

ARRVINDH SHRIRAMAN,
SANDHYA DWARKADAS, AND
MICHAEL L. SCOTT

tapping into parallelism with transactional memory



Arrvindh Shriraman is a graduate student in computer science at the University of Rochester. Arrvindh received his B.E. from the University of Madras, India, and his M.S. from the University of Rochester. His research interests include multiprocessor system design, hardware-software interface, and parallel programming models.

ashriram@cs.rochester.edu



Sandhya Dwarkadas is a professor of computer science and of electrical and computer engineering at the University of Rochester. Her research lies at the interface of hardware and software with a particular focus on concurrency, resulting in numerous publications that cross areas within systems. She is currently an associate editor for IEEE Computer Architecture Letters (and has been an associate editor for IEEE *Transactions on Parallel and Distributed Systems*).

sandhya@cs.rochester.edu



Michael Scott is a professor and past Chair of the Computer Science Department at the University of Rochester. He is an ACM Fellow, a recipient of the Dijkstra Prize in Distributed Computing, and author of the textbook *Programming Language Pragmatics* (3d edition, Morgan Kaufmann, 2009). He was recently Program Chair of TRANSACT '07 and of PPOPP '08.

scott@cs.rochester.edu

MULTICORE SYSTEMS PROMISE TO

deliver increasing performance only if programmers make thread-level parallelism visible in software. Unfortunately, multithreaded programs are difficult to write, largely because of the complexity of synchronization. Transactional memory (TM) aims to hide this complexity by raising the level of abstraction. Several software, hardware, and hybrid implementations of TM have been proposed and evaluated, and hardware support has begun to appear in commercial processors. In this article we provide an overview of TM from a systems perspective, with a focus on implementations that leverage hardware support. We describe the principal hardware alternatives, discuss performance and implementation tradeoffs, and argue that a classic “policy-in-software, mechanism-in-hardware” strategy can combine excellent performance with the flexibility to accommodate different system goals and workload characteristics.

For more than 40 years, Moore’s Law has packed twice as many transistors on a chip every 18 months. Between 1974 and 2004, hardware vendors used those extra transistors to equip their processors with ever-deeper pipelines, multi-way issue, aggressive branch prediction, and out-of-order execution, all of which served to harvest more instruction-level parallelism (ILP). Because the transistors were smaller, vendors were also able to dramatically increase the clock rate. All of that ended about four years ago, when microarchitects ran out of independent things to do while waiting for data from memory, and when the heat generated by faster clocks reached the limits of fan-based cooling. Future performance improvements must now come from multicore processors, which depend on explicit, thread-level parallelism. Four-core chips are common today, and if programmers can figure out how to use them, vendors will deliver hundreds of cores within a decade. The implications for software are profound: Historically only the most talented programmers have been able to write good parallel code; now everyone must do it.

Sadly, parallel programming is *hard*. Historically it has been limited mainly to servers, with “embarrassingly parallel” workloads, and to high-end scientific applications, with enormous data sets and enormous budgets. Even given a good division of labor among threads (something that’s often difficult to find), mainstream applications are plagued by the need to synchronize access to shared state. For this, programmers have traditionally relied on mutual exclusion locks, but these suffer from a host of problems, including the lack of composability (one can’t nest two lock-based operations inside a new critical section without introducing the possibility of deadlock) and the tension between concurrency and clarity: Coarse-grain lock-based algorithms are relatively easy to understand (grab the One Big Lock, do what needs doing, and release it) but they preclude any significant parallel speedup; fine-grained lock-based algorithms allow independent operations to proceed in parallel, but they are notoriously difficult to design, debug, maintain, and understand.

Transactional Memory (TM) aims to simplify synchronization by raising the level of abstraction. As in the database world, the programmer or compiler simply marks a block of code as “atomic”; the underlying system then promises to execute the block in an “all-or-nothing” manner isolated from similar blocks (transactions) in other threads. The implementation is typically based on *speculation*: It guesses that transactions will be independent and executes them in parallel, but watches their memory accesses just in case. If a conflict arises (two concurrent transactions access the same location, and at least one of them tries to write it), the implementation *aborts* one of the contenders, rolls back its execution, and restarts it at a later time. In some cases it may suffice to *delay* one of the contending transactions, but this does not work if, for example, each transaction tries to write something that the other has already read.

TM can be implemented in hardware, in software, or in some combination of the two. Software-only implementations have the advantage of running on legacy machines, but it is widely acknowledged that performance competitive with fine-grain locks will require hardware support. This article aims to describe what the hardware might look like and what its impacts might be on system software. We begin with a bit more detail on the TM programming model and a quick introduction to software TM. We then describe several ways in which brief, small-footprint transactions can be implemented entirely in hardware. Extension to transactions that overflow hardware tables or must survive a context switch are considered next. Finally, we describe our approach to hardware-accelerated software-controlled transactions, in which we carefully separate policy (in software) from mechanism (in hardware).

Transactional Memory in a Nutshell

Although TM systems vary in how they handle various subtle semantic issues, all are based on the notion of *serializability*: Regardless of implementation, transactions *appear* to execute in some global serial order. Writes by transaction A must never become visible to other transactions until A commits, at which time *all* of its writes must be visible. Moreover, writes by other transactions must never become visible to A partway through its own execution, even if A is doomed to abort (for otherwise A might perform some logically impossible operation with externally visible effects). Some TM systems relax the latter requirement by sandboxing A so that any erroneous operations it may perform do no harm to the rest of the program.

The principal motivation for TM is to simplify the parallel programming model. In some cases (e.g., if transactions are used in lieu of coarse-grain locks), it may also lead to performance improvements. An example appears in Fig. 1: If $X \neq Y$, it is likely that the critical sections of Threads 1 and 2 can execute safely in parallel. Because locks are a low-level mechanism, they preclude such execution. TM, however, allows it. If we replace the lock...unlock pairs with `atomic{...}` blocks, the typical TM implementation will execute the two transactions concurrently, aborting and retrying one of the transactions only if they actually conflict.

<p>Thread 1 lock(hash_tab.mutex) var = hash_tab.lookup(X); if(!var) hash_tab.insert(X); unlock(hash_tab.mutex)</p>	<p>Thread 2 lock(hash_tab.mutex) var = hash_tab.lookup(Y); if(!var) hash_tab.insert(Y); unlock(hash_tab.mutex)</p>
--	--

FIGURE 1: LOSS OF PARALLELISM AS A RESULT OF LOCKS [13]

IMPLEMENTATION

Any TM implementation based on speculation must perform at least three tasks: It must (1) detect and resolve conflicts between transactions executing in parallel; (2) keep track of both old and new versions of data that are modified speculatively; and (3) ensure that running transactions never perform erroneous, externally visible actions as a result of an inconsistent view of memory.

Conflict resolution may be *eager* or *lazy*. An eager system detects and resolves conflicts as soon as a pair of transactions have performed (or are about to perform) operations that preclude committing them both. A lazy system delays conflict resolution (and possibly detection as well) until one of the transactions is ready to commit. The losing transaction L may abort immediately or, if it is only about to perform its conflicting operation (and hasn't done so yet), it can wait for the winning transaction W to either abort (in which case L can proceed) or commit (in which case L may be able to occur after W in logical order).

Lazy conflict resolution exposes more concurrency by permitting both transactions in a pair of concurrent R-W conflicting transactions to commit so long as the reader commits (serializes) before the writer. Lazy conflict resolution also helps in ensuring that the conflict winner is likely to commit: If we defer to a transaction that is ready to commit, it will generally do so, and the system will make forward progress. Eager conflict resolution avoids investing effort in a transaction L that is doomed to abort, but it may waste the work performed so far if it aborts L in favor of W and W subsequently fails to commit owing to conflict with some third transaction T. Recent work [17, 22] suggests that eager management is inherently more performance-brittle and livelock-prone than lazy management. The performance of eager systems can be highly dependent on the choice of *contention management* (arbitration) policy used to pick winners and losers, and the right choice can be application-dependent.

Version management typically employs either *direct update*, in which speculative values are written to shared data immediately and are *undone* on abort, or *deferred update*, in which speculative values are written to a log and *re-done* (written to shared data) on commit. Direct update may be somewhat cheaper if—as we hope—transactions commit more often than they abort.

Systems that perform lazy conflict resolution, however, must generally use deferred update, to enable parallel execution of (i.e., speculation by) conflicting writers.

A BRIEF LOOK AT SOFTWARE TM

To track conflicts in the absence of special hardware, a software TM (STM) system must augment a program with instructions that read and write some sort of metadata. If program data are read more often than written (as is often the case), it is generally undesirable for readers to modify metadata, since that tends to introduce performance-sapping cache misses. As a result, readers are invisible to writers in most STM systems and bear full responsibility for detecting conflicts with writers. This task is commonly rolled into the problem of *validation*—ensuring that the data read so far are mutually consistent.

State-of-the-art STM systems perform validation on every nonredundant read. The supporting metadata varies greatly: In some systems, a reader inspects a modification timestamp or writer (owner) id associated with the location it is reading. In other systems, the reader inspects a list of Bloom filters that capture the write sets of recently committed transactions [21]. In the former case, metadata may be located in object headers or in a hash table indexed by virtual address.

Figure 2 shows the overhead of an STM system patterned after TL2 [5], running the STAMP benchmark suite [12]. This overhead is embedded in every thread, cannot be amortized with parallelism, and in fact tends to increase with processor count, owing to contention for metadata access. Here, versioning adds 2%–150% to program run time, while conflict detection and validation add 10%–290%. Static analysis may, in some cases, be able to eliminate significant amounts of redundant or unnecessary validation, logging, and memory fence overhead. Still, it seems reasonable to expect slowdowns on the order of factors of 2–3 in STM-based code, relative to well-tuned locks, reducing the potential for their adoption in practice.

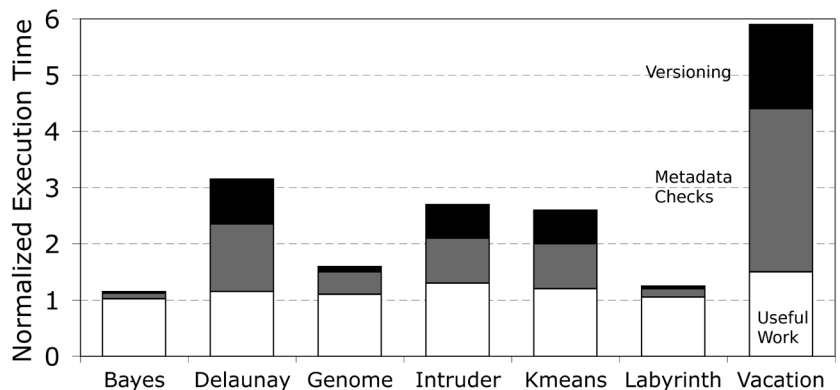


FIGURE 2: EXECUTION TIME BREAKDOWN FOR SINGLE-THREAD RUNS OF A TL2-LIKE STM SYSTEM ON APPLICATIONS FROM STAMP, UNINSTRUMENTED CODE RUN TIME = 1

Hardware for Small Transactions

On modern processors, locks and other synchronization mechanisms tend to be implemented using compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) instructions. Both of these options provide the ability to read a single memory word, compute a new value, and update the word, atomically. Transactional memory was originally conceived as a way to extend this capability to multiple locations.

HERLIHY AND MOSS

The term “transactional memory” was coined by Herlihy and Moss in 1993 [9]. In their proposal (“H&M TM”), a small “transactional cache” holds speculatively accessed locations, including both old and new values of locations that have been written. Conflicts between transactions appear as an attempt to invalidate a speculatively accessed line within the normal coherence protocol and cause the requesting transaction to abort. A transaction commits if it reaches the end of its execution while still in possession of all speculatively accessed locations. A transaction will always abort if it accesses more locations than will fit in the special cache, or if its thread loses the processor as a result of preemption or other interrupts.

OKLAHOMA UPDATE

In modern terminology, H&M TM called for eager conflict resolution. A contemporaneous proposal by Stone et al. [23] envisioned lazy resolution, with a conflict detection and resolution protocol based on two-phase commit. Dubbed the “Oklahoma Update” (after the Rogers and Hammerstein song “All er Nuthin’”), the proposal included a novel solution to the doomed transaction problem: As part of the commit protocol, an Oklahoma Update system would immediately restart any aborted competing transactions by branching back to a previously saved address. By contrast, H&M TM required that a transaction explicitly poll its status (to see if it was doomed) prior to performing any operation that might not be safe in the wake of inconsistent reads.

AMD ASF

Recently, researchers at AMD have proposed a multiword atomic update mechanism as an extension to the x86-64 instruction set [6]. Their Advanced Synchronization Facility (ASF), although not a part of any current processor roadmap, has been specified in considerable detail. As H&M TM does, it uses eager conflict resolution, but with a different contention management strategy: Whereas H&M TM resolves conflicts in favor of the transaction that accessed the conflicting location first, ASF resolves it in favor of the one that accessed it last. This “requester wins” strategy fits more easily into standard invalidation-based cache coherence protocols, but it may be somewhat more prone to livelock. As Oklahoma Update does, ASF includes a provision for immediate abort.

SUN ROCK

Sun’s next-generation UltraSPARC processor, expected to ship in 2009 [7], includes a thread-level speculation (TLS) mechanism that can be used to implement transactional memory. As do H&M TM and ASF, Rock [24] uses eager conflict management; as does ASF, it resolves conflicts in favor of the

requester. As do Oklahoma Update and ASF, it provides immediate abort. In a significant advance over these systems, however, it implements true processor checkpointing: On abort, all processor registers revert to the values they held when the transaction began. Moreover, all memory accesses within the transaction (not just those identified by special load and store instructions) are considered speculative.

STANFORD TCC

Although still limited (in its original form) to small transactions, the Transactional Coherence and Consistency (TCC) proposal of Hammond et al. [8] represented a major break with traditional concepts of memory access and communication. Whereas traditional threads (and processors) interact via individual loads and stores, TCC expresses all interaction in terms of transactions.

Like the multi-location commits of Oklahoma Update, TCC transactions are lazy. Individual writes within the transaction are delayed (buffered) and propagated to the rest of the system in bulk at commit time. Commit-time conflict detection and resolution employ either a central hardware arbiter or a distributed two-phase protocol. As in Rock, doomed transactions suffer an immediate abort and roll back to a processor checkpoint.

DISCUSSION

A common feature of the systems described in this section is the careful leveraging of existing hardware mechanisms. Eager systems (H&M TM, ASF, and Rock) leverage existing coherence protocol actions to detect transaction conflicts. In all five systems, hardware avoids most of the overhead of both conflict detection and versioning. At the same time, transactions in all five can abort simply because they access too much data (overflowing hardware resources) or take too long to execute (suffering a context switch). Also, although the systems differ in both the eagerness of conflict detection and resolution and the choice of winning transaction, in all cases these policy choices are embedded in the hardware; they cannot be changed in response to programmer preference or workload characteristics.

Unbounded Transactions

Small transactions are not sufficient if TM becomes a generic programming construct that can interact with other system modules (e.g., file systems and middleware) that have much more state than the typical critical section. It also seems unreasonable to expect programmers to choose transaction boundaries based on hardware resources. What is needed are low-overhead “unbounded” transactions that hide hardware resource limits and persist across system events (e.g., context switches, system calls, and device interrupts).

To support unbounded transactions, a TM system must virtualize both conflict detection and versioning. In both cases, the obvious strategy is to mimic STM and move transactional state from hardware to a metadata structure in virtual memory. Concrete realizations of this strategy vary in hardware complexity, degree of software intervention, and flexibility of conflict detection and contention management policy. In this section, we focus on implementation tradeoffs, dividing our attention between hardware-centric and hybrid hardware-software schemes. Later, we will turn to hardware-accelerated

schemes that are fundamentally controlled by software, thereby affording policy freedom.

HARDWARE-CENTRIC SYSTEMS

Several systems have extended simple hardware TM (HTM) systems with hardware controllers that iterate through data structures housed in virtual memory. For example, the first unbounded HTM proposal, UTM [1], called for both an in-memory log of transactional writes and an in-memory descriptor for every fixed-size block of program data (to hold read-write permission bits). The descriptors (metadata) constituted an unbounded extension of the access tracking structures found in bounded (small-transaction) HTM. The log constituted an unbounded extension of bounded HTM versioning. Although located in virtual memory, both structures were to be maintained by a hardware controller active on every transactional read and write.

Subsequent unbounded HTM proposals have typically employed a two-level strategy in which a hardware controller implements small transactions in the same way as bounded HTM, but invokes firmware (or low-level software) handlers when space or time resources are exhausted. VTM [14], for example, uses deferred update and buffers speculative writes in the L1 cache as long as they fit. If a speculative line must be evicted owing to limited capacity or associativity, firmware (microcode) moves the line and its metadata to a data structure in virtual memory and maintains both a count of such lines and summary metadata (counting Bloom filters) for all evicted lines. On a context switch, a handler iterates through the entire cache and moves all speculative lines to this data structure. Subsequent accesses (when the count is nonzero) trigger firmware handlers that perform lookup operations of the in-memory data structures and summary metadata in order to detect conflicts (or fetch prior updates within the same transaction). Unfortunately, the cost of lookups is nontrivial.

Bloom-filter-based access-set tracking has also been used in direct-update systems. In LogTM-SE [25], a hardware controller buffers old values in an undo log residing in virtual memory, while speculative values update the original locations (which requires eager conflict resolution in order to avoid atomicity violations). Bloom filters are easy to implement in hardware and can be small enough to virtualize (save and restore) easily. Their drawback is imprecision. Although erroneous indications of conflict are not a correctness issue (since in the worst case, transactions can still execute one at a time), they may lead to lower performance [3].

Hardware-centric systems such as VTM and LogTM-SE hide most of the complexity of virtualization from the system programmer, resulting in a relatively simple run-time system. This simplicity, however, gives rise to semantic rigidity. Special instructions are needed, for example, to “leak” information from aborted transactions (e.g., for performance analysis). Similarly, policies that have first-order effects on performance (e.g., conflict resolution time, contention management policy) are fixed at system design time.

HYBRID APPROACHES

Hardware-centric approaches to unbounded TM demand significant investment from hardware vendors. Hybrid TM systems [4, 10] reduce this investment by adopting a two-level strategy in which the second level is in software. They begin with a “best-effort” implementation of bounded HTM; that is, they attempt to execute transactions in hardware, but the attempt

can simply fail owing to implementation limitations. Software is then expected to pick up the pieces and ensure that all transactions are supported. The key idea is to generate two code sequences for transactions: an STM-compatible version that can run on stock processors and a second version that invokes the best-effort HTM. To ensure high performance, the STM is deployed only when HTM fails. The challenge is to ensure that HTM and STM transactions interoperate correctly. This is achieved by instrumenting the HTM transactions so that every memory operation also checks for concurrent, conflicting STM transactions. If one exists, then the HTM transaction fails, since it lacks the ability to perform conflict resolution with respect to the STM transaction.

Although hybrid systems keep the hardware simple, the instrumentation for interoperability may add significant overhead to HTM transactions. More ambitious hybrid systems [2] may improve performance by implementing conflict detection entirely in hardware (using extra bits associated with main memory), while performing versioning in software. As did hardware-centric unbounded TM, hybrid TM suffers from policy inflexibility inherited from the all-hardware case, and from significant overhead whenever overflow occurs.

Hardware-Accelerated Software-Controlled Transactions

Experimental evidence suggests that although eager conflict management may avoid wasted work, lazy systems may exploit more parallelism, avoid performance pathologies, and eliminate the need for sophisticated (and potentially costly) contention management [11, 17, 22]. Intermediate strategies (e.g., *mixed* conflict management, which resolves write-write conflicts eagerly and read-write conflicts lazily) may also be desirable for certain applications. Unfortunately, the hardware-centric and hybrid TM systems that we have discussed so far embed the choice of both conflict resolution time and contention management policy in silicon.

Hardware-accelerated but software-controlled TM systems [15, 16, 20] strive to leave such policy decisions under software control, while using hardware mechanisms to accelerate both bounded and unbounded transactions. This strategy allows the choice of policy to be tuned to the current workload. It also allows the TM system to reflect system-level concerns such as thread priority. As in the designs covered earlier, existing hardware mechanisms must be carefully leveraged to avoid potential impact on common-case non-transactional code.

The key insight that enables policy flexibility is that information gathering and decision making can be decoupled. In particular, data versioning, access tracking, and conflict detection can be supported as decoupled/separable mechanisms that do not embed policy. Conflict resolution time and contention management policy can then be decided dynamically by the application or TM runtime system.

DECOUPLED VERSIONING

To support lazy conflict resolution, we proposed a deferred-update versioning mechanism we call Programmable Data Isolation (PDI) [15]. PDI allows selective use of processor-private caches as a buffer for speculative writes or for reading/caching the current version of locations being speculatively written remotely. PDI lines are tracked by augmenting the coherence protocol with a pair of additional states. Data associated with speculative writes is not propagated to the rest of the system, allowing multiple transactions to

speculatively read or write the same location. However, coherence actions *are* propagated, allowing remote caches to track the information necessary to return them to a coherent state, without resolving the detected conflict immediately.

To support cache overflow of speculative state, a hardware-based overflow table (akin to a software-managed translation lookaside buffer) is added to the miss path of the L1 cache. Replacement of a speculatively modified cache line results in it being written back to a different (software-specified) region of the process's virtual memory space. A miss in the overflow table results in a trap to software, which can then set up the necessary mapping. In other words, software controls where and how the speculative modifications are maintained while hardware performs the common case (in the critical path) operation of copying data into and out of the cache.

DECOUPLED CONFLICT DETECTION AND RESOLUTION

Access tracking can be performed in hardware by adding extra bits in the private cache to indicate a speculatively modified copy. However, this tracking is bounded by the size of the cache. Alternative forms of tracking for an unbounded amount of metadata include Bloom-filter signatures [3] and ECC bits in memory [2]. Our hardware [16] provides one set of Bloom filters on each processor to represent the read and write sets of the running thread and another to summarize the speculative read and write sets of all currently preempted threads. These signatures and, in some cases, the PDI state bits are checked on coherence protocol transitions in order to detect conflicts among concurrently executing transactions.

To decouple conflict detection from resolution time, we provide *conflict summary tables* (CSTs) that record the occurrence of conflicts without necessarily forcing immediate resolution. More specifically, CSTs indicate the *transactions* that conflict, rather than the locations on which they conflict. This information concisely captures what a TM system needs to know in order to resolve conflicts at some potentially future time. Software can choose when to examine the tables and can use whatever other information it desires (e.g., priorities) to drive its resolution policy.

When a transaction commits, its speculative state is made visible to the rest of the system. To avoid the doomed transaction problem without software polling or sandboxing, conflicting transactions must be alerted and aborted immediately. We enable such aborts with a mechanism known as *alert-on-update* (AOU). This mechanism adds one extra bit, set under software control, to each tag in the cache. When the cache controller detects a remote write of a line whose bit is set, it notifies the local processor, effecting an immediate branch to a previously registered handler. This mechanism can be very lightweight, since the handler invocation is entirely at the user level. By ensuring immediate aborts, AOU avoids the need for validation, thereby eliminating a large fraction of the cost for the metadata checks shown in Figure 2. By choosing what (data, metadata, or transaction status word) and when (at access or commit time) cache lines are tagged as AOU, software can choose between object-based and block-based granularity and among eager, mixed, and lazy conflict resolution.

Using AOU, PDI, signatures, and CSTs, we have developed a series of software-controlled, hardware-accelerated TM systems. RTM-Lite [15, 20] uses AOU alone for validation and conflict detection in a software TM framework (RSTM [18]). RTM-Lite is able to achieve up to a 5x speedup over RSTM on a single thread. RTM [15] uses both AOU and PDI to eliminate validation and

versioning/copy overhead for transactions that fit in the cache. RTM is able to achieve up to an 8.7x speedup over RSTM. At the same time, it achieves only 35%–50% of the single-thread throughput of coarse-grain locks. The remaining overhead is due to software metadata updates and to the indirection needed for compatibility with transactions that fall back to software after overflowing the cache space available to PDI.

FlexTM [16] uses all four mechanisms to achieve flexible policy control without the need for software-managed metadata. The resulting single-thread performance is close to that of coarse-grain locks, demonstrating that eliminating per-access software overheads is essential to realizing the full potential of TM. Scalability is also improved relative to RTM-Lite and RTM. In contrast to other systems supporting lazy conflict resolution (e.g., TCC), FlexTM avoids the need for commit-time conflict detection: A processor's CSTs, which are purely local structures, identify the transactions with which the running transaction conflicts. Software can easily iterate through those transactions, aborting each. Experimental results [15–17, 22] confirm the ability to improve throughput by tailoring conflict resolution time and contention management policy based on application access patterns and overall system goals. The decoupled nature of the various hardware mechanisms also allows them to be used for a variety of non-TM-related tasks, including debugging, security, fast locks, and active messages.

Conclusion

The goal of Transactional Memory is to simplify synchronization in shared-memory parallel programs. Pure software approaches to implementing TM systems suffer from performance limitations. In this article, we presented an overview of emerging hardware support for TM that enhances performance, but with some limitations. The technology is still in its infancy, and widespread adoption will depend on the ability to support a wide spectrum of application behaviors and system requirements. Enforcing a single policy choice at design time precludes this flexibility. Hence, we advocate hardware acceleration of TM systems that leave policy in software. We described a set of mutually independent (decoupled) hardware mechanisms consistent with this approach and presented a series of systems that use this hardware to eliminate successive sources of software TM overhead. Decoupling facilitates incremental development by hardware vendors and leads to mechanisms useful not only for TM, but for various other purposes as well [15, 16, 19].

Several challenges remain. We need developers to integrate TM with existing systems, introduce new language constructs, and develop the necessary tool-chains. We also need to support composability and allow existing libraries to coexist with TM. Finally, we need to resolve a variety of challenging semantic issues, through a combination of formalization and experience with realistic applications. We hope this article will help to foster that process by stimulating broader interest in the promise of transactional memory.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant Nos. CCF-0702505, CNS-0411127, CNS-0615139, CNS-0834451, and CNS-0509270 and by NIH Grant Nos. 5 R21 GM079259-02 and 1 R21 HG004648-01.

REFERENCES

- [1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie, "Unbounded Transactional Memory," *Proc. of the 11th Int'l Symp. on High Performance Computer Architecture*, San Francisco, CA, Feb. 2005.
- [2] L. Baugh, N. Neelakantan, and C. Zilles, "Using Hardware Memory Protection to Build a High-Performance, Strongly Atomic Hybrid Transactional Memory," *Proc. of the 35th Int'l Symp. on Computer Architecture*, Beijing, China, June 2008.
- [3] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," *Proc. of the 33rd Int'l Symp. on Computer Architecture*, Boston, MA, June 2006.
- [4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.
- [5] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," *Proc. of the 20th Int'l Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.
- [6] S. Diestelhorst and M. Hohmuth, "Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory," presented at Workshop on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods (EPHAM), Boston, MA, Apr. 2008 (in conjunction with CGO).
- [7] A. Gonsalves, "Sun Delays Rock Processor by a Year," *EE Times*, 7 Feb. 2008: <http://www.eetimes.com/rss/showArticle.jhtml?articleID=206106243>.
- [8] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proc. of the 31st Int'l Symp. on Computer Architecture*, Munich, Germany, June 2004.
- [9] M. Herlihy and J.E. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. of the 20th Int. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.
- [10] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, New York, March 2006.
- [11] V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Adaptive Software Transactional Memory," *Proc. of the 19th Int'l Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.
- [12] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," *Proc. of the 2007 IEEE Int'l Symp. on Workload Characterization*, Seattle, WA, Sept. 2008.
- [13] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proc. of the 34th Int'l Symp. on Microarchitecture*, Austin, TX, Dec. 2001.
- [14] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," *Proc. of the 32nd Int'l Symp. on Computer Architecture*, Madison, WI, June 2005.
- [15] A. Shriraman, M.F. Spear, H. Hossain, S. Dwarkadas, and M.L. Scott, "An Integrated Hardware-Software Approach to Flexible Transactional Memory," *Proc. of the 34th Int'l Symp. on Computer Architecture*, San Diego, CA, June 2008.

2007. Earlier but expanded version available as TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[16] A. Shriraman, S. Dwarkadas, and M.L. Scott, "Flexible Decoupled Transactional Memory Support," *Proc. of the 25th Int'l Symp. on Computer Architecture*, Beijing, China, June 2008. Earlier version available as TR 925, Dept. of Computer Science, Univ. of Rochester, Nov. 2007.

[17] A. Shriraman and S. Dwarkadas, TR 939, Dept. of Computer Science, Univ. of Rochester, Sept. 2008.

[18] M.F. Spear, V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," *Proc. of the 20th Int'l Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[19] M.F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M.L. Scott, "Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors" (poster paper), *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[20] M.F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M.L. Scott, "Nonblocking Transactions without Indirection Using Alert-on-Update," *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[21] M.F. Spear, M.M. Michael, and C. von Praun, "RingSTM: Scalable Transactions with a Single Atomic Instruction," *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[22] M.F. Spear, L. Dalessandro, V.J. Marathe, and M.L. Scott, "Fair Contention Management for Software Transactional Memory," *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[23] J. M. Stone, H.S. Stone, P. Heidelberger, and J. Turek, "Multiple Reservations and the Oklahoma Update," *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993.

[24] M. Tremblay and S. Chaudhry, "A Third-Generation 65 nm 16-Core 32-Thread Plus 32-Scout-Thread CMT," *Proc. of the Int'l Solid State Circuits Conf.*, San Francisco, CA, Feb. 2008.

[25] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Valos, M.D. Hill, M.M. Swift, and D.A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," *Proc. of the 13th Int'l Symp. on High Performance Computer Architecture*, Phoenix, AZ, Feb. 2007.