Transactional Memory Today[1]

Michael Scott
Computer Science Department
University of Rochester, NY, USA
scott@cs.rochester.edu

It was an honor and a privilege to be asked to participate in the celebration, at PODC 2014, of Maurice Herlihy's many contributions to the field of distributed computing—and specifically, to address the topic of transactional memory, which has been a key component of my own research for the past decade or so.

When introducing transactional memory ("TM") to people outside the field, I describe it as a sort of magical merger of two essential ideas, at different levels of abstraction. First, at the language level, TM allows the programmer to specify that certain blocks of code should be atomic without saying how to *make* them atomic. Second, at the implementation level, TM uses speculation (much of the time, at least) to execute atomic blocks in parallel whenever possible. Each dynamic execution of an atomic block is known as a *transaction*. The implementation guesses that concurrent transactions will be mutually independent. It then monitors their execution, backing out and retrying if (and hopefully only if) they are discovered to conflict with one another.

The second of these ideas—the speculative implementation—was the focus of the original TM paper, co-authored by Maurice with Eliot Moss [22]. The first idea—the simplified model of language-level atomicity—is also due largely to Maurice, but was a somewhat later development.

1 Motivation

To understand the original motivation for transactional memory, consider the typical method of a nonblocking concurrent data structure. The code is likely to begin with a "planning phase" that peruses the current state of the structure, figuring out the operation it wants to perform, and initializing data—some thread-private, some visible to other threads—to describe that operation. At some point, a critical *linearizing instruction* transitions the operation from "desired" to "performed." In some cases, the identity of the linearizing instruction is obvious in the source code; in others it can be determined only by reasoning in hindsight over the history of the structure. Finally,

[1]Based on remarks delivered at the Maurice Herlihy 60th Birthday Celebration, Paris, France, July 2014

the method performs whatever "cleanup" is required to maintain long-term structural invariants. Nonblocking progress is guaranteed because the planning phase has no effect on the logical state of the structure, the linearizing instruction is atomic, and the cleanup phase can be performed by any thread—not just the one that called the original operation.

Two issues make methods of this sort very difficult to devise. The first is the need to effect the transition from "desired" to "performed" with a single atomic instruction. The second is the need to plan correctly in the face of concurrent changes by other threads. By contrast, an algorithm that uses a coarse-grained lock faces neither of these issues: writes by other threads will never occur in the middle of its reads; reads by other threads will never occur in the middle of its writes.

2 The Original Paper

While Maurice is largely celebrated for his theoretical contributions, the original TM paper was published at ISCA, the leading architecture conference, and was very much a hardware proposal. We can see this in the subtitle—"Architectural Support for Lock-Free Data Structures"—and the abstract: "[TM is] ... intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion."

The core idea is simple: a transaction runs almost the same code as a coarse-grain critical section, but with special load and store instructions, and without the actual lock. The special instructions allow the hardware to track conflicts between concurrent transactions. A special end-of-transaction commit instruction will succeed (and make transactionally written values visible to other threads) only if no concurrent conflicting transaction has committed. Here "conflict" means that one transaction writes a cache line that another reads or writes. Within a transaction, a special validate instruction allows code to determine whether it still has a chance to commit successfully—and in particular, whether the loads it has performed to date remain mutually consistent. In response to a failed validate or commit, the typical transaction will loop back (in software) and start over.

Looking back with the perspective of more than 20 years, the original TM paper appears remarkably prescient. Elision of coarse-grain locks remains the principal use case for TM today, though the resulting algorithms are "lock-free" only in the informal sense of "no application-level locks," not in the sense of livelock-free. Like almost all contemporary TM hardware, Herlihy & Moss (H&M) TM was also a "best-effort-only" proposal: a transaction could fail due not only to conflict or to overflow of hardware buffers, but to a variety of other conditions—notably external interrupts or the end of a scheduling quantum. Software must be prepared to fall back to a coarse-grain lock (or some other hybrid method) in the event of repeated failures.

Speculative state (the record of special loads and stores) in the H&M proposal was kept in a special "transactional cache" alongside the "regular" cache (in 1993, processors generally did not have multiple cache layers). This scheme is still considered viable today, though commercial offerings vary: the Intel Haswell processor leverages the regular L1 data cache [40]; Sun's unreleased Rock machine used the processor store buffer [10]; IBM's zEC12 uses per-core private L2s [25].

In contrast with current commercial implementations, H&M proposed a "responder wins" coherence strategy: if transaction A requested a cache line that had already been speculatively read or written by concurrent transaction B, B would "win" and A would be forced to abort. Current machines generally do the opposite: "responder loses"—kill B and let A continue. Responder-loses has the advantage of compatibility with existing coherence protocols, but responder-wins turns out

to be considerably less vulnerable to livelock. Nested transactions were not considered by H&M, but current commercial offerings address them only by counting, and subsuming the inner transactions in the outer: there is no way to abort and retry an inner transaction while keeping the outer one live.

Perhaps the most obvious difference between H&M and current TM is that the latter uses "modal" execution, rather than special loads and stores: in the wake of a special tm-start instruction, all ordinary memory accesses are considered speculative. In keeping with the technology of the day, H&M also assumed sequential consistency; modern machines must generally arrange for tm-start and commit instructions to incorporate memory barriers.

While designers of modern systems—both hardware and software—think of speculation as a fundamental design principle—comparable to caching in its degree of generality—this principle was nowhere near as widely recognized in 1993. In hindsight, the H&M paper (which doesn't even mention the term) can be seen not only as the seminal work on TM, but also as a seminal work in the history of speculation.

3 Subsequent Development

Within the architecture community, H&M TM was generally considered too ambitious for the hardware of the day, and was largely ignored for a decade. There was substantial uptake in the theory community, however, where TM-like semantics were incorporated into the notion of universal constructions [3, 5, 24, 28, 35]. In 1997, Shavit and Touitou coined the term "Software Transactional Memory," in a paper that shared with H&M the 2012 Dijkstra Prize [33].

And then came multicore. With the end of uniprocessor performance scaling, the difficulty of multithreaded programming became a sudden and pressing concern for researchers throughout academia and industry. And with advances in processor technology and transistor budgets, TM no longer looked so difficult to implement. Near-simultaneous breakthroughs in both software and hardware TM were announced by several groups in the early years of the 21st century.

Now, another decade on, perhaps a thousand TM papers have been published (including roughly a third of my own professional output). Plans are underway for the 10th annual ACM TRANSACT workshop. Hardware TM has been incorporated into multiple "real world" processors, including the Azul Vega 2 and 3 [7]; Sun Rock [10]; IBM Blue Gene/Q [36], zEnterprise EC12 [25], and POWER8 [6]; and Intel Haswell [40]. Work on software TM has proven even more fruitful, at least from a publications perspective: there are many more viable implementation alternatives—and many more semantic subtleties—than anyone would have anticipated back in 2003. TM language extensions have become the synchronization mechanism of choice in the Haskell community [16], official extensions for C++ are currently in the works (a preliminary version [1] already ships in gcc), and research-quality extensions have been developed for a wide range of other languages.

4 Maurice's Contributions

Throughout the history of TM, Maurice has remained a major contributor. The paragraphs here touch on only a few of his many contributions. With colleagues at Sun, Maurice co-designed the DSTM system [18], one of the first software TMs with semantics rich enough—and overheads low enough—to be potentially acceptable in practice. Among its several contributions, DSTM

introduced the notion of out-of-band *contention management*, a subject on which Maurice also collaborated with colleagues at EPFL [13, 14]. By separating safety and liveness, contention managers simplify both STM implementation and correctness proofs.

In 2005, Maurice collaborated with colleagues at Intel on mechanisms to virtualize hardware transactions, allowing them to survive both buffer overflows and context switches [30]. He also began a series of papers, with colleagues at Brown and Swarthmore, on transactions for energy efficiency [12]. With student Eric Koskinen, he introduced *transactional boosting* [20], which refines the notion of conflict to encompass the possibility that concurrent operations on abstract data types, performed within a transaction, may commute with one another at an abstract level—and thus be considered non-conflicting—even when they would appear to conflict at the level of loads and stores. With student Yossi Lev he explored support for debugging of transactional programs [21]. More recently, again with the team at Sun, he has explored the use of TM for memory management [11].

Perhaps most important, Maurice became a champion of the promise of transactions to simplify parallel programming—a promise he dubbed the "transactional manifesto" [19]. During a sabbatical at Microsoft Research in Cambridge, England, he collaborated with the Haskell team on their landmark exploration of *composability* [16]. Unlike locks, which require global reasoning to avoid or recover from deadlock, transactions can easily be combined to create larger atomic operations from smaller atomic pieces. While the benefits can certainly be oversold (and have been—though not by Maurice), composability represents a fundamental breakthrough in the creation of concurrent abstractions. Prudently employed, transactions can offer (most of) the performance of fine-grain locks with (most of) the convenience of coarse-grain locks.

5 Status and Challenges

Today hardware TM appears to have become a permanent addition to processor instruction sets. Run-time systems that use this hardware typically fall back to a global lock in the face of repeated conflict or overflow aborts. For the overflow case, hybrid systems that fall back to software TM may ultimately prove to be more appropriate. STM will also be required for TM programs on legacy hardware. The fastest STM implementations currently slow down critical sections (though not whole applications!) by factors of 3–5, and that number is unlikely to improve. With this present status as background, the future holds a host of open questions.

5.1 Usage Patterns

TM is not yet widely used. Most extant applications are actually written in Haskell, where the semantics are unusually rich but the implementation unusually slow. The most popular languages for research have been C and C++, but progress has been impeded, at least in part, by the lack of high quality benchmarks.

The biggest unknown remains the breadth of TM applicability. Transactions are clearly useful—from both a semantic and a performance perspective—for small operations on concurrent data structures. They are much less likely to be useful—at least from a performance perspective—for very large operations, which may overflow buffer limits in HTM, run slowly in STM, and experience high conflict rates in either case. No one is likely to write a web server that devotes a single large transaction to each incoming page request. Only experience will tell how large transactions can become and still run mostly in parallel.

When transactions *are* too big, and frequently conflict, programmers will need tools to help them identify the offending instructions and restructure their code for better performance. They will also need advances, in both theory and software engineering, to integrate transactions successfully into pre-existing lock-based applications.

5.2 Theory and Semantics

Beyond just atomicity, transactions need some form of condition synchronization, for operations that must wait for preconditions [16, 37]. There also appear to be cases in which a transaction needs some sort of "escape action" [29], to generate effects (or perhaps to observe outside state) in a way that is not fully isolated from action in other threads. In some cases, the application-level logic of a transaction may decide it needs to abort. If the transaction does not restart, but switches to some other code path, then information (the fact of the abort, at least) has "leaked" from code that "did not happen" [16]. Orthogonally, if large transactions prove useful in some applications, it may be desirable to parallelize them internally, and let the sub-threads share speculative state [4]. All these possibilities will require formalization.

A more fundamental question concerns the basic model of synchronization. While it is possible to define the behavior of transactions in terms of locks [27], with an explicit notion of abort and rollback, such an approach seems contrary to the claim that transactions are simpler than locks. An alternative is to make atomicity itself the fundamental concept [8], at which point the question arises: are aborts a part of the language-level semantics? It's appealing to leave them out, at least in the absence of a program-level abort operation, but it's not clear how such an approach would interact with operational semantics or with the definition of a data race.

For run-time–level semantics, it has been conventional to require that every transaction—even one that aborts—see a single, consistent memory state [15]. This requirement, unfortunately, is incompatible with implementations that "sandbox" transactions instead of continually checking for consistency, allowing doomed transactions to execute—at least for a little while—down logically impossible code paths. More flexible semantics might permit such "transactional zombies" while still ensuring forward progress [32].

5.3 Language and System Integration

For anyone building a TM language or system, the theory and semantic issues of the previous section are of course of central importance, but there are other issues as well. What should be the syntax of atomic blocks? Should there be atomic expressions? How should they interact with existing mechanisms like try blocks and exceptions? With locks?

What operations can be performed inside a transaction? Which of the standard library routines are on the list? If routines must be labeled as "transaction safe," does this become a "viral" annotation that propagates throughout a code base? How much of a large application must eschew transaction-unsafe operations?

In a similar vein, given the need to instrument loads and stores inside (but not outside) transactions, which subroutines must be "cloned"? How does the choice interact with separate compilation? How do we cope with the resulting "code bloat"?

Finally, what should be done about repeated aborts? Is fallback to a global lock acceptable, or do we need a hybrid HTM/STM system? Does the implementation need to adapt to observed abort patterns, avoiding fruitless speculation? What factors should influence adaptation? Should

it be static or dynamic? Does it need to incorporate feedback from prior executions? How does it interact with scheduling?

5.4 Building and Using TM Hardware

With the spread of TM hardware, it will be increasingly important to use that hardware well. In addition to tuning and adapting, we may wish to restructure transactions that frequently overflow buffers. We might, for example—by hand or automatically—reduce a transaction's memory footprint by converting a read-only preamble into explicit (nontransactional) speculation [2, 39]. One of my students has recently suggested using advisory locks (acquired using nontransactional loads and stores) to serialize only the portions of transactions that actually conflict [38].

Much will depend on the evolution of hardware TM capabilities. Nontransactional (but immediate) loads and stores are currently available only on IBM Power machines, and there at heavy cost. Lightweight implementations would enable not only partial serialization but also ordered transactions (i.e., speculative parallelization of ordered iteration) and more effective hardware/software hybrids [9, 26]. As noted above, there have been suggestions for "responder-wins" coherence, virtualization, nesting, and condition synchronization. With richer semantics, it may also be desirable to "deconstruct" the hardware interface, so that features are available individually, and can be used for additional purposes [23, 34].

6 Concluding Thoughts

While the discussion above spans much of the history of transactional memory, and mentions many open questions, the coverage has of necessity been spotty, and the choice of citations idiosyncratic. Many, many important topics and papers have been left out. For a much more comprehensive overview of the field, interested readers should consult the book-length treatise of Harris, Larus, and Rajwar [17]. A briefer overview can be found in chapter 9 of my synchronization monograph [31].

My sincere thanks to Hagit Attiya, Shlomi Dolev, Rachid Guerraoui, and Nir Shavit for organizing the celebration of Maurice's 60th birthday, and for giving me the opportunity to participate. My thanks, as well, to Panagiota Fatourou and Jennifer Welch for arranging the subsequent write-ups for BEATCS and SIGACT News. Most of all, my thanks and admiration to Maurice Herlihy for his seminal contributions, not only to transactional memory, but to nonblocking algorithms, topological analysis, and so many other aspects of parallel and distributed computing.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich, editors. Draft Specification of Transaction Language Constructs for C++. Version 1.1, IBM, Intel, and Sun Microsystems, Feb. 2012.

[2] Y. Afek, H. Avni, and N. Shavit. Towards Consistency Oblivious Programming. In *Proc. of the 15th Intl. Conf. on Principles of Distributed Systems*, pages 65-79. Toulouse, France, Dec. 2011.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-Free Made Fast. In *Proc. of the 27th ACM Symp. on Theory of Computing*, 1995.

[4] K. Agrawal, J. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[5] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, Velen, Germany, June–July 1993.

[6] H. W. Cain, B. Frey, D. Williams, M. M. Michael, C. May, and H. Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *Proc. of the 40th Intl. Symp. on Computer Architecture*, Tel Aviv, Israel, June 2013.

[7] C. Click Jr. And now some Hardware Transactional Memory comments. Author's Blog, Azul Systems, Feb. 2009. `blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html`.

[8] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, Cambridge, MA, Sept. 2010. Earlier but expanded version available as TR 959, Dept. of Computer Science, Univ. of Rochester, July 2010.

[9] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar. 2009.

[11] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proc. of the 30th ACM Symp. on Principles of Distributed Computing*, San Jose, CA, June 2011.

[12] C. Ferri, A. Viescas, T. Moreshet, I. Bahar, and M. Herlihy. Energy Implications of Transactional Memory for Embedded Architectures. In *Wkshp. on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods (EPHAM)*, Boston, MA, Apr. 2008. In conjunction with *CGO*.

[13] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[14] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, Aug. 2005.

[15] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[16] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[17] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, second edition, 2010.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[19] M. Herlihy. The Transactional Manifesto: Software Engineering and Non-blocking Synchronization. In *Invited keynote address, SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[20] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[21] M. Herlihy and Y. Lev. tm_db: A Generic Debugging Library for Transactional Programs. In *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Raleigh, NC, Sept. 2009.

[22] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[23] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. Technical Report 1594, Dept. of Computer Sciences, Univ. of Wisconsin–Madison, June 2007.

[24] A. Israeli and L. Rappoport. Disjoint-Access Parallel Implementations of Strong Shared Memory Primitives. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.

[25] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[26] A. Matveev and N. Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures*, Montreal, PQ, Canada, July 2013.

[27] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[28] M. Moir. Transparent Support for Wait-Free Transactions. In *Proc. of the 11th Intl. Wkshp. on Distributed Algorithms*, 1997.

[29] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005.

[31] M. L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, 2013.

[32] M. L. Scott. Transactional Semantics with Zombies. In *Invited keynote address, 6th Wkshp. on the Theory of Transactional Memory*, Paris, France, July 2014.

[33] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99-116, Feb. 1997. Originally presented at the *14th ACM Symp. on Principles of Distributed Computing*, Aug. 1995.

[34] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation Tradeoffs in the Design of Flexible Transactional Memory Support. *Journal of Parallel and Distributed Computing*, 70(10):1068-1084, Oct. 2010.

[35] J. Turek, D. Shasha, and S. Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, Vancouver, BC, Canada, Aug. 1992.

[36] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[37] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[38] L. Xiang and M. L. Scott. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *Proc. of the 27th ACM Symp. on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[39] L. Xiang and M. L. Scott. Software Partitioning of Hardware Transactions. In *Proc. of the 20th PPoPP*, San Francisco, CA, Feb. 2015.

[40] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization. In x. f. H.-P. Computing, editor, *Proc., SC2013: High Performance Computing, Networking, Storage and Analysis*, pages 1-11. Denver, Colorado, Nov. 2013.