Transactional Memory Support for C++

Authors: Victor Luchangco, victor.luchangco@oracle.com
Michael Wong, michaelw@ca.ibm.com
with other members of the transactional memory study group (SG5), including:
Hans Boehm, hans.boehm@hp.com
Justin Gottschlich, justin.gottschlich@intel.com
Jens Maurer, jens.maurer@gmx.net
Paul McKenney, paulmck@linux.vnet.ibm.com
Maged Michael, maged.michael@gmail.com
Mark Moir, mark.moir@oracle.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Michael Spear, spear@cse.lehigh.edu
Document number: N3919
Date: 2014-02-14
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong, michaelw@ca.ibm.com (head of SG5)
Revision: 1 (EWG approved version)

1 Introduction

Transactional memory supports a programming style that is intended to facilitate parallel execution with a comparatively gentle learning curve. This document describes a proposal developed by SG5 to introduce transactional constructs into C++ as a Technical Specification. It is a revision of N3718, an earlier document with the same title, presented at the September 2013 meeting in Chicago. However, this document is self-contained, and can be read independently of N3718. At the Chicago meeting, we received encouraging and helpful feedback, which we have used to revise this proposal.

This proposal is based in part on the *Draft Specification for Transactional Constructs in C++ (Version 1.1)* published by the Transactional Memory Specification Drafting Group in February 2012. It represents a pragmatic basic set of features, and omits or simplifies a number of controversial or complicated features from the *Draft Specification*. Our goal has been to focus SG5's efforts towards a basic set of features that is useful and can support progress towards possible inclusion in the C++ standard.

In addition to a description of the proposal, this document contains a summary of some of feedback we received at the Chicago meeting and of the discussion within SG5 since that meeting. Unlike N3718, it does not include examples or precise wording changes, as we have not yet revised them to reflect the changes in the proposal since N3718. However, these changes, summarized in Section 2.1, are mostly superficial.

2 Overview

We introduce two kinds of blocks to exploit transactional memory: *synchronized blocks* and *atomic blocks*. Synchronized blocks behave as if all synchronized blocks were protected by a single global recursive mutex. Atomic blocks (also called *atomic transactions*, or just *transactions*) appear to execute atomically and not concurrently with any synchronized block (unless the atomic block is executed within the synchronized block). Some operations are prohibited within atomic blocks because it may be impossible, difficult, or

expensive to support executing them in atomic blocks; such operations are called *transaction-unsafe*. An atomic block also specifies how to handle an exception thrown but not caught within the atomic block.

Some noteworthy points about synchronized and atomic blocks:

Data races Operations executed within synchronized or atomic blocks do not form data races with each other. However, they may form data races with operations not executed within any synchronized or atomic block. As usual, programs with data races have undefined semantics.

Exceptions When an exception is thrown but not caught within an atomic block, the effects of operations executed within the block may take effect or be discarded, or `std::abort` may be called. This behavior is specified by an additional keyword in the atomic block statement, as described in Section 4. An atomic block whose effects are discarded is said to be *canceled*. An atomic block that completes without its effects being discarded, and without calling `std::abort`, is said to be *committed*.

Transaction-safety As mentioned above, transaction-unsafe operations are prohibited within an atomic block. This restriction applies not only to code in the body of an atomic block, but also to code in the body of functions called (directly or indirectly) within the atomic block. To support static checking of this restriction, we introduce a keyword to declare that a function or function pointer is transaction-safe, and augment the type of a function or function pointer to specify whether it is transaction-safe. We also introduce an attribute to explicitly declare that a function is *not* transaction-safe.

To reduce the burden of declaring functions transaction-safe, a function is assumed to be transaction-safe if its definition does not contain any transaction-unsafe code and it is not explicitly declared transaction-unsafe. Furthermore, unless declared otherwise, a non-virtual function whose definition is unavailable is assumed to be transaction-safe. (This assumption does *not* apply to virtual functions because the callee is not generally known statically to the caller.) These assumptions are checked at link time.

2.1 Changes since N3718

We made several changes based on feedback and discussion at and since the Chicago meeting, including:

- The *atomic transactions* and *relaxed transactions* of N3718 were renamed atomic blocks and synchronized blocks respectively (and the relevant keywords were changed to refect this).
- Synchronized blocks are defined before and without reference to atomic blocks.
- We eliminated the use of *escape* to refer to an exception being thrown but not caught within a transaction (and the relevant keywords were changed to reflect this).
- Synchronized blocks may be nested within atomic blocks. (In N3718, atomic transactions could be nested within relaxed transactions, but relaxed transactions could not be nested within atomic transactions.)
- We decided to expand the set of functions in the standard library designated transaction-safe, and the set of exceptions that can cancel an atomic block. (This document does not fully reflect the intended change; instead, it includes relevant comments at appropriate places.)

We believe that these changes clarify the concepts being defined and may facilitate programmers' adoption of synchronized and atomic blocks. See Section 7 for further discussion.

3 Synchronized Blocks

A *synchronized block* has the following form:

`synchronized` { *body* }

The evaluation of any synchronized block synchronizes with every evaluation of any synchronized block (whether it is an evaluation of the same block or a different one) by another thread, so that the evaluations of non-nested synchronized blocks across all threads are totally ordered by the synchronizes-with relation. That is, the semantics of a synchronized block is equivalent to having a single global recursive mutex that is acquired before executing the body and released after the body is executed (unless the synchronized block is nested within another synchronized block). Thus, an operation within a synchronized block never forms a data race with any other operation within a synchronized block (the same block or a different one).

Note: *Entering and exiting a nested synchronized block (i.e., a synchronized block within another synchronized block) has no effect.*

Jumping into the body of a synchronized block using `goto` or `switch` is prohibited.

Use of synchronized blocks Synchronized blocks are intended in part to address some of the difficulties with using mutexes for synchronizing memory access by raising the level of abstraction and providing greater implementation flexibility. (See *Generic Programming Needs Transactional Memory* by Gottschlich and Boehm in Transact 2013 for a discussion of some of these issues.) With synchronized blocks, a programmer need not associate locks with memory locations, nor obey a locking discipline to avoid deadlock: Deadlock cannot occur if synchronized blocks are the only synchronization mechanism used in a program.

Although synchronized blocks can be implemented using a single global mutex, we expect that some implementations of synchronized blocks will exploit recent hardware and software mechanisms for transactional memory to improve performance relative to mutex-based synchronization. For example, threads may use speculation and conflict detection to evaluate synchronized blocks concurrently, discarding speculative outcomes if conflict is detected. Programmers should still endeavor to reduce the size of synchronized blocks and the conflicts between synchronized blocks: poor performance is likely if synchronized blocks are too large or concurrent conflicting evaluations of synchronized blocks are common. In addition, certain operations, such as I/O, cannot be executed speculatively, so their use within synchronized blocks may hurt performance.

4 Atomic Blocks

An *atomic block* can be written in one of the following forms:

`atomic_noexcept` { *body* }
`atomic_commit` { *body* }
`atomic_cancel` { *body* }

The keyword following `atomic` is the atomic block's *exception specifier*. It specifies the behavior when an exception escapes the transaction:

`atomic_noexcept`: This is undefined behavior and is not allowed; no side effects of the transaction can be observed.

`atomic_commit`: The transaction is committed and the exception is thrown.

`atomic_cancel`: If the exception is transaction-safe (defined below), the transaction is canceled and the exception is thrown. Otherwise, it is undefined behavior. In either case, no side effects of the transaction can be observed.

An exception is *transaction-safe* if its type is `bad_alloc`, `bad_array_length`, `bad_array_new_length`, `bad_cast`, `bad_typeid`, or a scalar type.

Comment: *Based on feedback on our earlier proposal (i.e., N3718), we intend to expand the set of transaction-safe exceptions. However, we have not investigated this sufficiently to expand it at this time.*

Code within the body of a transaction must be *transaction-safe*; that is, it must not be transaction-unsafe. Code is *transaction-unsafe* if:

- it contains an initialization of, assignment to, or a read from a volatile object;
- it is a transaction-unsafe `asm` declaration (the definition of a transaction-unsafe `asm` declaration is implementation-defined); or
- it contains a call to a transaction-unsafe function, or through a function pointer that is not transaction-safe (see Section 5).

Note: *The dynamic initialization of function-local statics is transaction-safe (assuming the code in the initialization expression is transaction-safe) even though it likely involves some nonatomic synchronization under the covers. However, see Section 6.*

Note: *Synchronization via locks and atomic objects is not allowed within atomic blocks (operations on these objects are calls to transaction-unsafe functions).*

Comment: *This restriction may be relaxed in a future revision of the Technical Specification.*

Jumping into the body of an atomic block using `goto` or `switch` is prohibited.

The body of an atomic block appears to take effect atomically: no other thread sees any intermediate state of an atomic block, nor does the thread executing an atomic block see the effects of any operation of other threads interleaved between the steps within the atomic block.

The evaluation of any atomic block synchronizes with every evaluation of any atomic or synchronized block by another thread, so that the evaluations of non-nested atomic and synchronized blocks across all threads are totally ordered by the synchronizes-with relation. Thus, a memory access within an atomic block does not race with any other memory access in an atomic or synchronized block. However, a memory access within an atomic block may race with conflicting memory accesses not within any atomic or synchronized block. The exact rules for defining data races are defined by the memory model.

Note: *As usual, programs with data races have undefined semantics.*

Note: *Although it has no observable effects, a canceled atomic block may still participate in data races.*

Note: *This proposal provides "closed nesting" semantics for nested atomic blocks.*[1]

Use of atomic blocks Atomic blocks are intended in part to replace many uses of mutexes for synchronizing memory access, simplifying the code and avoiding many problems introduced by mutexes (e.g.,

[1] For a description of closed nesting, see *Transactional Memory* by Harris, Larus and Rajwar, for example.

deadlock). We expect that some implementations of atomic blocks will exploit hardware and software transactional memory mechanisms to improve performance relative to mutex-based synchronization. Nonetheless, programmers should still endeavor to reduce the size of atomic blocks and the conflicts among atomic blocks and with synchronized blocks: poor performance is likely if atomic blocks are too large or concurrent conflicting executions of atomic and synchronized blocks are common.

5 Transaction-Safety for Functions

A function declaration may specify the `transaction_safe` keyword or the `transaction_unsafe` attribute.

Declarations of function pointers and typedef declarations involving function pointers may specify the `transaction_safe` keyword (but not the `transaction_unsafe` attribute).

A function is *transaction-unsafe* if

- any of its declarations specifies the `transaction_unsafe` attribute,
- it is a virtual function that does not specify the `transaction_safe` keyword and does not override a function whose declaration specifies the `transaction_safe` keyword,
- any of its parameters are declared volatile,
- it is a constructor or destructor whose corresponding class has a non-static volatile data member, or
- its definition contains transaction-unsafe code as defined in Section 4.

Note: *This definition covers lambdas and implicitly defined member functions.*

Note: *A function with multiple declarations is transaction-unsafe if any of its declarations satisfies the definition above.*

No declaration of a transaction-unsafe function may specify the `transaction_safe` keyword. A function is *transaction-safe* if it is not transaction-unsafe. The transaction-safety of a function is part of its type.

Note: *A transaction-safe function cannot overload a transaction-unsafe function with the same signature, and vice versa.*

A function pointer is *transaction-safe* if it is declared with the `transaction_safe` keyword. A call through a function pointer is transaction-unsafe unless the function pointer is transaction-safe.

A transaction-safe function pointer is implicitly convertible to an ordinary (i.e., not transaction-safe) function pointer; such conversion is treated as an identity conversion in overloading resolution.

A compiler-generated constructor/destructor/assignment operator for a class is transaction-unsafe if any of the corresponding operations on any of the class's direct base classes is transaction-unsafe.

A member function declared with the `transaction_safe` keyword or `transaction_unsafe` attribute in a base class preserves that attribute in any derived class, unless that member is redefined or overridden. Functions brought into a class via a `using` declaration preserve the attribute in the original scope. A virtual function of transaction-safe type must not be overridden by a virtual function of transaction-unsafe type.

Because a compilation unit might not contain all declarations of a function, the transaction safety of a function is confirmed only at link time in some cases.

Transaction-Safety of Functions in the Standard Library Certain functions in the standard library are designated as transaction-safe. See N3862 for details.

Allocation and deallocation functions are necessarily transaction-unsafe, and user-defined `new` and `delete` functions are also transaction-unsafe. See Section 7.7 for further discussion of this issue.

6 Outstanding Issues

We have yet to resolve two outstanding issues from N3718:

- The proposed wording changes for the memory model in N3718 assume that code that could observe a violation of the atomicity of an atomic transaction would necessarily form a data race with that transaction. However, this assumption is invalid because concurrent initialization of function-local static variables are not racy. We think we should add wording to the memory model specification to directly ensure the atomicity of atomic blocks.

- We specify that when the exception assumptions of an atomic block are violated, `std::abort` rather than `std::terminate` is called. This avoids the question of the status of the transaction from the perspective of a termination handler. However, in some cases, normal exception behavior (independent of atomic blocks) requires `std::terminate` to be called, raising the question we hoped to avoid.

See Section 7.14 of N3718 for more discussion of these issues.

7 Discussion of Feedback and Other Issues

This proposal is a mostly expository and syntactic revision from the proposal in N3718, presented at the C++ meeting in Chicago in September 2013, incorporating feedback we received there and subsequent discussion within SG5. In this section, we summarize some of the feedback we received at the Chicago meeting and the subsequent discussion within SG5, which we hope will help readers understand the changes relative to N3718, and also some aspects we opted not to change. For issues discussed before the Chicago meeting, please refer to the relevant section of N3718.

7.1 Making the proposal more modular

At the Chicago meeting, several people expressed concern that our transactions were unnecessarily entangling separable notions of *synchronization* and *exception safety*, the former having to do with coordinating access to shared memory and the latter with dealing with exceptional cases. They also worried that enabling atomic transactions to provide exception safety was a substantial undertaking that would hold up progress on the more pressing need for the synchronization provided by relaxed transactions (everyone, it seemed, agreed that mutexes were inadequate).

After much discussion, we agreed to clarify that atomic transactions and relaxed transactions embodied rather different ideas, and that an implementation could usefully provide relaxed transactions without atomic transactions, and to achieve this by specifying relaxed transactions without reference to atomic transactions. We also thought that we could avoid some confusion by not referring to both as transactions, and thus started to refer to relaxed transactions as *synchronized blocks*. (See Section 7.2 for more discussion on terminology.)

In rewriting the proposal, we considered whether we ought to emphasize the distinction even more by splitting the proposal into two, one describing synchronized blocks and the other describing atomic transactions (and transaction-safety). However, there was unanimous consensus against this among the active

participants of SG5. Among the cited concerns were that it could derail momentum on atomic transactions and that splitting synchronized blocks into a separate proposal that doesn't reference atomic transactions might lead to their developing without due consideration of their intended interaction with atomic transactions. We also noted that we explicitly chose to propose a Technical Specification because we expect the proposal to evolve as we gain experience with these constructs as they are actually used by programmers. As such, implementors need not implement the entire proposal for compliance, but we should not encourage them to abandon atomic transactions by splitting them off entirely.

7.2 Terminology

As mentioned above, we found that the term *transaction* evoked different notions to different people, and thus sometimes introduced more confusion than intuition, particularly in its use to refer to both atomic and relaxed transactions. Also, since we decided to describe relaxed transactions prior to and without reference to atomic transactions, it would be confusing to call them *relaxed* transactions. Therefore, we renamed them *synchronized blocks*, following the intuition expressed by several people at the Chicago meeting that they provide what they think of as synchronization. We have reservations about this term, in part because of its use in Java, but have not yet found a preferred alternative.

After revising the proposal as described above, some people opined that we ought to also rename atomic transactions (or just *transactions* at that point). We decided to revert back to an early syntax, in which transactions were introduced by the keyword `atomic` (this had been abandoned in the *Draft Specification*, and in N3718, in part to unify atomic and relaxed transactions, but had been revived by some at the Chicago meeting). A concern with using `atomic` is whether it would introduce syntactic or conceptual confusion with C++ atomic objects. However, we think that place in the grammar is sufficiently distinguished to avoid syntactic confusion, and that conceptual connection to atomic objects may actually be an advantage, because the concepts are similar. With this syntactic change, we called the blocks *atomic blocks*.

The use of *escape* to refer to an exception thrown but not caught within a block seemed to confuse more than aid understanding at the Chicago meeting, so we eliminated it in this revision.

7.3 Nesting synchronized blocks within atomic blocks

In N3718, relaxed transactions were transaction-unsafe, and so could not be executed within an atomic transaction. This restriction was natural in N3718 because the motivation for relaxed transactions was to allow a transaction to contain transaction-unsafe code: a transaction guaranteed to execute only transaction-safe code (as any code executed within an atomic transaction must be) should be designated an atomic transaction rather than a relaxed transaction. However, this restriction is not necessary, and is less compelling in this proposal, which more cleanly distinguishes synchronized blocks and atomic blocks, and specifically envisions that synchronized blocks might be implemented prior to atomic blocks, so that code intended to be atomic might be in a synchronized block rather than an atomic block. Thus, we agreed to allow synchronized blocks within atomic blocks.

7.4 Static checking of transaction-safety

The value to statically check the transaction-safety was questioned at the Chicago meeting, particularly because the `transaction_safe` keyword and changes to function and function-pointer types are needed only to enable this static check. Herb Sutter noted that in the past, when presented with the choice of introducing linguistic complexity or forgoing static checking, the C++ committee has typically favored forgoing static checking. Nonetheless, most of SG5 felt that there was significant value to static checking, not only in the greater assurance it provides, but also in helping to "transactionalize" existing programs

safely. (See *Transactionalizing Legacy Code* by Vyas, Liu and Spear in Transact 2013, for example.) If we ultimately decide to forgo static checking, then it may be helpful to introduce an optional annotation that enables static checking by a separate tool.

7.5 Choice of transaction-safe standard library functions

In N3718, we noted that we deliberately chose to designate only a small set of standard library fuctions as transaction-safe. Our reasoning was that this minimized work that had to be done immediately and provided greater flexibility for the future. (See Section 7.5 of N3718 for more discussion of this point.) However, people at the Chicago meeting pointed out several people at the Chicago meeting pointed out that this restriction may hinder adoption. Therefore, we have begun investigating how to make more of the standard library transaction-safe. This issue is discussed in a separate document (N3862).

7.6 Choice of transaction-safe exceptions

In N3718, we also noted that we deliberately defined a small set of exceptions as transaction-safe, with the intention to expand this as necessary. As with the standard library functions, we intend to revisit this decision and expland this set more aggressively.

7.7 Allocation and deallocation within atomic blocks

Standard containers are a vital building block of most application programs, and use of containers should be permitted within atomic blocks. Even the most simple of containers, such as `std::list` or `std::vector`, allocate and free memory for basic operations such as adding an element to the container. In some cases—at essentially unpredictable times—an allocation function must call out to the operating system to obtain additional storage. However, operating system calls are transaction-unsafe, which conflicts with the goal of supporting memory allocation within atomic blocks.

One implementation technique (used in gcc) is to abort (cancel) hardware-based transaction execution upon encountering a call to an allocation function and redo the transaction in software. The transaction is suspended while the allocation function executes, i.e. the allocation function does not execute in transaction context. When the allocation function returns the requested memory range, transactional execution of the calling code resumes. If that execution eventually encounters a need to cancel the transaction, a previously registered cleanup handler deallocates the memory.

This machinery would be entirely invisible to the application, except that C++ provides for user-replaceable allocation and deallocation functions (see Section 18.6 of the C++ standard). Repeated calls to these functions are thus observable. As a first step, spurious calls to allocation and deallocation functions (caused by transaction cancel and redo) must be expressly allowed, provided they occur in matched pairs.

As a next step, it might be desirable to start executing the allocation function while still in (hardware) transaction context. In most cases, the requested memory is probably available in a thread-local quick-list, not requiring an operating system call. For such a fast-path, canceling the hardware transaction and redoing in software would be seriously detrimental to performance. Only when the quick-list is exhausted should execution drop out of the transaction to allow operating system calls for acquiring more memory.

What should the standard-specified interface for allocation functions look like? It certainly makes a difference if the allocation function starts executing within a surrounding transaction, or as-if entirely outside any transaction. In the former case, what is the mechanism to explicitly request suspending the transactional execution to permit operating system calls? What is the mechanism to register a cleanup handler for a potential subsequent cancel? Is there maybe general value in such mechanisms, beyond allocation functions? This is unexplored design space in the context of this proposal. The most conservative way forward is

to disallow user replacement of allocation and deallocation functions if a call to such appears within a transaction anywhere in the program. This preserves all options for future evolution.

7.8 Transaction-safety as part of type for overloading resolution

In the current proposal, transaction-safety is not used in overloading resolution: it is not permissible to define two functions whose headers differ only in whether they are transaction-safe. At the Chicago meeting, Faisal Vali suggested that relaxing this restriction (i.e., by permitting transaction-safety to be used in overloading resolution) might address some problems encountered with the current proposal. In particular, this would allow a function to use one definition when called within an atomic block and another when called not within an atomic block. Such functionality would also enable programmers to take different paths depending on whether they are executing within an atomic block. However, supporting such a change will likely have ripple effects throughout the system, and thus requires further study before it can be incorporated into this proposal.

7.9 Restricting code permitted within synchronized blocks

We may want to restrict the operations that may be executed within a synchronized block, to enable certain implementation techniques and/or reduce the likelihood of unacceptable performance. For example, we may forbid I/O within a synchronized block, which would invalidate any speculative evaluation of the block.

Unless function calls are forbidden within synchronized blocks, statically checking any such restriction would likely require extending function types. As discussed in Section 7.4, such extension may receive pushback from the C++ committee.

7.10 Domains for synchronized blocks

Several people expressed concern that although "single global lock" semantics may seem attractive for its simplicity, it may induce too much synchronization in realistic large programs. Thus, it may be desirable to reduce synchronization by allowing synchronized blocks to specify "*domains*", and requiring (evaluations of) synchronized blocks to be ordered only if they have the same domain. In that case, a programmer could specify different domains for synchronized blocks known to never conflict. We might allow a synchronized block to specify multiple domains, and a synchronized block that does not specify any domain might be considered to have every domain (i.e., its evaluations must be ordered with respect to the evaluations of every other synchronized block).

Syntactically, the domain might be specified as an "argument" between the synchronized keyword and the body of the block:

`synchronized` (*domain*) { *body* }

If every synchronized block has exactly one domain (and the system does not recognize any relation among the domains), then this is logically equivalent to having different mutexes for each domain. However, this notion is more general than mutexes if synchronized blocks can specify multiple domains.

Introducing domains reintroduces many of the difficulties of mutex-based synchronization. For example, domains would likely be associated with memory locations, and the programmer would need to determine the correct domain(s) for each synchronized block. Also, a nested synchronization block may introduce deadlock, unless it specifies only domains specified by its (dynamically) enclosing synchronization block. If we impose such a restriction, then a synchronized block specifying a domain not specified by its enclosing synchronized block would be unsafe code, as discussed in Section 7.9.

If we do have a notion of unsafe code, and the definition of unsafe code ensures that a safe synchronized block (i.e., one that does not execute unsafe code) can be rolled back, then some of the problems with having multiple domains can be avoided for safe synchronized blocks. In particular, deadlock can be avoided. In this case, it is important to specify when the semantics guarantees that deadlock is avoided (whether it is statically checked).

7.11 Condition synchronization

Recent experience suggests that support for condition synchronization would be helpful. (See, for example, the experience reports by Vyas, Liu and Spear and by Skyrme and Rodriguez in Transact 2013.) As specified, we cannot use condition variables for this purpose because a synchronized block does not specify a mutex that can be passed to the condition variable. Although there is interest in providing such a mechanism, we do not yet have a specific proposal for it.

8 Related Documents

Some related documents and papers are listed below:

N3341: *Transactional Language Constructs for C++*

N3422: *SG5: Software Transactional Memory (TM) Status Report*

N3423: *SG5: Software Transactional Memory (TM) Meeting Minutes*

N3529: *SG5: Transactional Memory (TM) Meeting Minutes 2012/10/30-2013/02/04*

N3544: *SG5: Transactional Memory (TM) Meeting Minutes 2013/02/25-2013/03/04*

N3589: *Summary of Progress Since Portland towards Transactional Language Constructs for C++*

N3591: *Summary of Discussions on Explicit Cancellation in Transactional Language Constructs for C++*

N3592: *Alternative cancellation and data escape mechanisms for transactions*

N3690: *Programming Languages — C++*

N3695: *SG5 Transactional Memory (TM) Meeting Minutes 2013/03/11-2013/06/10*

N3717: *SG5 Transactional Memory (TM) Meeting Minutes 2013/06/24-2013/08/26*

N3718: *Transactional Memory Support for C++* (an earlier version of this proposal)

N3725: *Original Draft Specification of Transactional Language Constructs for C++, Version 1.1 (February 3, 2012)*

N3862: *Transactionalizing the C++ Standard Library*

Ali-Reza Adl-Tabatabai, Victor Luchangco, Virendra J. Marathe, Mark Moir, Ravi Narayanaswamy, Yang Ni, Dan Nussbaum, Xinmin Tian, Adam Welc, Peng Wu. *Exceptions and Transactions in C++*. USENIX Workshop on Hot Topics in Parallelism (HotPar), 2009.

Justin Gottschlich, Hans Boehm. *Generic Programming Needs Transactional Memory*. Workshop on Transactional Computing (TRANSACT), 2013.

Trilok Vyas, Yujie Liu, Michael Spear. *Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached*. Workshop on Transactional Computing (TRANSACT), 2013.

Alexandre Skyrme and Noemi Rodriguez. *From Locks to Transactional Memory: Lessons Learned from Porting a Real-World Application*. Workshop on Transactional Computing (TRANSACT), 2013.

Tim Harris, James Larus, Ravi Rajwar. *Transactional Memory, 2nd edition*, in Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010.

Resources from the Transactional Memory Specification Drafting Group predating SG5 are available from `https://sites.google.com/site/tmforcplusplus/`.

9 Acknowledgement

This work is the combined dedication and contribution from many people from academia, industry, and research through many years of discussions and feedback. Some of those are all the authors and chairs of the original external TM group that produced the original TM specification, all of the current SG5 SG, as well as individuals such as Dave Abrahams, Zhihao Yuan, Faisal Vali, Chandler Carruth, Lawrence Crowl, Olivier Giroux, Dietmar Kuehl, Jeffrey Yaskin, Jonathan Wakely, Eric Nielbler, Detlef Vollmann, Ville Voutilainen, Nevin Liber, Bjarne Stroustrup, Herb Sutter, Bronek Kozicki, Tony Van Eerd, Steve Clamage, Sebastien Redl, Niall Douglas and many others whom we may have forgotten inadverdently to list.

10 Examples

The first example below illustrates how transactions can elegantly solve a generic programming problem that is not possible to solve with locks. Subsequent examples are intended to clarify the features specified in this proposal.

10.1 Example illustrating importance of transactions for generic programming

Below we show an attempt to use locks for generic programming, and explain a fundamental problem with it. After that, we show how the same problem can be elegantly solved using transactions. These examples are based on examples in *Generic Programming Needs Transactional Memory* by Justin Gottschlich and Hans Boehm (TRANSACT 2013).

```
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_;
};

class T {
public:
   ...
   T& operator=(T const &rhs) {
```

```
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;
```

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

Given the declarations above, the following program results in deadlock. There is no way to order the locks to avoid this.

```
// Globally define sack
concurrent_sack<T> sack;
```

```
Thread 1                                  Thread 2
--------                                  --------
                                          // acquires L.m_
                                          lock_guard<log> _(L);
// acquires sack::m_
sack.set(T());
                                          // tries to acquire sack::m_
                                          // (deadlock)
                                          L.add(sack.get().to_str());
                                          L.add("...");
// tries to acquire L.m_ (deadlock)
// if T::operator==()'s call to
// check_invariants() returns false
```

Next we revisit the same problem using transactions.

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


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

```
} L;

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

With these declarations, the problem can be solved as follows. Note that the order in which the transactions are invoked does not matter, because no named locks are involved that could be misordered leading to deadlock as shown in the prior example.

Instead, transactions are used for this generic programming example enabling the generic programmer to build the system the way he or she believes it should be built, without leaking the implementation details to the end programmer.

Likewise, the end programmer can program in the most natural fashion for him or her without worrying about violating some embedded locking order within the generic programming code that he or she is using.

```
// Globally define sack
concurrent_sack<T> sack;

Thread 1                          Thread 2
--------                          --------
                                  // begins local transaction
                                  atomic_cancel
                                  {
// begins sack transaction
sack.set(T());

// begins L transaction if
// T::operator=()'s  call to
// check_invariants()
// returns false

                                      // begins sack transaction,
                                      // then L transaction
                                      L.add(sack.get().to_str());
                                      L.add("...");
                                  }
```

10.2 Example demonstrating atomicity of atomic blocks

This simple bank account example demonstrates the atomicity of atomic blocks.

```
class Account {
    int bal;
  public:
    Account(int initbal) { bal = initbal; };
```

```
    void deposit(int x) {
      atomic_noexcept {
        this.bal += x;
      }
    };

    void withdraw(int x) {
      deposit(-x);
    };

    int balance() { return bal; }
}

void transfer(Account a1, a2; int x;) {
  atomic_noexcept {
    a1.withdraw(x);
    a2.deposit(x);
  }
};

Account a1(0), a2(100);

Thread 1                  Thread 2
--------                  --------

transfer(a1, a2, 50);     atomic_noexcept {
                            r1 = a1.balance() + a2.balance();
                          }
                          assert(r1 == 100);
```

The assert cannot fire, because the transfer happens atomically and the two calls to `balance` happen atomically.

10.3 Example demonstrating need for atomic_cancel

Here, we extend the above example slightly so that transactions are logged by a function that may throw an exception, for example due to allocation failure.

```
void deposit(int x) {
  atomic_cancel {
    log_deposit(x);        // might throw
    this.bal += x;
  }
}

void withdraw(int x) {
  deposit(-x);
}

void transfer(account a1, a2; int x;) {
  try {
    atomic_cancel {
      a1.withdraw(x);
```

```
      a2.deposit(x);
    } catch (...) {
      printf("Transfer failed");
    }
  }
}
```

If the call from `transfer()` to `a2.deposit()` throws an exception, we should not simply commit the transaction, because the withdrawal has happened but the deposit has not. Canceling the transaction provides an easy way to recover to a good state, without violating the invariant the transaction in `transfer()` is intended to preserve. In this simple example, an error message is printed indicating that the transfer did not happen.

10.4 Example illustrating limitation regarding types of exceptions that can escape and workaround

If `log_deposit()` might throw an exception that is not transaction-safe, programmers can work around this by translating the exception to one that is transaction-safe before allowing it to escape.

```
void deposit(int x) {
  atomic_cancel {
    try {
      log_deposit(x);             // might throw
    } catch (Exception e) {
      throw TamedException(e);    // Produces transaction-safe exception
                                  // based on original exception
    }
    this.bal += x;
  }
}
```

10.5 Example illustrating that partial effects of cancelled transactions cannot be observed

```
#define TOO_BIG 17

int X = 0;

void do_something(int x) {
  atomic_cancel {
    X = x;
    if (x > 5)
      throw TOO_BIG;
  }
}

Thread 1                    Thread 2
--------                    --------

do_something(random());     atomic_noexcept {
                              r1 = X;
                            }
                            assert(r1 <= 5);
```

The assert cannot fire because a transaction that writes a value greater than 5 is canceled and therefore its effects cannot be observed by other threads.

10.6 Examples illustrating that partial effects of transactions that cause std::abort to be called cannot be observed

The first example shows that partial effects of a `atomic_noexcept` transaction that throws an exception cannot be observed by other threads.

```
int X = 0;

Thread 1                              Thread 2
--------                              --------

atomic_noexcept {                     int x;
  X = 1;                              atomic_noexcept {
  throw 0;                              x = X;
}                                     }
                                      assert(X==0);
```

For another example, suppose Thread 1 instead executes:

```
atomic_cancel {
  X = 1;
  throw SomeFancyException();
}
```

Again, Thread 2's assert cannot fire because partial effects of the `atomic_cancel` transaction that throws a non-transaction-safe exception cannot be observed by other threads.

10.7 Examples illustrating syhchronized blocks and non-races between accesses within transactions (including synchronized blocks)

Suppose we add the following method to the `Account` class shown in Section 10.2.

```
void print_balances_and_total (account a1, a2) {
  synchronized {
    printf("First account balance: %ld", a1.balance());
    printf("Second account balance: %ld", a2.balance());
    printf("Total: %ld", a1.balance() + a2.balance());
  }
}
```

Observations:

- This program is data-race-free: all concurrent accesses are within transactions.
- The synchronized block cannot be replaced with an atomic block, as I/O is not transaction-safe (due to calls to `printf`, which is a transaction-unsafe function).
- Balances will be consistent and total will equal sum of balances displayed.
- If we eliminate the synchronized block from this example (so the calls to `balance()` in `print_balances_and_total()` are not in transactions), then this program is racy.

10.8 Examples illustrating use of transaction_safe

A simple example explicitly declaring a function to be transaction-safe at its definition. This example is correct only if there is no previous declaration of `deposit`, or if the first such declaration is also explicitly transaction-safe.

```
void deposit(int x) transaction_safe {  // OK, deposit is transaction-safe
  atomic noexcept {
    this.bal += x;
  }
}
```

If a function is explicitly declared transaction-safe, this must (also) be included on the first declaration:

```
void foo();

void foo() transaction_safe {    // ERROR: must be on first declaration
  x++;                           //        if included at all
}
```

The next example illustrates that syncrhonized blocks are still allowed to be transaction-safe. Note that this example passes compilation and it is different from N3718, the first version of this proposal that was discussed in Chicago in September, 2013.

```
void foo() transaction_safe {
  ...
  synchronized {   // Passes in the current proposal; different then N3718
    ...
  }
}
```

10.9 Examples illustrating use of transaction_unsafe

A function declared transaction-unsafe cannot subsequently be declared transaction-safe.

```
[[transaction_unsafe]] void foo();

void foo() transaction_safe {   // ERROR: inconsistent declarations
   ...
}
```

A function declared transaction-unsafe cannot be called in an atomic block.

```
[[transaction_unsafe]] void foo();

void bar() {
  atomic_noexcept {
    foo();   // ERROR: foo is explicitly transaction_unsafe
  }
}
```

10.10 Examples illustrating "safe by default" (no error)

Functions (such as `foo()` in the following example) that do not include any transaction-unsafe code and do not call any functions that are not transaction-safe are implicitly transaction-safe. Furthermore, a function called within an atomic block (or an explicitly transaction-safe function) is assumed to be transaction-safe. Therefore, the following example compiles and links successfully.

```
common.h
--------

void foo();

file1.cxx
---------

#include "common.h"

void bar() {
  atomic_noexcept {
    foo();
  }
}

file2.cxx
---------

#include "common.h"

void foo() {
  // only transaction-safe stuff here
}
```

If `common.h` instead contained

```
void foo() transaction_safe;
```

the example would still compile and link successfully.

10.11 Example illustrating "safe by default" (error)

This example is similar to the one above, but the definition of `foo` contains transaction-unsafe code. As before, both files compile successfully. However, they do not link successfully, because compilation of `file1.cxx` assumed `foo()` to be transaction-safe, but its definition in `file2.cxx` is not.

```
common.h
--------

void foo();


file1.cxx
---------
```

```
#include "common.h"

void bar() {
  atomic_noexcept {
    foo();
  }
}

file2.cxx
---------

#include "common.h"

void foo() {
  printf("unsafe");              // transaction-unsafe due to I/O
}
```

If `common.h` instead included:

```
void foo() transaction_safe;
```

then compilation of `file2.cxx` would fail because `foo` contains something transaction-unsafe (I/O).
If `common.h` instead included:

```
[[transaction_unsafe]] void foo();
```

then compilation of `file1.cxx` would fail because `foo` is declared transaction-unsafe.

10.12 Examples illustrating transaction-safe function pointers

The following example illustrates combinations of transaction-safe and transaction-unsafe function and function pointers.

```
void (*fp)();
void (*tsfp)() transaction_safe;

void safefunc() {
  // nothing that is transaction-unsafe
}

void unsafefunc() {
  printf("Hello");
}

void bar() {
  fp = &unsafefunc;     // OK
  fp = &safefunc;       // OK
  tsfp = &unsafefunc;   // ERROR: can't assign transaction-unsafe function
                        //        to transaction-safe function pointer
  tsfp = &safefunc;     // OK
  fp = tsfp;            // OK: implicit conversion
  tsfp = fp;            // ERROR: can't assign transaction-unsafe function
                        //        pointer to transaction-safe one
```

```
  (*fp)();              // OK
  (*tsfp)();            // OK
  atomic noexcept {
    (*tsfp)();          // OK
    (*fp)();            // ERROR: call through fp transaction-unsafe
                        //        because fp is not transaction-safe
  }
}
```

10.13 Example illustrating function-local static initialization in atomic blocks

Consider this example:

```
std::pair<int,int>  f(int i) {
       static int x = i;
       static int y = i;
       return std::pair(x,y);
}

Thread 0                                Thread 1
--------                                --------

atomic_noexcept {                       f(1);
    auto r1 = f(0);
}
```

Because the transaction is an atomic block, it is not possible for `r1` to get (0,1): if Thread 0 initializes `x` to 0, then `f(1)`'s attempt to initialize `x` comes after Thread 0's transaction has completed, so Thread 1 does not initialize `y` before Thread 0 does (but see Section 6).

In contrast, of Thread 0 instead used a synchronized block, it *would* be possible for `r1` to get (0,1). This is because Thread 1's call to `f(1)` could start after Thread 0's synchronized block has initialized `x` but has not yet initialized `y`. There is no synchronization in the program to prevent this possibility, whereas the requirement for atomic blocks to be atomic does. (We note that concurrent, unsynchronized initialization of the same function-local static variable is explicitly not racy; see C++ standard section 6.7 paragraph 4.)

10.14 Examples illustrating virtual functions and overriding

A member function declared with a `transaction_safe` keyword or `transaction_unsafe` attribute in a base class preserves that attribute in any derived class, unless that member is redefined or overridden. Functions brought into a class via a `using` declaration preserve the attribute in the original scope. A virtual function of transaction-safe type must not be overridden by a virtual function of transaction-unsafe type.

```
struct B {
  virtual int f() {
    printf("not safe");  // not transaction-safe
    return 0;
  }

  virtual int g() transaction_safe {
    return 1;            // OK, transaction-safe definition
  }
```

```
  virtual int h() {
    return 1;            // not transaction-safe; must be explicit on
  }                      // virtual functions
}:

struct D : B {
  virtual int f() transaction_safe { // OK, transaction-safe override
                                     // of transaction-unsafe virtual function
     return 5;                       // OK, transaction-safe definition
  }

  virtual int g() {       // implicitly declared transaction-safe
     printf("not safe");  // ERROR: call to transaction-unsafe function
     return 0;
  }

  virtual int h() transaction_safe {   // ERROR: overridden virtual
                                       // function not explicitly
                                       // transaction-safe
     return 5;
  }
}:
```

The following example demonstrates that a function inherited from a base class remains transaction-safe; the same is true for transaction-unsafe.

```
struct B {
  void f() transaction_safe;
};

struct D : B {
  // when naming B::f through D, B::f stays transaction_safe
};

void g() {
  atomic {
    D d;
    d->f();   // ok, call to transaction-safe function
  }
}
```