Generic Programming Needs Transactional Memory

Justin E. Gottschlich
justin.e.gottschlich@intel.com
Intel Labs

Hans-J. Boehm
hans.boehm@hp.com
HP Labs

Abstract

Locks are the most widely used synchronization mechanism for threads. It is well-known that naive use of locks can easily lead to program failures through deadlock. Such deadlock is usually avoided through careful lock ordering. We argue that this approach is incompatible with several increasingly important programming practices that rely on libraries invoking ("calling-back") essentially unknown client code. Template-based generic programming in C++ is probably the most extreme in its use of call-backs, in that often almost every operator used in a generic function is potentially defined by client code.

Template functions are essential to C++. Much of the standard library consists of template functions and classes. We expect the same to be increasingly true for libraries designed for concurrency that support synchronization. We argue that if locks are used for synchronization in such code we have no reliable methodology for avoiding deadlock; we need an alternate synchronization mechanism. We argue that transactional memory can extract us from this predicament.

Unlike much of the debate surrounding transactional memory, we do not start with a focus on performance. We argue instead that transactional memory provides a desperately needed programmability improvement, which we have learned how to implement with sufficient performance to make it viable. We believe this sheds new light on the benefits of, and requirements for, transactional memory.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Parallel Programming

General Terms: Algorithms, Design

Keywords: Transactional Memory, Generic Programming

1. Introduction

Locks (or *mutexes*) are the most widely used mechanism to synchronize shared-memory access in multithreaded programs in today's software [15]. A program that uses more than one lock must generally follow a program-defined sequential locking order for the locks that are acquired by the program, otherwise a deadlock can occur as shown in Figure 1.

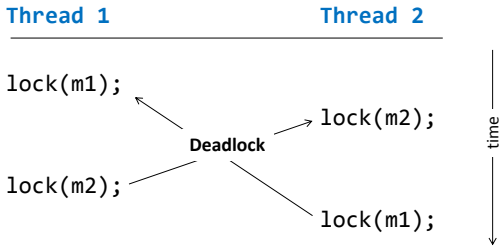

Figure 1. Deadlock Caused by Violation of Sequential Locking Order.

The deadlock shown in Figure 1 occurs because Thread 1 orders its lock acquisitions in the sequence of `(m1, m2)`, while Thread 2 orders the same lock acquisitions in the sequence of `(m2, m1)`. Thread 1 acquires `m1` while Thread 2 acquires `m2` and then each thread attempts to acquire the other lock without releasing the first one. Each thread will wait indefinitely for the other thread to release the lock it needs to make forward progress.

In simple cases, this typically has an easy solution. It may be possible to arrange Thread 2 such that it acquires `m1` and `m2` in the opposite order, enforcing consistent lock ordering. If `m1` is acquired in a separate nested function call, it may be possible to turn `m1` into a reentrant lock (`recursive_mutex` in C++11). It can then be acquired not only in the nested call after `m2`, but also preemptively before acquiring `m2`.

2. The problem with callbacks

Suppose now that instead of Thread 2 explicitly acquiring locks in order `(m2, m1)`, it calls a library function `f(g)` as in Figure 2, where `f` first acquires `m2`, and then calls `g`, which acquires `m1` with `m2` still held. This is fundamentally the same situation, with the same resulting deadlock as before.

If we assume that `f` is implemented in a library, and the function passed as `g` is provided by the main application, there is no longer a clear methodology for avoiding the problem. The author of `f` is unlikely to even be aware of the lock `m1`, and is thus no longer in a position to ensure lock ordering. The only likely way to avoid deadlock is to ensure that no locks are acquired by the library routine `f` and held

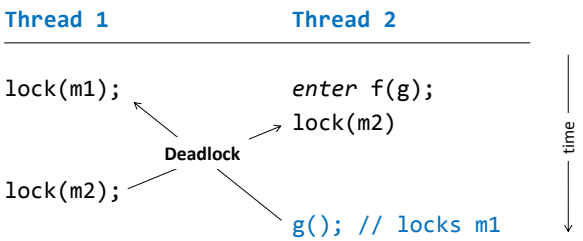

Figure 2. Deadlock with Generic Function.

during the callback. Yet, this is often difficult and unnatural in practice, as we explore further in Section 4.

This problem has been encountered many times before. The observer pattern is a common use of callbacks and appears as the poster child for the difficulty of thread programming in [20], for precisely this reason.[1] The Linux/GNU `dl_iterate_phdr` function provides a callback based interface acquiring locks internally, and web-based accounts of resulting deadlocks are easy to find.

Another fairly common use of what amounts to a callback interface is reference counting with synchronous destruction of objects, for example by using C++11's `shared_ptr` facility. An assignment to a reference counted pointer may lead to invocation of the destructor for the previously referenced object, which often leads to further reference count decrements, and further destructor invocations, possibly of objects that were indirectly referenced by the original pointer value, but whose implementation is not understood by the author of the code containing the pointer assignment. As is discussed in [3, 4], this again leads to deadlocks, among other issues.[2]

3. Generic programming: call-backs everywhere

From a software engineering perspective, software reusability and maintainability are key characteristics for successful large scale software development [9, 12]. Many modern programming languages, like Java, Scala, and Python, support software reusability and maintainability by providing the programmer with various ways to write portions of their software in a generic fashion [2, 19, 23]. Once the generic outline of the algorithm is captured it can then be used generally for various data types each with type-specialized functionality.

In C++, template functions are the most common way to write generic algorithms [1, 26]. A template function usually captures the portions of an algorithm necessary for all data types, or a range of data types, while omitting details that are specific to any particular data type. An example of such a template function is shown in Figure 3, where the greatest of three variables of type `T` is returned.

[1] Lee even points out that transactions provide a solution, but seems to downplay their impact [20].

[2] A very similar problem in early Java implementations is described in [27], where it is misattributed to the Java specification.

```
template <typename T>
T const & max3(T const &a1, T const &a2,
               T const &a3)
{
  T max(a1);
  if (a2 > max) max = a2;
  if (a3 > max) max = a3;
  return max;
}
```

Figure 3. A Template Function that Returns the Maximum Value of Three Variables.

Implicit in Figure 3's code are calls to `operator>()` for type `T` invoked, for example, by the expression `a2 > max`. By embedding a call to `operator>()` in this fashion, the `max3()` template function abstracts away the specific implementation details associated with `operator>()` for any particular type, enabling the programmer to independently specify what it means for a variable of such type to be less than another variable of the same type. Effectively the call to the `>` operator becomes a callback to client code. The author of `max3()` has no idea whether it will invoke a built-in operator or user-defined function, or, in the latter case, which locks it might acquire. It is quite likely that when `max3()` was written, the relevant implementation of `operator>()` did not yet exist.

What makes generic programming different from our prior examples is that many or most of the function calls and operator invocations depend on type parameters, and are thus effectively callbacks. That includes not only the `operator>()` uses, but also the assignments (`operator=()`), and the constructor used to initialize the `max` variable. It also includes the syntactically invisible destructor for `max`, and even possibly the syntactically invisible construction and destruction of expression temporaries. These constructors and destructors are likely to acquire locks if, for example, the constructor takes possession of resources from a shared free list that are returned by the destructor.

In order to enforce a lock ordering, the author of any generic function acquiring locks (or that could possibly be called while holding a lock) would have to reason about the locks that could potentially be acquired by any of these operators, which appears thoroughly intractable.

In the next section we illustrate this more concretely with a more complete example.

4. C++ template and locks example

Deadlocks are generally "programming-in-the-large" problems. For small applications that can be easily and fully understood by a single programmer, it is usually possible to avoid deadlocks, perhaps at the cost of convoluted code. Nonetheless, to fully appreciate the issue, we feel it is useful to present a concrete example. Here we do so.

```
template <typename T>
class concurrent_sack
{
public:
   ...
   void set(T const &obj) {
      lock_guard<mutex> _(m_);
      item_ = obj;
   }
   T const & get() const {
      lock_guard<mutex> _(m_);
      return item_;
   }
private:
   T item_;
   mutex m_;
};
```

Figure 4. A Concurrent Sack that Holds One Item.

Figure 4 presents a `concurrent_sack` data structure, which is designed to illustrate a concurrent container. It can hold a single element. Multiple threads can set and retrieve that element without synchronization in the client. Thus accesses are explicitly protected by a lock. This is done using the C++11 standard `lock_guard` class, whose constructor invokes the `lock()` function on the provided object, and whose destructor invokes the corresponding `unlock()` function. This is C++'s standard mechanism for exception-safe lock acquisition [18].

Figure 5 defines a class `T` with an explicitly defined assignment operator, which occasionally logs messages to a `log` object `L`. The `log` class provides a member function to add an entry to the log. As would be expected, it is safe for use from multiple threads. It also provides explicit `lock()` and `unlock()` member functions, which can be used to ensure that a thread that needs to log consecutive messages can do so. This API design is similar to that used by POSIX `stdio`.

Now assume that one thread (Thread 2) needs to generate two contiguous log messages. One of the messages relies on a value extracted from a `concurrent_sack<T>`. Thus, thread 2 explicitly acquires first the lock on `L`, and then the lock on the `concurrent_sack`. If it runs concurrently with a `set` operation on the same `concurrent_sack<T>`, which happens to generate a log entry, caused by a failed `check_invariants` call within class `T`'s `operator=()`, the program will deadlock as demonstrated in Figure 6.

4.1 Why is this problem so difficult?

The generic programming deadlock scenario shown in Figure 6 is challenging to address because it is neither the fault of the generic library writer who wrote `concurrent_sack`

```
class log {
public:
   ...
   void add(string const &s) {
      lock_guard<recursive_mutex> _(m_);
      l_ += s;
   }
   void lock() { m_.lock(); }
   void unlock() { m_.unlock(); }
private:
   recursive_mutex m_;
   string l_;
} L;

class T {
public:
   ...
   T& operator=(T const &rhs) {
      if (!check_invariants(rhs))
      { L.add("T invariant error"); }
   }
   bool check_invariants(T const& rhs)
   { return /* type-specific check */; }
   string to_str() const { return "..."; }
};
```

Figure 5. A Reentrant Safe Log Class and an Example Class T That Uses It.

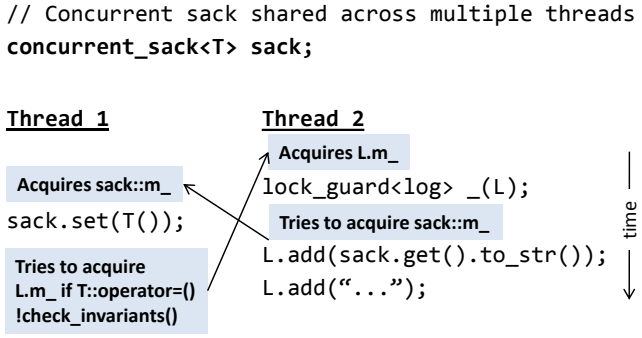

Figure 6. A Deadlock Using the Concurrent Sack with the Log Class.

nor the end programmer who wrote the `T` and `log` class. In both cases, neither programmer violated any known locking order and neither can be expected to properly fix the problem (at least, not by using locks).

The generic library writer cannot address the problem, because she has no way of knowing the deadlock will arise or how it will arise. The main program developer, on the other hand, can identify the deadlock if she has explicit

knowledge of the implementation details used to implement concurrent_sack. [3] Yet, even with such implementation knowledge, the specific details of the generic algorithm can (and should be allowed to) freely change without notifying the end-user, as long as the interface and functionality remain the same. Because of this, any attempt by the main program developer to permanently fix the problem by restructuring her software to align with the structure of the generic algorithm may be broken as soon the generic algorithm is updated.

To worsen matters, Figure 6's deadlock is likely to remain dormant in normal executions. That is, the deadlock will not occur in those executions where class T does not violate its invariants by always returning true for check_invariants(). This means that this deadlock has a lower probability of occurring because its dynamic state requirements might be rarely exercised when compared to other deadlocks that occur regardless of the program's dynamic state, such as the deadlock shown in Figure 1. Without reliable bug reproduction, fixing such bugs can be challenging and verifying such fixes are correct even more so [11, 24].

5. C++ template and transactions example

One of the core tenets of transactional memory is the promise to avoid deadlock [13]. In this section, we illustrate the importance of that promise by using transactions as a replacement for locks to synchronize the programming example we outlined in Section 4. With this approach, we are able to avoid the deadlock problem we previously encountered.

```
template <typename T>
class concurrent_sack
{
public:
   ...
   void set(T const &obj) {
      __transaction { item_ = obj; }
   }
   T const & get() const {
      __transaction { return item_; }
   }
private:
   T item_;
};
```

Figure 7. Transaction-Revised Concurrent Sack.

Figure 7 shows the transaction-revised concurrent_sack template class. The only change that we made to this class is that it now synchronizes its set() and get() methods with transactions instead of lock_guards [18]. All other implementation details of concurrent_sack are identical to the original.

```
class log {
public:
   ...
   void add(string const &s) {
      __transaction { l_ += s; }
   }
private:
   string l_;
} L;

class T {
public:
   ...
   T& operator=(T const &rhs) {
      if (!check_invariants(rhs))
      { L.add("T invariant error"); }
   }
   bool check_invariants(T const& rhs)
   { return /* type-specific check */; }
   string to_str() const { return "..."; }
};
```

Figure 8. Transaction-Revised Reentrant Safe Log Class and an Example Class T That Uses It.

We also updated the log class to use transactions, rather than locks, as shown in Figure 8. The updated log class uses a transaction to synchronize access to its add() method, which internally writes to L's shared string, l_. We have included the T class in Figure 8 for completeness, but it remains unchanged from the original.

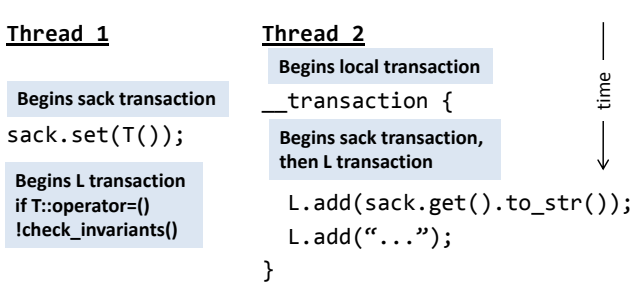

Figure 9. A Deadlock-Free Execution Using the Transaction-Revised Concurrent Sack with the Transaction-Revised Log Class.

Let us now consider the identical program execution that was previously described in Section 4. This time, let us use

[3] Although such knowledge fundamentally challenges a key motivation of generic programming; that is, abstraction of implementation details.

the new *transaction-revised* classes as shown in Figures 7 and 8. As before, a thread (Thread 2) needs to generate two contiguous log messages, one of which relies on a value extracted from `concurrent_sack<T>`. Because these operations must be placed in a consecutive sequence in `L`'s string `l_`, thread 2 must synchronize `l_` for the entire operation. Thread 2 achieves this by placing both `add()` calls within a local transaction as shown in Figure 9. This local transactions ensures that the memory modified across both `add()` calls remains isolated for duration of the entire local transaction. When the local transaction commits, the resulting state is seen by other threads a single atomic event.

When thread 1 performs its `sack.set()` operation it begins the transaction embedded within the `set()` method. When both threads 1 and 2 execute in the order shown in Figure 9, their respective transactions will conflict due to overlapped access to `sack`'s `item_`. Depending on the outcome of the `check_invariants()` method from class `T`, they may also conflict on their access to `L`'s `l_`. In either case, the transactional conflicts will not result in deadlock. At worst, one transaction will be aborted and restarted; at best, one transaction will be momentarily stalled while the other transaction commits. This example not only demonstrates the power of transactions to overcome the deadlock issue we demonstrated in Section 4, but it also shows that a transactional solution can result in fewer lines of code and, we believe, is easier to reason about than the lock-based example, resulting in more maintainable software.

6. Why transactions fit for generic programming

Generic programming generally aims to express an idea abstractly so it can be used for many different purposes [1]. When locks are used with generic programming, some degree of the locks' specificity is leaked into the abstract idea, making it less abstract and introducing the ordering constraints we discussed in Section 4.

Transactions, on the other hand, more naturally align with generic programming because they are a higher order abstraction than locks. Transaction only demarcate the section of the code where shared-memory accesses must be synchronized. Unlike locks, they place no ordering constraints (i.e., lock ordering) on the program.

Transactions also do not explicitly specify the way in which concurrency will be achieved, be it through a hardware, software, or hybrid TM [7, 14, 25]. By abstracting away such details, the underlying TM system that is used can be changed, potentially at run-time [29], without changing the generic program's algorithm. This widens the space in which the original generic program can be used because different environments have different concurrency constraints, such as transactions that execute locally on a single multicore machine versus transactions that execute over a distributed network [6, 16, 21].

Furthermore, we believe that with the arrival of real hardware transactional memory support in both IBM's Blue Gene/Q and Intel's Haswell processors [16, 17, 22], many of the early concerns about transactional performance [5] are likely to be, at the very least, partially addressed. The early results of Wang et al. on IBM's Blue Gene/Q processor demonstrate such a result and have begun to shed a much needed light on the performance viability of transactional memory in real hardware [28].

7. Conclusion

Locks are a poor match for generic programming. The need to acquire locks in the right order is at fundamental odds with the desire to write truly general code that does not depend on implementation details of the argument type. Though the problem is fundamentally similar to previously encountered issues with callbacks and locking, generic programming makes it far more pervasive and unmanageable.

We illustrated the pitfalls of such approaches when used with C++ template-based programming, which is perhaps the most common form of generic software. C++ templates amplify the issues, since it can be quite difficult to even identify the sites of potential callbacks, especially of constructor and destructor calls for compiler-introduced temporaries. But the fundamental problem is not specific to C++ templates. Rather, it impacts the entire generic programming ecosystem. Any generic code that invokes ("calls-back") arbitrary client-side code may result in deadlock.

Given the abundant use of C++ templates in the C++ Standard Template Library [18], the recent addition of thread support directly into C++11, and the increasing combination of the two, we believe it is imperative that an alternative solution to locks be found for multithreaded C++ template code. We showed that transactional memory is a viable alternative, and we believe it is the most practical one.

Although the issues with locks and callbacks have been previously recognized, we believe their impact has been understated, and the interactions with generic programming have not been highlighted. Prior examples of problems cases were much narrower than we believe the problem actually is.

This discussion sidesteps the usual arguments about transactional memory (cf. [5, 8]) because it is rooted in correctness, not performance. Our reasoning applies, only more so, if threads are used as a structuring mechanism on a uniprocessor. It is perhaps strongest in cases in which synchronization performance is not an issue because there is only infrequent use of shared data that must be protected.

Since our argument is not based on performance, it applies even to "transactions" implemented as a a single global lock. However transactional memory promises to support this deadlock-free programming discipline while preserving application scalability on multicore machines, and potentially, with the recent announcements of hardware transac-

tional memory [16, 17, 22], while preserving, or even enhancing, single-core application performance.

As has been previously observed [10, 30], we can start to obtain the benefits of transactional memory without wholesale conversion of existing code. For example, existing locks acquired only by "leaf" functions that do not hold the lock while calling unknown code cannot introduce deadlock.

References

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] D. M. Beazley. *Python Essential Reference*. Addison-Wesley Professional, 4th edition, 2009.

[3] H.-J. Boehm. Destructors, finalizers, and synchronization. In *Proceeedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, pages 262–272, 2003.

[4] H.-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HotPar*, 2009.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[6] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 39–52, New York, NY, USA, 2011. ACM.

[8] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *CACM*, 54(4):70–77, 2011.

[9] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[10] J. E. Gottschlich and J. Chung. Optimizing the concurrent execution of locks and transactions. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2011.

[11] J. E. Gottschlich, G. A. Pokam, and C. L. Pereira. Concurrent predicates: Finding and fixing the root cause of concurrency violations. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2012.

[12] D. R. Hanson. *C interfaces and implementations: techniques for creating reusable software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[13] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPoPP*, pages 48–60. ACM, 2005.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[16] Intel Corporation. Intel architecture instruction set extensions programming reference (Chapter 8: Transactional synchronization extensions). 2012.

[17] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM system z. In *Proceedings of the 45th International Symposium on Microarchitecture (MICRO)*, December 2012.

[18] N. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Prentice Hall, 2012.

[19] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[20] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[21] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions: In Concurrent and Distributed Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[22] R. Merritt. IBM plants transactional memory in CPU. *EE Times*, 2011.

[23] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[24] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multcore x86 processors. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*, December 2011.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*. Aug 1995.

[26] B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Reading, Mass., 1997.

[27] Sumatra Project. The Java hall of shame. `http://www.cs.arizona.edu/projects/sumatra/hallofshame/`, retrieved 12/14/2012.

[28] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[29] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear. A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim.*, 8(4):54:1–54:23, Jan. 2012.

[30] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.