UNLOCKING CONCURRENCY

ALI-REZA ADL-TABATABAI, INTEL
CHRISTOS KOZYRAKIS, STANFORD UNIVERSITY
BRATIN SAHA, INTEL

Multicore programming with transactional memory

Multicore architectures are an inflection point in mainstream software development because they force developers to write parallel programs. In a previous article in *Queue*, Herb Sutter and James Larus pointed out, "The concurrency revolution is primarily a software revolution. The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance." [1] In this new multicore world, developers must write explicitly parallel applications that can take advantage of the increasing number of cores that each successive multicore generation will provide.

Parallel programming poses many new challenges to the developer, one of which is synchronizing concurrent access to shared memory by multiple threads. Programmers have traditionally used locks for synchronization, but lock-based synchronization has well-known pitfalls. Simplistic coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races. Furthermore, scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races.

TM (transactional memory) provides a new concurrency-control construct that avoids the pitfalls of locks and significantly eases concurrent programming. It brings to mainstream parallel programming proven concurrency-control concepts used for decades by the database community. Transactional-language constructs are easy to use and can lead to programs that scale. By avoiding deadlocks and automatically allowing fine-grained concurrency, transactional-language constructs enable the programmer to compose scalable applications safely out of thread-safe libraries.

Although TM is still in a research stage, it has increasing momentum pushing it into the mainstream. The recently defined HPCS (high-productivity computing system) languages—Fortress from Sun, X10 from IBM, and Chapel from Cray—all propose new constructs for transactions in lieu of locks. Mainstream developers who are early adopters of parallel programming technologies have paid close attention to TM because of its potential for improving programmer productivity; for example, in his keynote address at the 2006 POPL (Principles of Programming Languages) symposium, Tim Sweeney of Epic Games pointed out that "manual synchronization…is hopelessly intractable" for dealing with concurrency in game-play simulation and claimed that "transactions are the only plausible solution to concurrent mutable state." [2]

Despite its momentum, bringing transactions into the mainstream still faces many challenges. Even with transactions, programmers must overcome parallel programming challenges, such as finding and extracting parallel tasks and mapping these tasks onto a parallel architecture for efficient execution. In this article, we describe how

transactions ease some of the challenges programmers face using locks, and we look at the challenges system designers face implementing transactions in programming languages.

PROGRAMMING WITH TRANSACTIONS

A memory transaction is a sequence of memory operations that either executes completely (*commits*) or has no effect (*aborts*).[3] Transactions are *atomic*, meaning they are an all-or-nothing sequence of operations. If a transaction commits, then all of its memory operations appear to take effect as a unit, as if all the operations happened instantaneously. If a transaction aborts, then none of its stores appear to take effect, as if the transaction never happened.

A transaction runs in *isolation*, meaning it executes as if it's the only operation running on the system and as if all other threads are suspended while it runs. This means that the effects of a memory transaction's stores are not visible outside the transaction until the transaction commits; it also means that there are no other conflicting stores by other transactions while it runs.

Transactions give the illusion of serial execution to the programmer, and they give the illusion that they execute as a single atomic step with respect to other concurrent operations in the system. The programmer can reason serially because no other thread will perform any conflicting operation.

Of course, a TM system doesn't really execute transactions serially; otherwise, it would defeat the purpose of parallel programming. Instead, the system "under the hood" allows multiple transactions to execute concurrently as long as it can still provide atomicity and isolation for each transaction. Later in this article, we cover how an implementation provides atomicity and isolation while still allowing as much concurrency as possible.

The best way to provide the benefits of TM to the programmer is to replace locks with a new language construct such as `atomic { B }` that executes the statements in block `B` as a transaction. A first-class language construct not only provides syntactic convenience for the programmer, but also enables static analyses that provide compile-time safety guarantees and enables compiler optimizations to improve performance, which we touch on later in this article.

Figure 1 illustrates how an atomic statement could be introduced and used in an object-oriented language such as Java. The figure shows two different implementations of a thread-safe map data structure. The code in section A of the figure shows a lock-based map using Java's synchronized statement. The `get()` method simply delegates the call to an underlying non-thread-safe map

Lock-based vs. Transactional Map Data Structure

A

```
class LockBasedMap implements Map
{
   Object mutex;
   Map m;
   LockBasedMap(Map m) {
      this.m = m;
      mutex = new Object();
   }
   public Object get() {
      synchronized (mutex) {
         return m.get();
      }
   }
   // other Map methods
   . . .
}
```

B

```
class AtomicMap implements Map
{

   Map m;
   AtomicMap(Map m) {
      this.m = m;
   }
   public Object get() {
      atomic {
         return m.get();
      }
   }
   // other Map methods
   . . .
}
```

FIG 1

implementation, first wrapping the call in a synchronized statement. The synchronized statement acquires a lock represented by a mutex object held in another field of the synchronized hash map. This same mutex object guards all the other calls to this hash map.

Using locks, the programmer has explicitly forced all threads to execute any call through this synchronized wrapper serially. Only one thread at a time can call any method on this hash map. This is an example of coarse-grained locking. It's easy to write thread-safe programs in this way—you simply guard all calls through an interface with a single lock, forcing threads to execute inside the interface one at a time.

Part B of figure 1 shows the same code, using transactions instead of locks. Rather than using a synchronized statement with an explicit lock object, this code uses a new atomic statement. This atomic statement declares that the call to `get()` should be done atomically, as if it were done in a single execution step with respect to other threads. As with coarse-grained locking, it's easy for the programmer to make an interface thread safe by simply wrapping all the calls through the interface with an atomic statement. Rather than explicitly forcing one thread at a time to execute any call to this hash map, however, the programmer has instead declared to the system that the call should execute atomically. The system now assumes responsibility for guaranteeing atomicity and implements concurrency control under the hood.

Unlike coarse-grained locking, transactions can provide scalability as long as the data-access patterns allow transactions to execute concurrently. The transaction system can provide good scalability in two ways:

- It can allow concurrent read operations to the same variable. In a parallel program, it's safe to allow two or more threads to read the same variable concurrently. Basic mutual exclusion locks don't permit concurrent readers; to allow concurrent readers, the programmer has to use special *reader-writer locks*, increasing the program's complexity.
- It can allow concurrent read and write operations to different variables. In a parallel program, it's safe to allow two or more threads to read and write disjoint variables concurrently. A programmer can explicitly code fine-grained disjoint access concurrency by associating different locks with different fine-grained data elements. This is usually a tedious and difficult task, however, and risks introducing bugs such as deadlocks and data races. Furthermore, as we show in a later example, fine-grained locking does not lend itself to modular software engineering practices: In general, a programmer can't take software modules that use fine-grained locking and compose them together in a manner that safely allows concurrent access to disjoint data.

Transactions can be implemented in such a way that they allow both concurrent read accesses, as well as concurrent accesses to disjoint, fine-grained data elements (e.g., different objects or different array elements). Using transactions, the programmer gets these forms of concurrency without having to code them explicitly in the program.

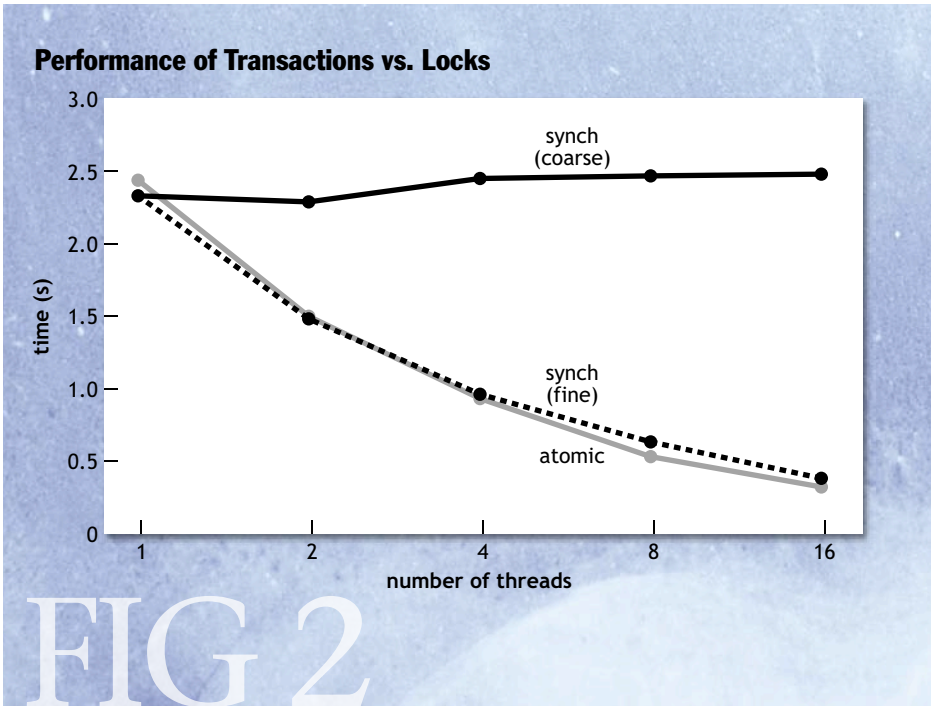

FIG 2

It is possible to write a concurrent hash-map data structure using locks so that you get both concurrent read accesses and concurrent accesses to disjoint data. In fact, the recent Java 5 libraries provide a version of HashMap, called ConcurrentHashMap, that does exactly this. The code for ConcurrentHashMap, however, is significantly longer and more complicated than the version

using coarse-grained locking. The algorithm was designed by threading experts and it went through a comprehensive public review process before it was added to the Java standard. In general, writing highly concurrent lock-based code such as ConcurrentHashMap is very complicated and bug prone and thereby introduces additional complexity to the software development process.

Figure 2 compares the performance of the three different versions of HashMap. It plots the time it takes to complete a fixed set of insert, delete, and update operations on a 16-way SMP (symmetric multiprocessing) machine.[4] As the numbers show, the performance of coarse-grained locking does not improve as the number of processors increases, so coarse-grained locking does not scale. The performance of fine-grained locking and transactional memory, however, improves as the number of processors increases. So for this data structure, transactions give you the same scalability and performance as fine-grained locking but with significantly less programming effort. As these numbers demonstrate, transactions delegate to the runtime system the hard task of allowing as much concurrency as possible.

Although highly concurrent libraries built using fine-grained locking can scale well, a developer doesn't necessarily retain scalability after composing larger applications out of these libraries. As an example, assume the programmer wants to perform a composite operation that moves a value from one concurrent hash map to another, while maintaining the invariant that threads always see a key in either one hash map or the other, but never in neither. Implementing this requires that the programmer resort to coarse-grained locking, thus losing the scalability benefits of a concurrent hash map (figure 3A). To implement a scalable solution to this problem, the programmer must somehow reuse the fine-grained locking code hidden inside the implementation of the concurrent hash map. Even if the programmer had access to this implementation, building a composite move operation out of it risks introducing deadlock and data races, especially in the presence of other composite operations.

Transactions, on the other hand, allow the programmer to compose applications out of libraries safely and still achieve scalability. The programmer can simply wrap a transaction around the composite move operation (figure 3B). The underlying TM system will allow two threads to perform a move operation concurrently as long as the two threads access different hash-table buckets in both underlying hash-map structures. So transactions allow a programmer to take separately authored scalable software components and compose them together into larger components, in a way that still provides as much concurrency as possible but without risking deadlocks because of concurrency control.

By providing a mechanism to roll back side effects, transactions enable a language to provide *failure atomicity*. In lock-based code, programmers must make sure that exception handlers properly restore invariants before releasing locks. This requirement often leads to complicated exception-handling code because the programmer must not only make sure that a critical section catches and handles all exceptions, but also track the state of the data structures used inside the critical section so that the exception handlers can properly restore invariants. In a transaction-based language, the atomic statement can roll back all the side effects of the transaction (automatically restoring invariants) if an uncaught exception propagates out of its block. This significantly reduces the amount of exception-handling code and improves robustness, as uncaught exceptions inside a transaction won't compromise a program's invariants.

Thread-safe Composite Operation

A

```
move(Object key) {
   synchronized(mutex) {
      map2.put(key, map1.remove(key));
   }
}
```

B

```
move(Object key) {
   atomic {
      map2.put(key, map1.remove(key));
   }
}
```

FIG 3

TRANSACTIONS UNDER THE HOOD

Transactional memory transfers the burden of concurrency management from the application programmers to the system designers. Under the hood, a combination of software and hardware must guarantee that concurrent transactions from multiple threads execute atomically and in isolation. The key mechanisms for a TM system are *data versioning* and *conflict detection*.

As transactions execute, the system must simultaneously manage multiple versions of data. A new version, produced by one of the pending transactions, will become globally visible only if the transaction commits. The old version, produced by a previously committed transaction, must be preserved in case the pending transaction aborts. With *eager versioning*, a write access within a transaction immediately writes to memory the new data version. The old version is buffered in an undo log. If the transaction later commits, no further action is necessary to make the new versions globally visible. If the transaction aborts, the old versions must be restored from the undo log, causing some additional delay. To prevent other code from observing the uncommitted new versions (loss of atomicity), eager versioning requires the use of locks or an equivalent hardware mechanism throughout the transaction duration.

Lazy versioning stores all new data versions in a write buffer until the transaction completes. If the transaction commits, the new versions become visible by copying from the write buffer to the actual memory addresses. If the transaction aborts, no further action is needed as the new versions were isolated in the write buffer. In contrast to eager versioning, the lazy approach is subject to loss of atomicity only during the commit process. The challenges with lazy versioning, particularly for software implementations, are the delay introduced on transaction commits and the need to search the write buffer first on transaction reads to access the latest data versions.

A conflict occurs when two or more transactions operate concurrently on the same data with at least one transaction writing a new version. Conflict detection and resolution are essential to guarantee atomic execution. Detection relies on tracking the *read set* and *write set* for each transaction, which, respectively, includes the addresses it read from and wrote to during its execution. We add an address to the read set on the first read to it within the transaction. Similarly, we add an address to the write set on the first write access.

Under *pessimistic conflict detection*, the system checks for conflicts progressively as transactions read and write data. Conflicts are detected early and can be handled either by stalling one of the transactions in place or by aborting one transaction and retrying it later. In general, the performance of pessimistic detection depends on the set of policies used to resolve conflicts, which are typically referred to as *contention management*. A challenging issue is the detection of recurring or circular conflicts between multiple transactions that can block all transactions from committing (lack of forward progress).

The alternative is *optimistic conflict detection* that assumes conflicts are rare and postpones all checks until the end of each transaction. Before committing, a transaction validates that no other transaction is reading the data it wrote or writing the data it read. The drawback to optimistic detection is that conflicts are detected late, past the point a transaction reads or writes the data. Hence, stalling in place is not a viable option for conflict resolution and may waste more work as a result of aborts. On the other hand, optimistic detection guarantees forward progress in all cases by simply giving priority to the committing transaction on a conflict. It also allows for additional concurrency for reads as conflict checks for writes are performed toward the end of each transaction. Optimistic conflict detection does not work with eager versioning.

The *granularity of conflict detection* is also an important design parameter. *Object-level* detection is close to the programmer's reasoning in object-oriented environments. Depending on the size of objects, it may also reduce overhead in terms of space and time needed for conflict detection. Its drawback is that it may lead to false conflicts, when two transactions operate on different fields within a large object such as a multidimensional array. *Word-level* detection eliminates false conflicts but requires more space and time to track and compare read sets and write sets. *Cache-line-level* detection provides a compromise between the frequency of false conflicts and time and space overhead. Unfortunately, cache lines and words are not language-level entities, which makes it difficult for programmers to optimize conflicts in their code, particularly with managed runtime environments that hide data placement from the user.

A final challenge for TM systems is the handling of *nested transactions*. Nesting may occur frequently, given the trend toward library-based programming and the fact that transactions can be composed easily and safely. Early systems automatically flattened nested transactions by subsuming any inner transactions within the outermost. While simple, the flattening approach prohibits explicit transaction aborts, which are useful for failure atomicity on exceptions. The alternative is to support partial

rollback to the beginning of the nested transaction when a conflict or an abort occurs during its execution. It requires that the version management and conflict detection for a nested transaction are independent from that for the outermost transaction. In addition to allowing explicit aborts, such support for nesting provides a powerful mechanism for performance tuning and for controlling the interaction between transactions and runtime or operating system services.[5]

It is unclear which of these options leads to an optimal design. Further experience with prototype implementations and a wide range of applications is needed to quantify the trade-offs among performance, ease of use, and complexity. In some cases, a combination of design options leads to the best performance. For example, some TM systems use optimistic detection for reads and pessimistic detection for writes, while detecting conflicts at the word level for arrays and at the object level for other data types.[6] Nevertheless, any TM system must provide efficient implementations for the key structures (read set, write set, undo log, write buffer) and must facilitate the integration with optimizing compilers, managed runtimes, and existing libraries. The following sections discuss how these challenges are addressed with software and hardware techniques.

SOFTWARE TRANSACTIONAL MEMORY

STM (software transactional memory) implements transactional memory entirely in software so that it runs on stock hardware. An STM implementation uses read and write barriers (that is, inserts instrumentation) for all shared memory reads and writes inside transactional code blocks. The instrumentation is inserted by a compiler and allows the runtime system to maintain the metadata that is required for data versioning and conflict detection.

Figure 4 shows an example of how an atomic construct could be translated by a compiler in an STM implementation. Part A shows an atomic code block written by the programmer, and part B shows the compiler instrumenting the code in the transactional block. We use a simplified control flow to ease the presentation. The `setjmp` function checkpoints the current execution context so that the transaction can be restarted on an abort. The `stmStart` function initializes the runtime data structures. Accesses to the global variables `a` and `b` are mediated through the barrier functions `stmRead` and `stmWrite`. The `stmCommit` function completes the transaction and makes its changes visible to other threads. The transaction gets validated periodically during its execution, and if a conflict is detected, the transaction is aborted. On an abort, the STM library rolls back all the updates performed by the transaction, uses a `longjmp` to restore the context saved at the beginning of the transaction, and reexecutes the transaction.

Since TM accesses need to be instrumented, a compiler needs to generate an extra copy of any function that may be called from inside a transaction. This copy contains instrumented accesses and is invoked when the function is called from within a transaction. The transactional code can be heavily optimized by a compiler—for example, by eliminating barriers to the same address or to immutable variables.[7]

Translating an Atomic Construct for STM

A User Code

```
int foo (int arg)
{
  ...
  atomic
  {
    b = a + 5;
  }
  ...
}
```

B Compiled Code

```
int foo (int arg)
{
  jmpbuf env;
  ...
  do {
    if (setjmp(&env) == 0) {
      stmStart();
        temp = stmRead(&a);
        temp1 = temp + 5;
        stmWrite(&b, temp1);
      stmCommit();
      break;
    }
  } while (1);
  ...
```

FIG 4

The read and write barriers operate on *transaction records,* pointer-size metadata associated with every piece of data that a transaction may access. The runtime system also maintains a *transaction descriptor* for each transaction. The descriptor contains its transaction's state such as the read set, the write set, and the undo log for eager versioning (or the write buffer for lazy versioning). The STM runtime exports an API that allows other components of the language runtime, such as the garbage collector, to inspect and modify the contents of the descriptor, such as the read set, write set, or undo log. The descriptor also contains metadata that allows the runtime system to infer the nesting depth at which data was read or written. This allows the STM to partially roll back a nested transaction.[8]

The write barrier implements different forms of data versioning and conflict detection for writes. For eager versioning (pessimistic writes) the write barrier acquires an exclusive lock on the transaction record corresponding to the updated memory location, remembers the location's old value in the undo log, and updates the memory location in place. For lazy versioning (optimistic writes) the write barrier stores the new value in the write buffer; at commit time, the transaction acquires an exclusive lock on all the required transaction records and copies the values to memory.

The read barrier also operates on transaction records for detecting conflicts and implementing pessimistic or optimistic forms of read concurrency. For pessimistic reads the read barrier simply acquires a read lock on the corresponding transaction record before reading the data item. Optimistic reads are implemented by using data versioning; the transaction record holds the version number for the associated data.[9]

STM implementations detect conflicts in two cases: the read or write barrier finds that a transaction record is locked by some other transaction; or in a system with optimistic read concurrency, the transaction finds, during periodic validation, that the version number for some transaction record in its read set has changed. On a conflict, the STM can use a variety of sophisticated conflict resolution schemes such as causing transactions to back off in a random manner, or aborting and restarting some set of conflicting transactions.

STMs allow transactions to be integrated with the rest of the language environment, such as a garbage collector. They allow transactions to be integrated with tools, such as debuggers. They also allow accurate diagnostics for performance tuning. Finally, STMs avoid baking TM semantics prematurely into hardware.

STM implementations can incur a 40-50 percent overhead compared with lock-based code on a single thread. Moreover, STM implementations incur additional overhead if they have to guarantee isolation between transactional and nontransactional code. Reducing this overhead is an active area of research. Like other forms of TM, STMs don't have a satisfactory way of handling irrevocable actions such as I/O and system calls, nor can they execute arbitrary precompiled binaries within a transaction.

HARDWARE ACCELERATION FOR TM

Transactional memory can also be implemented in hardware, referred to as HTM (hardware transactional memory). An HTM system requires no read or write barriers within the transaction code. The hardware manages data versions and tracks conflicts transparently as the software performs ordinary read and write accesses. Apart from reducing the overhead of instrumentation, HTM systems do not require two versions of the functions used in transactions and work with programs that call uninstrumented library routines.

HTM systems rely on the *cache hierarchy* and the *cache coherence protocol* to implement versioning and conflict detection. Caches observe all reads and writes issued by the processors, can buffer a significant amount of data, and are fast to search because of their associative organization. All HTM systems modify the first-level caches, but the approach extends to lower-level caches, both private and shared.

To track the read set and write set for a transaction, each cache line is annotated with R and W *tracking bits* that are set on the first read or write to the line, respectively. When a transaction commits or aborts, all tracking bits are cleared simultaneously using a *gang* or *flash reset* operation.

Caches implement data versioning by storing the working set for the undo log or the data buffer for the transactions. Before a cache write under eager versioning, we check if this is the first update to the cache line within this transaction (W bit reset). In this case, the cache line and its address are added to the undo log using additional writes to the cache. If the transaction aborts, a hardware or software mechanism must traverse the log and restore the old data versions.[10]

In lazy versioning, a cache line written by the transaction becomes part of the write buffer by setting its W bit.[11] If the transaction aborts, the write buffer is instantaneously flushed by invalidating all cache lines with the W bit set. If the transaction commits, the data in the write buffer becomes instantaneously visible to the rest of the system by resetting the W bits in all cache lines.

To detect conflicts, the caches must communicate their read sets and write sets using the cache coherence protocol implemented in multicore chips. Pessimistic conflict detection uses the same coherence messages exchanged in existing systems.[12] On a read or write access within a transaction, the processor will request shared or exclusive access to the corresponding cache line. The request is transmitted to all other processors that look up their caches for copies of this cache line. A conflict is signaled if a remote cache has a copy of the same line with the R bit set (for an exclusive access request) or the W bit set (for either request type). Optimistic conflict detection operates similarly but delays the requests for exclusive access to cache lines in the write set until the transaction is ready to commit. A single, bulk message is sufficient to communicate all requests.[13]

Even though HTM systems eliminate most sources of overhead for transactional execution, they nevertheless introduce additional challenges. The modifications HTM requires in the cache hierarchy and the coherence protocol are nontrivial. Processor vendors may be reluctant to implement them before transactional programming becomes pervasive. Moreover, the caches used to track the read set, write set, and write buffer for transactions have finite capacity and may *overflow* on a long transaction.

Long transactions may be rare, but they still must be handled in a manner that preserves atomicity and isolation. Placing implementation-dependent limits on transaction sizes is unacceptable from the programmer's perspective. Finally, it is challenging to handle the transaction state in caches for deeply nested transactions or when interrupts, paging, or thread migration occur.[14]

Several proposed mechanisms *virtualize* the finite resources and *simplify* their organization in HTM systems. One approach is to track read sets and write sets using signatures based on Bloom filters. The signatures provide a compact yet inexact (pessimistic) representation of the sets that can be easily saved, restored, or communicated if necessary. The drawback is that the inexact representation leads to additional, false conflicts that may degrade performance. Another approach is to map read sets, write sets, and write buffers to virtual memory and use hardware or firmware mechanisms to move data between caches and memory on cache overflows.

An alternative virtualization technique is to use a *hybrid* HTM-STM implementation. Transactions start using the HTM mode. If hardware resources are exceeded, the transactions are rolled back and restarted in the STM mode.[15] The challenge with hybrid TM is conflict detection between software and hardware transactions. To avoid the need for two versions of the code, the software mode of a hybrid STM system can be provided through the operating system with conflict detection at the granularity of memory pages.[16]

A final implementation approach is to start with an STM system and provide a small set of key mechanisms that targets its main sources of overhead.[17] This approach is called HASTM (hardware-accelerated STM). HASTM introduces two basic hardware primitives: support for detecting the first use of a cache line, and support for detecting possible remote updates to a cache line. The two primitives can significantly reduce the read barrier in general instrumentation overhead and the read-set validation time in the case of optimistic reads.

CONCLUSIONS

Composing scalable parallel applications using locks is difficult and full of pitfalls. Transactional memory avoids many of these pitfalls and allows the programmer to compose applications safely and in a manner that scales. Transactions improve the programmer's productivity by shifting the difficult concurrency-control problems from the application developer to the system designer.

In the past three years, TM has attracted a great deal of research activity, resulting in significant progress.[18] Nevertheless, before transactions can make it into the mainstream as first-class language constructs, there are many open challenges to address.

Developers will want to protect their investments in existing software, so transactions must be added incrementally to existing languages, and tools must be developed that help migrate existing code from locks to transactions. This means transactions must compose with existing concurrency features such as locks and threads. System calls and I/O must be allowed inside transactions, and transactional memory must integrate with other transactional resources in the environment. Debugging and tuning tools for transactional code are also challenges, as transactions still require tuning to achieve scalability and concurrency bugs are still possible using transactions.

Transactions are not a panacea for all parallel program-

ming challenges. Additional technologies are needed to address issues such as task decomposition and mapping. Nevertheless, transactions take a concrete step toward making parallel programming easier. This is a step that will clearly benefit from new software and hardware technologies. Q

AUTHORS' NOTE

For extended coverage on the topic, refer to the slides from the PACT '06 (Parallel Architectures and Compilation Techniques) tutorial, "Transactional Programming in a Multicore Environment," available at http://csl.stanford.edu/~christos/publications/tm_tutorial_pact2006.zip.

REFERENCES

1. Sutter, H., Larus, J. 2005. Software and the concurrency revolution. *ACM Queue* 3 (7).
2. Sweeney, T. 2006. The next mainstream programming languages: A game developer's perspective. Keynote speech, Symposium on Principles of Programming Languages. Charleston, SC (January).
3. Herlihy, M., Moss, E. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. San Diego, CA (May).
4. Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.M., Saha, B., Shpeisman, T. 2006. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation*. Ottawa, Canada (June).
5. A. McDonald, A., Chung, J., Carlstrom, B.D., Cao Minh, C., Chafi, H., Kozyrakis, C., Olukotun, K. 2006. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*. Boston, MA (June).
6. Saha, B., Adl-Tabatabai, A., Hudson, R., Cao Minh, C., Hertzberg, B. 2006. McRT-STM: A high-performance software transactional memory system for a multicore runtime. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. New York, NY (March).
7. See reference 4.
8. See reference 6.
9. See reference 6.
10. Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*. Austin, TX (February).
11. Hammond, L., Carlstrom, B., Wong, V., Chen, M., Kozyrakis, C., Olukotun, K. 2004. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro* 24 (6).
12. See reference 10.
13. See reference 11.
14. Chung, J., Cao Minh, C., McDonald, A., Skare, T., Chafi, H., Carlstrom, B., Kozyrakis, C., Olukotun, K. 2006. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA (October).
15. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA (October).
16. See reference 14.
17. Saha, B., Adl-Tabatabai, A., Jacobson, Q. 2006. Architectural support for software transactional memory. In *Proceedings of the 39th International Symposium on Microarchitecture*. Orlando, FL (December).
18. Transactional Memory Online Bibliography; http://www.cs.wisc.edu/trans-memory/biblio/.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ALI-REZA ADL-TABATABAI is a principal engineer in the Programming Systems Lab at Intel Corporation. He leads a team developing compilers and scalable runtimes for future Intel architectures. His current research concentrates on language features supporting parallel programming for future multicore architectures.

CHRISTOS KOZYRAKIS (http://csl.stanford.edu/~christos) is an assistant professor of electrical engineering and computer science at Stanford University. His research focuses on architectures, compilers, and programming models for parallel computer systems. He is working on transactional memory techniques that can greatly simplify parallel programming for the average developer.

BRATIN SAHA is a senior staff researcher in the Programming Systems Lab at Intel Corporation. He is one of the architects for synchronization and locking in the next-generation IA-32 processors. He is involved in the design and implementation of a highly scalable runtime for multicore processors. As a part of this he has been looking at language features, such as transitional memory, to ease parallel programming.