

**Document Number:** P1875R0

**Date:** 2019-10-06

**Target:** SG5, SG1

**Authors:** Michael Spear, Hans Boehm, Victor Luchangco, Michael Scott, Michael Wong

**Reply to:** [spear@lehigh.edu](mailto:spear@lehigh.edu), [hboehm@google.com](mailto:hboehm@google.com)

**Proposed ship vehicle:** Technical Specification

# Transactional Memory Lite Support in C++

## Abstract

We propose a transactional memory facility that is simpler and easier-to-implement than [N4514](#), the current “Technical Specification for C++ Extensions for Transactional Memory”. Our goal is to make it easier to experiment with a conforming facility, and thus to gain more confidence in this feature for eventual inclusion in the standard. The basic semantics are similar to the current TS, but the facility is less intrusive, and has been reduced to a single type of transaction. The required implementation effort is almost entirely determined by the desired performance; a minimal low performance implementation is trivial.

## 1. Motivation

SG5 was formed to consider how to support programs that use *transactional memory* (TM), a mechanism proposed to make it easier to write concurrent programs. The urgency of this task was increased by the advent of hardware support for transactional memory in commercial processors. The [Technical Specification for C++ Extensions for Transactional Memory](#) (TM TS) was approved in 2015, but it has not been widely implemented and used. Some have expressed concerns about its breadth, and questioned whether all features proposed therein are necessary. There is interest in having some support for transactional memory, even if it covers fewer use cases. A more incremental approach to adding transactional memory support is likely to be more effective. In this spirit, we propose a “lite” version of support for transactional memory. The idea is to support some important use cases with as small an addition to the language as possible.

## 2. Proposal

We propose the addition of a single construct, the atomic block. Syntax: `atomic { ... }`, presumably employing `atomic` as a contextual keyword.

The execution of code within an atomic block is called a transaction. Transactions are guaranteed to appear atomic with respect to other transactions. Specifically

1. Two transactions *conflict* if they access the same location and at least one of them writes that location.
2. If transactions A and B conflict, then either the end of A inter-thread happens-before the beginning of B or the end of B inter-thread happens-before the beginning of A.

Transactions never form data races with other transactions. However, there is no implied synchronization between transactions and nontransactional code, other than what may be implied by other aspects of the C++ specification. In particular, logically concurrent access to any location by transactional and nontransactional code is considered a data race (unless both accesses only read the location), and thus results in undefined behaviour..

An implementation must specify which kinds of operations are supported (permitted to occur) in the dynamic extent of a transaction. At a minimum, this must include: ordinary (non-atomic, non-volatile) reads and writes, ordinary (non-exceptional) control flow, and calls to `constexpr` functions (even if their arguments are not constants), and to functions defined in the current compilation unit whose bodies would themselves be permissible inside an atomic block. Nested atomic blocks are also allowed: they behave as if the inner occurrence of the `atomic` keyword were elided. Exceptions are not allowed to escape transactions: if executing an atomic block results in an exception that is not caught within the dynamic extent of the transaction, the behavior of the program is undefined. Similarly, the behavior of a program that executes an unsupported operation within the dynamic scope of a transaction is undefined.

An implementation may support (i.e., guarantee atomicity for) additional kinds of operations within transactions. Implementations are encouraged but not required to provide static and/or run-time warnings for programs that perform unsupported operations.

### 3. Syntactic Alternatives and Prototype Implementations

While TM support can be most directly expressed via the addition of lexically scoped transactions as a language feature, we have also considered two implementation alternatives. These have the advantage that they do not require any changes to the compiler front end, and make it easy for programmers to begin prototyping both (a) TM implementations, and (b) TM-based applications. We describe and compare these alternatives below. Both have been implemented as LLVM plugins, and are available for download from <http://github.com/mfs409/to-be-determined>.

#### 3.1 A Lambda-Based Execution Framework

The first alternative is a lambda-based execution framework for running transactions. With this framework, a programmer requests that a block of code be run as a transaction by wrapping it in a lambda expression, and passing it to a special TM library. Instead of `atomic { ... }`, we anticipate the syntax looking something like the following::

```
std::tm_exec([&]{ ... });
```

The strengths of this approach include:

- No changes are required to the front end of the compiler.
- For systems with hardware TM support, it is possible to implement `std::tm_exec` entirely as a library: `std::tm_exec` can (1) start the hardware transaction, (2) execute the lambda, and then (3) commit the hardware transaction, falling back to a hidden global reentrant mutex lock in the event of repeated failure.
- For systems without hardware TM support, it is possible to implement `std::tm_exec` by serializing all transactions on a single unnamed global reentrant mutex lock. Such an implementation would not provide scalability, and would unexpectedly allow system calls and accesses to atomic variables in transactions. This would be undesirable because it cannot be supported by more interesting implementations, and would not be easily diagnosable. However, in the interest of making this as easy as possible to implement, we do not want to preclude such implementations, in the initial TS phase..
- For systems that desire to add software TM support, the lambda naturally allows a compiler to capture and instrument accesses to memory in the local scope, and to track/detect invalid operations, such as calls to functions outside the translation unit, accesses to `volatile` and `atomic` variables, etc.

The weaknesses of this approach include:

- Lambda creation and management can introduce run-time overhead.

- Since the transaction body is a separate function, it cannot access `varargs` of its parent scope.
- The syntax of using a lambda is less elegant than an `atomic` keyword.

A full implementation of this framework is available at <https://github.com/mfs409/llvm-transmem/tree/sg5-lambda-stm>. The main implementation is an LLVM plugin that supports several software TM algorithms, as well as hardware TM and serialization of transactions on a reentrant mutex lock. Within that repository, there are also two “lite” implementations, which are library-only, corresponding to a system that serializes all transactions on a reentrant mutex, and a system that uses hardware TM (with fall-back to a mutex).

### 3.2 An RAII Framework

The second alternative is to use the “resource acquisition is initialization” (RAII) idiom to mark transaction bodies. That is, instead of writing `atomic { ... }`, a programmer would write:

```
{std::transaction_scope ts(); ... }
```

Most of the merits of this approach are the same as for the lambda approach. In particular:

- No changes to the front end of the compiler are required.
- For systems with hardware TM support, and in systems that choose to serialize all transactions on a global unnamed reentrant mutex lock, TM support can be achieved entirely in a library. Starting a transaction is performed in the `std::transaction_scope` constructor. Committing the transaction is performed by the destructor.

In addition, the first two weaknesses of the lambda approach are avoided. That is:

- There is no overhead for lambda creation and management.
- All variables of the parent scope are accessible, even those (like `varargs`) that cannot be captured by a lambda.

The weaknesses of this approach include:

- Supporting software TM requires more complexity than the lambda approach, since instrumentation must be performed at a finer granularity than function scope.
- The compiler instrumentation for supporting software and hybrid TM is more complex than the lambda approach..

We plan to make available a repository containing an implementation of this framework. The majority of the code is identical to the lambda implementation discussed above. There are also “lite” implementations, as with the lambda implementation.

### 3.3 Summary of Alternatives

Below, we compare the three implementation options:

	Lexical Scope	Lambda	RAII
Front-End Modifications	Yes	No	No
Back-End Modifications for Hardware TM	No	No	No
Complexity of Software TM Back-End	High	Low	Medium
Limits on Transactions	No	Yes (e.g., <code>varargs</code> )	No

Elegance	High	Medium	Low
Static Checking	Required (for control flow)	Optional	Optional
Supports Library-Only Implementations for Fast Prototyping	No	Yes	Yes
Implementation Status	Not started	Complete	In-Progress

#### 4. Other Alternatives Considered

We considered various ways to limit the size of a transaction, to make it easier to guarantee the desired atomic behavior efficiently, particularly with hardware transactional memory. For example, we could limit the number of locations that a transaction can access (i.e., the size of its read and write sets). We could restrict control flow to avoid large, possibly infinite loops. However, we decided to keep the rules simple and leave it to the programmer and/or implementation to avoid performance issues that might be introduced by, for example, transactions that always fall back to locks because they are too large to complete successfully on a particular hardware transactional memory implementation.

We considered expanding the kinds of operations that transactions must support. In particular, we gave consideration to letting transactions access atomics. We decided against requiring atomics support because it could impose a cost on programs that use atomics but not transactions.

We discussed requiring that compilers reject programs that contain volatile accesses, atomic accesses, syscalls, escaping exceptions, and other “bad” things, but decided against it because it is possible that a programmer will know that those things never happen at run time. For example, code containing such operations might occur only within conditionals that never trigger in the dynamic scope of a transaction. In addition, we did not want to rule out implementations that support such features.

We discussed explicitly allowing transactions to call functions defined in other translation units, but decided against it because it might require changes to the ABI (e.g., transactional and nontransactional copies of the bodies of separately compiled functions) and to the type system (making “transaction safety” a “viral” property).

#### 5. Implementation Issues

This proposal is intended to admit a wide variety of implementations. At one extreme, an implementation (e.g., one focused on the use of hardware transactional memory) might support only the bare minimum of required operations inside transactions, and refuse to compile any program with an atomic block that has an unsupported operation in its body or in the body of any function it calls, directly or indirectly. At the other extreme, an implementation (e.g., one focused on ease of implementation within the compiler) might acquire a single, anonymous global lock at the beginning of each atomic block, release that lock at the end, and permit arbitrary (otherwise well defined) code to be executed in-between.

To facilitate experimentation with this proposal, we have made a prototype implementation available as an LLVM plugin. The code is available under <https://github.com/mfs409/llvm-transmem>. Details of the implementation are available at <http://www.cse.lehigh.edu/~spear/papers/zardoshti-taco-2019.pdf>.

Our implementation is an LLVM plugin that uses the “lambda” option from Section 3.1. The plugin is a .so file. To use the plugin, simply compile code with clang++, and include the flags `-Xclang -load -Xclang $(PLUGIN_SO)`. We provide several software and hardware-accelerated TM algorithms. To link to an algorithm, include the appropriate library in the list of files to link.

The plugin searches the translation unit for lambdas passed to `tm_exec()`. For each, it finds all functions reachable from the lambda. For each function it creates a clone. In each clone, it replaces all loads/stores with library calls for performing the equivalent load or store. It also replaces each function call in a clone with a call to the corresponding cloned function. Upon encountering a forbidden instruction, the plugin inserts a preceding call to serialize all transactions. This has the effect of reducing behavior to that of a program in which all transactions are protected by a single global lock.

The plugin also clones the body of the lambda, and inserts a branch as the first instruction. The cloned body is instrumented the same way as reached functions. When the program is linked against a hardware TM implementation, for which instrumentation is not needed, the lambda branches to the original (uninstrumented) code. When it is linked against a software TM, `tm_exec()` will direct the lambda to run the instrumented code.

The TM libraries provide correct instrumentation for loads and stores. They also log calls to `malloc` and `free`; frees are deferred until transactions commit, and mallocs are freed if a transaction aborts. When a transaction starts in HTM mode, it does not use the instrumented path. Instead, it branches once in the lambda and then takes the original uninstrumented path. We also provide some “hybrid” TMs, which attempt to use hardware TM but can fall back to software TM.

Our implementation includes numerous additional features, to include support for calls to functions in separate translation units. Despite these additional features, the entire LLVM plugin is on the order of 2K lines of commented code, and TM libraries are typically under 800 lines of commented code. The support for TM does not require changes to any of the other parts of the compiler. At compile time, all transformations have worst case overhead linear in the number of instructions in each function. At run time, the overheads are (1) a branch in the lambda, (2) the cost of calling a lambda, and (3) function call overhead for reaching the instrumentation. Note that cost #3 is not incurred for hardware TM. While the plugin supports features not discussed in this document, they will not introduce run-time overheads for programs that exclusively use TM as described in this document.

## 6. Future Directions

This proposal is explicitly intended to be limited, and to avoid design choices that preclude extending the functionality to allow greater flexibility and expressivity for transactions in the future.

## 7. Straw Polls to be Proposed

Is this simpler approach to transactional memory in C++, which allows for library-only implementations, preferable to the more robust proposal from 2015 ([N4180](#))?

Among the three main options (keywords, lambda, RAII) for declaring a transactional scope, which is best?

## 8. FAQ

### Can I use parts of the STL inside transactions?

STL functions that are `constexpr` are guaranteed to work in transactions. Compatibility of all other parts of the STL is implementation-defined.

### Will this allow me to make good use of hardware transactional memory?

While this depends on the implementation and on the underlying hardware, yes, it is very much our intent to make hardware TM easily accessible in C++ (with fallback, e.g., to a global lock, as is conventional on “best effort” TM hardware).

### What are the limits on hybrid or software transactional memory algorithms that can be used with this language support?

The proposed language support assumes “word-based” (byte-based) transactional memory that maintains a single version of program data. It does not forbid nonblocking, hybrid, or software transactional memory algorithms as long as they are word-based and single version.

**What does this proposal abandon that the earlier proposal supported?**

Compile-time guarantees of transaction safety (hardware TM compatibility) for `atomic` blocks.

The guaranteed ability to call functions of other compilation units from within `atomic` blocks.

The option to allow escaping exceptions to cancel `atomic` blocks.

The lock-like semantics of `synchronized` blocks, which were guaranteed to support a richer variety of operations inside transactions.