Document Number: P0231R0
Date: 2016-02-12
Authors: Victor Luchangco, victor.luchangco@oracle.com
Michael Spear, spear@cse.lehigh.edu
Michael Wong, fraggamuffin@gmail.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, LEWG
Reply to: Michael Wong fraggamuffin@gmail.com (Chair of SG5)

Extending the Transactional Memory Technical Specification to Support Commit Actions

Abstract

The Technical Specification for C++ Extensions for Transactional Memory (TM TS) [1] proposed by *Study Group 5 (SG5): Transactional Memory* was recently approved. This paper extends the Technical Specification to enable transactions to introduce code whose execution is deferred until *after* the transaction completes. Such *commit actions* are motivated both by real-world experience [2,3] and in the research literature [4,5]. This paper presents syntactic and semantic extensions to the Technical Specification to support commit actions in atomic and synchronized blocks.

Changes from previous versions

N/A

1. Introduction

This paper introduces new syntax and semantics for *commit actions*, which are operations whose execution is deferred until after an enclosing transaction[1] completes. It is the result of discussions within SG5 on extensions to the Technical Specification for C++ Extensions for Transactional Memory [1] (henceforth, the TM TS). This paper is based on both research proposals, and the real-world experience of Wyatt Technology [2].

[1] We use *transaction* to refer to the dynamic extent of an atomic block or synchronized block. (Atomic blocks and synchronized blocks are treated identically in this paper.)

There are a number of motivations for deferring operations until after a transaction commits. These include supporting user-defined transaction-safe allocators [4], enabling transactions to interact with condition variables [5], supporting error logging and debugging statements within transactions [3], and enabling transactions to call code that may have side effects [2]. In the current TM TS, some of these behaviors are supported through synchronized blocks. However, synchronized blocks that contain irreversible operations cause serialization: no other synchronized or atomic block can be executed at the same time as the block containing the irreversible operation.

In this paper, we describe an extension to the TM TS that allows arbitrary code to be deferred until the completion of the top-level transaction. Our focus is on describing the behavior and the syntax in intuitive terms. Core Standard wording will be reviewed in a separate paper.

A variant of commit actions has been available in the GCC transactional memory implementation since GCC 4.7. Its utility has been shown in a number of cases, including the implementation of transaction-safe condition variables [5] and the transactionalization of memcached [3]. Deferred operations were also used to build a runtime library supporting safe I/O and syscalls from transactions [6].

2. The std::transaction_defer mechanism

We propose a new function, std::transaction_defer, which takes a single argument: a function object (usually defined by a lambda expression). The behavior of std::transaction_defer depends on the context. When called from outside of transaction, std::transaction_defer executes its argument (the function object) immediately. When called from within a transaction, the execution of the function is deferred until after the transaction completes. More precisely, the execution of this function is sequenced after the outermost transaction and before any other code that is sequenced after the outermost transaction.

If a transaction defers several operations, they are all sequenced after the outermost transaction completes, and they execute in the order implied by the dynamic order of their std::transaction_defer calls. That is, all deferred functions are accumulated at the level of the outermost transaction, and are executed in the order they are deferred, with each operation being sequenced before the operation that follows it in this order. Each deferred operation must be completed before the next operation in this order is begun.

Note that there is no restriction on the number of calls to std::transaction_defer that can be made within a single transaction, nor on the operations that are allowed within the function passed to std::transaction_defer.

3. Examples and Pitfalls

Some subtle cases arise because std::transaction_defer may be called within nested transactions and within transactions executed by deferred functions. We give several examples to illustrate

the correct behavior in these cases. In these examples, we assume single-threaded code, so that accesses to variable from transactional and nontransactional contexts do not result in data races.

First, a deferred function is not executed until the outermost transaction completes. Thus:

```
int x = 0;
atomic {
   atomic {transaction_defer([]{x++;});};
   x = 6;
}
// x == 7;
```

Second, functions deferred within a single (outermost) transaction are executed in the order in which they are passed to std::transaction_defer, regardless of whether they are called from within a nested transaction:

```
atomic {
   transaction_defer([]{printf("A");});
   transaction_defer([]{printf("B");});
   atomic {
       transaction_defer([]{printf("C");});
   }
   transaction_defer([]{atomic{transaction_defer([]{printf("D");});}});
   transaction_defer([]{printf("E");});
}
// output: ABCDE
```

Third, if transaction A defers calls to functions B and C, function B uses a transaction internally, and B's transaction defers a call to function D, then the execution of D must complete before C begins because the execution of D after B's transaction is part of the execution of B:

```
int x;
atomic {
   transaction_defer([]{x = 5;});
   transaction_defer([]{atomic{transaction_defer([]{x *= 10;});};});
   transaction_defer([]{x += 5;});
}
// x == 55
```

As with other uses of lambdas, we must take care when accessing variables captured by reference in lambdas that define deferred operations. For example, if the lambda captures (by reference) a variable that is local to the transaction, then the behavior of any access to that variable is undefined because the lambda is deferred until after the block completes, at which point the variable is out of scope. This is analogous leaking the address of a stack variable. To avoid this problem, the programmer could capture the variable by value, or, if that is not possible, allocate a new region on the heap, and copy the value to that region and capture the

newly allocated region by reference. Then the lambda can access this region and free it afterwards, without concern that the region has gone out of scope.

Also, if a deferred operation is defined by a lambda captures by reference a shared variable that is not local to the transaction, then access to that shared variable must be synchronized to ensure that there are no data races because the deferred operation is executed outside any transaction. For example, to print a value, the lambda could use a transaction to copy the value to a local buffer, and then print the buffer, or it could use a synchronized block to print the value.

4. Implementation Issues

We can think of calls to std::transaction_defer as appending elements to a list whose elements (i.e., the functions passed to std::transaction_defer) will be processed when the outermost transaction completes. Since there is no limit on the number of calls to std::transaction_defer within a transaction, an implementation must not assume a fixed bound on the number of entries in such a list.

The lack of restrictions on code that is deferred means that an implementation must support unbounded recursive deferred operations. For example, the following code is legal:

```
void a(int x) {
   atomic {if (x > 0) transaction_defer([]{b(x-1);});} }

void b(int x) {
   atomic {if (x > 0) transaction_defer([]{a(x-1)});}
}
```

5. Acknowledgments

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

6. References

1. Wong (ed). “Technical Specification for C++ Extensions for Transactional Memory”. N4514
2. Hall. “Industrial Experience with Transactional Memory at Wyatt Technology”. N4438.
3. Ruan et al. “Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached”. ASPLOS 2014.
4. Ni et al. “Design and Implementation of Transactional Constructs for C/C++”. OOPSLA 2008.
5. Wang et al. “Transaction-Friendly Condition Variables”. SPAA 2014.
6. Volos et al. "xCalls: Safe I/O in Memory Transactions". EuroSys 2009.