**Ubiquity Symposium**

# The Multicore Transformation

## The Future of Synchronization on Multicores

***by Maurice Herlihy***

### Editor's Introduction

*Synchronization bugs such as data races and deadlocks make every programmer cringe—traditional locks only provide a partial solution, while high-contention locks can easily degrade performance. Maurice Herlihy proposes replacing locks with transactions. He discusses adapting the well-established concept of data base transactions to multicore systems and shared main memory.*

 

**Ubiquity Symposium**

# The Multicore Transformation

## The Future of Synchronization on Multicores

***by Maurice Herlihy***

An unwelcome side-effect of the advent of multicore processors is that system designers and software engineers can no longer count on increasing clock speed to improve perceived performance. Instead, Moore's law has shifted to mean that each generation of processors will provide more cores, leaving clock speed essentially flat. This transition will not be easy, because conventional synchronization techniques based on locks, monitors, and conditions are unlikely to be effective in such a demanding environment.

### Classical Multiprocessor Synchronization

Traditionally, concurrent programs have typically used some form of "locks" to synchronize access to shared data structures. Locks, however, have well-known limitations. Coarse-grained locks, which protect relatively large amounts of data, generally do not scale: Threads block one another even when they do not really interfere, and the lock itself becomes a source of contention.

Fine-grained locks can mitigate these scalability problems, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, as the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a cache miss, page fault, or context switch, other running threads may be blocked. Locks inhibit concurrency because they must be used conservatively*:* a thread must acquire a lock whenever there is a possibility of synchronization conflict, even if such conflict is actually rare.

A further serious limitation of locks is that they do not compose easily. For example, consider a simple FIFO queue that provides methods to enqueue and dequeue items. To allow concurrent

 

access, the queue has a private lock field, acquired when each method is called, and released when it returns. It is considered good software engineering practice to keep all lock manipulations internal to method calls, so that careless users will be less likely to jeopardize the integrity of the data structure.

This approach is simple and safe for a single object, but now suppose we want to transfer an item from one queue to another. We want this transfer to be atomic: It appears the item is always in one queue or the other, but never in both and not in either. Could we lock both queues, do the transfer, and then unlock them both? Unfortunately, this plan requires exposing both the objects' locks and their locking conventions to clients, greatly complicating the object's interface, and greatly increasing the likelihood of synchronization bugs. There does not appear to be any simple and elegant solution to the problem of composing modular, lock-based data structures.

Locks can sometimes give rise to certain performance anomalies. Priority inversion occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. Convoying occurs when a thread holding lock is descheduled, cause threads that require that lock to queue up, unable to progress. Even after the lock is released, it may take some time to drain the queue, in much the same way that an accident can slow traffic even after the debris has been cleared away. Finally, it is prone to deadlock if threads attempt to lock the same objects in different orders. Today, when concurrent applications are rare, these problems are typically avoided or alleviated by skilled specialists. Soon, when nearly all applications encompass some form of concurrency, less skilled programmers may face the same challenges.

Lock-free data structures can sometimes provide better performance than lock-based data structures, but they may be complicated to design and verify, and they suffer from the some lack of composability.

**Transactional Memory**

For our purposes, a *transaction* is a sequence of steps executed by a single thread. Transactions are "serializable," meaning transactions appear to execute sequentially, in a one-at-a-time order. Transactions are often (but not always) executed "speculatively." A speculative transaction that succeeds is said to "commit," and its effects become visible to other threads, while one that fails is said to "abort" (or cancel), and its effects are discarded.

 

Transactions have long been ubiquitous in the database community. A complete discussion of their role is beyond the scope of this article, but the reader is referred to excellent books by Gray and Reuter [1] and by Lewis et al. [2] for a systematic presentation of what database transactions are, how they work, and where they came from.

The transactions proposed for multicore synchronizations satisfy the same formal serializability and atomicity properties as database-style transactions, but their intended use is very different. Unlike database transactions, which may affect large quantities of long-lived data, multicore transactions are short-lived activities that access a relatively small number of objects in primary memory. The effects of database transactions are persistent, meaning they survive crashes and power failures; committing a transaction involves backing up changes on a disk. Multicore transactions are not persistent, and involve no explicit disk I/O.

Transactions address many of the flaws associated with locking. Transactions compose in a natural way, via nesting. Through the use of careful speculation and contention management, the underlying run-time system can avoid priority inversion, convoying, and deadlocks. Recent studies [3, 4] comparing the use of transactions and locks suggest that software based on transactional synchronization is clearer, developed faster, and has fewer errors than similar software that uses locks. Nevertheless, the question is far from settled: the performance of software transactions remain an issue, and there are still open questions about how best to support system calls, and how transactional and non-transactional thread can interact.

**Programming with Transactions**

Concurrent programming models based on transactions are becoming commonplace. They are present in a variety of modern languages, including Clojure [5], Scala [6] Haskell [7], Java [8], Python [9], and others, either through native language support or indirectly though libraries.

Here is a simple example showing how to program with transactions, expressed using the widely available GCC 4.7 transactional memory language constructs. Figure 1 shows a transactional implementation of a simple circular queue. The `enq()` and `deq {}` methods consist of the standard sequential code (Lines 15-18 and 23-26) each surrounded by a **`transaction_atomic`** block (Lines 14 and 22).

(The **`commit-on-escape`** qualifier specifies if control exits the transaction through an uncaught exception, then the results of the transaction should not be discarded.) To illustrate

 

how easily transactions can be composed, we have also shown a `transfer ()` function, external to the queue class, which atomically transfers an item from one queue to another.

```
1  class  Queue {
2    int   capacity ;        // max size
3    int   size ;            // actual size
4    int   head;             // next to dequeue
5    int   tail ;            // next to enqueue
6    int * items ;           // array of items
7  public :
8    Queue(int  initial ) {
9      items = new int[ initial ];
10     capacity = initial ;
11     size = head = tail = 0;
12   }
13   void enq(int x) {
14    transaction atomic _commit on escape {
15      if ( size == capacity) throw new FullException();
16      items [ tail ] = x;
17      tail = ( tail +1) % capacity;
18      size ++;
19    }
20   }
21   int deq() {
22    transaction atomic _commit on escape {
23      if ( size == 0) throw new EmptyException();
24      int result = items[head ];
25      head = (head+1) % capacity;
26      size --;
27    }
28   }
29 }
30 // external to class
31 void transfer (Queue* q1, Queue* q2) {
32   transaction atomic _ commit on escape {
33     q2->enq(q1->deq());
34   }
35  };
```

**Figure 1: A Transactional Queue.**

As an exercise, the reader might consider how this class would be implemented using locking. The queue would most likely provide a protected lock field, and each method would acquire the lock on entry and release it when it returns, taking care also to release the lock when throwing an exception. As noted, the `external transfer ()` function would be difficult to write

 

without either making the lock field public or introducing another layer of locks. Moreover, care must be taken to avoid deadlocks.

**Implementing Transactional Memory**

Transactional memory implementations can be roughly divided into and hardware transactional memory [10], where functionality is provided directly an instruction set architecture, and software transactional memory [11], where functionality is provided in software.

Hardware transactional memory has progressed from an active research area to a commercial reality. Hardware support for transactions is supported by Intel's Haswell processor [12] and will soon to be provided by IBM's Power architecture [13].

Hardware transactional memory works by adapting standard multiprocessor cache-coherence protocols. Simplifying slightly, when a thread reads or writes a memory location on behalf of a transaction, that value is loaded into the L1 cache, and that cache entry is flagged as transactional. Transactional writes are accumulated in the L1 cache, and are not written back to memory while the transaction is active. If another thread invalidates a transactional entry, that transaction is aborted and restarted. If a transaction finishes without having had any of its entries invalidated, then the transaction commits by marking its transactional entries as valid or as dirty, and allowing the dirty entries to be written back to memory in the usual way.

Hardware transactions, while efficient, are typically limited by the size and associativity of the last-level cache, the length of a scheduling quantum, and restrictions on the kinds of instructions that can be executed within a transaction. For these reasons, existing hardware transactional memory implementations are best effort: The underlying platform does not guarantee that any transaction will complete successfully, although most transactions will do so in practice. For these reasons, programs that use hardware transactions typically require a software backup.

There are many proposed implementations of software transactional memory, and it is impossible to do justice to them here. In a managed language such as Java or C#, it is natural to organize synchronization and recovery around objects. Each transactional object might provide a lock accessible only to the run-time system. The first time a transaction accesses an object; it acquires the lock, and makes a back-up copy of the object. If the transaction commits (completes successfully), it releases its locks and discards its back-up copies, while if it aborts, it

 

restores the original object states from their back-up copies before releasing its locks. Additional mechanisms would be provided to detect or avoid deadlocks.

In less-structured languages such as C or C++, it makes sense to do synchronization and recovery at the granularity of uninterpreted memory blocks instead of objects.

Of course, existing software transactional memory implementations are much more sophisticated, employing a variety of clever conflict detection, recovery, and scheduling mechanisms. The reader is referred to Harris et al. [14] for an excellent survey of both hardware and software transactional memory, including the variety of implementation techniques that have been proposed.

## Conclusion

The author predicts that direct hardware support for transactions will have a pervasive effect across the software stack, affecting how we implement and reason about everything from low-level constructs like mutual exclusion locks, to concurrent data structures such as skip-lists or priority queues, to system-level constructs such as read-copy-update (RCU), all the way to run-time support for high-level language synchronization mechanisms. Although transactions can alleviate many of the well-known shortcomings of legacy synchronization constructs, we believe that such a pervasive change will present new challenges and opportunities very distinct from the familiar issues we face today.

## References

[1] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[2] Lewis, P. M., Bernstein, A. J., and Kifer, M. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2001.

[3] Pankratius, V., and Adl-Tabatabai, A.-R. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA '11).* ACM Press, New York, 2011, 43-52.

[4] Rossbach, C. J., Hofmann, O. S., and Witchel, E. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10).* ACM Press, New York,, 2010, 47-56.

 

[5] Halloway, S. *Programming Clojure*. Pragmatic Programmers. Pragmatic Bookshelf, 2009.

[6] Scala STM Expert Group. ScalaSTM, 2010; http://nbronson.github.com/scala-stm/.

[7] Harris, T., Marlow, S., Jones, S. L. P., and Herlihy, M. Composable memory transactions. *Commun. ACM* 51, 8 (2008), 91-100.

[8] The Deuce STM Group. Deuce STM - Java Software Transactional Memory, 2008; https://sites.google.com/site/deucestm/.

[9] Python Application Kit. TrellisSTM, 2008; http://peak.telecommunity.com/DevCenter/TrellisSTM/.

[10] Herlihy, M., and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM Press, New York, 1993, 289-300.

[11] Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95).* ACM, 1995, 204-213.

[12] Reinders, J. Transactional Synchronization in Haswell. *Intel Developer Zone*, (Feb. 7 2012); http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/.

[13] Cain, H. W., Michael, M. M., Frey, B., May, C., Williams, D., and Le, H. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13).* ACM Press, New York, 2013, 225-236.

[14] Harris, T., Larus, J. R., and Rajwar, R. *Transactional Memory, 2nd edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

**About the Author**

Maurice Herlihy has an A.B. in mathematics from Harvard University, and a Ph.D. in Computer science from M.I.T. He has served on the faculty of Carnegie Mellon University, on the staff of DEC Cambridge Research Lab, and is currently a professor in the Computer Science Department at Brown University. He is the recipient of the 2003 Dijkstra Prize in Distributed Computing, the 2004 Gödel Prize in theoretical computer science, the 2008 ISCA influential paper award, the

 

2012 Edsger W. Dijkstra Prize, and the 2013 Wallace McDowell award. He received a 2012 Fulbright Distinguished Chair in the Natural Sciences and Engineering Lecturing Fellowship, and he is fellow of the ACM and a member of the National Academy of Engineering.

**DOI:** 10.1145/2618405

 