Transactional Memory Should Be an Implementation Technique, Not a Programming Interface

Hans-J. Boehm

HP Laboratories

HPL-2009-45

Keyword(s):

transactional memory, locks

Abstract:

Transactional memory is often advocated as an easier-to-use replacement or locks that avoids any possibility of deadlock. Recently, as more care has been exercised in precisely specifying its semantics, a number of researchers have observed that probably the most attractive semantics for transactional memory systems is based on "single global lock atomicity", i.e. on the semantics of a single global lock. We argue that this should be taken one step further: The synchronization operations seen by the programmer should really just be locks, possibly with some syntactic sugar for easier programming with a single global lock. Use as a deadlock-free lock replacement does not require any rollback primitive, or any other constructs that expose properties of the implementation. And it appears that such extensions add considerable complexity.

External Posting Date: February 21, 2009 [Fulltext]   Approved for External Publication
Internal Posting Date: February 21, 2009 [Fulltext]

Transactional Memory Should Be an Implementation Technique, Not a Programming Interface

This will appear as a position paper in the USENIX HOTPAR '09 workshop.

Hans-J. Boehm
HP Laboratories

Abstract

Transactional memory is often advocated as an easier-to-use replacement for locks that avoids any possibility of deadlock. Recently, as more care has been exercised in precisely specifying its semantics, a number of researchers have observed that probably the most attractive semantics for transactional memory systems is based on "single global lock atomicity", i.e. on the semantics of a single global lock. We argue that this should be taken one step further: The synchronization operations seen by the programmer should really just be locks, possibly with some syntactic sugar for easier programming with a single global lock.

Use as a deadlock-free lock replacement does not require any rollback primitive, or any other constructs that expose properties of the implementation. And it appears that such extensions add considerable complexity. Instead, the implementation should strive to optimize coarse-grain locks, for example by implementing them using transactional techniques.

1 Introduction

Probably the most widely used approach to parallelizing applications in shared memory environments is to write programs as multiple threads, sharing a complete address space. This approach is neither universally optimal, nor likely to disappear from the scene. It is probably unique in allowing relatively painless incremental parallelization of hot sections of existing sequential code.

Existing multithreaded applications typically use locks to ensure mutual exclusion during access to concurrent data structures. There is a fairly simple way to view threads and shared variables in this environment [2, 8, 5]:

1. We define a sequentially consistent execution in the usual way [20], essentially as an interleaving of steps from the threads. Lock acquisitions and releases of a given lock must start with an acquisition, and then alternate in the interleaved sequence, the subsequence of steps corresponding to each thread must reflect the sequential semantics of that thread, and each shared variable reference must "see" the last preceding assignment in this interleaving. For example, if counter variable `c` were initially zero, two threads $T1$ and $T2$ each acquired lock `l`, incremented counter `c`, and then released the lock, a sequentially consistent execution might be:

   ```
   T1: lock(l);
   T1: tmp1 = c;  (sees 0)
   T1: c = tmp1 + 1; (stores 1)
   T1: unlock(l);
   T2: lock(l);
   T2: tmp2 = c;  (sees 1)
   T2: c = tmp2 + 1; (stores 2)
   T2: unlock(l);
   ```

   If instead $T2$ did not acquire the lock, but simply incremented *c*, a possible sequentially consistent execution would be

   ```
   T1: lock(l);
   T1: tmp1 = c; (sees 0)
   T2: tmp2 = c; (sees 0)
   T1: c = tmp1 + 1; (stores 1)
   T2: c = tmp2 + 1; (stores 1)
   T1: unlock(l);
   ```

2. We define two steps performing memory operations to *conflict* if they access the same memory location and one of them is a write.

3. A program allows a *data race* (on a particular input) if there is a sequentially consistent execution (i.e. interleaving of steps) in which two conflicting steps performing memory operations on ordinary

data (i.e. not synchronization objects) corresponding to different threads appear next to each other, i.e. could occur in parallel. The first version of our parallel counter increment program above does not allow a data race, since accesses to `c` in the interleaving must be separated by locking operations, while the second one (without the lock acquisition in *T2*) does. In fact the sequentially consistent execution we gave above contains two conflicting operations from different threads as the third and forth step.

4. We then require that a program that does not allow a data race exhibits the behavior of one of its sequentially consistent executions.

5. For the purposes of this paper, we consider programs that allow data races to be erroneous. This is consistent with Ada [30], Posix [18] threads, and the current draft of C++0x[1], the next C++ standard [19, 8]. Some languages, notably Java, attempt to provide stronger properties [23], but these attempts have been only partially successful [3, 28].

We repeat this presentation here to emphasize that it is surprisingly simple, especially in light of the confusion that has historically surrounded the basic programming rules for threads and locks. [6, 7, 4, 8] All mainstream language specifications now appear to be converging on this approach[2] [30, 23, 8, 18].

In spite of its relative simplicity, it appears to be widely believed that this model is already pushing the complexity envelope for the kind of mainstream parallel programming we will need with ubiquitous multicore processors, and our emphasis should be on simplifying it further, rather than complicating it.

1.1 Atomic sections and transactional memory

The use of locks for mutual exclusion unfortunately has a well-known down side. If two threads acquire the same two locks in opposite order, they can cause a deadlock, with each thread holding exactly one of the locks. Although, at first glance, this may appear easily avoidable by insisting on a fixed lock ordering, this is hard to do in practice.

Consider a piece of C++ code that performs a pointer assignment `x = y` while holding a lock l. It is increasingly common to use reference counted shared pointers, such as `shared_ptr` [10] from the Boost library or the committee draft for C++0x, the next C++ standard [19]. In that case, the assignment operator invokes user-defined destructor operations on the old value of `x`. These in turn may recursively invoke further destructors for objects indirectly referenced from the old value of `x`. The code containing the original pointer assignment generally has no knowledge of the types of these objects or the locks their destructors may acquire. Yet it must somehow ensure that *all* of these locks follow l in the lock ordering.

A more elaborate Java-based illustration of the problem is the observer pattern example from [21].[3]

It is well-known that this problem can be avoided by replacing lock-based mutual exclusion with *atomic sections* that ensure that code inside atomic sections cannot interfere with atomic section code in other threads. Unlike lock-based synchronization, the programmer does not specify a specific lock object to use; mutual exclusion is provided across *all* atomic sections in the program. In Java, one might write "`atomic` $\{ S \}$" instead of "`synchronized (` *lock_obj* `)` $\{ S \}$".

Constructs along these lines can be implemented with widely different techniques. They have been studied primarily in the context of *transactional memory* systems (cf. [17, 29, 16, 1]) that allow multiple threads to proceed concurrently into atomic sections, possibly rolling back one or more of them when conflicting accesses are discovered.

But atomic sections may also be implemented in other ways. For example, they could be implemented in Java simply by declaring a globally accessible object `the_lock`, shared by all code in a program, and then replacing all atomic sections "`atomic` $\{ S \}$" by "`synchronized (the_lock)` $\{ S \}$". This is of course not likely to result in a scalable implementation, and there has been significant amount of research on either semi-automatically [24] or fully automatically [13, 31, 9] assigning distinct finer-grained locks to atomic sections. Although it is not always easy to reproduce the scalability of transactional memory techniques with these approaches, they do reduce the contention-free cost back down to that of locks.

2 Remaining Problems with Atomic Sections

Although atomic sections address an important problem with locks, they so far have enjoyed only minimal use in real applications. The fundamental reasons for this are:

1. We do not fully understand their semantics. In spite of the long history of transactional memory, many of the fundamental issues were not really exposed until recently (cf. [14]). Most of the issues that previously arose for locks also apply in a transactional setting. Unlike with locks, it is unclear whether we are approaching a consensus on its resolution. A number of very recent papers [25, 26, 15, 12], often postdating our original submission, take a position

similar to ours. But, for example, [22] expresses a position that is almost opposite.

2. Depending on the resolution of the semantics issues, it may be difficult to reuse existing code in a transactional setting.
3. We do not yet understand how to consistently obtain performance similar to, or better than, that obtained with locks, particularly in low contention situations such as SPECjbb [32], which we all hope are common in real applications. Performance is again influenced substantially by decisions about the underlying semantics.

Here we focus on the first issue, though we are clearly also motivated by the other two.

3 Preserving Simple Semantics for Atomic Sections

Since our goal is to introduce atomic sections as a simplification of locks, it makes little sense to significantly complicate the semantic model in the process.

The "sequential consistency for data-race-free programs" approach appears to be fundamentally as simple as possible. Sequential consistency has long been viewed as the most natural model of parallel program execution [20]. Any attempt to extend it to programs containing data races (as [20] originally proposed) imposes expensive constraints on both hardware and compiler, since they must ensure that memory references to possibly-non-local locations must be performed in order, requiring memory fence instructions, and inhibiting common compiler optimizations, such as common subexpression elimination. [5, 8]. These restrictions are rarely useful to portable code, in part because the resulting language semantics depends on hardware access granularity.

If we view an atomic section as the acquisition of a single global lock, i.e. as having "single global lock atomicity" semantics (cf. [32, 25]), then it is trivial to extend this model to atomic sections. Effectively, we apply the same rules as above, with the following additional restriction on the interleaving of steps in a sequentially consistent execution: An atomic section entry may not appear in the interleaving until all prior atomic section entries by other threads also have matching exits in the interleaving. This is essentially the same rule we would use for a single global reentrant lock.

Note that this addresses all of the other semantic issues normally discussed in connection with transactional memory semantics. For example it addresses all of the safety properties cataloged in [32], without requiring them to be specifically addressed, or even mentioning them. It provides no guarantees for those properties that make assertions about programs with data races ("repeatable reads", "intermediate updates", "intermediate reads"). It does provide the other guarantees, such as "publication safety", and "privatization safety", since we promise sequential consistency in those cases.

This approach also has the advantage (also enjoyed by and pointed out by [32]) that the interaction between atomic sections and conventional synchronization mechanisms such as locks, condition variables, Java `volatiles` and C++0x `atomic<T>` variables are perfectly well defined, and combinations are useful.

4 What about explicit rollback?

Most transactional memory systems (e.g. [16] or the Intel STM [1, 26]) provide additional facilities that leverage the ability of traditional transactional memory implementations to roll back partially executed transactions. Typically these take the form of a `retry` and/or explicit `abort` statement. We argue here that although they are undoubtedly useful at times, they do not appear to belong in an interface designed to provide basic mutual exclusion.

We basically see two kinds of applications for these facilities, which we address in turn:

4.1 Providing failure atomicity

It appears to be fairly common to use explicit transaction aborts to recover from failure conditions detected during the transaction. Rolling back the transaction is an easy way to, for example, restore data structure invariants. Effectively the transaction rollback facility provides a convenient mechanism for "exception safety".

Although it is useful to roll back a computation in the event of failure, this use appears completely unrelated to the use of transactions for isolation between threads, i.e. as a lock replacement. In particular, it is clearly equally useful, and implementable at less cost, in single-threaded environments, or when operating on data structures private to a thread. Roll-back for failure atomicity also introduces a number of other tricky issues:

1. Rollback for failure atomicity does not interact well with with "irrevocable" actions, as used, for example, in [26]. A computation that, even under very exceptional conditions, performs an irreversible I/O action or thread communication, such as the example in the next section, cannot be reliably rolled back past that point. Even when a computation never performs an irreversible action, it is not clear how to make that information available to the programmer and/or compiler when cross-module calls are involved.

2. It is unclear how to reliably report the cause of a failure when a computation is rolled back, since the state leading to the failure, including objects that are likely to be of use describing it, will be rolled back (cf. [16, 11]).

Although we agree that it would be nice to provide failure atomicity by allowing rollback initiated with an explicit abort action, it appears sufficiently distinct from the concurrency control aspects of transactional memory, and introduces enough added difficulties, that we do not consider it further here.[4]

4.2 Thread Communication

Some transactional memory systems (e.g. [16]) provide a `retry` construct that allows transactions to be used for thread communications. When `retry` is executed, the transaction is aborted, and re-executed once one of the previously read values changes. This effectively allows a thread to wait until a condition is satisfied as a result of actions by another thread.

This facility effectively provides a replacement for the condition variables normally used with locks. Unfortunately, it does so in a way that appears to destroy the composition properties that are often used to justify transactional memory to start with.

For example, a call to a function `f()` may need to perform such communication when it needs to log an error. In a lock-based program, it is usually acceptable to call `f()` while an unrelated lock is held. In a system based on atomic sections, it is unclear how this can work. If we were to write

```
atomic {
    f();
}

f() {
    do_something();
    if (error) {
        request_logging_by_other_thread();
        wait_for_other_thread();
    }
}
```

the other thread should not see the request until the atomic section completes. But the atomic section can't complete until the other thread sees and processes the request. The communication here fundamentally violates the notion that atomic sections should appear indivisible, and no other thread should see an intermediate state.

Interestingly, if we use the single global lock interpretation of atomic sections, but implement `request_logging_by_other_thread()` using conventional locks and condition variables, with a different lock, such code continues to work correctly, as it does in a purely lock-based environment.

Thus it appears to us that `retry` is of limited utility: It is essentially only useful for code that "knows" it will never be executed inside a transaction. This is at odds with normal coding practices, in which most code is part of data abstractions that can be used for either thread-private data, or shared data for which the client ensures isolation/mutual exclusion.

We expect that more general purpose code requiring explicit thread communication will continue to be written with locks and condition variables. We see the primary benefit of atomic sections as simplifying the large amounts of code, often written by less experienced programmers, that do not directly perform this kind of inter-thread communication, though they may rely on a library to do so.

Thus we do not believe that it makes sense to complicate an atomic section interface with a facility like `retry` that is really intended to solve problems addressed by more expert programmers, but does not give them the tools to produce a complete solution.

5 What about progress guarantees?

As is pointed out in [22], the atomic section semantics we advocate differ from what transactional memory experts might expect, in that they may block progress of a thread where a transactional implementation of atomic sections would not. Consider the following example, closely related to one in [22]:

```
Thread 1:
  atomic {
    while(1) {}
  }

Thread 2:
  atomic {}
  print "Hello";
```

In our semantics, this is allowed to never print "Hello", since thread 1 may start running first, acquire the global lock, and thus prevent thread 2 from ever completing its atomic section.

Although this may be surprising, it actually reflects an intentional lack of progress guarantees in the existing Java memory model, which was motivated by completely different considerations. A non-preemptive uniprocessor scheduler would also fail to make progress here, and we concluded during the Java memory model discussions that such implementations should be allowed.

There is no apparent way to modify the preceding example to ensure that "Hello" can only be printed in a transactional, and not a lock-based implementation, unless we introduce either nested synchronization or have some way to force a fair execution. In the absence of those, it appears impossible to programmatically distinguish the two implementation styles. In our example, thread 2 cannot test whether the infinite loop in thread 1 has started without accessing a variable set inside the atomic section containing the loop. But if we introduced such a variable, this access would introduce a data race, nested synchronization, or a conflicting atomic section that could be delayed indefinitely even with a transactional implementation.

In the presence of nested synchronization, our semantics do assign nontrivial meaning to empty atomic sections. Assume v is a synchronization variable, e.g. a Java volatile variable, and x is an ordinary variable, and both are initially zero in:

```
Thread 1:
  atomic {
    v = 1;
    x = 1;
  }

Thread 2:
  while (!v) {}
  atomic {}
  x = 2;
```

Under our semantics, this does not contain a data race, since x = 2 cannot be executed until after the atomic section in thread 1 completes. And indeed a lock-based implementation will guard against the race. However, a transactional implementation that treats an empty atomic section as a no-op would not prevent the race.

However, as we illustrated earlier, even transactional implementations generally have to revert to locking in the presence of nested thread communication, such as here. This is done both by the Intel C++ STM, and in [32]. We expect that this mechanism can be used to ensure lock-like semantics in such cases.

(Note that thread creation is viewed as synchronization for this purpose. If a thread is created inside an atomic section with single global lock semantics, an empty atomic section in the child is guaranteed to wait for the parent to leave its atomic section. This appears to be a moderately common idiom with locks, which assigns nontrivial meaning to empty critical sections.)

Thus, in all cases, the single global lock semantics seem fundamentally compatible with both lock-based and transactional memory implementations.

However, all of this clearly needs further investigation, and precise statements and proofs.

6 Where Does That Leave Transactional Memory?

Atomic sections, as we have defined them here, add very little to the interface seen by programmers. The same code could have been written with an explicit global reentrant lock. They effectively provide a convention and a bit of "syntactic sugar".

However, by encouraging a programming style that relies on such a convention, essentially the ultimate coarse-grain locking convention, we are clearly relying on clever implementations to recover reasonable scalability. It's not yet clear to us what the best implementations will look like, but certainly transactional implementations, presumably with a fall-back to locks to handle IO and nested synchronization, are likely to be part of the solution. To our knowledge, this is consistent with the approach taken by [27] or Azul Systems' use of transactional memory as an optimization for Java locks.

7 Acknowledgments

In addition to comments on earlier drafts from the anonymous reviewers and Terence Kelly, this paper benefitted from discussions with many others, including Ali-Reza Adl-Tabatabai Dhruva Chakrabarti, Pramod Joisha, Tatiana Shpeisman, and Adam Welc, most of whom are believed to disagree with parts of it.

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 *SIGPLAN Conference on Programming Language Design and Implementation* (2006), pp. 26–37.

[2] Adve, S. V., and Hill, M. D. Weak ordering—A new definition. In *Proc. 17th Intl. Symp. Computer Architecture* (1990), pp. 2–14.

[3] Aspinall, D., and Sevcik, J. Java memory model examples: Good, bad, and ugly. VAMP07 Proceedings http://www.cs.ru.nl/~chaack/VAMP07/, 2007.

[4] Boehm, H., and Maclaren, N. Should volatile acquire atomicity and thread visibility semantics? C++ standards committee paper WG21/N2016 = J16/06-0086, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2016.html, April 2006.

[5] Boehm, H.-J. Threads basics. http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/threadsintro.html.

[6] Boehm, H.-J. Threads cannot be implemented as a library. In *Proc. Conf. on Programming Language Design and Implementation* (2005).

[7] Boehm, H.-J. Reordering constraints for pthread-style locks. In *Proc. 12th Symp. Principles and Practice of Parallel Programming* (2007), pp. 173–182.

[8] Boehm, H.-J., and Adve, S. Foundations of the c++ concurrency memory model. In *Proc. Conf. on Programming Language Design and Implementation* (2008), pp. 68–78.

[9] Cherem, S., Chilimbi, T., and Gulwami, S. Infering locks for atomic sections. In *PLDI 08* (2008), pp. 304–315.

[10] Colvin, G., Dawes, B., Adler, D., and Dimov, P. The Boost shared_ptr class template. http://www.boost.org/libs/smart_ptr/shared_ptr.htm, August 2005.

[11] Crowl, L., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Integrating transactional memory into c++. TRANSACT, http://www.cs.rochester.edu/meetings/TRANSACT07/papers/crowl.pdf, 2007.

[12] Delessandro, L., and Scott, M. L. Strong isolation is a weak idea. TRANSACT, http://transact09.cs.washington.edu/33_paper.pdf, 2009.

[13] Emmi, M., Fischer, J. S., Jhala, R., and Majumdar, R. Lock Allocation. In *Symposium on Principles of Programming Languages (POPL)* (2007), pp. 291–296.

[14] Grossman, D., Pugh, B., and Manson, J. What do High-Level Memory Models Mean for Transactions? In *Workshop on Memory System Performance and Correctness (MSPC)* (2006), pp. 63–69.

[15] Harris, T. Language constructs for transactional memory. Slides from POPL 09 invited talk http://research.microsoft.com/en-us/um/people/tharris/misc/2009-01jan-popl.pdf, January 2009.

[16] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Principles and Practice of Parallel Programming* (2005), pp. 48–60.

[17] Herlihy, M., and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA 93* (1993), pp. 289–300.

[18] IEEE, and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[19] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, programming language - C++ (committee draft). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf, 2008.

[20] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (1979), 690–691.

[21] Lee, E. A. The problem with threads. Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[22] Luchangco, V. Against lock-based semantics for transactional memory. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures* (2008), pp. 98–100.

[23] Manson, J., Pugh, W., and Adve, S. The Java memory model. In *Proc. Symp. on Principles of Programming Languages* (2005).

[24] McCloskey, B., Zhou, F., Gay, D., and Brewer, E. Autolocker: Synchronization Inference for Atomic Sections. In *POPL 06* (2006), pp. 346–358.

[25] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R., Saha, B., and Welc, A. Single global lock semantics in a weakly atomic stm. In *TRANSACT* (2008).

[26] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. Design and Implementation of Transactional Constructs for C/C++. In *OOPSLA* (2008), pp. 195–212.

[27] Rajwar, R., and Goodman, J. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Microarchitecture, IEEE/ACM International Symposium on* (Los Alamitos, CA, USA, 2001), IEEE Computer Society, pp. 294–305.

[28] Sevcik, J., and Aspinall, D. On validity of program transformations in the java memory model. In *ECOOP 2008* (2008), pp. 27–51.

[29] Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[30] United States Department of Defense. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*, 1983. Springer.

[31] Zhang, Y., Sreedhar, V. C., Zhu, W., Sarkar, V., and Gao, G. R. Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections. CAPSL Technical Memo Revised 65, University of Delaware, Mar. 2007.

[32] Ziarek, L., Welc, A., Adl-Tabatabai, A.-R., Menon, V., Shpeisman, T., and Jagannathan, S. A uniform transactional execution environment for java. In *ECOOP* (2008), pp. 129–154.

Notes

[1] The C++0x name is historical; it is actually expected to be approved in 2010 at the earliest.

[2] In most cases there are additional "wizards only" facilities that relax sequential consistency guarantees. We ignore those in this paper, since our primary goal is a simple programming model for mainstream programmers.

[3] Where it is credited to Mark Miller.

[4] Since the original submission, discussions with others, notably with Tatiana Shpeisman, pointed out that if rollback is provided, the multithreaded version will probably also need to provide isolation, in order to avoid rolling back multiple threads. Thus the two are not completely orthogonal. One of the reviewers provided a similar comment.