Document Number	P1875R2
Date:	**2021-03-14**
Target:	EWG, SG1
Authors:	Michael Spear, Hans Boehm, Victor Luchangco, Jens Maurer, Michael L. Scott, Michael Wong
Reply to:	**spear@lehigh.edu, hboehm@google.com**
Proposed ship vehicle	Technical Specification
Wording proposal	P2066

Transactional Memory Lite Support in C++

Abstract

We propose a transactional memory facility that simplifies N4514, the current "Technical Specification for C++ Extensions for Transactional Memory". Our goal is to make it easier to implement and 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 conforming implementation is easy to implement.

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 gain more traction with compiler writers and programmers. In this spirit, we propose a "lite" version of support for transactional memory (TM-Lite). The idea is to support some important use cases with only a small addition to the language.

2. Proposal

We propose the addition of a single construct, the atomic block. Syntax: `atomic do { ... }`, 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 non-transactional 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 non-transactional code is considered a data race (unless both accesses only read the location), and thus results in undefined behavior.

An implementation must specify which kinds of operations are supported (permitted to occur) in the dynamic extent of a transaction. At a minimum, these must include: primitive arithmetic operations on scalar data types, ordinary (non-`atomic`, non-`volatile`) reads and writes, local control flow (including exceptions caught within the transaction), and calls to (a) inline functions with reachable definitions and functions defined in the current compilation unit, provided their bodies would be permissible inside an atomic block, and (b) standard library functions other than for threads, I/O, and atomics. The core challenge in providing this support is ensuring the correctness of allocation and deallocation within transactions. (These were not included in the minimal required set of operations in the initial version of this document.) Nested atomic blocks are also allowed: they behave as if the inner occurrence of `atomic do` were elided. Exceptions are not allowed to escape transactions (including nested 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. The behavior of a program that executes an unsupported operation within the dynamic scope of a transaction is implementation-defined.

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 also considered two implementation alternatives. These alternatives have the advantage that they do not require any changes to the compiler front end, and can be implemented as libraries. Thus, they 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 in the same LLVM plugin, which is available for download from http://github.com/mfs409/llvm-transmem/.

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 `std::mutex` in the event of repeated failure.
- For systems without hardware TM support, it is possible to implement `std::tm_exec` by serializing all transactions using a single unnamed global `std::mutex`. Such an implementation would not provide scalability.

However, in the interest of making TM 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 `atomic do`.

As noted above, this approach to TM has been implemented as an LLVM plugin that supports several software TM algorithms, as well as hardware and hybrid TM, and serialization of transactions on a reentrant mutex lock. Within the repository, there are also two library-only implementations, 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 do { ... }`, 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`, 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 in 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 introduces more complexity than in the lambda approach, since instrumentation must be performed at a finer granularity than function scope.
- The compiler instrumentation to support software and hybrid TM is more complex than in the lambda approach.
- As pointed out in SG1 discussions, RAII raises the question of what the semantics are when a `transaction_scope` is declared with other than automatic storage duration.

Our LLVM plugin provides implementations of this alternative analogous to those of the lambda alternative. Full support for hybrid and software TM adds approximately 600 lines of code to 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	Medium	Low	Medium
Additional limits on Transactions	No	Yes (e.g., `varargs`)	No
Static Checking	Required (for control flow)	Optional	Optional
Implementation Status	Not started	Complete	Complete

Following consultation with SG1 (see Section 7), we believe that adding a context-sensitive keyword is preferable to either of these alternatives. Nonetheless, these alternatives may help programmers begin experimenting with the ideas in this TM-Lite proposal until such time as a context-sensitive keyword is integrated into a compiler front-end. We expect that code in the RAII style can be converted to use the context-sensitive keyword using simple find-and-replace.

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 enlarging the set of operations that transactions must support, in particular, allowing transactions to access `atomic` variables. We decided against requiring support for `atomic`s because it was unclear how to avoid imposing a cost on programs that use `atomic`s but not transactions.

We discussed requiring that compilers reject programs that contain `volatile` accesses, `atomic` accesses, syscalls, escaping exceptions, and other "bad" things. We decided against imposing this requirement because a programmer may 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 extend the specification with experimental support for 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 non-transactional copies of the bodies of separately compiled functions) and to the type system (making "transaction safety" a "viral" property).

We also discussed requiring implementations to support calls to other translation units by falling back to a global `mutex`. Although this may eventually turn out to be preferable, we felt that it was more important at this stage to allow as broad a range of implementations as possible, including those that require instrumenting all memory accesses inside a transaction.

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 the body of any function it calls, directly or indirectly. At another 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.

As noted above, we have made a prototype implementation available as an LLVM plugin to facilitate experimentation with this proposal. 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 offers both the "lambda" and RAII options described in Sections 3.1 and 3.2. The plugin is a .so file. To use the plugin, simply compile code with clang++, and include the flags `-Xclang -load -Xclang /path/to/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()`, or for calls to the RAII transaction object constructor. It finds all functions reachable from the lambda or from the scope of the object. For each function, it creates a clone, and replaces all loads and stores in the clone with library calls to perform the equivalent load or store. It also replaces each function call in the 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. It would be easy to modify the plugin to instead reject such programs.

For the lambda API, the plugin also clones the body of the lambda, and inserts a conditional test to choose between the two clones at the beginning. The cloned body is instrumented the same way as reached functions. For the RAII API, the plugin clones the control flow subgraph between the transaction object constructor and destructor, instruments the cloned graph, and inserts an initial conditional test. In either case, when the program is linked against a hardware TM implementation, for which instrumentation is not needed, the conditional tests direct execution to the original (uninstrumented) code. When the program is linked against a software TM, or when fall-back occurs at run time, the conditional tests direct execution to the instrumented code.

The TM libraries provide correct instrumentation for loads and stores. They also log calls to `malloc` and `free` (which are invoked by `new` and `delete`); `free`s are deferred until the transaction commits, and `malloc`s are freed if the transaction aborts. When a transaction starts in HTM mode, it does not use the instrumented path. Instead, it branches once 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, most notably support for calls to functions in separate translation units. Despite these additional features, the entire LLVM plugin is only about 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 when the transaction begins, (2) the cost of calling a lambda (lambda API only), and (3) potential 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, these features do not introduce run-time overheads for programs that exclusively use TM as described in this document.

6. Supporting the Standard Library

A goal of this proposal is to provide a lightweight transactional memory feature that is broadly usable. For example, we would like programmers to be able to use (most of) the standard library within atomic blocks. To support this, it is necessary to permit two advanced operations in atomic blocks. First, atomic blocks must allow programmers to perform allocation and deallocation operations. Second, atomic blocks must be allowed to catch the exceptions that they throw.

In this section, we discuss these requirements in more detail. For brevity, this section uses the term "allocation" to refer to any call to `new`, `delete`, or the equivalent C functions (such as `malloc` and `free`).

Without support for allocation, much of the standard library is incompatible with atomic blocks. Consider `std::vector`: since `push_back` *might* result in an allocation, it cannot be called from an atomic block if allocation is unsafe. Furthermore, while other operations such as `empty`, `size`, `max_size`, and `capacity` may be called safely from an atomic block, they cannot be called if a race would result. This means, for example, that a thread cannot call `push_back()` while another is using an atomic block to call `empty()`: the threads would potentially access the same internal state of the `vector` while relying on different concurrency control mechanisms. Note, too, that one cannot simply implement `push_back()` with atomic blocks, as doing so would most likely limit the instantiations of the container to types whose copy constructor is safe to call from an atomic block. In addition, allocation can result in an out-of-memory exception being thrown. Thus, supporting allocation within a transaction requires some support for exceptions within transactions.

6.1 Correctness Concerns

There are valid concerns about allocation and deallocation in atomic blocks. As an example, consider a program in which one thread executes:

```
atomic do { for (int i = 0; i < 100; ++i) a[i] = new Object(); }
```

while another thread executes:

```
for (int i = 0; i < 100; ++i) b[i] = new Object();.
```

Potential Race: We have not defined the implementation of `the allocator`. One might think that unless both new and delete are implemented as transactions, they would participate in races when called from atomic blocks that utilize the standard library.

Observable Interleavings: If `new` and `delete` are not implemented as transactions, then the allocation calls in the second thread can interleave with the allocation calls in the first, whereas a programmer could otherwise expect all of the calls of the first thread to appear to happen without interleaving with operations of any other thread. In a memory-starved environment, this can result in the second thread inferring the state of the first when a call to `new` fails. In general programs, it can result in the second thread inferring the state of the first by observing that its sequence of calls to `new` return different addresses than they would if the first thread executed as a single indivisible (“atomic”) operation.

System Call: An implementation of `new` may perform a system call in order to increase the program's virtual address range. These system calls are not supported by hardware implementations of TM. Absent hardware TM, such system calls will still produce visible changes to the process state, which would not be “atomic” with respect to the remainder of the atomic block.

Exceptions: A call to `new` by the first thread might throw an out-of-memory exception, and TM-Lite does not require atomic blocks to support exceptions.

We argue that these concerns do not compromise correctness. First, note that the implementation of allocator functions is defined to be thread-safe, but the thread safety mechanism is not specified. Thus, simultaneous calls to the allocator from atomic blocks and other code do not, on their own, constitute a race.

Second, the specification of `new` and `delete` does not deal with their implementation, and a variety of alternative allocators can be swapped, at run time, in most C++ programs. This means that concerns about observable interleavings are invalid: a correct C++ program cannot make assumptions about the return value of the allocator.

Third, while a potential allocation-related system call can produce changes to the process state, that state is not defined by C++. Thus, a program whose behavior relies on the process's allocation state is not standards-compliant. As a practical matter, the standard does not prevent an allocator implementation from using a background thread that modifies memory

mappings or other process state, so this behavior is no different from what should be expected in any case. Allocator-related system calls thus do not violate atomicity for standards-compliant C++ programs.

Fourth, while exceptions that escape an atomic block are intentionally forbidden, it is easy to support exceptions that are thrown and caught within the same atomic block: Since the proposed specification allows an atomic block to serialize at any time and for any reason, it is sufficient to serialize the block when an out-of-memory error is thrown, and then use the existing exception-handling support of the language implementation. As with the allocator, the implementation of exceptions is specified as being thread-safe, without any restriction on the mechanism for doing so. Likewise, while high-performance implementations of exceptions support may change the process state (e.g., by requesting that the loader extract symbols from the executable file), programs that rely on such behavior are not standards-compliant.

6.2 Performance Concerns

An important consideration is how support for allocation in atomic blocks will affect the performance of existing code. In particular, it is widely understood that high-performance implementations of atomic blocks will rely on some manner of speculative execution. This means that an allocation or deallocation may need to be "undone", which could impact the behavior of the allocator.

First, we observe that in hardware-accelerated implementations of TM-Lite, this issue does not arise: the hardware can "undo" an allocation or deallocation with no effort from the language implementation. Implementations that always serialize atomic blocks on a single lock are also trivial, since they never undo operations. By the same token, a hardware implementation that falls back to serializing with a global lock is not an issue. Performance concerns arise only for implementations that rely on software-based speculation (i.e., software or hybrid transactional memory).

A software or hybrid TM implementation could simply serialize upon any allocation or deallocation. This would not lead to good performance, however. A more realistic strategy, which is well explored in the research community and present in the existing TM TS, is to log allocations and defer deallocation until the commit point. If a speculation fails, the allocations are undone (this can change the return value of subsequent allocations, but as discussed previously, correct C++ programs cannot reliably reason about non-null returned values from the allocator). If an atomic block completes, then all deallocations are performed as part of the completion operation.

This strategy is correct and safe in general. However, it can result in performance pathologies, and become a quality of implementation issue. The canonical example is a single atomic block that has a high incidence of allocations and deallocations. An equivalent lock-based program could satisfy some of the allocations using memory that it previously deallocated, but naive deferral would not allow such optimization. Existing studies of "captured memory" have shown that an implementation can overcome this problem. Alternatively, a TM-Lite implementation could serialize allocation-heavy atomic blocks, while using deferral for typical atomic blocks.

6.3 How much of the standard library do we support?

Many standard library calls are relatively easy to support in any TM implementation style, since they are implemented entirely in headers, and thus fully visible for transformation by the compiler, if the implementation requires it. But other functions may be implemented in a separate translation unit. In a hardware-based implementation, usually with a global lock fallback, this causes no issues, since there is no need to instrument the separately-compiled code. However, this raises questions for a speculative, pure-software implementation.

Such implementations may still support such separately-compiled calls by falling back to a global lock when a transaction enters uninstrumented code. This has the disadvantages that the compiler must support such a fallback, and the fallback may introduce unexpected scalability issues. It also raises questions about whether the specification should require support of e.g. function pointers, when the use of library facilities like `std::function` provides similar facilities.

SG5 has gone back and forth on this issue. For the implementation styles we expect to be most popular (HTM, usually with single global lock fallback), it doesn't matter, but for others it might be important.

Our current position is to impose relatively weak restrictions on standard library use, but to otherwise keep support of language constructs like function pointers optional. This means that the compiler may not have to deal with the fallback for calls to separately compiled code, and it could be handled by the library. We hope that experience with the TS will help to determine whether this is a sensible position. It does have the advantage of exposing the tension here, but we do not have confidence this decision will stand in the long run.

6.4 Summary

Most of the arguments in this section are supported by our experience implementing a prototype TM-Lite implementation in LLVM. In particular, we found that serializing an atomic block that throws, and rejecting only those throwing atomic blocks that allow an exception to escape, did not have a significant impact on the complexity of the implementation. Likewise, it was easy to support logging of allocations and deferral of deallocation for software TM implementations. With these two changes in place, most of the C++ standard library (most notably strings, containers, iterators, and ranges) can be used in the prototype implementations of atomic blocks, even those that do not extend the compiler back end.

7. Future Directions

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

8. SG1 discussion

SG1 discussed the three different options for the syntax of atomic blocks. By a margin of 8-3, the atomic block language-based language syntax was favored over the lambda-based library approach (there were no votes for RAII).

The content in Section 6 was added to this paper after the discussion with EWG in August 2020, in response to questions about specifying the set of supported operations in atomic blocks.

9. Implementation Status and Details

As we have implemented a superset of the functionality proposed in this document, we can provide estimates of the implementation cost for many of the various aspects of language support for this "TM-Lite" proposal. These estimates are for an implementation in LLVM version 10.0.

Front-end support for `atomic` keyword	No estimate	We did not implement front-end support
Back-end support for HTM or lock only	< 100 lines of code	A handful of wrapper functions provide an interface to the TM library.
Back-end support for HTM and STM	< 2500 lines of code	Includes RAII and lambda APIs, support for cross-TU function calls, and an API for C programs
Library support for HTM or lock	< 200 lines of code	The library need only provide a means of interacting with HTM or `std::mutex`.
Library support for HTM and STM	< 2000 lines of code	Majority of code is for efficient data structures for run-time tracking of loads and stores

Note that the back-end implementation complexity for the RAII API offers a good estimate for the back-end implementation complexity of an implementation that uses `atomic do`.

10. FAQ

Can I use parts of the standard library inside transactions?
If Section 6 is approved, then most of the standard library can be used inside of transactions. Otherwise, only standard library functions that are constexpr are guaranteed to work, since they are declared inline and must have reachable definitions. Compatibility of all other parts of the standard library are implementation defined.

Will this proposal allow me to make good use of hardware transactional memory?
This depends on the implementation and on the underlying hardware. That said, the intent of this proposal is to make hardware TM easily accessible in C++ (with fallback, e.g., to a global lock, as is conventional with "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-granularity) transactional memory that maintains a single version of program data. It does not preclude 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. By dropping this, we also implicitly avoid the need to implement aborting only inner transactions, greatly simplifying the treatment of nested transactions.
- The lock-like semantics of `synchronized` blocks, which were guaranteed to support a richer variety of operations inside transactions.