Simplifying Transactional Memory Support in C++

PANTEA ZARDOSHTI, Lehigh University
TINGZHE ZHOU, Lehigh University
PAVITHRA BALAJI, Lehigh University
MICHAEL L. SCOTT, University of Rochester
MICHAEL SPEAR, Lehigh University

C++ has supported a provisional version of Transactional Memory (TM) since 2015, via a technical specification. However, TM has not seen widespread adoption, and compiler vendors have been slow to implement the technical specification. We conjecture that the proposed TM support is too difficult for programmers to use, too complex for compiler designers to implement and verify, and not industry-proven enough to justify final standardization in its current form.

To address these problems, we present a different design for supporting TM in C++. By forbidding explicit self-abort, and by introducing an executor-based mechanism for running transactions, our approach makes it easier for developers to get code up and running with TM. Our proposal should also be appealing to compiler developers, as it allows a spectrum of levels of support for TM, with varying performance, and varying reliance on hardware TM support in order to provide scalability.

While our design does not enable some of the optimizations admitted by the current technical specification, we show that it enables the implementation of robust support for TM in a small, orthogonal compiler extension. Our implementation is able to handle a wide range of transactional programs, delivering low instrumentation overhead and scalability and performance on par with the current state of the art. Based on this experience, we believe our approach to be a viable means of reinvigorating the standardization of TM in C++.

Additional Key Words and Phrases: Transactional Memory, LLVM, C++, Synchronization

ACM Reference format:
Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael Spear. 2019. Simplifying Transactional Memory Support in C++. 1, 1, Article 1 (May 2019), 25 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Transactional Memory (TM) [18] was originally envisioned as a hardware mechanism to simplify the creation of nonblocking data structures. Developers quickly realized that TM could be implemented entirely in software (STM) [38], and that transactional techniques could simplify parallel programing and even accelerate conventional lock-based programs, by replacing critical sections with transactions [25, 32, 33]. Interest in TM blossomed, leading to research proposals that integrated TM tightly into programming languages [1, 6, 7, 17, 21, 30]. The enthusiasm surrounding TM also resulted in an effort to standardize TM in C++. This effort began in 2007, and produced

XXXX-XXXX/2019/5-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the Transactional Memory Technical Specification for C++ (TMTS) [19] in 2015. Today, the GCC compiler supports the TMTS [16], and TM is implemented in hardware from Intel and IBM [29].

Surprisingly, TM has not seen broad adoption by programmers. The first obstacle is performance. Today's hardware TM (HTM) does not provide clear and consistent guarantees that transactions can commit [29, 45]. Hardware transactions that perform system calls, or whose memory accesses exceed cache capacity (either in terms of number of lines or associativity), may never commit. Software transactions require instrumentation on every store, to ensure that they can be rolled back in the event of a conflict, and on every read, to ensure serializability. This instrumentation leads to 50% slowdown or more at one thread for common transactional benchmarks.

The second obstacle to widespread TM adoption is ease-of-use. The TMTS exposes two flavors of transactions: "synchronized" transactions, which are roughly equivalent to critical sections protected by a single coarse-grained lock, and "atomic" transactions, which can be explicitly aborted by the programmer. While synchronized transactions can perform I/O and access `atomic` variables, they must serialize the entire TM before doing so, and programmers cannot tell when, or if, a synchronized transaction will serialize. In contrast, atomic transactions, even those that never self-abort, are statically guaranteed to be free of code that could require serialization [35]. The best tool that programmers have to achieve good performance is to favor the "atomic" flavor of transactions, but doing so requires changes to the type of every function reachable from transactions, so that the compiler can statically ensure that the full effect of the transaction can be undone. The burden to change these types ranges from significant, for programs that have separately-compiled source files, to impossible, for programs that use closed-source libraries. Surprisingly, static guarantees also encourage bad coding habits and hinder debugging: Assertions are forbidden in atomic transactions, since they could result in output when the predicate is not satisfied, and programmers cannot use `printf` debugging from atomic transactions, even for non-concurrent execution.

Third, there is a chicken-and-egg problem. GCC is the only major compiler to support the TMTS, and adding support required significant changes at all levels, to support new keywords, new function types, new run-time libraries, and new static analyses and transformations that ensure that software transactions can be rolled back and retried. Other compiler vendors have been reluctant to implement TM before it has been proven, programmers are reluctant to invest the effort to use TM until it is standardized, and standards bodies are hesitant to standardize a feature that has not shown its value in production software.

Three main audiences care about TM: application developers, compiler writers, and language designers. The current realization of TM poorly serves all three groups: The options the TMTS gives programmers each have significant drawbacks, the TMTS introduces too much complexity into the implementation of the compiler, and the TMTS impacts too many parts of the C++ specification. As a result, progress toward adding TM to C++ has stalled.

In this paper, we present an alternative approach to supporting TM in C++. In our approach, a programmer requests that a code region be run transactionally by passing a lambda to a library function. In a functional but low-concurrency implementation, the library can use a global lock to serialize lambda executions. In our LLVM implementation, a compiler plug-in finds all such lambdas and transforms them so that they are compatible with both HTM and STM execution. A parameter accompanying the lambda allows a programmer to customize the transaction's execution. This approach has profound effects on all three audiences:

- Programmers can use TM more easily: they do not need to verify the safety of all code reachable from a transaction, and they are given mechanisms for tuning performance (e.g., avoiding serialization and relaxing ordering at transaction boundaries).

- Compilers can support TM more easily: a correct implementation can be achieved exclusively as a run-time library. No changes to the intermediate representation, static transformations, or code generation are required.
- The languages community can standardize TM more easily: TM does not complicate the memory model, and the burden of standards compliance is minimal since TM can be supported exclusively as a library. In domains where TM is not applicable (e.g., GPUs), supporting TM will not increase compiler implementation and verification effort.

The remainder of this paper is organized as follows. In Section 2, we discuss background material and related work in the areas of transactional memory and memory instrumentation. Then, in Section 3, we describe our design for supporting TM in C++. Section 4 discusses our implementation of TM support, as an LLVM plug-in and run-time library. We evaluate performance in Section 5. Section 6 concludes.

2 BACKGROUND AND RELATED WORK

2.1 Transactional Memory

Since its inception, TM has been a fundamentally speculative technique [18, 38]. The key idea is that multiple threads attempt to run transactions in parallel, and either custom hardware or a run-time software system keeps track of the locations accessed by concurrent transactions. When conflicts among transactions do not manifest, the transactions can complete (commit), but when conflicts exist, some subset of the actively running transactions must either stall or else undo their speculative effects, roll back (abort), and try again.

Most HTM proposals rely upon or extend the hardware cache coherence mechanism in order to detect conflicts among transactions. While there is tremendous nuance to research proposals for HTM, with a variety of mechanisms to detect conflicts [4], the strategy employed by commercially available HTM has been to avoid significant changes to the cache coherence protocol. Then, once a transaction begins, the cache controller need only verify that lines of data accessed by the transaction do not leave the cache. If an eviction of a transactional line becomes necessary, then the transaction aborts instead.

Consequently, HTM implementations are "best effort": with very few exceptions, a hardware transaction is not guaranteed to commit. This is even true when transactions run in isolation: context switches or device interrupts can cause a hardware transaction to fail, as can memory access patterns that lead to capacity or conflict misses. As a result, practical uses of HTM rely on a fallback path. The simplest and most popular is to fall back to a single global lock [45]. This has clean semantics, though weaker than those provided by hardware transactions [3].

STM implementations instrument the loads and stores of a transactional region – e.g, by replacing them with function calls. The behavior of these calls depends on the STM algorithm. In "eager" STM [15, 36], locations are locked upon first write, and the previous value of the location is saved to an undo log. Committing is fast in eager STM, which need only validate reads, release locks, and reclaim the log, but aborts must undo writes. In "lazy" STM [12, 14], no locks are acquired until the commit point. Writes are stored in a per-thread collection, and every read must perform a lookup in that collection to ensure that a transaction sees its own speculative state. At commit time, a lazy STM must acquire locks, validate reads, update values, and release locks.

Clearly STM will have higher latency than HTM. It is also difficult to ensure lock-based semantics for STM [27], since the commit of a writer transaction does not immediately cause conflicting readers to abort. Nonetheless, STM has access to more run-time information, which it can use to adapt to the workload to achieve good performance. This adaptivity includes "contention management" [37], the decision as to which transaction(s) should delay or abort when a conflict manifests, as well

```
// Source Code (running sequentially)
synchronized {
    counter++
}
// OR
atomic {
    counter++
}
```

```
// Transformed Code
if TxBegin() == UNINST then
    counter++
else {
    // Transformed Code
    tmp = TxRead(&counter)
    TxWrite(&counter, tmp + 1)
}
TxEnd()
```

Listing 1. Example instrumentation performed by GCC.

as dynamically switching switching between eager and lazy modes. This flexibility, coupled with its unbounded capacity, makes STM a scalable fallback for programs whose transactions cannot complete in hardware [5, 9, 26, 34], as well as an implementation option for systems without HTM (such as ARM CPUs).

2.2 The C++ Transactional Memory Technical Specification

The C++ Transactional Memory Technical Specification (TMTS) [19] proposes standard language constructs for TM. In the spirit of C++, it aims to avoid any performance penalty for programs that do not use TM, to introduce minimum additional overhead versus directly programming to HTM (when HTM is available), and to provide more safety than would be likely if programmers invented their own TM implementations.

The TMTS enables programmers to declare that a lexically scoped block should run as a transaction (either a "synchronized" block or an "atomic" block). The differences between these blocks are subtle: an atomic block contains code explicitly intended to be a transaction: it cannot perform I/O or system calls, and it cannot communicate with other threads through `volatile` or `atomic` variables (or through types built upon atomicity, such as mutexes). The compiler must statically check that an atomic block could safely abort and roll back at any time. Since this property holds, a programmer is allowed to explicitly "cancel" an atomic block, by allowing an exception to escape its boundaries. There are three variants of atomic blocks, corresponding to the cases where an escaping transaction commits the transaction and propagates, unrolls the transaction and propagates, or causes the program to terminate.

Synchronized blocks are allowed to do anything that can be done from a lock-protected critical section, including I/O and system calls. Whenever a synchronized block attempts to perform an operation that would not be able to be rolled back upon subsequent conflict (an "irrevocable" operation [41, 44]), execution transitions to a serial mode, in which the synchronized block can continue without the risk of a conflict subsequently manifesting. A synchronized block cannot cancel itself.

The TMTS does not introduce special transactional data types, but to support separate compilation, it introduces "transaction-safe" function types. This type enables static checking that code reachable from an atomic block is free of irrevocable operations. It also informs the compiler about functions requiring instrumentation for STM support. Ruan et al. found that the principal benefit of transaction-safe functions is not the ability to cancel, but the static guarantee of speculation safety that they provide [35]. They argue that while this static checking creates a significant burden on programmers (by necessitating that function types be changed across the entire control flow graph rooted at the atomic transaction), the resulting static guarantee of freedom from serialization is a strong benefit versus synchronized blocks.

To provide correct instrumentation without impacting the performance of non-transactional code, the TMTS implementation in GCC clones every transaction-safe function, and instruments the clone. In the clone, every memory read and write is replaced with a function call to an STM implementation. The top-most lexical scope of the transaction is also duplicated, so that HTM code paths can run uninstrumented code and avoid function calls on loads and stores. Examples of this transformation and instrumentation appear in Listing 1. To handle virtual methods and function pointers, the TMTS implementation in GCC creates a mapping from functions to their clones, and modifies the linker to load this mapping at load time. In instrumented code, calls to virtual methods and calls through function pointers are redirected through the map.

In principle, the TMTS is agnostic as to how STM (or an STM fallback for HTM) is implemented. However, lexical scoping favors eager STM: Suppose v is a local variable declared within a transaction's scope. If the address of v is passed to a transaction-safe function that changes the value of v and then returns, do subsequent accesses to v require instrumentation? In lazy STM, they *might*, because the new value of v could be in the transaction's redo log; without instrumentation, the thread would not see its own write to *v*. Efforts to transform the popular STAMP TM benchmark [28] to use the TMTS and lazy STM [20] discovered many instances in which the compiler made assumptions that the STM was eager, which necessitated a rewrite of benchmarks to get them to behave correctly with lazy STM.

2.3 Memory Instrumentation in LLVM

The LLVM compiler infrastructure can be extended through a robust plugin architecture [22]. Recently, several groups have used this architecture to achieve memory instrumentation, similar to that provided by GCC in its implementation of the TMTS. In a precursor to the TMTS, Christie et al. used LLVM to instrument C++ code for HTM and eager STM [8]. This work, along with that of Ni et al. for STM [30], formed the foundation upon which the TMTS was built. More recently, Denny et al. extended C with new types for persistence, and extended LLVM's intermediate representation (IR) to understand these types. This enabled both static checking of common programming bugs for persistent memory, and transformation of the IR to provide transactional semantics for regions that accessed persistent memory [13].

In the context of IoT devices, Baghsorkhi and Margiolas created an LLVM-based framework to form transactional regions of flexible granularity [2]. By designing firmware along with their compiler extensions, they were able to achieve fault tolerance for devices with unreliable power sources, and to tune the application in order to balance between coarse transactions, which amortize overhead, and fine-grained transactions, which increase the likelihood of the application making forward progress.

3 REDESIGNING TM SUPPORT IN C++

A guiding principle in C++ is the concept of the "zero-overhead abstraction" [42]. Zero-overhead abstractions do not introduce run-time cost to programs that do not use them, and cannot be outperformed by a custom implementation of that abstraction. Zero-overhead abstractions should also be easy to use, and their use should have a local effect on a code base. In our view, the TMTS fails these goals. From the programmer's perspective:

- Transaction safety is inferred for templated functions, but not for other functions. For programs with multiple source files, manually changing function types requires whole-program modification. When an uncommon code path performs I/O (e.g., `printf` debugging, error logging), an attempt to use atomic transactions becomes futile [35].

```
void double_transfer(account a1, account a2, account to, float amount)
transaction_safe
{
    // maximize amount to transfer from a1 while preserving account rules
    float a1x = get_transferrable_amount(a1, amount);
    transfer(a1, to, a1x);      // in case a1 == a2
    // repeat for a2
    float a2x = get_transferrable_amount(a2, amount - a1x);
    // Must undo previous transfer and effects of get_transferrable_amount if
    // insufficient funds
    if (a1x + a2x != amount)
        throw InsufficientFunds(from);
    transfer (a2, to, a2x);
}
```

Listing 2. An example of throwing an exception to cancel the calling transaction.

- It is not clear that cancellation can be used reliably, because a programmer decides to cancel in a nested scope (a function called by a transaction), but the effect on the transaction (whether partial results are committed or undone) is decided at transaction scope.

At the language implementation level:

- Lexical scoping of transactions makes it difficult to properly checkpoint the stack, especially for variables local to the parent of the transaction scope. This complicates implementation of some STM and Hybrid TM algorithms [9, 26, 34].
- While transaction-safe functions enable static linking among clones, dynamic lookup is still required for function pointers and virtual methods.
- TM support is spread across every stage of the compiler, as well as the linker.

We now present an alternative approach to supporting TM in C++. Our guiding design principles are:

- Localize compiler support for TM as much as possible, ideally to no more than a single compiler pass and a single run-time library.
- Simplify the programming model to avoid whole-program modification, even if it restricts programmer options or results in less predictable performance.
- Increase opportunities for developers to tune transaction behavior.

3.1 Change #1: Eliminate Transaction Cancellation

There is, to the best of our knowledge, no published work showing an application that requires transaction cancellation, perhaps because cancellation requires the sort of whole-program reasoning that TM was supposed to obviate. Consider the hypothetical bank transaction in Listing 2. The programmer begins moving funds, and at some point determines that an account lacks the balance to complete the transaction. Can the programmer throw an exception to cancel the transaction? If the code is called from a synchronized block or nontransactional code, then an exception cannot have the desired effect. If the code is called from an atomic block, then whether partial results commit or roll back depends on the variant of atomic block that was used. Furthermore, the semantics of cancellation complicate the language-level memory model considerably [11, 39], limit the use of HTM, and require changes to the low-level implementation of C++ exceptions. In light of these shortcomings, we forbid cancellation, and propagate the resulting benefits throughout the entire

```
atomic {
    if (y)
        return f(x);
    ...
    return g(x);
}
```

```
TM_EXEC(flags, [&](){
    if (y) {
        result = f(x);
        return; // escape lambda
    }
    ...
    result = g(x);
});
return result;
```

Listing 3. Control flow with lambda-based transactions is more complicated than with lexically-scoped transactions.

TM design, while leaving the door open to adding cancellation in the future if experience and standards committee enthusiasm warrants it.

Benefits. Elimination of explicit cancellation simplifies programming and debugging: programmers do not need to change function types, and can add `printf` statements during debugging, without breaking their code. Without cancellation, compliant compilers can support TM at varying levels of quality and implementation effort, ranging from virtually no compiler changes to changes as complex as the TMTS requires. Note that all of these options could be appealing: On systems without HTM and with few cores, TM often performs best if implemented as a single global reentrant mutex lock [43]. For these systems, no per-access instrumentation will ever be needed. For hypothetical future systems with robust HTM capabilities, the compiler can be equally simple, since transactions would always use HTM. On existing systems with many cores but either best-effort HTM or no HTM, the sort of instrumentation proposed by the TMTS would still be advantageous.

3.2 Change #2: Replace Lexical Scope with Executors

Our second proposed change is to replace lexically-scoped transactions with an "executor" model, in which lambdas are passed to a library. This change slightly complicates the application programmer's experience of using TM, but affords greater tunability and easier implementation of TM libraries. To understand why this change complicates programming, consider Listing 3. Since the lambda is a function call, transactions within functions cannot directly `return` from the calling function, and transactions within loops cannot directly `break` or `continue`. Instead, programmers must introduce a shadow variable in the calling scope, and then branch based on this shadow variable after the transaction completes.

Benefits. We believe that the benefits of an executor style outweigh the costs. First, since keywords are replaced with a function call to a standard library, it is much simpler to extend the TM so that programmers can tune the behavior of individual transactions. For example, the `flags` parameter shown in Listing 3 could be used to indicate that the transaction should always be run irrevocably, to express priority, or to tune backoff parameters. The value of this parameter could even be computed at run time (e.g., based on the size of a collection passed to the transaction).

At the TM implementation level, the use of lambdas makes it easier to implement TM, because the function call to the lambda, which is made by the executor, creates a new stack frame. This makes it trivial for the implementation to determine when a read or write is to a transaction-local variable, even when stack addresses are passed to functions that may also receive heap addresses. Since both reads and writes to the stack are easy to discover, lazy hybrid TM systems no longer require any static analyses to determine when to avoid logging – the entire decision can be made

dynamically. This, again, reduces the complexity of the implementation by making it easier to localize TM support to a single compiler pass and a run-time library.

Drawbacks. In many published TM papers, algorithms are evaluated based on their ability to scale for workloads that are primarily read-only. In Listing 3, suppose that `f()` does not perform any writes, and that `y` is true. In that case, the code on the left would finish by updating a register, and never modifying shared memory – it would be read-only. In contrast, the code on the right would not be read-only, since `result` would be updated by writing to memory outside of the transaction. We evaluate this situation in Section 5.

In addition, the implementation of lambdas can introduce run-time inefficiencies. For example, in the LLVM compiler, we found that the structure holding references to captured variables can be allocated on the stack or on the heap, depending on heuristics and the availability of interprocedural analysis about the lifetime of lambdas. Clearly, invoking an allocator on every lambda will hurt performance, and could impede scalability. We do not offer any solution at this time, but note that the increasing popularity of executor-style libraries for C++ should lead to a virtuous cycle, in which one domain's optimizations for lambdas benefit all other executors, including ours.

Supporting the C Language. An unwritten requirement of the TMTS is to support the use of TM in C code. Since lambdas are not part of the C language, we overload the `TM_EXEC` function so that it can also take a function pointer and `void*` argument. While not as elegant as lambdas, this approach enables C programs to use TM. It also affords more control to C++ programmers who wish to avoid the above issues with heap allocation of lambda structures.

3.3 Change #3: Replace Function Types with Attributes

Our final proposed change is to eliminate the `transaction_safe` extension to function types, and replace it with an optional attribute. When the compiler encounters a function with the attribute, it has the option of cloning the function and instrumenting the clone. Note that this change does not compromise correctness: Given that we have already removed self abort, static checking of safety is no longer required. By removing the type information on functions, our proposal is free of any new keywords or language-specific static checks. This makes it possible to support high-quality TM entirely as a transformation over the IR, without any changes to the compiler front end.

From a programmability perspective, this change makes it much easier for programmers to get started using TM: once they identify transactional boundaries, they can compile the code, and it will work. As we show in Section 4, for transactions that do not call code in separately compiled source files, it is straightforward to identify the full call graph and instrument it, thereby avoiding any serialization. Doing so does not increase implementation complexity, since the call graph discovery that our approach requires is already required by the TMTS, to support templated functions. Furthermore, error messages associated with typed functions are often opaque, especially since incorrect marking in header files can lead to errors that are not detected until link time.

To avoid serializing on calls to functions in separate compilation units, an implementation can maintain a function-to-clone map, akin to that required by the TMTS. To identify the functions that must have attributes to prevent serialization, build flags enable our system to report (either at link time or run time) functions that require attributes. The run-time support is straightforward [23, 24], and the compiler analysis is no more complex than the support required by the TMTS. Furthermore, it is not required: it helps the programmer achieve better performance, but its absence does not compromise correctness.

Benefits. The main benefit of this change is its impact on developer velocity: there is no prolonged period during which code does not compile due to the incremental addition of TM. In contrast,

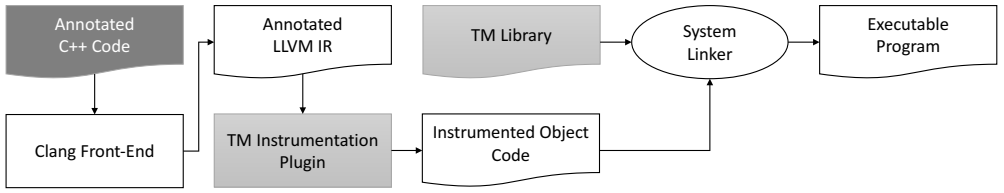


Fig. 1. Overall system design. Programmer involvement is limited to adding attributes to the source code (dark gray). New components in the compilation toolchain are shaded light gray.

to use atomic transactions with the TMTS, programmers are currently advised to replace all uses of atomic variables with mini-transactions, to comment out assertions and error logging, and to avoid common features like `shared_ptr` [35]. These are not benign changes: they make code less maintainable, lead to program-wide edits, and can even cause memory leaks. Worse, they are unnecessary when they only affect uncommon code paths (e.g., error logging), especially when HTM is available.

Drawbacks. Above, we mentioned that incorrect typing of functions as `transaction_safe` can lead to link-time errors. On the other hand, under the TMTS, STM transactions that call functions from other source files can statically link to the instrumented clones of those functions. In contrast, under our design every such function call must be transformed to perform a dynamic lookup that either finds the instrumented clone or else serializes the transaction and calls the original.[1] This results in a lookup to a thread-safe hash table. The mechanism is identical to that required by the TMTS for virtual methods, function pointers, and synchronized blocks.

3.4 Design Summary

To summarize, our design for executor-style transactions in C++ does not introduce new requirements for programmers. Rather than mark transactions as lexical regions, programmers wrap transactions in lambdas and pass them to an executor. Instead of proactively changing the types of functions to eliminate potential serialization and ensure self-abort safety, programmers reactively add attributes to functions in response to observed serialization during testing. Other features of the TMTS, such as transaction nesting, are supported, and incur no more overhead than in the TMTS. The main omission is the elimination of self-abort, which is not widely used, but the benefits are a simpler programming model, a simpler and more localized implementation in the compiler, and a path by which compiler designers can support both HTM and low-scalability STM without significant effort.[2]

4 IMPLEMENTATION

In this section, we describe our implementation of TM support for C++. In order to enable a scalable code path for STM and HTM fallback, we add a transformation pass to the compiler, which creates and instruments clones for functions reachable from transactions. We also provide a run-time library that implements the `TM_EXEC` function, as well as per-access instrumentation for STM and HTM fallback. Figure 1 depicts the position of these components in the compilation workflow.

[1] In some cases, the program will need to abort the transaction and then re-start in serialized mode.

[2] Due to our experience working with the TMTS, we are not able to conduct an unbiased study to support our claim of simplicity. We encourage the reader to perform such a study contrasting our implementation with the TMTS, using a methodology similar to that in [31].

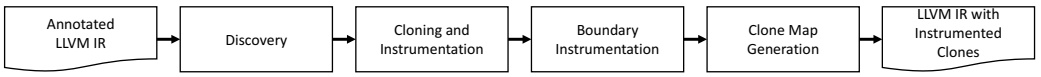


Fig. 2. Code Region Plugin Design

4.1 LLVM Extensions

Our implementation supports the language design from Section 3. It is implemented as a single `ModulePass` in LLVM, and consists of about 2K lines of commented C++ code. The pass is activated via a command-line parameter to the `clang++` front-end, during the translation from source code to machine code, and transforms the LLVM IR in linear time. It is implemented without any modifications to the LLVM source code itself, and interacts only with parts of the LLVM API that are common to versions 5.0 and above. To use the plugin, one need only add one compiler flag and add one link step to existing Makefiles.

Figure 2 depicts the workflow of our `ModulePass`. There are four stages: discovery, in which each transaction's control flow graph is created; cloning and instrumentation, in which functions reachable from transactions are cloned, and the clones' memory accesses instrumented to achieve transactional semantics; transaction boundary instrumentation; and a step that creates mappings from functions to their clones, for use during program execution. Through these steps, our extension ensures that every interaction with memory from a transactional region becomes a function call to a library that provides the appropriate memory instrumentation, and that the boundaries of each instrumented memory region call the library to checkpoint thread state before transactions start, and commit effects when transactions end.

4.1.1 Discovery. The main obligation of our implementation is to ensure that every memory access within the dynamic control flow graph of a transaction is either (a) transformed into a library call that performs the access, or (b) dominated by a library call that ensured the calling transaction is either using HTM, or irrevocable and running in isolation. There are three ways that a function could be part of this dynamic graph: it could be the lambda (or C function) passed to the executor, it could be a function that was explicitly marked by the programmer, or it could be reachable from one of the above.

The discovery step identifies the names of all functions passed to the executor (the compiler assigns a unique hidden name to each lambda body), and all functions with attributes. (Note that in LLVM IR, attributes appear only for functions whose definition and declaration are both in the current module.) It uses these to populate a worklist, and then iterates over the bodies of functions in the worklist to identify previously-unseen, transaction-reachable functions. Upon completion, we have a list of all functions that are both (a) reachable from a transaction, and (b) defined in the current source file. This is a single pass over the IR for a source file, and completes in linear time.

4.1.2 Cloning and Instrumentation. For each function that we discovered, we create a clone. Cloning is necessary to ensure that TM support does not introduce overhead on code that does not use it – such code should always call the original, unmodified version of the cloned function. Similarly, HTM will use the original, uninstrumented code. After cloning, we store a mapping from the original function to the clone, to facilitate future lookups. Then, we instrument the clone. Broadly, there are five actions we may take during instrumentation.

(1) The most common case is that an instruction has no need for instrumentation, in which case it is not transformed. This includes register-register instructions, which are the most abundant instructions in the LLVM instruction set.
(2) For LLVM instructions that access memory (loads, stores, memory-related LLVM intrinsics, and other special instructions), we replace the instruction with a call to the TM run-time library. The library call is expected to provide the same behavior as the IR, but with transactional semantics.
(3) For LLVM instructions that transfer control flow to another function in the same source file, we use the mapping to locate the cloned function, and we replace the original call with a call to the clone.
(4) When the instruction transfers control to a function in another source file (i.e., one for which we do not have a definition), we replace the call with a call to the TM library, which takes as a parameter a reference to the original function, and returns a reference to the appropriate version of the function at runtime. The return value is then called. In this manner, we support dynamic translation of function pointers, virtual methods, and functions defined in other files.
(5) For IR instructions that do not have correct transactional semantics (e.g., that include `volatile` memory accesses or self-modifying code), we prefix the instruction with a call to the library's irrevocability mechanism. When the call returns, the transaction will be running in isolation, and the corresponding instruction will be safe to execute.

Like discovery, instrumentation is a single linear-time pass over the IR.

Note that LLVM's C++ front-end (clang++) produces IR that contains calls to the C++ library (i.e., for exceptions). While it is always safe to treat these C++ library calls as lacking transactional semantics (case 5), they can also be handled as special cases, like LLVM intrinsics (case 2).

4.1.3 Boundary Instrumentation. The primary obligation of the boundary instrumentation phase is to ensure that the executor is able to run the instrumented body of the transaction when STM is in use, and the uninstrumented body when HTM is available. For the lambda (C++) interface to the executor, we already cloned the lambda body in the previous step. In this step, we alter the signature of the lambda to take an extra parameter, and we add a branch to the lambda body, which calls either the instrumented clone or the original code, depending on the value of the parameter. In this manner, the library implementation of the executor can control which path to execute.

For the C interface to the executor, we have also cloned the function. However, in contrast to the C++ interface, the function passed to the executor may be called from nontransactional contexts. Thus it is not appropriate to change the original function body's signature. Instead, we substitute a four-parameter call to the executor for the original three-parameter call. That is, in addition to passing flags, a function, and an argument packet, we also pass a reference to the clone of the function. In this manner, the executor can decide which function to run. In cases where the clone is not available, we retain the original three-argument call, whose implementation we describe in Section 4.2

4.1.4 Clone Map Generation. When an instrumented function calls another function whose definition is not in the same source file, the instrumentation step will replace the call with a library call that performs a dynamic lookup. That lookup must have access to the full set of function/clone pairs that were produced during the instrumentation of all source files of the program, so that it can either direct execution to the clone, or else serialize the transaction.

As we will discuss in Section 4.2, the TM library is responsible for maintaining a data structure that associates function references with clone references. Whenever compiling a C or C++ source

file, the clone map generation step adds a static initializer to the output file. The body of the initializer contains calls to the TM library to register each entry from the list produced during the discovery step. This code executes automatically when a shared object file is loaded. We also produce a static destructor, which unregisters functions when the object file is unloaded. In this manner, both statically and dynamically linked files are able to contain instrumented code. Furthermore, the mechanism used (static initializers) is already supported by linkers, as it is required by C++. Thus we do not require modifications to the linker in order to support TM.

4.2 Library Implementation

To date, we have implemented several STM and HTM algorithms in our system, most notably NOrec [12], a version of the eager TM from GCC [15], and a library to support Intel's TSX HTM.

In STM publications, the algorithm usually exports four public functions: one to begin a transaction, one to commit it, one to read from memory, and one to write to memory. This is a simplification of the true ABI needed to support TM in C++. A more instructive reference is the GCC implementation of the TMTS.

In a practical implementation there must be as many read and write functions as there are sizes of primitive types. In the GCC implementation of the TMTS, these read and write functions are further parameterized by access patterns (such as write-after-write and read-followed-by-write), which make it possible for the TM library to coalesce certain overheads. Additionally, to checkpoint variables from a (lexically-scoped) transaction's enclosing stack frame, GCC's TM implementation has one lightweight logging function per primitive type.

Beyond these memory access functions, the GCC TM ABI provides functions to start and try to commit transactions (the same functions are used for nested and top-level transactions), to allocate and deallocate memory, to perform bulk memory operations (memcpy, memset, memmove), and to look up function clones. There are also functions to manage exceptions, so that an exception can escape a transaction that is canceled, and to serialize a synchronized transaction.

In our design, the library must expose three categories of functions. The first comprises functions that relate to coarse transaction behavior. The second comprises functions that correspond to LLVM IR instructions. The third comprises instructions that mediate functionality of the C++ standard library. In the first category, there are three executor functions to manage transactions: one to execute lambdas, and two to support C programs, depending on whether the clone was found at compile time or not. These functions create a transactional context, execute the function (or its clone), and then commit the transaction. For C programs, the executor will perform a lookup in the clone table if a clone was not provided via the four-parameter call.

The first category also includes three functions to manage the clone table: one to insert a mapping, another to remove a mapping, and a third to perform lookups. The clone table must be thread-safe, in order to support dynamic linking and loading, but thread safety is relatively inexpensive, since it can piggyback on existing synchronization in the TM implementation. Finally, the library provides an `unsafe` function, which transitions a transaction to irrevocable mode.

In the second category, our libraries must provide transactional analogues to LLVM IR instructions that access memory. We conducted a top-to-bottom evaluation of the LLVM IR to determine which instructions were affected. Of the hundreds of instructions in the LLVM IR, only a small number require our attention. They are listed in Table 1.

In the third category, our libraries provide transaction-safe versions of common functions from the C and C++ libraries. These are also listed in Table 1. At the present time, this category only includes `malloc`, `free`, and the C library intrinsics. We initially produced replacements for C++

exception instructions, but we have not been able to find any transactional programs that use exceptions, so we removed this support, and instead prefix them with calls to `unsafe`.

The most common instructions in Table 1 are instructions to access memory, for instructions that call functions (including `invoke`, which calls a function that may throw an exception), and memory intrinsics. In addition, there are a handful of loads and stores between memory and vector registers. For each instruction, we can choose to implement it in a transactional manner, or to prefix it with `unsafe`. In some cases, such as `llvm.clear_cache`, the instruction is never generated for C++ programs, and can be ignored. In the applications we evaluated in Section 5, only 23 of these instructions needed transactional versions: the 8 loads and 8 stores of primitive (non-volatile, non-atomic) types, control flow functions (invoke, call), malloc and free, and the three C library intrinsics. All other instructions were handled by prefixing them with a call to `unsafe`. Supporting these 23 instructions required about 600 lines of library code per TM algorithm, with significant code sharing.

Table 1. Memory-related LLVM Instructions and C/C++ function calls inserted by LLVM during compilation.

Category	Instruction	Description
Control Flow	`call`, `invoke` `llvm.trap`, `llvm.debugtrap`	`invoke` is for exceptions that might `throw`; `traps` are for debugging.
Allocation	`malloc`, `free`	These appear as both C function calls and compiler intrinsics.
Memory Access	`load`, `store`	There are 8 versions of each, for the 8 primitive types in LLVM IR.
SIMD Memory Access	`llvm.masked.load`, `llvm.masked.store` `llvm.masked.scatter`, `llvm.masked.gather`	There are 24 versions of each (8 types × 3 mask patterns).
Memory Access (Synchronization)	`volatileload`, `volatilestore`, `atomicload` `atomicstore`, `atomicrmw`, `cmpxchg`	For accessing synchronization variables and device registers.
C Library Intrinsics	`llvm.memcpy`, `llvm.memmove` `llvm.memset`	These appear as both C function calls and compiler intrinsics.

C Library Intrinsics (Synchronization)	`llvm.memcpy.element.unordered.atomic`
	`llvm.memmove.element.unordered.atomic`
	`llvm.memset.element.unordered.atomic`	

Exception Support	`llvm.init.trampoline`, `__cxa_begin_catch`
	`__cxa_end_catch`, `__cxa_allocate_exception`
	`__cxa_free_exception`, `__cxa_throw`
	`__cxa_rethrow`	

Other	`llvm.clear.cache`, `llvm.load.relative`	Not applicable to C++ code.

5 EVALUATION

5.1 Benchmarks and Experiments

Our LLVM plugin consists of about 2K lines of commented C++ code, and can be run as a compile-time pass for LLVM 5.0 and above. To validate the correctness of the plugin, we developed an ad-hoc unit test suite with 6K lines of code (158 unique tests) that produce every LLVM IR instruction that accesses memory (including vector operations and self-modifying code), affects control flow (including exceptions), or causes interaction across source code files.

The goal of our evaluation is not to show that our approach is superior to GCC's implementation of the TMTS, though performance is often as good if not better. Rather, it is to show that the philosophy behind our approach has merit. There are three dimensions on which this merit can be evaluated.

(1) When TM implementation is reduced to an absolute minimum, how much overhead does our instrumentation introduce versus uninstrumented code?
(2) For popular TM benchmarks and workloads, does our approach yield performance competitive with that of the GCC implementation of the TMTS?
(3) How much programmer effort is required to use our approach, relative to the TMTS?

The third of these is the most subjective, but also the easiest to address: transforming programs to use our approach, once they had been transactionalized using GCC, was a simple matter of text substitution. Thus we claim that our approach is not significantly harder than using the TMTS.

To evaluate the other two dimensions, we conducted experiments on a Dell PowerEdge R640 with two Intel Xeon Platinum 8160 CPUs (24 cores/48 threads per CPU) at 2.1GHz, with 196 GB of RAM. Experiments are the average of five trials; to avoid NUMA effects, we limited execution to a single CPU socket.

We looked at three sets of applications. First, we measured traditional data structure microbenchmarks (a non-resizable hash table with 64k buckets, and a balanced binary search tree). We assigned each thread an 80/10/10 mix of lookups, inserts, and removes, with keys drawn from a universe of 2^{20} elements, and ran each test for 5 seconds. These experiments are valuable for measuring the latency of transactions and fundamental scalability. They also represent the case (data structures) for which TM is most likely to deliver long-term benefit. Second, we looked at STAMP [28], the traditional TM benchmark suite, using the default parameters. We used the version from Kilgore et al. [20], which conforms to the TMTS. Finally, we looked at a handful of TM versions of open-source applications [35, 47]: the PBZip2 file compression/decompression algorithm, the x265 High Efficiency Video Coding (HEVC/H.265) encoder, and memcached.[3] For x265 and PBZip2, we used 1.1GB input files.

In the remainder of this section, we present a performance evaluation of our design. There are three questions we seek to answer:

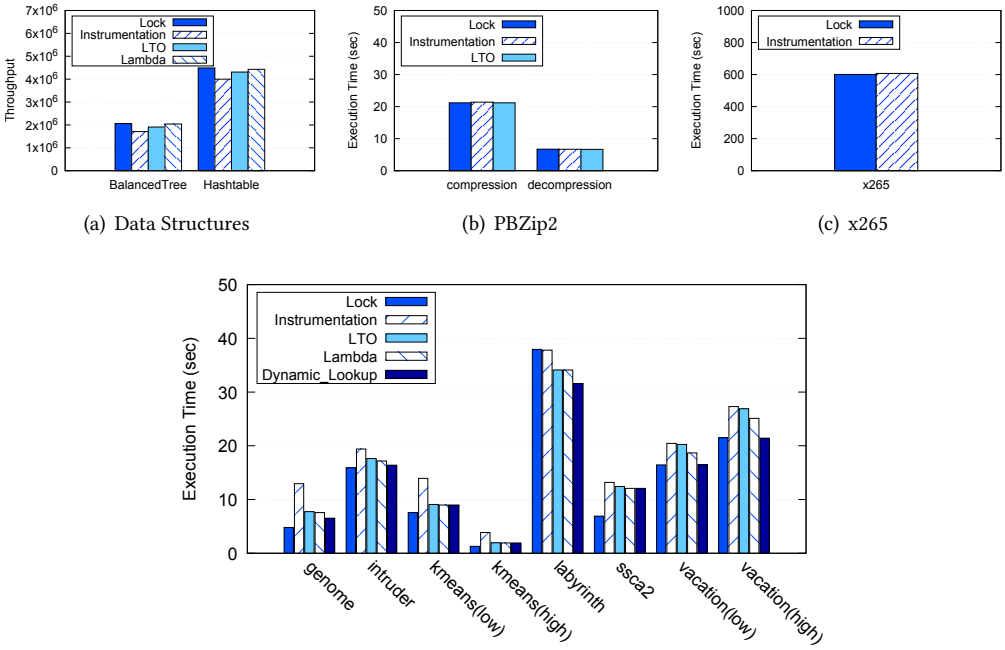
(1) How significant is the latency introduced?
(2) What impact does our design have on scalability?
(3) How does the performance of our design compare to the state of the art?

5.2 Latency of Instrumentation

We begin our evaluation by looking at the latency that our instrumentation introduces for single-threaded code. Figures 3 and 4 present results for all of the benchmarks we consider in this paper. "Lock" refers to an experiment in which there is no TM: each transaction is implemented as a critical section, protected by a single global mutex. This is the baseline, since it is unlikely that any TM implementation, either hardware or software, could have lower single-thread latency. "Instrumentation" refers to the use of our system to instrument the programs' transactions. This means that each transaction became a lambda passed to an executor, and that each memory access within each transaction became a function call that (1) incremented a per-thread count of the number of API calls, and (2) performed the appropriate load or store.

The data structure experiments depict throughput, where higher is better. All other experiments measure time, and lower is better. Every data point is the average of five trials. Variance was inconsequential. In the data structure experiments, where millions of transactions commit per second, we see that the overhead of instrumentation is highest, reaching almost 30% in the worst case. The HashTable has about 10 memory accesses per transaction, while the tree has about 100. The difference between the data structures highlights the cost of per-access function calls.

[3]We were unable to get the TM version of memcached to compile with the latest version of GCC, so we do not include memcached results.



(a) Data Structures (b) PBZip2 (c) x265

(d) STAMP

Fig. 3. Single-threaded execution, to measure the latency of instrumentation.

In contrast, for PBZip2, where transactions are short and the streaming nature of the workload provides many opportunities to cover TM latencies, the raw overhead of function calls and lambdas is negligible. While transactions are larger in x265, it is also a streaming workload, with cache misses and I/O to hide the latency of instrumentation. Each benchmark in STAMP falls somewhere between the extremes of high-locality data structures and streaming workloads. For example, Genome sees a significant slowdown, because it spends most of its time traversing linked lists.

To gain more understanding of the relative costs of per-access instrumentation versus boundary instrumentation, we turned on LLVM's link-time optimization ("LTO"). In these tests, every function call that corresponds to memory access instrumentation is inlined, and the instrumentation itself is optimized. For example, this means that if a basic block performed n reads of integers, "Inst" would have four function calls and four counter increments, whereas "LTO" would have zero function calls and a single increment by four. In most cases, this reduced overheads significantly.

In the data structure microbenchmarks, as well as in STAMP, we still see some instrumentation overhead. The next source of overhead that we identified was related to lambdas. Running a transaction typically takes two nested function calls. The first call is to the `TM_EXEC` function, which, in turn, calls the body of the lambda. In most cases, LTO was not inlining these calls. In the "Lambda" bars, we made the body of `TM_EXEC` visible to the compiler, so that lambda calls could be inlined *before* LTO, and then LTO applied at the end of compilation. This alone brought BalancedTree performance to within 4% of the baseline, and HashTable to within 8%.

We do not report the benefit of "Lambda" for the streaming workloads, since they already had negligible overhead. The merit of "Lambda" optimizations for STAMP was more variable, depending

Table 2. Count of dynamic function translation to find clones.

	Genome	Intruder	Labyrinth	Vaction (high)	Vaction (low)
Default	156M	321M	364K	642M	905M
Optimized	103M	257M	0	559M	781M

on the ratio of memory accesses to transactions, and the density of memory accesses per basic block. KMeans saw an improvement of 94% for LLVM+LTO, leaving little room for Lambda to improve performance versus the baseline. In Vacation, there were roughly 500 memory accesses per transaction, and while LTO alone did little to reduce overhead, Lambda played a more noteworthy role. SSCA2 saw little benefit from either optimization.

A significant difference between STAMP and the other benchmarks is that STAMP transactions often call functions that are defined in multiple source files. In our design, these calls must always perform a dynamic lookup to find the instrumented clone. While the lookup takes $O(1)$ time, it is a cost that is not shared by the TMTS: in the TMTS, the types of functions indicate if they will have clones, and thus these function calls can be linked statically. To assess the impact of added dynamic calls, we restructured the build so that all transactional instrumentation occurred immediately before LTO. At this point, all source files have been merged into a single file containing LLVM IR, and thus lookup is only necessary for function pointers and virtual method calls (the latter of which do not occur in STAMP). The final bar in Figure 3, labeled "Dynamic_Lookup", shows the impact.

KMeans and SSCA2 are unaffected by this change, since their transactions do not call functions in other source files. For the other benchmarks, we see a significant improvement. To further assess the impact, we measured the number of function translations that occurred. The result appears in Table 2. The number of dynamic lookups does not vary among Instrumentation, LTO, and Lambda. In the table, they are reported as "Default"; "Dynamic_Lookup" is reported as "Optimized".

The first observation is that some lookups are unavoidable. Since STAMP was originally written in C, its generic ordered set data structures (lists, trees) use function pointers to define the functions that determine the order among set elements. These function calls are unavoidable, and require dynamic lookup. The second observation is that reducing dynamic lookups has a noticable impact on performance: Vacation, Genome, and Intruder perform on par with the baseline uninstrumented code.

These experiments identified two sources of overhead that are a consequence of our design. The first overhead results from how transactions are called, with the use of lambdas adding a measurable function call overhead. If we had been willing to hard-code details of the TM implementation into the compiler, we would have been able to achieve the same reduction in this cost as we saw with "Lambda." The second overhead results from how calls to functions in separate source files are translated to the corresponding clones during execution. For programs that are able to use whole-program link-time optimization, this cost can be driven down by delaying transactional instrumentation until link time.

A natural question is whether GCC gains a fundamental advantage by embedding more information about TM into the compiler, and avoiding many of the overheads we identified. In Figure 4, we present the single-thread latency for our system and GCC. For each compiler we consider two variants: "Lock", which is as above, and "Instrumentation", which uses the compiler's TM instrumentation (for LLVM, "Dynamic_Lookup") to call a library. Within the library, there is no instrumentation. From the experiments, we see that LLVM's TM overheads are typically less significant than GCC's. The exceptions are KMeans, which receives special discussion in Section 5.4, and SSCA2, for which our system outperformed GCC at all thread counts.

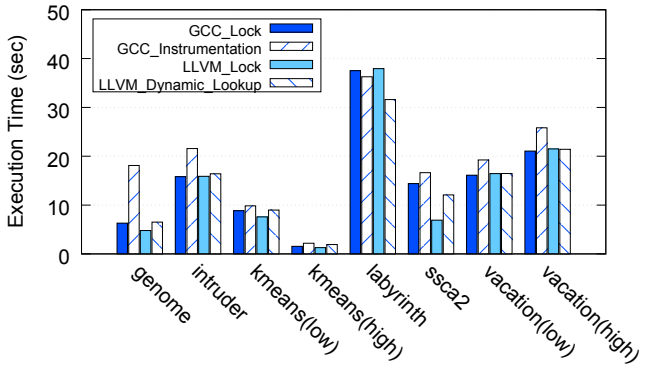


Fig. 4. Single-threaded execution, to compare the latency of GCC instrumentation with LLVM instrumentation.

5.3 Scalability

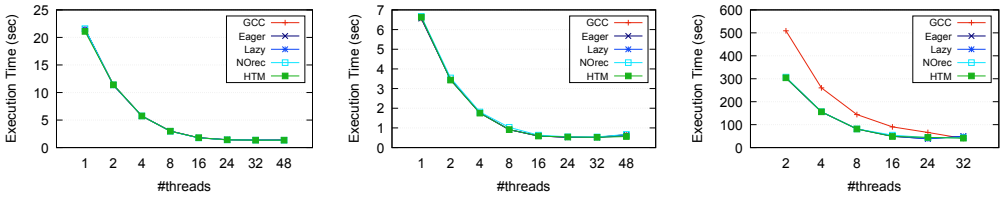
Next we assess the scalability of the benchmarks when running with multiple threads. Though it is not an ideal comparison, we compare against the TMTS by running each program with GCC's STM implementation ("GCC"). We compare against LLVM+LTO, using several TM algorithms:

- Eager: a good-faith approximation of GCC's STM algorithm, which is based on TinySTM [15].
- Lazy: a version of GCC's STM algorithm, using commit-time locking.
- NOrec: the lazy NOrec algorithm [12], which uses values instead of locks to detect conflicts among transactions.
- HTM: Intel TSX, with fallback to a single global lock after 8 consecutive aborts.

We emphasize that the point of these experiments is not to claim that one approach is better than the other. Rather, our aim is to determine if there are any scalability bottlenecks in our system that do not appear in the state-of-the-art implementation of the TMTS. Note that some variation in latency between GCC and LLVM is unavoidable, since they optimize code differently. However, since the STM algorithms are similar, we do not expect to see differences in scalability.

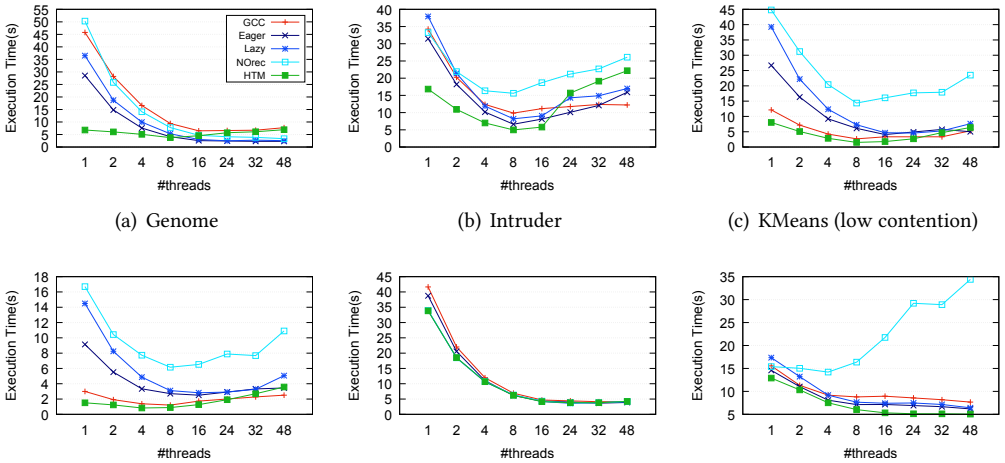
We begin with the streaming workloads, PBZip2 and x265, in Figure 5. For PBZip2, both compression and decompression behave identically for all TM approaches. Simply put, the transactions are small enough, and the I/O significant enough, that minor differences in TM implementation do not matter. X265 is more complex. First, note that x265 employs a pool of 8 worker threads. We report the number of non-frame-pool threads on the x axis, noting that the first such thread is a dedicated pool manager thread. Second, LLVM seems to do a better job vectorizing with SIMD instructions than GCC, producing a significant difference in performance at low thread counts. This difference persists, with all TM implementations converging by the time the CPU is fully employed.

Next, we present results for STAMP in Figure 6. The most significant observation is that the individual benchmarks within STAMP are different enough to show a separation among the STM algorithms. This is particularly true for NOrec, which fares poorly for workloads with a large number of small writing transactions (e.g., SSCA2). Among workloads that consistently scale (Genome, Labyrinth, SSCA2, Vacation), the trends are the same for GCC with the TMTS and LLVM with our TM design. The only exceptions are that HTM begins to degrade once hyperthreading is activated, and NOrec is more apt to degrade at high thread counts. In Intruder, where all algorithms degrade after 8 threads, our algorithms degrade more quickly. This result occurs because GCC will serialize



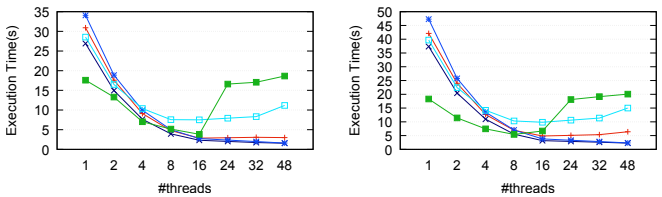
(a) PBZip2 Compression (1.1GB file) (b) PBZip2 Decompression (1.1GB file) (c) x265 Encoding (1.1GB file)

Fig. 5. Multi-threaded execution of streaming workloads. With the exception of GCC on x265, curves in each graph are essentially superimposed.



(a) Genome (b) Intruder (c) KMeans (low contention)

(d) KMeans (high contention) (e) Labyrinth (f) SSCA2



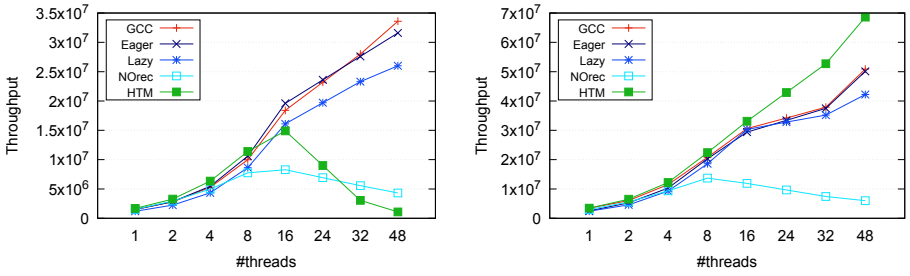
(g) Vacation (low contention) (h) Vacation (high contention)

Fig. 6. Multi-threaded execution of STAMP benchmarks, using the default parameters

transactions that abort 100 times in a row, whereas our implementations retry until succeeding, which risks livelock. While we have not implemented sophisticated per-transaction contention management [37], it would be straightforward to add to our system, since our library-based design allows passing additional flags to `TM_EXEC`.

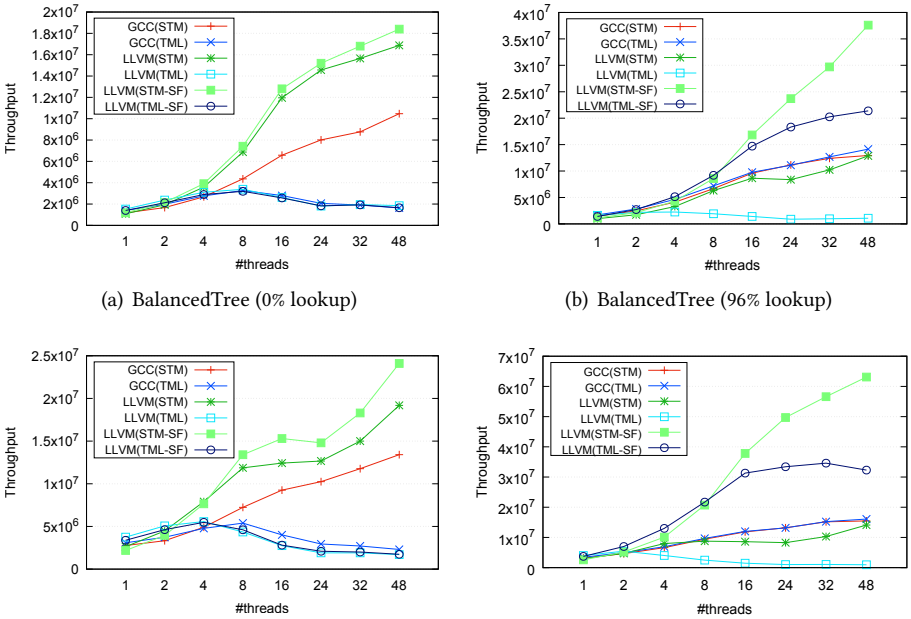
While LLVM tends to have lower latency for every workload, KMeans is an outlier. We defer discussion of KMeans until Section 5.4

Lastly, we assess the performance of the HashTable and BalancedTree data structures in Figure 7. Our first finding was that these data structures scaled worse than expected, for both GCC and



(a) BalancedTree (b) HashTable

Fig. 7. Multi-threaded execution of the data structure microbenchmarks



(a) BalancedTree (0% lookup) (b) BalancedTree (96% lookup)

(c) HashTable (0% lookup) (d) HashTable (96% lookup)

Fig. 8. Impact of stack-frame optimization for read-only transactions.

LLVM. Upon further evaluation, we were able to identify commit-time quiescence[4] as a significant contributor to poor scaling. We edited the GCC code to disable quiescence, and we modified our implementation of Eager so that a run-time flag to the executor could disable quiescence on a transaction-by-transaction basis. With quiescence disabled, both workloads exhibit good (and equivalent) scaling out to the full number of hardware threads on the CPU. However, we again observe that for HTM, scalability degrades with hyperthreading. Additionally, NOrec's scalability was much worse than expected.

[4]Quiescence is a mechanism that prevents a thread from progressing past the end of a transaction until after all concurrent transactions have reached a "safe point". It is used to ensure correctness when transactions transition data from shared to thread-private modes [40, 46].

In Section 3.2, we discussed the costs of representing transaction bodies as lambdas. The most significant was that a software transaction would incur the cost of an instrumented write when updating variables on its caller's stack frame (see, for example, Listing 3). This can impede scalability in data structure workloads, in which a set lookup must return either true or false. A lambda that updates a local variable in its parent's scope does so by writing to an address in the parent stack frame, not via a register. This is treated as a heap write by our instrumentation, and prevents such a transaction from taking the "read-only" fast-path in STM instrumentation. This behavior explains the performance of NOrec in Figure 7: it has a known bottleneck in its commit of writing transactions. To gain a deeper understanding of this scenario, we re-ran additional data structure tests, reported in Figure 8.

For these added tests, we considered two lookup ratios: 0% and 96%, where remaining operations are split evenly among inserts and removes, so that the data structure sizes remain stable. We initially consider four systems: GCC using its default STM (ml_wt), GCC using its implementation of the "TML" algorithm [10] (gl_wt), LLVM using our implementation of the GCC default STM, and LLVM using our implementation of TML. TML is not a general-purpose STM algorithm: read-only transactions can run concurrently, but a writing transaction causes all concurrent readers to abort, and to block until the writer completes.

In all experiments, our implementation of TML fails to scale, because every transaction performs an instrumented write: inserts and removes modify the data structure, and lookups write their return value to their caller's stack. In contrast, the GCC implementation of TML scales well for the 96% lookup tests, because its lookup transactions do not artificially become writers.

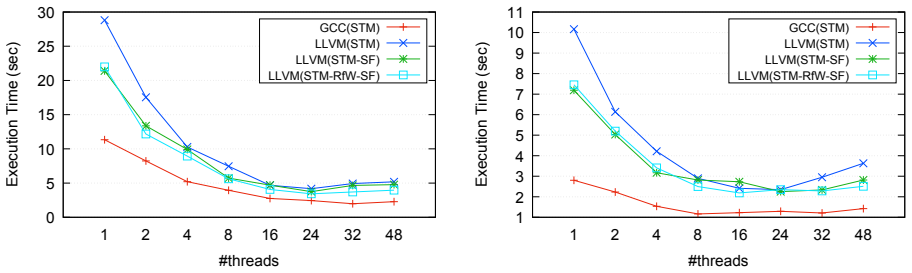
Our implementation of GCC's STM algorithm underperforms for the 96% lookup cases. This, again, is because every transaction becomes a writer. While the lookup transactions do not cause each other to abort, they do experience cache-level contention when committing (this contention is a consequence of how the STM algorithm commits writer transactions). The effect is more pronounced in the HashTable tests, since the transacitons are smaller: when all transactions are small writers, the counter becomes a bottleneck.

In the 0% lookup case, we see that both implementations of TML fail to scale, as expected. GCC's default STM scales, but especially for the HashTable test, it experiences contention on a shared counter in its commit, which reduces scalability. Our implementation of GCC's algorithm now outperforms GCC, confirming that its relatively worse performance at 96% was due to GCC taking advantage of read-only optimizations that our implementation could not employ. Extending these results back to Figure 7, we can conclude that NOrec's poor performance was indeed because our system does not allow STM implementations to optimize read-only transaction commit.

To further support this claim, we added a new annotation to our system, which lets a programmer indicate that accesses to the top of the non-transactional stack do not require instrumentation. The performance of our algorithms with this "stack frame" optimization are marked "-OPT" in the figure. With the optimization in place, the algorithms scale as expected. We expect that production systems would perform this optimization automatically.

5.4 Stack Frame and Read-for-Write Optimizations in KMeans

As discussed previously, the KMeans benchmark from STAMP showed better performance for GCC than for LLVM. In analyzing the output of the two compilers, we found two explanations. The first is that KMeans transactions perform many reads and writes of local variables in the enclosing stack frame. The second is that GCC's implementation of the TMTS optimizes five patterns for reading and writing a variable, such as read-followed-by-write (RfW). In KMeans, this pattern occurs frequently when data points are assigned to clusters. GCC identifies the RfW pattern at



(a) KMeans (low) (b) KMeans (high)

Fig. 9. Impact of employing Read-for-Write (RfW) and stack frame (SF) optimizations for the KMeans workloads.

compile time, and calls the RfW version of read functions. To assess whether these differences could explain the performance we observed, Figure 9 presents results when we applied our stack frame optimizations, and also added support for RfW to LLVM.

The figure presents results for applying RfW after stack-frame optimizations. We also looked at their impact before stack-frame optimizations. Placement did not matter: RfW had a marginal impact in all cases. However, the stack frame optimizations significantly reduced latency. For the specific case of KMeans, in which every transaction performs writes, this outcome was due to transactions using variables that are local to the enclosing stack frame, which were being accessed with full instrumentation by our system. While the stack frame optimization did not avoid calling the TM library on each of these accesses, it reduced the amount of work done by the library. The reduction in instrumentation, but not in calls to the library, manifests as performance that is significantly better, but still worse than GCC, for the "SF" curves. Refactoring the code to cache some accesses to avoid repeated instrumented accesses to these local variables reduced overhead by another 10%–15%, but still could not compete with GCC, which avoids these overheads entirely.

5.5 Summary

We identified four sources of overhead for our TM support, relative to the state of the art implementation of the TMTS in C++. Below, we summarize these overheads, and discuss mitigations.

- Lambdas – Our design has more overhead to run transactional code, because it uses lambdas instead of lexically scoped blocks. This is a consequence of our design focus on minimizing changes to compiler front ends. We showed that this cost can be reduced if the compiler has details of the implementation of `TM_EXEC` early in the compilation process.
- Dynamic Lookup – While both the TMTS and our design must provide dynamic lookup to find clones of functions called through virtual dispatch and function pointers, our design also incurs dynamic lookup overhead for calls to functions defined in separate source files. This is a consequence of our design emphasizing ease of use for programmers, wherein we eliminated the requirement to change the types of functions called from transactions. We showed that this cost can be reduced if instrumentation is performed immediately before link-time optimization, instead of during translation of source files into separate object files.
- Read-for-Write – The GCC implementation of the TMTS can statically optimize certain memory access patterns, most notably the case where a read of some location is postdominated by a write to that same location. We showed that optimization only makes a difference in one STAMP benchmark, and can easily be replicated by our system.

- Stack accesses – Our decision to use lambdas to represent transaction bodies means that a transaction's effects can never be limited to registers: there is always a memory write. These writes do not affect HTM, but cause STM algorithms to treat read-only transactions as writers, which can impede scalability. We then showed that programmers can add one optional annotation before transaction bodies to mitigate this cost. Even without mitigation, our implementations perform well, outperforming GCC in almost every case.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new design for TM extensions to C++. Our design uses an executor model instead of lexically scoped transactions, and it supports a subset of the features in the TMTS: specifically, it does not allow transactions to explicitly cancel themselves. These design points admit significantly less complexity in the implementation of TM: the compiler extension consists of a single 2K line `ModulePass`, which operates over LLVM IR without requiring any changes to the parser or language front end. They also transform the programmer experience: instead of struggling to get a program to compile with TM, developers can focus on tuning their use of TM to achieve good performance.

In our evaluation, we showed that our approach introduces modest instrumentation overhead, ranging from 2% to 40% versus uninstrumented code. Our implementation scaled as well as GCC's more advanced TM implementation when using the same TM algorithms, and did not introduce higher instrumentation overhead. We were able to achieve this performance across all TMTS-compliant applications that we could find, suggesting that our approach is robust enough to handle complex transactional programs.

As future work, we plan to integrate intraprocedural static analysis to identify and eliminate redundancies. We are encouraged by both the rapid pace of development in LLVM and its modular framework, and believe that both will enable advances that drive down the cost of TM. We hope that the lower overheads on the horizon — and the lower implementation burden posed by our design — may be the tipping point that leads at last to the standardization of TM in C++.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, as well as our colleagues in the SG5 group, for their feedback and advice. At Lehigh University, this work was supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) under grant CCF-1723624, as well as NSF grant CAREER-1253362. At the University of Rochester, this work was supported by NSF grants CCF-1422649 and CCF-1717712, and by a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel, Google, or the National Science Foundation.

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. 2006. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*. Ottawa, ON, Canada.

[2] Sara Baghsorkhi and Christos Margiolas. 2018. Automating efficient variable-grained resiliency for low-power IoT systems. In *In Proceedings of the 2018 International Symposium on Code Generation and Optimization*. Vienna, Austria.

[3] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. 2006. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters* 5, 2 (Nov. 2006).

[4] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*. San Diego, CA.

[5] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. 2014. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, AB, Canada.

[6] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. 2006. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*.

[7] Keith Chapman, Antony Hosking, and Eliot Moss. 2016. Hybrid STM/HTM for nested transactions on OpenJDK. In *In Proceedings of the 31rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*. Amsterdam, Netherlands.

[8] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. 2010. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys 2010 Conference*. Paris, France.

[9] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael Spear. 2011. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, CA.

[10] Luke Dalessandro, Dave Dice, Michael L. Scott, Nir Shavit, and Michael Spear. 2010. Transactional Mutex Locks. In *Proceedings of the Euro-Par 2010 Conference*. Ischia-Naples, Italy.

[11] Luke Dalessandro, Michael L. Scott, and Michael Spear. 2010. Transactions as the Foundation of a Memory Consistency Model. In *Proceedings of the 24th International Symposium on Distributed Computing*. Cambridge, MA.

[12] Luke Dalessandro, Michael Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*. Bangalore, India.

[13] Joel Denny, Seyong Lee, and Jeffrey Vetter. 2016. ANVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems.. In *In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. Kyoto, Japan.

[14] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*. Stockholm, Sweden.

[15] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT.

[16] Free Software Foundation. 2012. Transactional Memory in GCC. (2012). http://gcc.gnu.org/wiki/TransactionalMemory.

[17] Tim Harris, Mark Plesko, Avraham Shinar, and David Tarditi. 2006. Optimizing Memory Transactions. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*. Ottawa, ON, Canada.

[18] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, CA.

[19] ISO/IEC JTC 1/SC 22/WG 21. 2015. Technical Specification for C++ Extensions for Transactional Memory. (May 2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

[20] Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, , and Michael Spear. 2015. Transactional Tools for the Third Decade. In *Proceedings of the 10th ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR.

[21] Guy Korland, Nir Shavit, and Pascal Felber. 2010. Noninvasive Concurrency with Java STM. In *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers*. Pisa, Italy.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. Palo Alto, CA.

[23] Heiner Litz, Ricardo Dias, and David Cheriton. 2015. Efficient Correction of Anomalies in Snapshot Isolation Transactions. *ACM Transactions on Architecture and Code Optimization* 11, 4 (Dec. 2015), 65:1–65:24.

[24] Yujie Liu, Justin Gottschlich, Gilles Pokam, and Michael Spear. 2015. TSXProf: Profiling Hardware Transactions. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*. San Francisco, CA.

[25] José F. Martínez and Josep Torrellas. 2002. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA.

[26] Alexander Matveev and Nir Shavit. 2015. Reduced Hardware NORec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Istanbul, Turkey.

[27] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. 2008. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[28] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Seattle, WA.

[29] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland, OR.

[30] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*. Nashville, TN, USA.

[31] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. San Jose, CA.

[32] Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*. Austin, TX.

[33] Ravi Rajwar and James R. Goodman. 2002. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA.

[34] Wenjia Ruan and Michael Spear. 2015. Hybrid Transactional Memory Revisited. In *Proceedings of the 29th International Symposium on Distributed Computing*. Tokyo, Japan.

[35] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 2014. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, UT.

[36] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*. New York, NY.

[37] William N. Scherer III and Michael L. Scott. 2005. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*. Las Vegas, NV.

[38] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. Ottawa, ON, Canada.

[39] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. 2009. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*. Calgary, AB, Canada.

[40] Michael Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2008. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*. Luxor, Egypt.

[41] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. 2008. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*. Portland, OR.

[42] Bjarne Stroustrup. 2012. Foundations of C++ (Keynote Lecture). In *Proceedings of the European Joint Conference on Theory and Practice of Software*. Tallinn, Estonia.

[43] Takayuki Usui, Yannis Smaragdakis, Reimer Behrends, and Jacob Evans. 2009. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*. Raleigh, NC.

[44] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[45] Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, CO.

[46] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. 2008. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[47] Tingzhe Zhou, PanteA Zardoshti, and Michael Spear. 2017. Practical Experience with Transactional Lock Elision. In *Proceedings of the 46th International Conference on Parallel Processing*. Bristol, UK.