Document Number D0xxxR0

Date: 2017-10-18

Project: ISO JTC1/SC22/WG21: Programming Language C++, SG5, SG1

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

Emails:

Reply to:

Proposed API for Reduced TM Support in C++

Introduction

The C++ Technical Specification for Transactional Memory (TMTS) introduces support for Transactional Memory (TM) through the addition of two new keywords and an extension to the type system:

- `atomic` blocks indicate regions of code that ought to run as "transactions". An atomic block can take one of three forms, related to its behavior when an exception escapes the block: `atomic_cancel` indicates that the memory effects of the block should be undone, and the exception propagated; `atomic_commit` indicates that the memory effects of the block should be committed to memory, and the exception propagated; and `atomic_noexcept` indicates that there is a run-time error if an exception escapes the transaction boundary. Atomic blocks may only perform *safe* operations (e.g., those whose only side effects are to memory). That is, they may not have any side effects that are visible to other threads prior to the block committing, and they may only perform operations that can be proven by the compiler to be safe.
- `synchronized` blocks are semantically equivalent to regions of code protected by the same implicit coarse-grained reentrant global lock. That is, two synchronized blocks may run concurrently only if the run-time system can prove that the concurrent execution of the blocks would not manifest a data race. Furthermore, synchronized and atomic blocks may execute concurrently without races: if a synchronized block and an atomic block have a conflict, the run-time system will resolve the conflict.
- Transaction-safe functions are proven by the compiler to be free of unsafe operations. That is, they may only call other safe functions, and perform non-volatile memory operations. By extending the type of functions, it is possible for separately compiled functions to be called by an atomic block.

To illustrate these behaviors, consider the following example:

```
void bar() transaction_safe;
void bar2();
void foo(){
    atomic_cancel {
        x[rand()]++;
        bar();
    }
}
void foo2() {
    synchronized {
```

```
          if (y == 17)
            { foo();
              bar();
          }
          else {
              bar2();
          }
      }
  }
```

Many threads may simultaneously call `foo` and `foo2`. When `y==17`, all of these threads may execute concurrently without introducing a race, and as long as they do not access the same memory locations, they will commit in parallel, using hardware TM if it is available, and a software TM library otherwise. If `y != 17`, then when a thread runs `foo2`, all concurrent transactions will be aborted, and no transactions will be attempted, until the calling thread completes. This is because `bar2` is not known to be transaction-safe, and thus the run-time system may conservatively serialize to ensure correctness.

There are several properties illustrated by the example:

- `atomic` blocks are statically checked for the absence of unsafe operations. Thus they are guaranteed not to require serialization.
- `atomic` and `synchronized` blocks interact cleanly and naturally.
- `synchronized` blocks need not cause serialization, if at run time they only perform safe operations.

1. Implementation and Usage Implications of the TMTS

The compiler is expected to produce two versions of any function that is transaction-safe. One version is appropriate for executing from non-transactional contexts, as well as when HTM is available. The other contains software instrumentation, so that a library can monitor its loads and stores to ensure correctness. This version is appropriate for use in software TM. In the instrumented version, calls to transaction-safe functions are replaced with calls to the corresponding instrumented versions. The compiler is free to perform the same instrumentation in synchronized blocks, but it is not mandatory.

Programmers should prefer atomic blocks, due to the static guarantee of safety. However, synchronized blocks play an essential role: they allow system calls and other unsafe operations over shared mutable data. Note that it is reasonable for a programmer to expect identical performance and behavior from an `atomic_commit` block and a `synchronized` block that only calls transaction-safe functions.

2. Practical Experience with the TMTS

Experience with the TMTS is limited to date. It is available in the GCC compiler infrastructure, and there are several research papers that use the TMTS. However, some researchers report that the GCC implementation is buggy (issues include linkage of auto-generated safe versions of template instantiations and internal compiler errors related to the transactional instrumentation). Note that there is a chicken-and-egg problem: GCC relies on bug reports, so until there is adoption, GCC will remain buggy. The reasons for limited adoption include:

- Implementation challenges: The TMTS is complex, and only GCC supports it. Until other compilers support it, the decision to use TM implies lock-in.
- Performance: Software TM is slow, and the GCC implementation does not offer any performance benefits over simpler approaches to accessing HTM.
- Complexity: Research use of the TMTS has focused on a simple application of TM: lock elision. Lock elision takes a lock-based program and attempts to use TM to replace locks with transactions. Thus it does not require the complex rules governing exceptions. Furthermore, many real-world critical sections have unsafe operations on uncommon code paths (e.g., error logging), which result in the entire family of atomic blocks offering limited value.

3. An Alternative Approach

We propose a new API for accessing the features of synchronized blocks. In this new API, there are no new keywords. Rather, the API requires programmers to pass lambdas to a function, which then executes those lambdas in a manner equivalent to the `synchronized` blocks of the TMTS. An example appears below:

```
void bar();
int foo(){

    std::tm_synchronized([](){
        x[rand()]++;
        bar();
    });
}
```

There are two important attributes of this example. The first is syntactic: we are passing lambdas to a standard library function, instead of introducing new keywords. The second is that now `bar()` is not known to be transaction-safe, and thus it will cause serialization. There are several mitigating factors:

> If `bar` is in the same translation unit as `foo`, then it is straightforward for a compiler to create correct instrumentation and avoid serialization. Note that this is already achieved in GCC for template instantiations, and is easily extended to functions defined in the same translation unit. Note, too, that since safety is not a property of a type, the creation of instrumented versions of functions does not change the meaning of the program or create future problems if unsafe code is added to bar. Similarly, the call to `bar` can be instrumented so that an instrumented version, if any, can be found at run time. This feature is already present in GCC to handle lookup of instrumented versions of functions reached by a function pointer. The only requirement on the compiler is to produce per-translation-unit initialization code, similar to a static constructor, for creating function-to-instrumented-version mappings. An implementation is free to allow annotations that inform the compiler of the safety of functions not defined in the translation unit. An early version of the TMTS provided this feature, then called `transaction_callable`, to allow programmers to request an instrumented version of functions that were not safe, but which rarely performed unsafe actions (e.g., only performed logging in exceptional circumstances). Extending this proposal with such an annotation is straightforward, and would not impact this proposal in any substantial way.

4. Implementation issues

There are four levels of implementation issues being addressed by this proposal:

1. Reentrant global lock. Advantages: simple -- does not require any modifications to the compiler front-end or back-end; implements semantics of TMTS `synchronized` blocks. Disadvantage: disappoints users as it does not obtain the concurrency that the name of the function suggests.
2. HW elision of the global lock. Advantages: same as for reentrant global lock, plus good performance for workloads that fit within the hardware capacity constraints. Disadvantage: still no benefit for workloads incompatible with hardware support, or for systems without hardware support.
3. Compiler extensions to recognize calls to `tm_synchronized` with a lambda and to instrument (clone) for STM; serialize if across translation units. Advantages: enables better performance across a variety of hardware and workloads. Disadvantage: increased compiler complexity. Note: this is still compatible with HW elision.
4. Compiler extensions to instrument/clone even if passing a functor that was obtained from somewhere else across translation units. In such an implementation it may be worthwhile adding the `transaction_callable` annotation. Advantages: best possible performance. Disadvantages: increased compiler complexity, and a possibility that code using the annotation may be rejected by legacy compilers.

5. Implementation Status

Mike Spear's LLVM plugin is under 2K, but does not address exception landing pads yet; run-time STM libraries would likely be fewer than 2K additional lines, and run-time HTM libraries would likely be fewer than 1K additional lines.

6. Discussion Straw poll

Below are some unstructured topics that could merit further discussion:

- This proposal loses the static guarantee of the absence of serialization, provided by `atomic` blocks. It is possible to provide some of this information at run time through a "profiling" mode in a library.
- Even implementation level 4 can be achieved effectively: we conservatively estimate under 10K lines of well-commented LLVM plugin code. Adding additional arguments to the call to `std::tm_synchronized` would be a simple way to add functionality (e.g., exception behavior, lightweight ordering).
- This proposal does not preclude the introduction of transaction-safe functions and static checking at a later time.
- TODO: need to mention that we could add support for other features through additional functions to the std namespace. For example, SG5 has considered a mechanism for deferring operations until a transaction commits, through a single new function call.
- Should the single-lock fallback be a spinlock or a lock that can yield the CPU?