM, it is best to avoid unnecessary synchronization between parts of an application that are intended to run in parallel. In general, TM does *not* divide a task into pieces that can be executed in parallel. However, it does make it easier to do so. For many problems it is difficult or impossible to effectively break a task into independent pieces that can be run in parallel without synchronization. With TM, the problem can be broken up into pieces that *usually* don't conflict, which is substantially easier. These pieces can be run in parallel using TM. If the pieces usually don't conflict, then they are executed in the uncontended case for which TM systems are generally optimized. When conflicts do occur, TM detects them and ensures correct execution.

The abstraction layer provided by TM also allows relatively complex mechanisms to be used that cannot (reasonably) be embedded in application code. An example is read sharing. Suppose an application has some data that is read frequently and written only occasionally. HTM that is implemented on top of a cache coherence protocol (for example, Sun's Rock processor) can access such data without acquiring exclusive ownership of any cacheline. If such data is protected by a lock, acquiring the lock requires exclusive ownership of its cacheline, resulting in additional coherence traffic, additional latency, and poor scalability. For a software example, we have used SNZI objects [11] to improve the scalability of STM read sharing [16].

These arguments concern performance, scalability, portability, and maintainability of application code. This overlooks a critical advantage of TM, namely that it makes it much easier for programmers to achieve *correct* implementations of concurrent algorithms and data structures. When our group introduced DSTM [13] (the first STM to support dynamic data structures), we used it to implement a transactional red-black tree. Apart from translating to DSTM's ugly experimental interface, this was essentially no more difficult that writing a sequential red-black tree. Because DSTM is a nonblocking STM, the result was a nonblocking red-black tree. At that time, this was by far the most sophisticated nonblocking data structure in existence. Developing such nonblocking data structures directly is *very* challenging, but TM hides that complexity from the programmer.

In a more recent illustration of how TM can make concurrent programming easier, we used HTM to implement SNZI objects [11] with a fairly straightforward algorithm, whereas the existing algorithms are publishable results! The simple HTM-based algorithms substantially outperformed the more complex software-only ones. This and other examples in which we have used HTM to simplify code and/or improve its performance and scalability are presented in detail in [6].

Next we discuss the variety of ways in which TM can be exposed to programmers, or can be used transparently.

STM library interface: Transactions can be used by programming directly to an STM library interface. This approach is workable for small experiments, and has been important for testing and evaluating prototype systems.

Language support: Programming to the interface of an STM library quickly becomes tedious and error prone, and it is clear that language support is needed to enable productive use by mainstream programmers. A draft specification developed by researchers in our group together with others at IBM and Intel [1], outlines such language support for the context of C++, and several groups are developing experimental systems based on it. Language features and interfaces have been proposed for various other contexts as well.

Transparent use of TM: TM can be used to improve system software in various ways, including performance, scalability, and less direct benefits, such as simplifying code. For one example, our group recently showed that efficient work stealing queues can be implemented with much simpler code than existing software implementations by using HTM [6]. The Java HotSpot™ Virtual Machine (JVM™) uses a complex work stealing algorithm for parallel garbage collection, which has been responsible for a number of difficult concurrency bugs. The cost of these bugs might have been avoided

[2] Using TM in one context will *not* eliminate disadvantages of continued use of locking in another. We make this rather obvious statement because of the all-too-common "argument" that such sentences represent a claim that TM will completely eliminate disadvantages of locking, which would require it to replace all locks everywhere. Again, we do not view this as a goal of TM.

if it were possible to use a simpler HTM-based algorithm. This would not require Java programmers to change their code or even be aware of TM.

Another way in which TM can be exploited without departing from existing programming models is Transactional Lock Elision (TLE) [7, 8], which uses HTM to attempt to execute critical sections protected by the same lock in parallel. For example, a JVM can be modified to use this technique in the implementations of synchronized blocks and methods for unmodified Java programs [7, 8]. TLE can also be used in native execution contexts, such as C or C++, with little or no impact on application source code.

We now turn to specific TM implementation approaches and the kinds of applications that may benefit from them.

Software TM: STM has the advantage that researchers can build or download an STM and experiment with it on standard hardware. As a result, there has been a wealth of research on implementing STM. While numerous improvements have been made and continue to be made, STMs built to date entail significant overhead as compared to simple code protected by a lock, due to the need to instrument every load and store performed in a transaction, as well as for initializing transactions and committing them. Thus, even though STM can provide scalability, while a single lock cannot, STM may fail to provide a performance benefit over a single lock in some cases. In other cases, the scalability of STM allows it to outperform a single lock implementation.

We expect the overhead of STM to continue to decrease through compiler optimizations and other innovation, though clearly it will always be noticeable. Just as importantly, as discussed in Section 4, characteristics of a workload have significant influence on its chances of profitably using STM. In particular, if STM is used sparingly to handle tricky synchronization to coordinate tasks that can be performed mostly without synchronization, then the parallelism it helps enable is more important than the overhead of the few and small transactions that are used. On the other hand, if transactions are used excessively, especially if large transactions are used frequently, then the overhead of STM *is* critical and the likelihood of conflicts between transactions is higher. Such applications are much less likely to derive any benefit from STM. Exploring the spectrum between these points and determining "how much is too much" is an important topic for ongoing research, both for characterizing applications that can make good use of STM, as well as for optimizing STM systems for those cases.

Hardware TM: HTM can have much lower overhead than STM, because there is no need to instrument every memory access in a transaction, and hardware can exploit existing mechanisms such as caches and cache coherence protocols directly to detect conflicts. In contrast, STM often requires expensive and complicated algorithms for the same purpose. For similar reasons, most HTM designs provide *strong atomicity* [2], meaning that concurrent nontransactional and transactional accesses to the same memory location are allowed. Providing strong atomicity using STM is usually expensive, and sometime infeasible (for example, when legacy libraries are used that cannot be recompiled).

On the other hand, HTM is less flexible than STM, and existing implementations are subject to certain limitations that do not apply to STM. For example, transactions on Rock [7, 8] cannot exceed the limitations of certain processor structures, such as caches, TLBs, write buffers, etc. As a result, a software alternative is often needed. The complexity of having this software alternative can be hidden from the programmer in *some* cases (see below), but not all.

We have demonstrated the use of HTM to significantly simplify some concurrent code and/or improve its performance for a number of purposes [6]. Some of these examples require strong atomicity, some require a software alternative, and some depend on an assumption that certain classes of small, simple transactions will eventually commit so that no software alternative is needed. Furthermore, in several cases, low latency for hardware transactions is critical to successfully exploiting them.

Hardware-Software Combinations: Several proposals for supporting TM use hardware support to improve performance and/or simplify software, but depend on STM to avoid the limitations of the assumed hardware support. For example, Hybrid TM (HyTM) [5] and Phased TM (PhTM) [17] both attempt to use HTM to commit a transaction, but if it (repeatedly) fails, resort to more expensive STM to commit the transaction. Such systems aim to exploit HTM to the extent it is available and effective, but to transparently hide the details of the HTM and its limitations from the programmer. While we expect to improve on existing prototypes, it seems clear that there will always be a significant difference in performance between HTM transactions and STM ones.

For some applications, this difference may be unacceptable, in which case TM—or at least the current systems—may not be suitable. In other applications, transactions may always or almost always succeed using HTM. For example, in [6] we explain how PhTM provides a correct and scalable work stealing queue for which operations always succeed using HTM, except for occasional resizing operations, which can exceed the store buffer capacity. Because such operations are rare, the additional overhead of using STM to complete them is of little consequence. In yet other applications, the main requirement is to complete as soon as possible, and if that means some transactions execute more slowly using STM, but enough are successful with HTM to improve overall execution time, the use of TM is beneficial despite the difference in performance between HTM and STM.

Finally, such systems are useful for developing transactional applications before executing them on a simulator, or before a future HTM feature becomes available.

3. TM myths and misconceptions

TM Myth # 1: *TM is intended as a replacement for all locks.* As discussed, we don't think any serious TM researcher believes this. There is interesting research aimed at making TM interact better with various existing mechanisms, and allowing code to be executed in transactions that cannot be meaningfully supported by most designs today. But nobody could reasonably argue that TM will replace *all* locks or other synchronization constructs.

TM Myth # 2: *Locks compose and/or composition is not needed and/or is harmful.* Sloppy language has obscured this debate somewhat. It is true that lock-based implementations of two operations on a data structure cannot be straightforwardly composed into a single one. It is also true that, if one is willing to expose locking conventions to callers, then something like composition *can* be achieved. But the constraints imposed by such techniques make composition in most TM systems much cleaner and more flexible.

We have also heard the argument that composition will lead to transactions that are too large and performance will suffer unacceptably. Just about anything can be overused or misused. That is not an argument against supporting it. The debate should be about how *well* TM can be used by various kinds of programmers, not how poorly.

TM Myth # 3: *TM is primarily intended for improving performance under heavy contention.* This myth is probably a result of the use by many TM researchers (ourselves included) of microbenchmarks that aim to examine the behavior of TM systems under the worst conditions possible: frequent conflicts. Robustness under contention is important, but this does not imply that this is the purpose for which TM is intended. We discuss this issue further in Section 4.

TM Myth # 4: *Using TM requires the entire application to be rewritten and/or recompiled.* Using TM in one module does not automatically require changes elsewhere. Regarding recompilation, there is some truth to this claim in some contexts. In a system that provides the features specified in [1], only code that can be called (directly or indirectly) from within a transaction must be recompiled. In other contexts, such as when strong atomicity is required, full recompilation may be necessary. This requirement is of little consequence in some contexts, and unacceptable in others.

TM Myth # 5: *Programmers will never be able to debug TM or diagnose performance problems.* It is true that, if programmers in general are to use transactions explicitly, then debuggers will need to understand about transactions in order to present them in a meaningful way to users, and performance and profiling tools will need new functionality to assist in diagnosing performance problems related to TM. We and others are working on TM support for such tools [12, 15]. Not only are we confident that such tools can effectively support TM, but we think that TM and its infrastructure can actually make it *easier* to debug and profile transactional programs than with existing programs and tools.

TM Myth # 6: *TM is not useful because it can't do X, for some X.* As discussed already, it seems this class of (non)argument has arisen from the (incorrect, in our opinion) belief that TM proponents claim that TM is the solution to all problems in all contexts.

TM Myth # 7: *TM converts deadlocks to livelocks, and you'd rather have a deadlock than a livelock.* Many existing prototypes allow the possibility of livelock, avoiding it in practice using techniques such as backoff. However, TM does *not* fundamentally introduce livelock. TM systems can provide a variety of strong progress conditions, for example by becoming more conservative in response to excessive conflicts and retries. In contrast, once deadlock occurs in a lock-based program, there is usually no correct way for the threads involved to continue operating.

We do agree that it is easier to examine a system if it is stopped. TM-aware tools can be used to cause a program to stop when TM-related events—such as the number of retries of a transaction exceeding a specified threshold—occur, allowing it to be examined in a debugger [15]. Unlike with deadlock, the system can then resume execution.

TM Myth # 8: *TM makes programming harder because the programmer must follow new rules that are hard to understand.* This claim is made mostly due the requirement of many STMs that programmers avoid concurrent transactional and nontransactional accesses to the same data. This is inconvenient, to be sure. However, a similar challenge comes with lock-based programming. Depending on the context, semantics of "racy" programs that allow synchronized and unsynchronized code to access the same data concurrently are either undefined (as in the forthcoming C++ standard), or depend on subtle aspects of memory consistency models. In fact it is *easier* to avoid such races using transactions than using locks because programmers using locks must not only ensure that concurrent accesses to the same variable are protected, but also that they are protected by the *same* lock. Transactional programmers need only ensure that such accesses are within transactions.

TM Myth # 9: *TM makes programming harder because of the need to retry transactions, manage contention between them, etc.* Using various approaches already mentioned, reacting to transaction failures, backing off if necessary, and deciding how and when to retry is all handled by system code that is hidden from the regular programmer.

TM Myth # 10: *The overhead posed by STM may likely overshadow its promise.* To the extent that we can parse this sentence, which is enshrined in large font in [4], it seems to be a conjecture that the cost of STM fundamentally outweighs its benefit. In our opinion, the data presented in [4] do not support such a claim. First, the authors use a small number of existing STM systems on a handful of benchmarks to

reach this conclusion. There is no evidence that those STM systems are anywhere close to optimal (we are confident that they are not), and furthermore no evidence that the benchmarks used have any value in predicting the potential value of STM. Indeed, the authors of [10] were unable to reproduce the results of [4], and when they investigated why, they discovered that the parameters of one "standard" benchmark had been modified to increase contention and thereby reduce performance, and that the results of another experiment were due to architectural features unrelated to TM. Even using the same limited set of benchmarks, they were able to achieve results that were much more encouraging for STM than those reported in [4]. Naturally, this begs the question of what the "right" workloads are to evaluate TM prototypes. This is the subject of the next section.

4. Evaluating TM prototypes

To date, TM prototypes have mostly been evaluated using the following types of workloads:

Microbenchmarks: Each thread repeatedly performs a randomly chosen operation on some shared data structure—such as a hash table, linked list, or red-black tree—as quickly as possible. Parameters control the mix of operations. For example, for a hash table we control the probability that each operation is a `get`, `put`, or `lookup` operation. We then graph throughput against the number of threads. We usually compare against a single-lock implementation, because that is what can be achieved using locks with a similar level of programming complexity to using a transaction. Occasionally, a fine-grained locking solution is not much more difficult, so we compare against that too [8].

Such microbenchmarks are valuable not because they represent realistic workloads, but because they allow us to examine the behavior of a TM prototype in a variety of conditions. Single-threaded throughput gives an indication of the overhead of transactions, and workloads that *should* scale well allow us to evaluate the scalability of a TM prototype under heavy use with few conflicts.

Finally, by choosing workloads in which conflicts are more frequent, we can examine the stability of the system under heavy contention. By examining how quickly and how severely performance deteriorates, we get an indication of how well the system handles contention.

Microbenchmarks are also valuable for debugging and tuning a system, because the code is small enough that we can examine it in detail, and because we have the flexibility to control parameters to test hypotheses, etc.

Thus, microbenchmarks are a critical tool in evaluating TM implementations. Clearly, though, they are *not* likely to be representative of well-designed transactional applications, which should use transactions sparingly and such that conflicts between transactions are relatively rare. By blindly using microbenchmarks to evaluate TM prototypes, we risk optimizing for the wrong workloads.

Applications: A number of useful benchmark suites (e.g. STAMP [3]) are available, including some workloads that are at least somewhat more realistic than microbenchmarks. As experimental implementations of transactional language features have begun to emerge, it has become feasible to build larger transactional applications, so hopefully more applications will become available. Nonetheless, there are still few real transactional applications, and we are still far from identifying a set that is representative of real workloads.

While we are interested in new applications, we continue to rely primarily on microbenchmarks to evaluate our TM systems. However, we are changing some aspects of how we use them, in part to respond to the above-mentioned risk of optimizing for the wrong workloads. The way we have typically used microbenchmarks is unrepresentative of real workloads in at least two ways.

First, while single-thread performance in such microbenchmarks can give an indication of the system's overhead, this overestimates the impact of such overhead on applications that use transactions sparingly for synchronization, while seeking to minimize synchronization overall. This is one reason we believe that the authors of [4] are mistaken in their conclusion (see TM Myth # 10 in Section 3).

Second, single-thread performance is not a faithful indicator of multi-threaded performance with low contention, because cache behavior differs significantly between these two scenarios (cache hit rates are likely to be much higher in the former than in the latter). Because STMs typically access metadata in addition to each data access, this factor may *under*estimate the impact of STM on performance.

These shortcomings can be addressed at least in part by making threads pause between transactions, and controlling the (distribution of) pause times. This allows us to better characterize the kinds of workloads that fare best with various TM implementations. Such exploration can give guidance about what kinds of applications are likely to extract most benefit from TM, and insight into what metrics may be most useful for evaluating TM implementations.

Finally, the target architecture for a given implementation is important. In our group, we have explored how to make transactions *fast* (such as TL2 [9]) and how to make transactions *scalable* (such as SkySTM [16]). These efforts have illuminated difficult tradeoffs between speed and scalability. We are currently exploring STM designs that are as fast as possible while being "scalable enough" for realistic applications running on the single-chip multicore systems of the next few years, as we believe this is likely to be the context for most STM applications in that timeframe. We are evaluating our TM systems across a range of workloads, with an eye towards exploring characteristics of workloads that perform well in various contexts, as well as to building systems that are fast, but include mechanisms sufficient to achieve scalable performance on single-chip multicore systems.

5. Concluding remarks

We have discussed various forms of transactional memory in various contexts, and characteristics of applications and workloads that may profitably exploit them. We have also sought to dispel a few myths and misconceptions about transactional memory, and to discuss how ongoing research should evaluate new ideas, taking into account what workload characteristics may be representative of realistic workloads. We believe applications can exploit small transactions sparingly in order to facilitate scalable synchronization in support of parallel applications. We are less sure that it will make sense for applications to make heavier use of transactions or use larger transactions, but it remains to be seen where the boundaries lie, and to more fully characterize what kinds of applications can profitably use various kinds of TM.

References

[1] A.-R. Adl-Tabatabai and T. Shpeisman (Eds.). Draft specification of transactional language constructs for C++, version 1.0. `http://research.sun.com/scalable/pubs/C++-transactional-constructs-1.0.pdf`, Aug. 2009.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proceedings of the 12th Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[6] D. Dice, Y. Lev, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2010.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, New York, NY, USA, 2009.

[8] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006.

[10] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, July 2010.

[11] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC '07: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2007.

[12] M. Herlihy and Y. Lev. tm_db: A generic debugging library for transactional programs. In *Proceedings of the 18th IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 136–145, Washington, DC, USA, 2009.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[15] Y. Lev. Debugging and proling of transactional programs. `www.cs.brown.edu/people/levyossi/Thesis`, Apr. 2010.

[16] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Workshop on Transactional Computing (Transact)*, February 2009. research.sun.com/scalable/pubs/ TRANSACT2009-ScalableSTMAnatomy.pdf.

[17] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007. research.sun.com/scalable/pubs/ TRANSACT2007-PhTM.pdf.

[18] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[19] P. McKenney. Transactional memory everywhere?, September 2009. `paulmck.livejournal.com/10264.html`.

[20] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, (10):99–116, 1997.