DOI: 10.1145/1364782.1364800

Is TM the answer for improving parallel programming?

BY JAMES LARUS AND CHRISTOS KOZYRAKIS

Transactional Memory

AS COMPUTERS EVOLVE, programming changes as well. The past few years mark the start of a historic transition from sequential to parallel computation in the processors used in most personal, server, and mobile computers. This shift marks the end of a remarkable 30-year period in which advances in semiconductor technology and computer architecture improved the performance of sequential processors at an annual rate of 40%–50%. This steady performance increase benefited all software, and this progress was a key factor driving the spread of software throughout modern life.

This remarkable era stopped when practical limits on the power dissipation of a chip ended the continual increases in clock speed and limited instruction-level parallelism diminished the benefit of increasingly complex processor architectures. The era did not stop because Moore's Law[a] ended. Semiconductor technology is still capable of doubling the transistors on a chip every two years. However, this flood of transistors now increases the number of independent processors on a chip, rather than making an individual processor run faster. The resulting computer architecture, named Multicore, consists of several independent processors (cores) on a chip that communicate through shared memory. Today, two-core chips are common and four-core chips are coming to market, and there is every reason to believe that the number of cores will continue to double for a number of generations. On one hand, the good news is that the peak performance of a Multicore computer doubles each time the number of cores doubles. On the other hand, achieving this performance requires a program execute in parallel and scale as the number of processors increase.

Few programs today are written to exploit parallelism effectively. In part, most programmers did not have access to parallel computers, which were limited to domains with large, naturally parallel workloads, such as servers, or huge computations, such as high-performance computing. Because mainstream programming was sequential programming, most existing programming languages, libraries, design patterns, and training do not address the challenges of parallelism *programming*. Obviously, this situation must change before programmers in general will start writing parallel programs for Multicore processors.

A primary challenge is to find better abstractions for expressing parallel computation and for writing parallel programs. Parallel programming encompasses all of the difficulties of sequential programming, but also introduces the hard problem of coordinating interactions among concurrently executing tasks. Today, most parallel

a. The doubling every 18–24 months of the number of transistors fabricable on a chip.

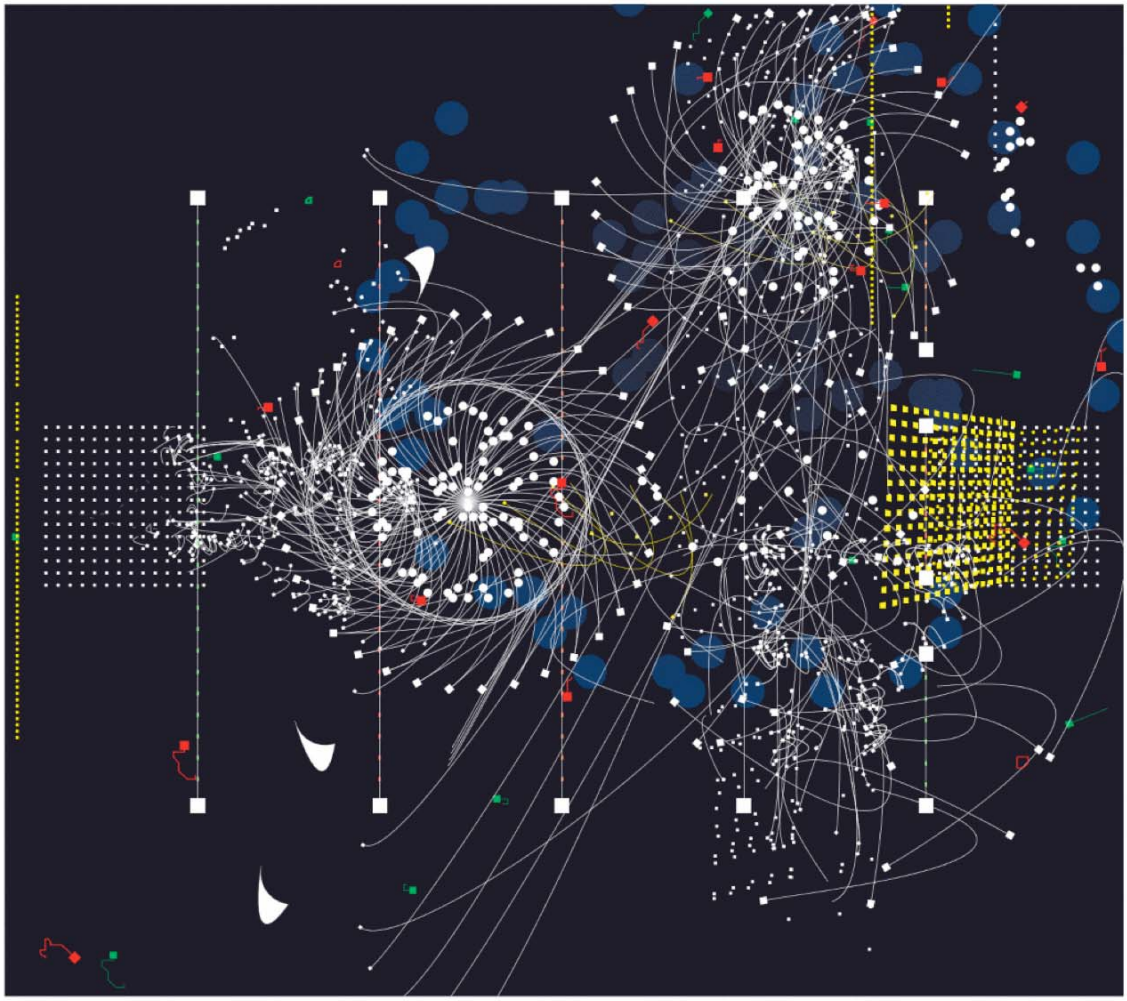


ILLUSTRATION BY STUDIO TONNE / ZEEGENRUSH.COM

programs employ low-level programming constructs that are just a thin veneer over the underlying hardware. These constructs consist of threads, which are an abstract processor, and explicit synchronization (for example, locks, semaphores, and monitors) to coordinate thread execution. Considerable experience has shown that parallel programs written with these constructs are difficult to design, program, debug, maintain, and—to add insult to injury—often do not perform well.

Transactional memory (TM)—proposed by Lomet[19] and first practically implemented by Herlihy and Moss[13]—is a new programming construct that offers a higher-level abstraction for writing parallel programs. In the past few years, it has engendered considerable interest, as transactions have long been used in databases to isolate concurrent activities. TM offers a mechanism that allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, concurrently executing parts of the program. TM offers a promising, but as yet unproven mechanism to improve parallel programming.

What is Transaction Memory?

A transaction is a form of program execution borrowed from the database community.[8] Concurrent queries conflict when they read and write an item in a database, and a conflict can produce an erroneous result that could not arise from a sequential execution of the queries. Transactions ensure that all queries produce the same result as if they executed serially (a property known as "serializability"). Decomposing the semantics of a transaction yields four requirements, usually called the "ACID" properties—atomicity, consistency, isolation, and durability.

TM provides lightweight transactions for threads running in a shared address space. TM ensures the atomicity and isolation of concurrently executing tasks. (In general, TM does not provide consistency or durability.) Atomicity ensures program state changes effected by code executing in a transaction are indivisible from the perspective of other, concurrently executing

transactions. In other words, although the code in a transaction may modify individual variables through a series of assignments, another computation can only observe the program state immediately before or immediately after the transaction executes. *Isolation* ensures that concurrently executing tasks cannot affect the result of a transaction, so a transaction produces the same answer as when no other task was executing. Transactions provide a basis to construct parallel abstractions, which are building blocks that can be combined without knowledge of their internal details, much as procedures and objects provide composable abstractions for sequential code.

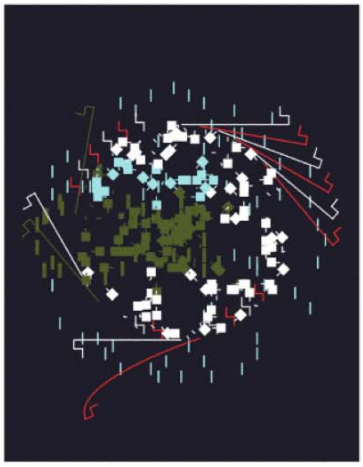
TM Programming Model. A programming model provides a rationale for the design of programming language constructs and guidance on how to construct programs. Like many aspects of TM, its programming model is still the subject of active investigation.

Most TM systems provide simple atomic statements that execute a block of code (and the routines it invokes) as a transaction. An atomic block isolates the code from concurrently executed threads, but a block is not a replacement for general synchronization such as semaphores or condition variables.[2] In particular, atomic blocks by themselves do not provide a means to coordinate code running on parallel threads.

Automatic mutual exclusion (AME), by contrast, turns the transactional model "inside-out" by executing most of a program in transactions.[15] AME supports asynchronous programming, in which a function starts one or more asynchronous computations and later rendezvouses to retrieve their results. This programming model is a common way to deal with unpredictable latency in user-directed and distributed systems. The atomicity provided by transactions ensures that an asynchronous computation, which executes at an unpredictable rate, does not interfere with other, simultaneously active computations.

Advantages of Transactional Memory. Parallel programming poses many difficulties, but one of the most serious challenges in writing correct code is coordinating access to data shared by several threads. Data races, deadlocks, and poor scalability are consequences of trying to ensure mutual exclusion with too little or too much synchronization. TM offers a simpler alternative to mutual exclusion by shifting the burden of correct synchronization from a programmer to the TM system.[9] In theory, a program's author only needs to identify a sequence of operations on shared data that should appear to execute atomically to other, concurrent threads. Through the many mechanisms discussed here, the TM system then ensures this outcome.

Harris and Peyton-Jones[11] argued that, beyond providing a better programming abstraction, transactions



also make synchronization composable, which enables the construction of concurrency programming abstractions. A programming abstraction is composable if it can be correctly combined with other abstractions without needing to understand how the abstractions operate.

Simple locking is not composable. Consider, as an example, a class that implements a collection of bank accounts. The class provides thread-safe `Deposit` and `Withdraw` operations to add and remove money from a bank account. Suppose that we want to compose these operations into a thread-safe `Transfer` operation, which moves money from one account to another. The intermediate state, in which money was debited but not credited, should not be visible to other threads (that is, the transfer should be atomic). Since `Deposit` and `Withdraw` lock an account only during their operation, properly implementing `Transfer` requires understanding and modifying the class's locking discipline by adding a method to either lock all accounts or lock a single account. The latter approach allows non-overlapping transfers to execute concurrently, but introduces the possibility of deadlock if a transfer from account A to B overlaps with a transfer from B to A.

TM allows the operations to be composed directly. The `Deposit` and `Withdraw` operations each execute in a transaction, to protect their manipulations of the underlying data. The `Transfer` operation also executes in a transaction, which subsumes the underlying operations into a single atomic action.

Limitations of Transactional Memory. Transactions by themselves cannot replace all synchronization in a parallel program.[2] Beyond mutual exclusion, synchronization is often used to coordinate independent tasks, for example, by ensuring that one task waits for another to finish or by limiting the number of threads performing a task.

Transactions by themselves provide little assistance in coordinating independent tasks. For example, consider a producer-consumer programming relationship, in which one task writes a value that another task reads. Transactions can ensure the tasks' shared accesses do not interfere. However, this pattern is expensive to implement with transactions, whose goal is to shield a task from interactions with other tasks. If the consumer transaction finds the value is not available, it can only abort and check for the value later. Busy waiting by aborting is inefficient since an aborted transaction rolls back its entire computation. A better solution is for the producer to signal the consumer when the value is ready. However, since a signal is not visible in a transaction, many TM systems provide a guard that prevents a transaction from starting execution until a predicate becomes true.

Haskell TM introduced the `retry` and `orElse` constructs as a way for a transaction to wait until an event occurs and to sequence the execution of two transactions.[11] Executing a `retry` statement causes the surrounding transaction to abort. It does not reexecute until a location it previously

ILLUSTRATION BY STUDIO TONNE / ZEEBENRUSH.COM

read changes value, which avoids the crudest form of busy waiting in which a transaction repeatedly reads an unchanging value and aborts. The `orElse` construct composes two transactions by starting the second one only if the first transaction fails to commit. This pattern—which arises in situations as simple as checking for a value in a cache and recomputing it if necessary—is difficult to express otherwise, since a transaction's failure and reexecution is transparent to other computation.

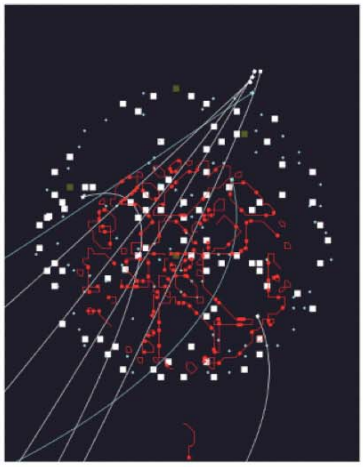
We still do not understand the trade-offs and programming pragmatics of the TM programming model. For example, the semantics of nested transactions is an area of active debate. Suppose that code in a transaction O invokes a library routine, which starts its own transaction I. Should the two transactions interact in any way, and if so, what are the implications for the TM implementation and for programmers building modular software and libraries? Consider when transaction I commits. Should its results be visible only to code in transaction O (closed nesting) or also to other threads (open nesting)? If the latter, what happens if transaction O aborts? Similarly, if transaction I aborts, should it terminate transaction O as well, or should the inner transaction be rolled back and restarted independently?

Finally, the performance of TM is not yet good enough for widespread use. Software TM systems (STM) impose considerable overhead costs on code running in a transaction, which saps the performance advantages of parallel computers. Hardware TM systems (HTM) can lower the overhead, but they are only starting to become commercially available, and most HTM systems fall back on software for large transactions. Better implementation techniques are likely to improve both types of systems and are an area of active research.

Transactional Memory Implementation

TM can be implemented entirely in software (STM) or with specialized hardware support (HTM). Many different implementation techniques have been proposed, and this paper, rather than surveying the literature, focuses on a few key techniques. A more complete overview is available elsewhere.[18]

Most TM systems of both types implement optimistic concurrency control in which a transaction executes under the assumption that it will not conflict with another transaction. If two transactions conflict, because one modifies a location read or modified by the other, the TM system aborts one of the transactions by reversing (rolling back) its side effects. The alternative pessimistic concurrency control requires a transaction to establish exclusive access to a location (for example, by acquiring a lock) before modifying it. This approach also



may abort and roll back a transaction, in case of deadlock.

Software Transactional Memory. The initial paper on STM by Shavit and Touitou[29] showed it was possible to implement lock-free, atomic, multi-location operations entirely in software, but it required a program to declare in advance the memory locations to be accessed by a transaction.

Herlihy et al.'s Dynamic STM (DSTM)[14] was the first STM system that did not require a program to declare the memory locations accessed by a transaction. DSTM is an object-granularity, deferred-update STM system, which means that a transaction modifies a private copy of an object and only makes its changes visible to other transactions when it commits. The transaction exclusively accesses the copy without synchronization. However, another transaction can access the original, underlying object while the first transaction is still running, which causes a logical conflict that the STM system detects and resolves by aborting one of the two transactions.

An STM system can detect a conflict when a transaction first accesses an object (early detection) or when the transaction attempts to commit (late detection). Both approaches yield the same results, but may perform differently and, unfortunately, neither is consistently superior. Early detection prevents a transaction from performing unnecessary computation that a subsequent abort will discard. Late detection can avoid unnecessary aborts, as when the conflicting transaction itself aborts because of a conflict with a third transaction.

Another complication is a conflict between a transaction that only reads an object and another that modifies the object. Since reads are more common than writes, STM systems only clone objects that are modified. To reduce overhead, a transaction tracks the objects it reads and, before it commits, ensures that no other transaction modified them.

DSTM is a library. An object manipulated in a transaction is first registered with the DSTM system, which returns a `TMObject` wrapper for the object (as illustrated in the accompanying figure). Subsequently, the code executing the transaction can open the `TMObject` for read-only or read-write access, which returns a pointer to the original or cloned object, respectively. Either way, the transaction manipulates the object directly, without further synchronization.

A transaction ends when the program attempts to commit the transaction's changes. If the transaction successfully commits, the DSTM system atomically replaces, for all modified objects, the old object in a `Locator` structure with its modified version.

A transaction T can commit successfully if it meets two conditions. The first is that no concurrently executing transaction modified an object read by T. DSTM tracks the objects a transaction opened for reading and validates the entries in this read set when the transaction attempts to commit. An object in the read set is valid if its version is unchanged since transaction

T first opened it. DSTM also validates the read set every time it opens an object, to avoid allowing a transaction to continue executing in an erroneous program state in which some objects changed after the transaction started execution.

The second condition is that transaction T is not modifying an object that another transaction is also modifying. DSTM prevents this type of conflict by only allowing one transaction to open an object for modification. When a write-write conflict occurs, DSTM aborts one of the two conflicting transactions and allows the other to proceed. DSTM rolls the aborted transaction back to its initial state and then allow it to reexecute. The policy used to select which transaction to abort can affect system performance, including liveness, but it should have no effect on the semantics of the STM system.[28]

The performance of DSTM, like other STM systems, depends on the details of the workload. In general, the large overheads of STM systems are more expensive than locking on a small number of processors. However, as the number of processors increases, so does the contention for a lock and the cost of locking. When this occurs and conflicts are rare, STMs have been shown to outperform locks on small benchmarks.

Deferred-Update Systems. Other deferred-update STM systems investigated alternative implementation techniques. Harris and Fraser's WSTM system detects conflicts at word, not object, granularity. This approach can avoid unnecessary conflicts if two transactions access different fields in an object, but it complicates the implementation sufficiently that few STM systems adopted the idea (although, HTM systems generally detect conflicts at word or cache line granularity). WSTM also was the first STM system integrated into a programming language. Harris and Fraser extended Java with an atomic statement that executed its block in a transaction, for example:

```
atomic {
  int x = lst.head;
  lst = lst.tail;
  …
}
```

The construct also provided an optional guard that prevents a transaction from executing until the condition becomes true.

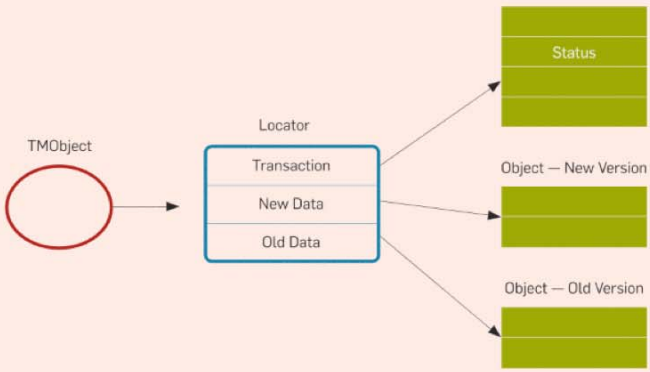
Considerable research has investigated the policies that select which transaction to abort at a conflict.[10, 28] No one policy performs best in all situations, though a policy called "Polka" performed well overall. Under this policy, each transaction tracks the number of objects it has open and uses this count as its priority. A transaction attempting to acquire access to an object immediately aborts a conflicting, lower-priority transaction. If the acquiring transaction's priority is lower, it backs off N times, where N is the difference in priority, with an exponentially increasing interval between the retries. The transaction aborts and reexecutes if it cannot acquire an object within N attempts.

Direct Update Systems. In a direct-update STM system, transactions directly modify an object, rather than a copy.[1, 12, 27] Eliminating the copy potentially is more efficient, since it does not require a clone of each modified object. However, direct-update systems must record the original value of each modified memory location, so the system can restore the location if the transaction aborts. In addition, a direct update STM must prevent a transaction from reading the locations modified by other, uncommitted transactions, thereby reducing the potential for concurrent execution.

Direct update STM systems also require a lock to prevent multiple transactions from updating an object concurrently. Because of the high cost of fair multiple reader-single writer locks, the systems do not lock a read-only object and instead rely on read-set validation to detect concurrent modification of read-only objects. These lock sets incur the same high overhead cost as in deferred-update systems.

The locks used to prevent multiple writes to a location, however, raise the possibility of stalling many transactions when a transaction is suspended or descheduled while holding locks. Deferred-update STM systems typically use non-blocking data structures, which prevented a failed thread from obstructing other threads. Direct-update STM systems provide similar forward progress guarantees to an application by detecting and aborting failed or blocked threads.

A transacted object in the DSTM system.



TMObject is a handle for the object. It points to a Locator, which in turn points to the Transaction that opened the object, the original ("old) version of the object, and the transaction's private ("new") clone of the object.

Hardware Support for Transactional Memory

The programming effort necessary to exploit parallelism is justified if the new code performs better or is more responsive than sequential code. Even though the performance of recent STM systems scales with the number of processors in a Multicore chip, the overhead of the software systems is significant. Even with compiler optimizations, a STM thread may run two

to seven times slower than sequential code. [22, 26]

HTM can improve the performance of STM. While still an active area of research, proposed systems fall into two broad categories: those that accelerate key STM operations and those that implement transactional bookkeeping directly in hardware.

Hardware Acceleration for STM. The primary source of overhead for an STM is the maintenance and validation of read sets. To track a read set, an STM system typically invokes an instrumentation routine at every shared-memory read. The routine registers the object's address and optionally performs early conflict detection by checking the object's version or lock. To validate a transaction, the STM must traverse the read set and ensure each object has no conflicts. This instrumentation can impose a large overhead if the transaction does not perform a large amount of computation per memory access.

The hardware-accelerated STM (HASTM) by Saha et al. was the first system to propose hardware support to reduce the overhead of STM instrumentation.[26] The supplementary hardware allows software to build fast filters that could accelerate the common case of read set maintenance. HASTM provides the STM with two capabilities through per-thread mark bits at the granularity of cache blocks. First, software can check if a mark bit was previously set for a given block of memory and that no other thread wrote to the block since it was marked (conflict detection). Second, software can query if potentially there were writes by other threads to any of the memory blocks that the thread marked (validation).

HASTM proposed implementing mark bits using additional metadata for each block in the per-processor cache of a Multicore chip. The hardware tracks if any marked cache block was invalidated because it was evicted from the cache or written to by another thread. An STM uses mark bits in the following way. The read instrumentation call checks and sets the mark bit for the memory block that contains an object's header. If the mark bit was set, indicating that the transaction previously accessed the object, it is not added to the read set again. To validate the transaction, the STM queries the hardware to determine if any marked cache blocks were invalidated. If not, all objects accessed through instrumentation were private to the thread for the duration of the transaction and no further validation is required. If some marked blocks were invalidated, the STM must rely on software-based validation to check the version numbers or locks for all objects in the read set. This expensive validation step determines if a marked block was evicted because of limited cache capacity or because of true conflicts between concurrent transactions.

The programming effort necessary to exploit parallelism is justified if the new code performs better or is more responsive than sequential code.

HASTM allows transactions to span system events such as interrupts, context switches, and page faults, as the mark bits function only as a filter. If servicing a system event causes the eviction of some marked blocks, a pending transaction can continue its subsequent execution without aborting. The transaction will simply fall back on software validation before it commits. Similarly, HASTM allows a transaction to be suspended and its speculative state inspected by a component such as a garbage collector or a debugger running in another thread.

It is also possible to accelerate STM conflict detection without modifying hardware caches. First-level caches are typically in the critical path of a processor and interact with complex subsystems such as the coherence protocol. Even minor changes to caches can affect the processor's clock frequency and increase design and verification complexity. The signature-accelerated STM (SigTM) proposed by Cao Minh et al. uses hardware signatures to encode pessimistically the read set and write set for software transactions.[22] A hardware Bloom filter outside of the caches computes the signatures.[b] Software instrumentation provides the filters with the addresses of the objects read or written within a transaction. To detect conflicts, hardware in the computer monitors coherence traffic for requests for exclusive accesses to a cache block, which indicates a memory update. The hardware tests if the address in a re-

b. A Bloom filter efficiently represents a superset of the elements in a set and allows fast set membership queries. The use of Bloom filters for dependency detection in thread-level speculation and transactions was originally proposed by Ceze et al.[6]

quest is potentially in a transaction's read or write set by examining the transaction's signatures. If so, the memory reference is a potential conflict and the STM can either abort a transaction or turn to software validation.

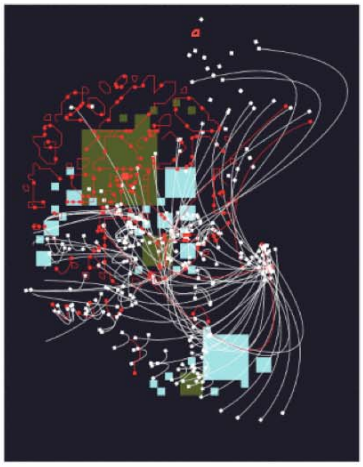
Both HASTM and SigTM accelerate read set tracking and validation for STM systems. Nevertheless, the architectures differ. SigTM encodes read set and write sets whose size exceeds the size of private caches. Capacity and conflict misses do not cause software validation, as with HASTM. On the other hand, SigTM uses probabilistic signatures, which never miss a true conflict, but may produce false conflicts due to address aliasing in a Bloom filter. Moreover, the hardware signatures are relatively compact and easy to manipulate, so they can be saved and restored across context switches and other interruptions. In HASTM, the mark bits may be lost if a processor is used to run other tasks. On the other hand, SigTM signatures track physical addresses and their content must be discarded after the virtual page mapping is modified.

Hardware acceleration for read set management has been shown to improve the performance of lock-based, direct-update, and deferred-update STM systems by a factor of two.[22, 26] Additional improvements are possible with hardware mechanisms that target version management for the objects written by the STM.[31] Nevertheless, since most programs read significantly more objects than they write, these performance improvements are small.

Hardware Transactional Memory. The interest in full hardware implementation of TM (HTM) dates to the initial two papers on TM by Knight[16] and Herlihy and Moss[13] respectively. HTM systems require no software instrumentation of memory references within transaction code. The hardware manages data versions and tracks conflicts transparently as software performs ordinary read and write accesses. Eliminating instrumentation reduces program overhead and eliminates the need to specialize function bodies so they can be called within and outside of a transaction.

HTM systems rely on a computer's cache hierarchy and the cache coherence protocol to implement versioning and conflict detection. Caches observe all reads and writes issued by a processor, can buffer a significant amount of data, and can be searched efficiently because of their associative organization. All HTMs modify the first-level caches, but the approach extends to higher-level caches, both private and shared. To illustrate the organization and operation of HTM systems, we will describe the TCC architecture in some detail and briefly mention the key attributes of alternative designs.

The Transactional Coherence and Consistency (TCC) system is a deferred-



update HTM that performs conflict detection when a transaction attempts to commit.[21] To track the read set and write set, each cache block is annotated with R and W tracking bits, which are set on the first read or write access to the block from within the transaction. Cache blocks in the write set act as a write buffer and do not propagate the memory updates until the transaction commits.

TCC commits transactions using a two-phase protocol. First, the hardware acquires exclusive access to all cache blocks in the write set using coherence messages. If this step is successful, the transaction is considered validated. Next, the hardware instantaneously resets all W bits in the cache, which atomically commits the updates by this transaction. The new versions of the data are now globally accessible by all processors through the normal coherence protocol of a Multicore chip. If validation fails, because another processor is also trying to commit a conflicting transaction, the hardware reverts to a software handler, which may abort the transaction or attempt to commit it under a contention management policy. When a transaction commits or aborts, all tracking bits are simultaneously cleared using a gang reset operation. Absent conflicts, multiple transactions may be committing in parallel.

Conflict detection occurs as other processors receive the coherence messages from the committing transaction. Hardware looks up the received block address in the local caches. If the block is in a cache and has its R or W bit set, there is a read-write or a write-write conflict between the committing and the local transaction. The hardware signals a software handler, which aborts the local transaction and potentially retries it after a backoff period.

Similar hardware techniques can support HTM systems with direct memory updates or early detection of conflicts.[23] For direct updates, the hardware transparently logs the original value in a memory block before its first modification by a transaction. If the transaction aborts, the log is used to undo any memory updates. For early conflict detection, the hardware acquires exclusive access to the cache block on the first write and maintains it until the transaction commits. Under light contention, most HTM designs perform similarly. Under heavier contention, deferred updates and late conflict detection lead to fewer pathological scenarios that can be easily handled with a backoff policy.[3]

The performance of an HTM thread is typically within 2%–10% of the performance of non-transactional code. An HTM system can outperform a lock-based STM by a factor of four and the corresponding hardware-accelerated STM by a factor of two.[22] Nevertheless, HTM systems face several system challenges that are not an issue for STM implementations. The caches used to track the read set, write set, and data versions have finite capacity and may overflow on a long transaction. The transactional state in caches is large and is difficult to save and restore at interrupts, context switching, and paging

ILLUSTRATION BY STUDIO TONNE / ZEEGENRUSH.COM

events. Long transactions may be rare, but they still must execute in a manner that preserves atomicity and isolation. Placing implementation-dependent limits on transaction sizes is unacceptable from a programmer's perspective.

A simple mechanism to handle cache overflows or system events is to ensure the offending transaction executes to completion.[21] When one of these events occurs, the HTM system can update memory directly without tracking the read set, write set, or old data versions. At this point, however, no other transactions can execute, as conflict detection is no longer possible. Moreover, direct memory updates without undo logging preclude the use of explicit abort or retry statements in a transaction.

Rajwar et al. proposed Virtualized TM (VTM), an alternative approach that maintains atomicity and isolation for even if a transaction is interrupted by a cache overflow or a system event.[25] VTM maps the key bookkeeping data structures for transactional execution (read set, write set, write buffer or undo-log) to virtual memory, which is effectively unbounded and is unaffected by system interruptions. The hardware caches hold the working set of these data structures. VTM also suggested the use of hardware signatures to avoid redundant searches through structures in virtual memory.

A final technique to address the limitation of hardware resources is to use a hybrid HTM–STM system.[7, 17] A transaction starts in the HTM mode using hardware mechanisms for conflict detection and data versioning. If HTM resources are exceeded, the transaction is rolled back and restarted in the STM mode with additional instrumentation. This approach requires two versions of each function, but it provides good performance for short transactions. A challenge for hybrid systems is to detect conflict between concurrently HTM and STM transactions.

Hardware/Software Interface for Transactional Memory. Hardware designs are optimized to make the common case fast and reduce the cost of correctly handling rare events. Processor vendors will follow this principle in introducing hardware support for transactional execution. Initial systems are likely to devote modest hardware resources to TM. As more applications use transactions, more aggressive hardware designs, including full-featured HTM systems, may become available.

Regardless of the amount of hardware used for TM, it is important that HTM systems provide functionality that is useful in developing practical programming models and execution environments. A significant amount of HTM research has focused on hardware/software interfaces that can support rich software features. McDonald et al. suggested four interface mechanisms for HTM systems.[20] The first mechanism is a two-phase commit protocol that architecturally separates transaction validation from committing its updates to memory. The second mechanism is transactional handlers that allow software to interface on significant events such as conflict detection, commit, or abort. Shriraman et al. suggest alert-on-update, a similar mechanism that invokes a software handler when HTM hardware detects conflicts or experiences overflows.[31] The third mechanism is support for closed and open-nested transactions. Open nesting allows software to interrupt the currently executing transaction and run some service code (for example, a system call) with independent atomicity and isolation guarantees. Other researchers suggest it is sufficient to suspend HTM transactions to service system calls and I/O operations.[24, 32] Nevertheless, if the service code uses transactions to access shared data in memory, the requirements of transaction pausing are not significantly different from those of open-nested transactions. Finally, both McDonald et al. and Sriraman et al. propose multiple types of load and store instructions what allow compilers to distinguish accesses to thread-private, immutable, or idempotent data from accesses to truly shared data. By providing such mechanisms, HTM systems can support software features ranging from conditional synchronization and limited I/O within transactions[5,32] to high-level concurrency models that avoid transaction aborts on memory conflicts if the application-level semantics are not violated.[4]

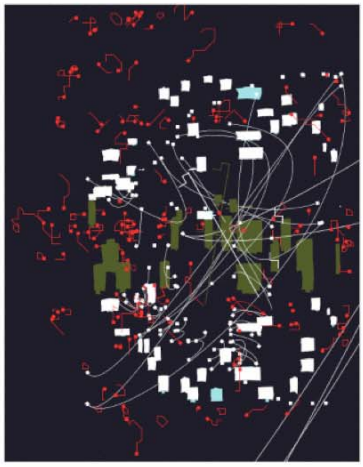


ILLUSTRATION BY STUDIO TONNE / ZEROENRUSH.COM

Open Issues

Beyond the implementation issues discussed here, TM faces a number of challenges that are the subject of active research. One serious difficulty with optimistic TM is that a transaction that executed an I/O operation may roll back at a conflict. I/O in this case consists of any interaction with the world outside of the control of the TM system. If a transaction aborts, its I/O operations should roll back as well, which may be difficult or impossible to accomplish in general. Buffering the data read or written by a transaction permits some rollbacks, but buffering fails in simple situations, such as a transaction that writes a prompt and then waits for user input. A more general approach is to designate a single privileged transaction that runs to completion, by ensuring it triumphs over all conflicting transactions. Only the privileged transaction can perform I/O (but the privilege can be passed between transactions), which unfortunately limits the amount of I/O a program can perform.

Another major issue is strong and weak atomicity. STM systems generally implement weak atomicity, in which non-transactional code is not isolated from code in transactions. HTM systems, on the other hand, implement strong atomicity, which provides a more deterministic programming model in which non-transactional code does not affect the atomicity of a transaction. This difference presents several problems. Beyond the basic question of which model is a better basis for writing software, the semantic differences makes it difficult to develop software that runs on both types of systems.

The least common denominator is the weak model, but erroneous programs will produce divergent results on different systems. An alternative viewpoint is that unsynchronized data accesses between two threads is generally an error, and if only one thread is executing a transaction, then there is insufficient synchronization between the threads. Therefore, the programming language, tools, runtime system, or hardware should prevent or detect unsynchronized sharing between transactional and non-transactional code, and a programmer should fix the defect.

Weakly atomic systems also face difficulties when an object is shared between transactional and non-transactional code.[30] Publication occurs when a thread makes an object visible to other threads (for example, by adding it to a global queue) and privatization occurs when a thread removes an object from the global shared space. Private data should be manipulatable outside of a transaction without synchronization, but an object's transition between public and private must be coordinated with the TM system, lest it attempt to roll back an object's state while another thread assumes it has sole, private access to the data.

Finally, TM must coexist and interoperate with existing programs and libraries. It is not practical to require programmers to start afresh and acquire a brand new set of transactional libraries to enjoy the benefits of TM. Existing sequential code should be able to execute correctly in a transaction, perhaps with a small amount of annotation and recompilation. Existing parallel code that uses locks and other forms of synchronization, must continue to operate properly, even if some threads are executing transactions.

Conclusion

Transactional memory by itself is unlikely to make Multicore computers readily programmable. Many other improvements to programming languages, tools, runtime systems, and computer architecture are also necessary. TM, however, does provide a time-tested model for isolating concurrent computations from each other. This model raises the level of abstraction for reasoning about concurrent tasks and helps avoid many insidious parallel programming errors. However, many aspects of the semantics and implementation of TM are still the subject of active research. If these difficulties can be resolved in a timely fashion, TM will likely become a central pillar of parallel programming. C

References

1. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, 2006). ACM, NY 26–37.
2. Blundell, C., Lewis, E.C., and Martin, M.M.K . Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters 5* (Nov. 2006).
3. Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M., and Wood, D.A. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture* (San Diego, CA, 2007). ACM, NY, 81–91.
4. Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, CA, 2007). ACM, NY. 56–67.
5. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, 2006). ACM, NY, 1–13.
6. Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture* (Boston, MA, 2006). ACM, NY, 227–238.
7. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 2006). ACM, NY. 336–346.
8. Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA. 1992.
9. Grossman, D. The transactional memory / garbage collection analogy. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Montreal, Canada, 2007). ACM, NY. 695–706.
10. Guerraoui, R., Herlihy, M., and Pochon, B. Polymorphic contention management. In *Proceedings of the 19th International Symposium on Distributed Computing* (Krakow, Poland, 2005). Springer Verlag, 303–323.
11. Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, 2005), ACM, NY, 48–60.
12. Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada, 2006). ACM, NY, 14–25.
13. Herlihy, M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. ACM, 1993, 289–300.
14. Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W.N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing* (Boston, MA, 2003). 92–101
15. Isard, M., and Birrell, A. Automatic mutual exclusion. In *Proceeding of the Usenix 11th Workshop on Hot Topics in Operating Systems* (San Diego, CA, 2007).
16. Knight, T.F. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Lisp and Functional Programming Conference*. ACM, NY.
17. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., and Nguyen, A. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, NY, 2006, 209–220.
18. Larus, J.R. and Rajwar, R. *Transactional Memory*. Morgan & Claypool, 2006.
19. Lomet, D.B. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software* (Raleigh, NC, 1977). ACM, NY, 128–137.
20. McDonald, A., Chung, J., Brian, D.C., Minh, C.C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*. ACM, 2006, 53–65.
21. McDonald, A., Chung, J., Chafi, H., Cao Minh, C., Carlstrom, B.D., Hammond, L., Kozyrakis, C., and Olukotun, K. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. (St Louis, MO, 2005). IEEE. 63–74.
22. Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture* (San Diego, CA, 2007) ACM, NY, 69–80.
23. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., and Wood, D.A. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture* (Austin, TX, 2006). IEEE, 254–265.
24. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., and Wood, D.A. Supporting Nested Transactional Memory in LogTM. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 2006). ACM, NY, 359–370.
25. Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture* (Madison, WI, 2005). ACM. NY, 494–505.
26. Saha, B., Adl-Tabatabai, A. R., and Jacobson, Q. Architectural support for software transactional memory. In *Proceedings of the 39th International Symposium on Microarchitecture* (Orlando, FL, 2006). IEEE, 185–196.
27. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., and Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006). ACM, NY, 187–197.
28. Scherer III, W.N., and Scott, M.L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Las Vegas, NV, 2005). ACM Press. 240–248.
29. Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada, 1995). ACM, NY, 204–213.
30. Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., and Saha, B. Enforcing isolation and ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, 2007). ACM, NY, 78–88.
31. Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., and Scott, M.L. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture* (San Diego, CA, 2007). ACM, NY, 104–115.
32. Zilles, C. and Baugh, L. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (Ottawa, Canada, 2006). ACM, NY.

James Larus (larus@microsoft.com) is a research area manager at Microsoft Research, Redmond, WA.

Christos Kozyrakis (christosee.stanford.edu) is an assistant professor of electrical engineering and computer science at Stanford University, Stanford, CA.