Transactional Memory Support for C

Authors: 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
Victor Luchangco, victor.luchangco@oracle.com
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: N1961

Date: 2015-09-23

Project: Programming Language C,

Reply-to: Michael Wong, michaelw@ca.ibm.com (Chair of SG5)

Revision: 1

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 WG21 SG5 to introduce transactional constructs into C++ as a Technical Specification.

This document is based on N4514 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf) which has been approved to be published as a Technical Specification for C++. This proposal mirrors that approved draft in semantics and syntax suitably updated for C based on our design choices as outlined in Section 8.

That proposal (N4514) 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.

2 Overview

The merits of Transactional Memory can be observed from a reading of Section 1 of N3341 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf) which is the first paper that was presented to WG21 before the formation of SG5. Its subsequent development leading to a C++ TS can be charted from the sequence of papers with increasing number in the Reference Section. Even prior to that, IBM, Intel, Sun (Oracle), and HP along with many academics have gathered in 2008 fortnightly to discuss a common high language level specification. That Industry group was joined by Red Hat before it was moved into WG21 with the acceptance of N3341.

There have also been many implementations of TM in industry. The most similar to this proposal was implemented in GCC 4.7 with semantics that is nearly identical to that which was approved in the C++ TS, with a different keyword. In so doing, we are standardizing for C existing practice with very little invention. It is also the right time as industrial, and consumer hardware has become affordable in the market in the form of Intel Haswell, and IBM's Power8 and zEC12 mainframes.

The presence or absence of hardware does not affect this specification. This is a pure software-only specification which has been proven by GCC 4.7 (as well as the Intel STM and IBM Alphaworks TM compiler before that). The presence of TM hardware could enhance the performance of the software implementation, but it could also complicate it, because the handoff between the hardware and software, for when the hardware fails is non-trivial even in research.

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*.

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.

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.

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 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 { *body* }

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: *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: *This proposal provides "closed nesting" semantics for nested atomic blocks (*[1] For a description of closed nesting, see *Transactional Memory* by Harris, Larus and Rajwar, for example).

Use of 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., 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 keyword ,
-
- any of its parameters are declared volatile,
-
- its definition contains transaction-unsafe code as defined in Section 4.

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.

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.

6 Static Checking vs Dynamic Checking

A previous design allowed dynamic checking of transaction safety through Safe-by-default (SBD) where the implementation is allowed to generate two versions of functions for cases where they apply and is necessary: one a transaction-safe and another that is transaction-unsafe. The implementation is allowed to choose at link time and possibly discard the unused one depending on the facility supported.

When this design was reviewed by C++, some feel the SBD solution would be non-portable depending on the quality of linker (say on older VMS platforms), or whether full program analysis would be enabled, which might even depend on what optimization was turned on. This breaks the spirit of a Standard which is about portability. However, an implementation is always allowed to do more and enable optionally SBD.

Those C++ members who objected also offered a solution which would resolve their objection. In fact, this was an earlier design before SBD and is implemented in GCC 4.7 based on N3725: Original Draft Specification of Transactional Language Constructs for C++. This proposal reflects that Static Checking design.

Specifically, we will require explicit annotation for inline function, or an inline function if declared without a body if it's used before the definition, or "plain" extern functions.
But for all other cases do not need to be annotated.

7. _Optimizied_for_synchronized Statement

With Safe-by-default generating two versions, there was no need for this attribute. With static-checking back in the design, we require this attribute to bring back speculation in synchronized blocks.
As with much of our design, there is implementation experience as this is an attribute which has been implemented in the Intel STM compiler but is named transaction_callable.

There are some library functions that could not be made transaction-safe. Examples are assert, fprintf, perror, and abort. Consider this example from memcache:

```
store_item(){ // in thread.c
  ...
  _Synchronized {
    ret = do_store_item(...)
  }
  ...
}
do_store_item() { // in memcached.c
  ...
  if (...)
  else if (...)
  else if (...)
    if (...)
    else if (...)
    else
      ...
      if (settings.verbose > 1) //A
        fprintf(stderr, "CAS: failure: expected %llu, got %llu\n", (unsigned long long) ITEM_get_cas(old_it), (unsigned
long long)ITEM_get_cas(it));
    ...
}
```

All the conditionals are there to show that even if the verbose test in Line A passes (when verbose is 2 or higher), it is still a very rare case that fprintf will run. However rare, do_store_item() cannot be made transaction-safe because it contains an unsafe code path with fprintf. This means that when it is called from a synchronized block from within store_item(), (because do_store_item() is not transaction-safe) it must serialize before the call to do_store_item(), since there is only an uninstrumented version of the function.

This is unsatisfactory for most of the normal paths of this function. So a new optimization function attribute keyword is needed: _Optimized_for_synchronized. This indicates to the compiler that (a) do_store_item() might be called from a synchronized block in another compilation unit, and (b) the programmer thinks that the compiler should generate an instrumented version of do_store_item() even though it has unsafe code, because doing so would incur serialization only for those control flows in which the fprintf() is reached.

In effect, a function annotated with keyword _Optimized_for_synchronized may have irrevocable operations and legacy function calls inside its lexical and dynamic scope. Such a function may call other similarly annotated functions within its lexical and dynamic scopes, and may contain synchronized or atomic blocks. Further, a function that is annotated as such may contain indirect function calls and virtual function calls even if it is unknown at compile-time whether the target of a function pointer is also annotated as such.

A further example demonstrates its use:

```
// translation unit 1
_Optimize_for_synchronized int f(int);

void g(int x) {
  _Synchronized {
    ret = f(x*x);
  }
}

// translation unit 2
extern int verbose;

_Optimize_for_synchronized int f(int x)
{
  if (x >= 0)
    return x;
  if (verbose > 1)
    std::cerr << "failure: negative x" << std::endl;
  return -1;
}
```

If the attribute were not present for f, which is not declared transaction_safe, a program might have to drop out of speculative execution in g's synchronized block every time when calling f, although that is only actually required for displaying the error message in the rare verbose error case.

8. Design choices

a. Atomic blocks but not explicit cancellation

There is no explicit cancellation in this C specification, but the C++ specification has three constructs to enable handling of C++ exceptions and user-directed cancellation. It is prefaced with this statement and the following in *italic* which is lifted verbatim from N3999.

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.

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.

During a discussion within SG5 on how to support TM in C, we actually defined fairly clearly how to support user-directed cancellation in C by reusing the break keyword to indicate at the end of an _Atomic block whether to (commit or) cancel, similar to the return value of a function.

Example (not necessarily good syntax from Jens Maurer):

```
_Atomic {
  break _Atomic;   // new: will cancel and continue execution at the closing brace
  goto good;       // will commit, as will any other control flow out of the _Atomic block
}

good:    // plain old label syntax, nothing special, also works in C++
```

So in this formulation, any transfer within the block such as continue (if there was a loop) and goto inside the bock does not commit (as it is currently in the PDTS).

The new "break _Atomic" statement may only appear lexically within an _Atomic block. Thus, it's a syntax error if it appears in foo() (and foo itself doesn't have a nested transaction). So, in the C world, the compiler can see all points where a given transaction would be cancelled. It's the user's responsibility to transfer control to such a point if he
wants a transaction to be canceled (e.g. by using function return values).
Break is a good choice as it is in both language. Some might say it adds another usage for break statement but that seems consistent with its other uses, without inventing a new keyword.

The other alternative is to add a new keyword specifically to do this job.
_Atomic_cancel

Then whatever keyword we would add to C would also be proposed for C++ to make portability simler.

Having some form of cancellation in C would make its syntax/semantics parallel closely to C++. Explicit abort/cancel and is superficially very attractive. It has somewhat different weaknesses in C and C++.
It doesn't seem that useful for error recovery in either case, since most possible error sources tend not to be transaction-safe. In C we would no longer have the issues with parts of the exception object getting deconstructed on the way out, but the control flow (through goto statements) would look ugly, and after an error you would have to manually unwind the call chain back up to the transaction.
In both cases it's unclear to some SG5 members whether people will eventually find enough use cases to warrant the complexity. But the experiment may be worthwhile in C as well as a TS similar to C++ TS.

If there is no explicit cancellation, then the only difference between synchronized and atomic blocks in C is the static checking for the absence of serialization points. Is it worth having two different keywords for this purpose?

But the main argument for keeping atomic transaction is compile-time support for speculative execution. For the user, no atomic blocks mean they could be disappointed to find synchronized block do not scale and abandon Transactional Memory. But with atomic blocks, they may be disappointed by the heavy machinery for so little benefit.

In the end, we think C would benefit from both, so decided by a narrow margin to offer both constructs initially, but withhold explicit cancellation, until we have more data from the C++ TS implementation. The committee may wish to reverse this decision and ask to extend it now for full equality between C and C++ TM TS.

If explicit cancellation stays out of the first version of the TM TS or C, then in a future updated second version of this TS for C, we could consider adding cancellation. But we would have to think about some of the issues such as:

- if a function foo() was called from a transaction, and foo() used cancellation, then what would happen if the transaction did not expect such behavior? Would we need a viral annotation not just for transaction_safe, but also for transaction_safe_maycancel?
    - We would limit that by making it a syntax error if it appears in foo() (and foo itself doesn't have a nested transaction).
- an explicit cancel would have to abort only the innermost transaction, as in C++. This means that it requires open (not flat) nesting support, which we could otherwise avoid. This is because open nesting implementation is much more complex, and current C++ TM TS only requires closed nesting.

b. transaction_safe annotation implementation
Marking a function as transaction_safe, by using the keyword effectively creates two versions of a function: one that is transaction-safe and one that is transaction-unsafe. This means that compiler will need to create mangling for two different versions. C++ has name mangling to enable this, even though the annotation is through an attribute.

C adorns it as a keyword to the function and compiler must now take account of this keyword and add extra mangling to find the two variants of the function with the same name. We feel this is not a problem since compilers know how to do this, generate two variants of a function with the same name, or use whole program analysis to discard the unneeded version at link-time.

We wanted to specifically call out this to make sure implementers are aware of this note.

9. Summary

We propose transactional memory semantics and syntax for adoption into C initially as a Technical Specification(TS). The syntax and semantics mirror The Transactional Memory TS that has already been accepted by C++ suitably adapted for C. There has been compiler implementation experience with Intel's STM compiler, IBM's Alphaworks and GNU since 4.7. Hardware already exists in the form of Intel's Haswell, and IBM's Power8/zEC12

As far as we know, GCC will start switching to the TM C++ TS now that it is approved, and the switch in syntax might take only a few PMs, given that the semantics is 90% the same. Enabling it for C might take similar time (not official, just my assumption). Work is already beginning to add this capability to clang by the author and graduate student team under the direction of Michael Spear, one of the SG5 contributors. It is anticipated that other commercial compilers will follow, especially Intel and IBM to support their specific hardware.

10. 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 Jens Maurer, Dave Abrahams, Zhihao Yuan, Faisal Vali, Chandler Carruth, Lawrence Crowl, Olivier Giroux, Dietmar Kuehl, Jeffrey Yaskin, Jonathan Wakely, Eric Niebler, Detlef Vollmann, Ville Voutilainen, Nevin Liber, Bjarne Stroustrup, Herb Sutter, Clark Nelson, Bronek Kozicki, Tony Van Eerd, Steve Clamage, Sebastien Redl, Niall Douglas, Rajan Bhakta and many others whom we may have forgotten inadvertently to list.

11 References

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 (February3, 2012)*

N3859: Transactional Memory Support for C++
N3861: Transactional Memory (TM) Meeting Minutes 2013/09/09-2014/01/20

N3862: *Transactionalizing the C++ Standard Library*

N3919: Transactional Memory Support for C++
N3999: Standard Wording for Transactional Memory Support for C++
N4000: Towards a Transaction-safe C++ Standard Library: std::list
N4001: SG5: Transactional Memory (TM) Meeting Minutes 2014/02/03-2014/05/19
N4037: Non-Transactional Implementation of Atomic Tree Move
N4179: Transactional Memory Support for C++: Wording (revision 2)
N4180: SG5 Transactional Memory Support for C++ Update
N4181: Transactionalizing the C++ Standard Library Updates
N4182: SG5: Transactional Memory (TM) Meeting Minutes 2014/07/14-2014/10/06

N4265: Transactional Memory Support for C++: Wording (revision 3)
N4272: Working Draft, Technical Specification for C++ Extensions for Transactional Memory
N4301: Working Draft, Technical Specification for C++ Extensions for Transactional Memory
N4302: Technical Specification for C++ Extensions for Technical Specification for C++ Extensions for Transactional Memory
N4338: Editor's Report: Technical Specification for C++ Extensions for Transactional Memory
N4396: National Body Comments: PDTS 19841, Transactional Memory
N4410: Responses to PDTS comments on Transactional Memory
N4438: Industrial Experience with Transactional Memory at Wyatt Technologies.
N4441: SG5: Transactional Memory (TM) Meeting Minutes 2015-03-23 and 2015-04-06
N4488: Responses to PDTS comments on Transactional Memory, version 2
N4513: Working Draft Technical Specification for C++ Extensions for Transactional Memory
N4514: Technical Specification for C++ Extensions for Transactional Memory
N4515: Editor's Report: Technical Specification for C++ Extensions for Transactional Memory

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/.