Simplifying Transactional Memory Support in C++

PanteA Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael Scott, and Michael Spear
Many of the ideas in this presentation and paper were heavily influenced by discussions in SG5, the ISO C++ Study Group tasked with standardizing Transactional Memory in C++
Transactional Memory

• A linguistic construct for declaring that the effects of a region of code ought to appear to happen:
  • Atomically: all at once, all-or-nothing
  • Consistently: program invariants hold throughout
  • in Isolation: intermediate states are not visible to other threads of a correctly synchronized program

• Why?
  • Fine-grained synchronization is hard to get right
  • Let a run-time mechanism and/or specialized hardware dynamically track conflicts between transactions
    • If no conflicts, concurrent regions can complete
    • On conflict, run-time system rolls back some transactions, retries them

```cpp
// x, y, and z
// are integers
// in shared memory
transaction {
    x++;  
    y++;  
    z++;  
}
```
Transactional Instrumentation

- Hardware (Intel TSX, IBM BG/Q, etc.) can monitor the memory accesses of a dynamic scope

- Software can achieve the same through instrumentation
  - Every load becomes a function call
  - Every store becomes a function call
  - Transaction boundaries become function calls

- Compiler should produce and optimize two code paths

```cpp
// x, y, and z are integers:
transaction {
    x++;
    y++
    z++;
}

if (take_checkpoint() == SW) {
    tmstore(&y, tmread(&x) + 1);
    tmstore(&y, tmread(&y) + 1);
    tmstore(&y, tmread(&y) + 1);
} else {
    x++;
    y++;
    z++;
}
commit_or_restore_checkpoint();
```
Why It’s Not Easy

• Some operations cannot* execute as part of a transaction
  • I/O, system calls, acquiring/releasing locks, accessing atomic<> variables, ...

• What should the compiler do:
  • If a transaction tries to do these sorts of things?
  • If it cannot tell if a transaction will try to do these sorts of things:
    • Due to separate compilation?
    • Due to unpredictable control flow (or even just assert() statements)?

* I’m ignoring 20 years of research prototypes 😊
2015: C++ TM Technical Specification (TMTS)

- Introduced two flavors of transactions
  - synchronized blocks:
    - Behave *as if* protected by a single global reentrant mutex
    - Correct if the code would be correct when implemented with such a lock
    - Programmer can expect speculative lock elision with fallback to lock
    - Easy to support: can fall back to a lock immediately, or as soon as anything gets difficult
    - Difficult to use: no diagnostics when a transaction will fall back to a lock
  
  - atomic blocks:
    - Compiler must prove that transactions will not attempt any forbidden behaviors
    - Allows transactions to abort themselves
    - Difficult to use: every function callable from a transaction must have transaction-safety as part of its type
    - Difficult to support: requires changes to every stage of the compiler
2020: State of Affairs

• Both flavors of transaction are difficult to use
  • Viral annotations, or no way to avoid serialization

• TMTS requires supporting both → hard to justify in environments that are unlikely to benefit from TM

• GCC, IBM, and Intel compilers initially supported TMTS, only GCC still does

• Non-volatile memory: the mechanics of TM support are needed by more than just concurrency
Focus of This Paper

• Can we create a strategy for supporting TM in C++ that is:
  • Easy to implement
  • Easy to use
  • Easy to extend

• Can we achieve good TM performance with that strategy?

• What are the challenges? What are the costs?
Nobody ships code that they haven’t profiled
Outline

• Motivation
• **Semantics and spelling of transactions**
• Compiler extensions
• Run-time library support
• Evaluation and optimization
• Conclusions

Latest version of all code available at [github.com/mfs409/llvm-transmem](https://github.com/mfs409/llvm-transmem)
Semantics of Transactions

• For most programmers, *atomicity* just means *critical section*
  • The thought of undoing a critical section doesn’t really make sense, no evidence of its use in GCC

• I/O in critical sections is great for debugging
  • assert() statements, printf(), etc.

• Proving absence of serialization is too painful (especially with separate compilation)
  • Need a sane back-up plan

• Detecting serialization is easy during testing and optimization
  • E.g., with a profiling TM library that logs serialization events
Consequences

- We remove self self-abort
  - A single, global, reentrant lock is a valid implementation of TM

- Upon unauthorized instruction, serialize, don’t reject the code
  - Makes it much easier to get started
  - Programmer can find serialization during profiling

- Programmers can annotate transaction-reachable functions to increase scalability
  - Not part of type, not checked by compiler, but does lead to functions being cloned.
Spelling of Transactions

• Transactions may access variables of enclosing scopes
  • Should these accesses be fully instrumented? Partially instrumented?
  • How much work for compiler to find the right amount of instrumentation?
  • What consequences for the compiler front-end?

• In the paper: use an executor syntax
  • No front-end changes: just uses library
  • Naturally captures scope, allows easy instrumentation
  • Treats enclosing scope as shared
    • Programmer or compiler hints can improve on this

```cpp
int local;
...
atomic {
    local++;
}
```

```cpp
int local;
...
atomic_exec ([&](){
    local++;
});
```
Spelling of Transactions

• What about C?
  • Function interface
  • Sufficient for backwards compatibility without changing compiler front-end
  • Definitely not elegant

• RAII syntax (not in paper)
  • Can be abused 😐
  • More robust (varargs, return, break, continue)
Outline

• Motivation
• Semantics and spelling of transactions
• Compiler extensions
• Run-time library support
• Evaluation and optimization
• Conclusions

Latest version of all code available at [github.com/mfs409/llvm-transmem](https://github.com/mfs409/llvm-transmem)
Compiler Extensions

• Goals:
  • Add one CXXFLAG and one LDFLAG to Makefile (more if using LTO)
  • Use LLVM plugin architecture (versus “our version of clang/LLVM”)
  • No overhead when TM not in use (e.g., no instrumentation of non-transactional code)
TM Plugin: Discovery and Cloning

• Discovery:
  • Find all transaction boundaries
  • Compute all reachable functions in same translation unit
  • Clone those functions

• Note: easy in paper, since transaction CFG’s root is always a function
  • A bit harder with subsequent RAII work
• Instrumentation of cloned functions:
  • Classified all LLVM IR instructions... most are naturally safe
  • Replace regular loads/stores with function calls
  • Replace in-TU function calls with calls to clones
  • Replace non-TU function calls and indirect calls with calls to clone lookup
  • Prefix unsafe instructions with call to serialize transaction

• Note: In C and Lambda APIs, root function is included
• C API: no changes
• Lambda API: insert branch in original root function to allow HTM on non-instrumented path
• RAII API: Need special instrumentation of the root’s basic blocks
TM Plugin: Clone Maps and Optimizations

- Produce a static initializer that inserts original function / cloned function pairs into a map
  - Enables dynamic lookup of non-TU clones

- Optimizations: remove redundant calls to serialize a transaction
TM Plugin: Wrap-Up

• In LLVM, constructors cannot be annotated

• Programmer may wish to annotate “pure” functions to avoid instrumentation

• Programmer may need to provide instrument-only versions of standard library functions

• Details and solutions/workarounds in paper
Outline

• Motivation
• Semantics and spelling of transactions
• Compiler extensions
• Run-time library support
• Evaluation and optimization
• Conclusions

Latest version of all code available at github.com/mfs409/llvm-transmem
Run-Time Library Support

• Implemented over a dozen STM/Hybrid TM/HTM strategies
  • Can handle any/all word-based STM algorithms

• Design is heavily templated and automated
  • Individual TM library implementations under 1K lines with comments
  • Lots of dynamic optimizations

• Subsequent work: support many persistent TM algorithms too [Zardoshti PACT 2019]

• Do not currently support optimizations that require static analysis
  • RfW, WaW, RaW
Outline

• Motivation
• Semantics and spelling of transactions
• Compiler extensions
• Run-time library support
• Evaluation and optimization
• Conclusions

Latest version of all code available at github.com/mfs409/llvm-transmem
Evaluation

• Platform:
  • HW: Xeon Platinum 8160 (2.1GHz, 24 cores / 48 threads) x 2 / 196 GB RAM
  • SW: LLVM 5.0, GCC 7.0 (TMTS) / Ubuntu 18.04

• Benchmarks:
  • Data structures (hash table, balanced tree)
  • Streaming workloads (PBZip2, x265)
  • STAMP TM benchmarks (genome, intruder, kmeans, labyrinth, ssca2, vacation)

• Questions:
  • How expensive is empty instrumentation? Does LTO help?
  • How scalable relative to state of the art?
  • What deficiencies exist? How easy are they to remedy?
Latency

- Baseline: in-lined calls to acquire / release lock at transaction boundaries

- No-op TM + LTO approaches same level of overhead
  - Exceptions: dynamic clone lookup, lambda overheads
Latency

• With actual transactions, latency comparable to GCC
  • Can’t compare directly, since lock-based baselines differ

• Takeaways:
  • Lambdas and dynamic lookup have a cost versus the ideal
  • Satisfactory performance for real TM, with a much simpler design than the TMTS
Scalability

• **STAMP**
  • Our “Eager” == GCC STM algorithm
  • Added other TMs, including HTM
  • Scalability generally equivalent to GCC, with lower latency

• Dynamic lookup is costly
  • Could resolve with better LTO
Scalability

- Our simplistic management of transactional stack frames hurts scalability and latency
- Adding hints (-SF) resolves scalability bottleneck, reduces latency
  - Improves STAMP KMeans, too
Performance Summary

• Common case: good performance (on par with TMTS)

• Inherent weaknesses vs TMTS
  • Lambdas have a cost $\rightarrow$ resolve with better LTO (or RAII)
  • Dynamic lookup $\rightarrow$ resolve with better LTO
  • Limited static analysis (e.g., RfW; see paper) $\rightarrow$ benefit is small, but easy to achieve
  • Stack frames are too simple $\rightarrow$ Mostly resolved via hints
Outline

• Motivation
• Semantics and spelling of transactions
• Compiler extensions
• Run-time library support
• Evaluation and optimization
• Conclusions
Conclusions and Future Work

• The implementation cost to support TM in C++ can be driven down by focusing on common use cases

• Simultaneously results in a easier-to-use feature
  • Quick to get up and running with transactions
  • Easy to profile / debug

• Several optimization opportunities remain

• Framework is open-source, easy to extend (e.g., for persistent TM)
Questions / Discussion

With special thanks to the ISO C++ SG5 group, the National Science Foundation, Intel, Google, and our reviewers