Document Number: P0366R0
Date: 2016-05-30
Authors: Victor Luchangco, victor.luchangco@oracle.com
Michael Spear, spear@cse.lehigh.edu
Michael Wong, michael@codeplay.com
with other members of the transactional memory study group (SG5), including:
Hans Boehm, hans.boehm@hp.com
Brett Hall, bretthall@fastmail.fm
Jens Maurer, jens.maurer@gmx.net
Maged Michael, magedm@us.ibm.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Project: Programming Language C++, SG5 Transactional Memory
Reply to: Michael Wong, michael@codeplay.com(Chair of SG5)
Revision: 1

Extending the Transactional Memory Technical Specification with an in_transaction Statement

Abstract
Changes from previous versions
Introduction
Motivation
Alternatives
Discussion of Alternatives
Synchronized Blocks
Experience
Conclusion and Recommendation
Acknowledgements
References

Abstract

This paper explores various ways to extend The Technical Specification for C++ Extensions for Transactional Memory (TM TS) [1] proposed by *Study Group 5 (SG5): Transactional Memory* to allow code to dynamically determine and exploit whether it is executed within a transaction.

Changes from previous versions

N/A

1. Introduction

One of the challenges of exploiting transactional memory (TM) is that certain operations are impossible or expensive to execute within a transaction. The Technical Specification for C++ Extensions for Transactional Memory (TM TS) [1] addresses this issue by designating such operations as *transaction-unsafe* and forbidding these operations from being executed within (the dynamic extent of) an atomic block. To enable static checking of this restriction, the TM TS also requires that functions called within an atomic block by declared *transaction-safe*. Such functions must not execute any transaction-unsafe operations, and must not call any functions that are not transaction-safe.

Sometimes, a programmer may want a function that executes different code depending on whether it is called from within a transaction. The TM TS does not provide any mechanism for a function to determine whether it is called from within a transaction. In this paper, we describe some motivating examples for such functionality, propose a few different ways to provide it, and discuss some issues raised by providing this functionality and the different ways of providing it. For now, we say that code is executed within a transaction if it is executed within the dynamic extent of an atomic or synchronized block, but this is one of the issues we discuss later.

2. Motivation

One motivation for executing different code within a transaction than outside any transaction is to preserve the performance of nontransactional code, without restricting the programmer. For example, consider the `memcpy()` function, which performs reads and writes of memory, and hence ought to be safe to use within transactions. However, nontransactional implementations typically exploit in-line assembly to efficiently do mathematical (SAXPY computations), bitwise (population count), and memory (`memcmp`) operations, among others. Because these operations are typically ***not*** transaction-safe, we must use another (presumably less efficient) implementation when executing within a transaction. By allowing a function to execute different code paths for transactional and nontransactional contexts, we can deliver peak performance for nontransactional code, without sacrificing transaction safety.

A second motivation is to explicitly alter the behavior of a function when it is called transactionally. One use case we have identified is exceptional behavior: Within the context of an `atomic_cancel` block, instead of calling `std::terminate` or `std::abort`, we may wish to throw an exception that causes the transaction to abort and catch this exception outside the transaction, exploiting the "failure atomicity" semantics specified by the TM TS. Another example is to elide (transaction-unsafe) diagnostic messages when executing within a transaction.

A third motivation is to facilitate the eventual migration of legacy code (to include user code, as well as the STL and portions of the C Standard Library) to an optimal transaction-safe implementation. In this setting, a programmer might initially create a naïve transaction-safe implementation, to enable broad use of transactions, and then optimize those components piecemeal, as the need arises.

3. Alternatives

We now propose six variant ways to provide this functionality. In lieu of a formal specification, we illustrate these alternatives by showing how to implement a transaction-safe `memcpy()` function given the two functions `memcpy_safe()` and `memcpy_asm()`, which respectively implement `memcpy()` using only basic memory operations, and using in-line assembly and vector instructions. (All that is relevant is that `memcpy_safe()` is transaction-safe and `memcpy_asm()` is not, and that we want to execute the latter when possible, i.e., when not within a transaction.) Most of these alternatives propose a new keyword or library function, and the proposed names are intended only to be suggestive, not final.

First, we could use a new keyword `in_transaction` to introduce a lexical structure with two blocks providing the two alternative code paths. The `memcpy()` function could be written:

```
void* memcpy(void* dest, void* src, size_t n) transaction_safe {
  in_transaction {
    return memcpy_safe(dest, src, n);
  } else {
    return memcpy_asm(dest, src, n);
  }
}
```

Second, we could provide a `std::in_transaction()` library function that returns `true` when called from within a transaction, and `false` otherwise. Furthermore, when this function is used as a condition, a code path in which it is guaranteed to return `false` is allowed to contain transaction-unsafe code even if it appears within a transaction-safe function. Using this function, we could write our `memcpy()` function as follows:

```
void* memcpy(void* dest, void* src, size_t n) transaction_safe {
  if (std::in_transaction()) {
    return memcpy_safe(dest, src, n);
  } else {
    return memcpy_asm(dest, src, n);
  }
}
```

A third option provides the `std::in_transaction()` library function described above, except that it does not allow transaction-safe code to appear within a protected branch of a transaction-safe function. Instead, this functionality is provided by a block, introduced by a new `not_in_transaction` keyword, that explicitly asserts that it is not within a transaction. If such a block is executed within a transaction, `std::abort` is called. With this alternative, `memcpy`() looks as follows:

```
void* memcpy(void* dest, void* src, size_t n) transaction_safe {
  if (std::in_transaction()) {
    return memcpy_safe(dest, src, n);
  } else {
    not_in_transaction { return memcpy_asm(dest, src, n); }
  }
}
```

A fourth alternative is to provide a transaction-safe `std::transaction_select()` library function that takes two function objects as arguments and call the first if within a transaction and the second if not within a transaction. With this alternative, `memcpy`() looks as follows:

```
void* memcpy(void* dest, void* src, size_t n) transaction_safe {
  void *retval;
  transaction_select(
    ([]() { retval = memcpy_safe(dest, src, n); }),
    ([]() { retval = memcpy_asm(dest, src, n); }));
  return retval;
}
```

A fifth alternative, already present in GCC, is based on an idea originally proposed by Intel [2]. This mechanism, called "`transaction_wrap`", indicates that some `transaction_safe` function should be used in place of the original function when that function is called from a transaction. With this alternative, `memcpy`() can be written as follows:

```
void* memcpy(void* dest, void* src, size_t n) {
    return memcpy_asm(dest, src, n);
}

void* memcpy_safe(void*, void*, size_t) transaction_safe
[[transaction_wrap(memcpy)]];
```

The sixth alternative is similar, but uses overloading rather than the `transaction_wrap` keyword. That is, it allows a function to have overloaded definitions that differ only in whether the function is declared transaction-safe. A transaction-safe definition is considered more specific than one that is not transaction-safe, and it is applicable only when called from within a transaction. Thus, `memcpy`() can be written as follows:

```
void* memcpy(void* dest, void* src, size_t n) {
    return memcpy_asm(dest, src, n);
}

void* memcpy(void* dest, void* src, size_t n) transaction_safe {
  return memcpy_safe(dest, src, n);
}
```

4. Discussion of Alternatives

We briefly enumerate the strengths and weaknesses of these mechanisms below.

In the case of `transaction_wrap`, the main benefit is that the original (unsafe) code does not require any modification. Instead, a compilation unit can include a header that declares that certain `transaction_safe` functions should wrap their unsafe counterparts, and within that translation unit, the safe code will be called from any transaction or `transaction_safe` version of a function emitted by the compiler. This approach is also compatible with C. Its weaknesses include that it operates at the granularity of functions, and that does not require those functions to be co-located with the unsafe original. This can create maintenance problems for programmers, since two versions of code, in two separate files, can be reached by the same call site. Similarly, debuggers and profilers will need to be updated to understand the new mechanism. From a verification standpoint, since the wrapping code cannot be proven to have the same behavior as the original code, `transaction_wrap` could pose problems for verification.

The overloading variant shares these advantages and disadvantages with `transaction_wrap`, and it may be more natural both for programmers and to integrate tool support. Although it requires a change to the type system, the basic TM proposal already requires such a change to support the static checking of `transaction_safe` functions, so this additional change might not add too much more overhead. However, resolving the overloading implied by this variant differs from existing overloading resolution in that it depends on the context in which the call is made (i.e., whether it is within a transaction), not just the types of its arguments. There are analogies to weak linking, which may be a problem on some operating systems.

In contrast, the other four variants (i.e., the first four variants) all require modifying legacy code: To make a function `transaction_safe`, one must edit the function to add an `in_transaction` block, or use the `in_transaction` or `transaction_select` function. On the other hand, code maintenance should be simplified, since both versions of a function should be stored in the same file.

A weakness of the `in_transaction` block variant is that it requires the addition of a new keyword to the language. It may also require somewhat awkward or repetitive code, if, for example, we want to use the transaction-safe code path in some cases when executing outside a transaction.

By using a `std::in_transaction()` function, the second variant avoids these issues. However, this approach requires significantly more nuance in the compiler. Consider the following example:

```
foo = std::in_transaction();
if (!compute(foo))
  asm(...);
```

In this case, this code is safe if and only if `compute(foo)` is true whenever is `foo` is true (so that `asm(...)` is not executed in that case). In general, determining this is undecidable, so we must spell out a set of statically checkable restrictions that will allow the compiler to conclude that this code is transaction-safe. This requirement seems onerous. (We also observed that if we generate separate implementations of transaction-safe functions for executing within and outside of transactions, then within those separate implementations, this function is fixed, so we might be able to exploit the "constexpr if" mechanism. However, it wasn't clear how to do this, or if it makes sense considering that some implementations make choose a different approach.)

The third variant avoids this issue by moving the responsibility to the programmer to indicate that the code path is not within a transaction. However, it introduces a keyword.

The fourth variant removes the need for a keyword by using a library function that takes function objects to mimic the `in_transaction` block of the first variant. It doesn't address the possible repetitiveness of the code, and requiring the code in the two paths to be encapsulated in function objects can make the code clunky, but the uses of this mechanism might be sufficiently contained that this is not a serious issue. Another strength of this approach is that behavior will be easier to verify, since the version of the function called from any context is known at compile time. However, since lambdas are used, this approach may not be compatible with C.

5. Synchronized Blocks

One issue to consider that is relevant to all variants is whether code within a synchronized block (but not an atomic block) should be considered as executing within a transaction. On one hand, transaction-unsafe code is permitted within a synchronized block. On the other hand, executing such code will likely rule out speculative implementations that are part of the motivation for using synchronized blocks rather than traditional locking. One possibility is to leave it undetermined, so that programmers using this mechanism within a synchronized block must be prepared for either alternative.

There is some nuance to this situation. Consider the following:

```
void safe(...) transaction_safe { ... }
void unsafe(...) {
     ...;
     safe();
     ...;
}

synchronized {
     safe(...);
     unsafe(...);
}
```

In this example, a transaction-safe function (`safe`) is called twice from the same synchronized block. In the first case, the transaction may be running speculatively. In the second, a call to an unsafe (and uninstrumented) function (`unsafe`) is made, and it, in turn, calls `safe`.

Since `unsafe` is not instrumented, in the case of `transaction_wrap`, it must call the uninstrumented version of `safe`. Note that the function is not likely to be rewritten on account of transactions, so it would not contain a `transaction_select` call either. However, the first call to `safe` might use the instrumented version. With `transaction_wrap`, then, it appears that the most reasonable semantics (all calls to a function made within a synchronized block have the same behavior) can only be achieved by exclusively calling the uninstrumented version of `safe`. That, however, is likely to lead to poor performance (unsafe calls often result in serialization of transactions). It could be possible for programmers to avoid this behavior by using a nested `atomic` block, and calling `safe` from it. Such an approach is cumbersome, to say the least.

Note that the first four variants do not have this problem: whether `in_transaction()` is defined as returning `true` or `false` when called in a `synchronized` block, the behavior will still be consistent.

6. Experience

The `transaction_wrap` mechanism has been available in the GCC TM implementation since GCC 4.9. It has been used in a number of research projects, but we do not have evidence of its use in production code.

In the Wyatt STM, a library implementation of TM, a function similar to `std::in_transaction()` was used to manage (unsafe) timing code in functions called from within a transaction [3].

7. Conclusion and Recommendation

Further discussion and experience are required.

8. Acknowledgements

We wish to acknowledge and thank committee members and others who have given us valuable feedback.

9. References

1. Wong (ed). “Technical Specification for C++ Extensions for Transactional Memory”. N4514
2. Ni et al. “Design and Implementation of Transactional Constructs for C/C++”. OOPSLA 2008.
3. Hall. “Industrial Experience with Transactional Memory at Wyatt Technology”. N4438.