Transactional Language Constructs for C++

Authors: Hans Boehm, HP, hans.boehm@hp.com
Justin Gottschlich, Intel, justin.e.gottschlich@intel.com
Victor Luchangco, Oracle, victor.luchangco@oracle.com
Maged Michael, IBM, maged.michael@acm.org
Mark Moir, Oracle, mark.moir@oracle.com
Clark Nelson, Intel, clark.nelson@intel.com
Torvald Riegel, Red Hat, triegel@redhat.com
Tatiana Shpeisman, Intel, tatiana.shpeisman@intel.com
Michael Wong, IBM, michaelw@ca.ibm.com

Document number: N3341=12-0031
Date: 2012-01-11
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong, IBM, michaelw@ca.ibm.com
Revision: 1

Introduction

We propose that transactional language constructs be integrated into the C++ programming language, initially as a technical report (TR). We believe that this TR should be considered by the Concurrency Subgroup. It has impact to both the Core Language and the Library Working Group. This paper should be of interest to anyone who would like parallel programming in C++ to be easier, and to more easily support a modular programming paradigm.

1. The Problem

Two types of synchronization supported by C++11 are mutexes, or locks, and atomics. Locks and atomics are basic abstractions used to access and mutate shared state, as defined by the Three Pillars of Concurrency [8]. From this view, these abstractions aim to avoid data races and synchronize objects in shared memory. Unfortunately, locks and atomics are notoriously difficult to use [9]. Simple coarse-grain locking strategies, where all program data are protected using one or few locks, lead to unnecessary serialization of program execution and loss of performance. Sophisticated fine-grain locking or use of atomics result in complex association between data and synchronization that protects access to that data. A failure to correctly maintain these associations throughout the program leads to concurrency errors, such as data races and deadlocks. Moreover, synchronization strategies designed to work well on one platform often perform poorly on a platform with a different number of hardware threads or a different cost for synchronization primitives.

Locks and atomics also have serious weaknesses when constructing larger programs out of smaller pieces; they are often described as not being composable. Because locks and atomics are so difficult to compose, they do not support modular programming well [2]. As C++ multithreaded programs increase in size and complexity, more advanced abstractions will be needed to mitigate the programming complexity that naturally arises from frequent use of synchronization in large-scale software systems.

Normally when a thread calls two functions in a sequence, even if each of them individually appears to execute atomically, the sequence no longer appears to be atomic; other threads can perceive the two calls as separate actions. To prevent other threads from visibly interleaving between the two calls, we must acquire a lock around the whole sequence. As a result, when we combine function calls into a larger program, we often end up with nested lock acquisitions: a function acquires a lock while one of its callers holds another one.

Unfortunately, once programs acquire more than one lock at a time, they may deadlock if multiple threads do so in an inconsistent order. The usual advice is to acquire locks only in a predetermined order. But there appears to be no practical way to do so in a large system. Edward Lee's well-known technical report "*The Problem with Threads*" presents a simple practical example of this kind of problem (the "observer" pattern) in a Java context, but it applies equally well to C++. Implementing this simple pattern with locks turns out to be nearly impossible. In spite of the technical report title, the problem lies with locks rather than threads – a concurrent transaction-based solution is not significantly harder than a sequential one.

The problem is compounded by common C++ programming practices. In order to enforce a lock order, each function call would be required to document which locks it can acquire. With this type of requirement, a template function `f<T>` would need to document its lock acquisitions. What if it assigns to a variable of type `T`? If, for example, `T` has `shared_ptr` type, its assignment operator can acquire any lock acquired by any destructor for an object reachable (via `shared_ptr`s) from a `T`. This is because the assignment of `shared_ptr` may delete any object that was directly or indirectly pointed to by that `shared_ptr`. This is far more detailed information than the implementor of `f<T>` is likely to know, or want to know. It implies a complete violation of any reasonable notion of modularity.

Functions that use atomics to perform shared memory operations are also not composable, but in a different sense: atomics provide no easy mechanism for combining the small atomic actions supplied by `atomic<T>` into larger atomic actions. The atomic<T> operations are not intended to compose into bigger atomic operations.

Parallel programming inherently entails increased complexity in developing programs, in reasoning about programs, and in reproducing bugs. By using a transaction---rather than one or more locks---to synchronize a section of code, the programmer is not required to specify which metadata (i.e., which named variable) is used to synchronize data. In addition to simplifying the resulting software, this approach alleviates the need of a fixed order of execution, which is required by locks to avoid deadlock. This results in simpler designs that are easier to write, reason about, and maintain. Furthermore, it enables specialized synchronization support for different platforms, which can be improved over time, without requiring changes to the application code.

2. The Proposal

We propose to integrate transactional language constructs, or for short, transactional memory[1] (TM), into the C++ programming language. Our proposal and its integration into C++ are described in detail in the "Draft Specification of Transactional Language Constructs for C++" [1]. Some of the key benefits of TM, compared to locks, are listed below.

1. In case studies, TM has been shown to reduce the challenge of parallel programming compared to mutual exclusion [5][6]. Although the concept of mutual exclusion is straightforward, programs that overwhelmingly use locks, an implementation technique to enforce mutual exclusion, can

[1] The term *transactional memory* is often used with a variety of meanings. In this paper we use this term to refer to a language-level transactional abstraction rather than an implementation of such an abstraction.

quickly become unwieldy even for expert parallel programmers. The above mentioned studies have shown that using transactions instead of locks resulted in simpler programs with significantly fewer programming errors.

2. TM makes it easier to develop deadlock-free programs, as functions that use only transactional synchronization can be composed without a risk of deadlock. This is a significant step beyond what can be said of systems that use locks for shared memory synchronization, as locks are inherently prone to deadlock.

3. TM supports a modular programming model; that is, a programming model where software is written by composing separate and interchangeable modules together. Modular programming is fundamental to C++. All of C++'s standard libraries and its generic programming language design support a modular programming model. In contrast to the problems with locks, which we described in the previous section, TM is composable.

4. TM raises a level of abstraction compared to mutual exclusion because transactions support a wider range of implementation approaches, including speculative execution. When programmers use transactions to control access to shared data they specify *what* is synchronized, not *how* it is synchronized. This enables a greater degree of implementation freedom to produce more efficient solutions.

5. TM can be designed such that it can be used in conjunction with most existing C++ concurrency mechanisms and, in cases when it cannot, it can be designed such that concurrency incompatibilities are identified as errors at compile-time.

Adding TM to C++ will improve the modularity of concurrent libraries, make C++ easier to teach and learn, and supply a programming model for future hardware. These benefits are in line with the fundamental spirit of C++ as directed by Bjarne Stroustrup [7] and many committee members.

2.1 A Brief Introduction to Transactional Memory

Like mutual exclusion, TM is a concurrency control mechanism. TM aims to simplify shared memory parallel programming by moving the complexity of shared memory management away from the programmer's view and placing it under the control of the implementation of the TM.

To access shared memory variables using TM, programmers use a *transaction*, which is a sequence of operations that execute in isolation from other transactions. From a programmer perspective, transactions behave as if they were executed one at a time, with no two transactions ever running concurrently. In order to improve performance, TM subsystems often execute transactions concurrently when possible. In particular, transactions that do not conflict (that is, they do not concurrently access the same location of memory where at least one of the accesses is a write) are typically executed concurrently by the TM subsystem.

In data-race free programs, transactions that do not contain other types of synchronization are *isolated*; that is, they are seen by other threads as occurring as a single operation where operations from other threads only occur before or after the execution of a transaction. The following example illustrates how a transaction might be used to implement a swap function.[2]

```
void swap(int &x, int &y)
{
   __transaction
   {
      int tmp = x;
      x = y;
      y = tmp;
   }
}
```

In this example, the write operations to both `x` and `y` are seen by other threads as occurring as a single, indivisible operation. There is no visible program state where `x = y` has occurred and `y = tmp` has not occurred. In any program state observable by other threads, either both `x = y` and `y = tmp` have occurred or neither have.

TM supports composition, whereby transactions can be nested within each other to any depth, which raises the level of abstraction for parallel programming when compared to other conventional concurrency control mechanisms, such as mutual exclusion. The following example illustrates, at a high level, the benefit of TM's composition.

```
class Account // class definition
{
public:
  Account(int amt) : bal_(amt) {}

  void withdraw(int amt){
    __transaction { bal_ -= amt; }
  }

  void deposit(int amt){
    __transaction { bal_ += amt; }
  }

  int balance(){
    __transaction { return bal_; }
  }
private:
  int bal_;
};

Account chk(100), sav(0); // Global accounts chk ($100) and sav ($0)

void transfer(int amt)
{
   // Larger transaction composed of two smaller transactions
   __transaction
   {
      chk.withdraw(amt);
      sav.deposit(amt);
   }
}
```

[2] Our specification [1] proposes two kinds of transactions, each denoted with a different keyword. However, the differences between these transactions are not relevant for the examples we present in this paper, so we use a single keyword to avoid unnecessary complication.

In the above example, the `transfer()` function implements a transaction that is composed of two smaller transactions, implemented within `withdraw()` and `deposit()`, for two shared memory objects, `chk` and `sav`, respectively. Because transactions compose, the combined operations of `chk.withdraw(amt)` and `sav.deposit(amt)` become an indivisible operation when placed within the context of a single, larger transaction. This enables the operations of moving money between two accounts to be seen as a single operation, while avoiding the previously described problems that arise when such behavior is enforced explicitly with locks. To demonstrate this, consider the following example that consists of two threads that execute concurrently (assume `chk.bal_ == 100`, `sav.bal_ == 0`, initially).

```
Thread 1                                Thread 2
--------                                --------
                                        int chkBal = 0, savBal = 0;

                                        __transaction
                                        {
transfer(100);                             chkBal = chk.balance();
                                           savBal = sav.balance();
                                        }
```

In the above example, the outer transaction in thread 2 can execute either before or after the transaction in the `transfer()` function from thread 1. If thread 2's outer transaction is executed before thread 1's `transfer()` function, `chkBal = 100` and `savBal = 0`. If it is executed after thread 1's `transfer()` function, `chkBal = 0` and `savBal = 100`. These are the only two legal program states that can be observed by thread 2's transaction. There is no program state such that thread 2's outer transaction can execute between the steps of thread 1's `transfer()` function and see `chkBal = 0` and `savBal = 0`, thus preventing the program state in which the transferred money has been withdrawn from the `chk` account but has not yet been deposited into the `sav` account.

2.2 Advanced Cases

More complex examples of TM such as multi-level transactional nesting, embedding non-transactional forms of synchronization within transactions, or throwing an exception from within a cancelled transaction, are explained in detail in the "Draft Specification of Transactional Language Constructs for C++" [1].

3. Interactions and Implementability

3.1 Interactions

The following incomplete list consists of language features that we believe should have defined behavior and support regarding their interaction with TM if TM is to be incorporated into C++:

- C++ 11 memory model and atomics
- Support for member initializers
- Support for C++ expressions
- Legacy code integration
- Structured block nesting
- Multiple function entry and exit points
- Polymorphism
- Exceptions

Details on how we propose TM interact with these C++ language features are described in the "Draft Specification of Transactional Language Constructs for C++" [1]. TM is not a one-off or special feature, but when used in combination with carefully designed integration with expressions, and function blocks can enable powerful expressions that would otherwise be challenging to write correctly and efficiently.

3.2 Implementability

In 2008, a group was formed to create the "Draft Specification of Transactional Language Constructs for C++," (the C++ TM Specification). Since that time, the C++ TM Specification drafting group has actively pursued the design of a practical TM specification for C++. The group is a joint consortium of companies including representatives from HP, IBM, Intel, Red Hat, and Sun/Oracle. Version 1.0 of the C++ TM Specification was released to the public in August 2009. The group plans to release version 1.1 of the C++ TM Specification in early 2012. This TR proposal is submitted on behalf of the C++ TM group.

The C++ TM Specification proposes transactional constructs in C++, which allow for a wide variety of implementations. Since the release of version 1.0, there have been several partially conforming compiler reference implementations of the C++ TM Specification, such as Intel's STM compiler, Sun's Studio compiler, and IBM's Alphaworks STM C++ compiler and, most recently, TM support in GCC v.4.7. The existence of multiple implementations of varying stages of the Specification indicate the possibility of a range of implementation approaches, as well as a non-trivial body of usage experience of TM with C++. Despite its simple usage model, TM masks a large underlying complexity which can best be tested through implementation and usage. For this reason, we urge the committee to accept this proposal as a technical report.

The C++ TM Specification is designed to integrate TM with existing C++ constructs, and has considerations for transaction statements, statements safe for transactional execution, transactional safety function attributes, transaction expressions, function transaction blocks, transaction cancel statements, exception behaviors, transaction attribute propagation, and fully defined behavior with C++11's concurrency memory model. Not all proposed features are fundamentally required to provide the core benefits of transactional constructs in C++. We look forward to working with the C++ Standards Committee to determine an appropriate set of features based on our proposal, taking into account ongoing experience in both implementation and usage of these features.

In addition to purely software based implementations of TM, our transactional abstractions are open-ended in terms of their support for integration of existing and future hardware transactional memory systems. This is important because of predicted future trends of integrating TM into CPU cores. Transactional language constructs provide one way to exploit such hardware features in a portable way. C++1x should be armed with a respective programming model as transactional hardware models become reality.

Conclusion

Parallel programming inherently entails increased complexity in developing programs, in reasoning about programs, and in reproducing bugs. Two of the fundamental types of synchronization for C++11, locks and atomics, not only fall short in reducing this increased complexity, they contribute to it. TM, on the other hand, makes parallel programming simpler by raising the programming level of abstraction when compared to locks and atomics. With TM, programmers need only specify *what* is synchronized, not *how* it is synchronized. This results in simpler designs that are easier to write, reason about, and maintain. Furthermore, it enables specialized synchronization support for different platforms, which can be improved over time, without requiring changes to the application code. TM also supports composition which enables a modular programming model where software can be written by composing separate and interchangeable modules together. Modular programming is fundamental to C++.

In this paper, we, the C++ TM drafting group, proposed that transactional language constructs be integrated into the C++ programming language, initially as a technical report. Our group is a joint consortium of companies including representatives from HP, IBM, Intel, Red Hat, and Sun/Oracle. Our TM proposal and its integration into C++ are described in detail in the "Draft Specification of Transactional Language Constructs for C++." The C++ TM Specification has been designed such that our transactional language constructs integrate well with existing C++11 synchronization types.

Since the release of version 1.0 of the C++ TM Specification, there have been several partially conforming compiler reference implementations of the specification as well as a non-trivial body of usage experience of TM with C++. Recently, a large subset of our transactional language constructs are supported within GCC 4.7. Adding TM to C++ will improve the modularity of concurrent libraries, make C++ easier to teach and learn, and supply a programming model for future hardware. These benefits are in line with the fundamental spirit of C++.

Acknowledgement

This proposal is the culmination of work by the following individuals who contributed to the "Draft Specification of Transactional Language Constructs for C++."

> Ali-Reza Adl-Tabatabai (Intel), Kit Barton (IBM), Hans Boehm (HP), Calin Cascaval (IBM), Steve Clamage (Oracle), Robert Geva (Intel), Justin Gottschlich (Intel), Richard Henderson (Red Hat), Victor Luchangco (Oracle), Virendra Marathe (Oracle), Maged Michael (IBM), Mark Moir (Oracle), Ravi Narayanaswamy (Intel), Clark Nelson (Intel), Yang Ni (Intel), Daniel Nussbaum (Oracle), Torvald Riegel (Red Hat), Tatiana Shpeisman (Intel), Raul Silvera (IBM), Xinmin Tian (Intel), Douglas Walls (Oracle), Adam Welc (Intel), Michael Wong (IBM), Peng Wu (IBM)

Appendix A. Prior Approaches

Mutual Exclusion

Mutual exclusion is perhaps the most common form of concurrency control for shared memory parallel programming. In general, mutual exclusion ensures program correctness by limiting access to shared memory variables to one thread at a time. Mutual exclusion achieves this restriction by using mutually exclusive locks, also known as just locks or mutexes. For a thread to access a shared memory variable, it must first acquire the lock that protects the shared memory variable. When a thread has completed its access to the shared memory variable, it releases the lock.

On the surface, the concept of mutual exclusion is straightforward and easy to apply. However, the majority of the parallel programming community agrees that mutual exclusion quickly becomes unwieldy if used in large-scale software. Furthermore, many experts believe it is notoriously challenging to write both correct and efficient large-scale multithreaded software using mutual exclusion [3].

Non-Blocking Atomic Primitives and C++ Atomics Types

A less frequent form of synchronizing access to shared memory variables is to use non-blocking atomic primitives, such as compare-and-swap (CAS) or load-linked store-conditional (LL/SC), or more recently, C++ atomic types to write non-blocking algorithms.

Although these approaches can often yield substantial concurrency, their nontrivial use is often limited to only expert parallel programmers as correctly building even simple data structures, such as a queue, using these kinds of synchronization primitives can be challenging [3].

Lock Elision

Another possible speculative technique, known as lock elision, has been proposed where locks are elided, that is, not acquired, in cases where conflicts in the data they protect are absent. Lock elision may reduce unnecessary synchronization, thereby increasing concurrent throughput. However, while lock elision can result in notable performance gains over lock-based systems that do not use lock elision, it still requires that programmers write correct and complete locking schemes for their multithreaded software. Therefore, lock elision does not significantly reduce, nor is it intended to reduce, the challenge of writing multithreaded software.

Appendix B. Existing TM Support

Programming languages that have directly integrated TM into the language include:

- Chapel
- Clojure
- Concurrent Haskell
- Fortress
- X10

Experimental support for TM has also been developed for the following languages:

- C
- C#
- Haskell
- Java
- Perl
- Python
- OCaml
- Scala (TM support proposed for standard library integration)

A large body of TM work exists for C++ using various approaches, including language changes and library support.

References

[1] A. Ald-Tabatabai, T. Shpeisman, and J. Gottschlich. Editors. 2009. “Draft Specification of Transactional Language Constructs for C++.” (http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/)

[2] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. “Composable memory transactions.” In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '05).

[3] M. Herlihy and N. Shavit. “The Art of Multiprocessor Programming.” Morgan Kaufmann. March 2008.

[4] M. M. Michael and M. L. Scott. 1996. “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.” In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (PODC '96).

[5] V. Pankratius and A. Adl-Tabatabai. 2011. “A study of transactional memory vs. locks in practice.” In SPAA 2011.

[6] C. J. Rossbach, O.S. Hofmann, and E. Witchel. “Is Transactional Programming Actually Easier?” In PPoPP 2010.

[7] Bjarne Stroustrup. 1995. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publ. Co., New York, NY, USA.

[8] H. Sutter. “The Pillars of Concurrency.” In Dr. Dobbs, July, 2007. (http://drdobbs.com/article/print?articleId=200001985&siteSectionName=)

[9] H. Sutter. “The Trouble with Locks.” In Dr. Dobbs, March, 2005.