Lightweight Language-Level Support for Transactional Memory

PanteA Zardoshti
Lehigh University
paz215@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

1 Introduction

The C++ Transactional Memory Technical Specification(TMTS) [1] has not seen widespread adoption, in large part due to its complexity. In particular, the TMTS requires changes to the parser, to support a new language construct; to the type system, to enable "safe" transactions that can be undone and which can only call other "safe" functions; to exception semantics, to support transactions that self-abort; and to code generation, to insert instrumentation for checkpointing stack state and to transform memory accesses into function calls. Without a compelling performance argument, these implementation and verification costs are too high, especially since many of these requirements only are necessary for software TM.

In this poster, we show that the elimination of support for self-abort, coupled with the use of an "executor" interface to the TM system, produces a dramatically simpler TM implementation. Our solution is self-contained in 2000 lines of commented LLVM plugin code, plus TM library implementations. It is completely orthogonal to the rest of the compiler implementation. It is resilient, able to correctly compile open-source TM applications. And it is fast, performing on par with the TMTS implementation in GCC.

2 Executor Interface

The first simplifying aspect of our work is the use of an executor interface, in which lambdas are passed to a function that guarantees transactional execution. Obviously, this eliminates the need for new keywords. More significantly, it simplifies the implementation in two ways. First, the use of lambdas simplifies checkpointing of local state. In the TMTS, a transactional region that conditionally writes to local variables of its enclosing scope must instrument those writes and roll them back on abort; this instrumentation is different from the checkpointing of heap accesses. With lambdas, these accesses become equivalent to heap accesses, because the executor framework makes clear that they are not to the same stack frame as the transaction body. This eliminates local variable checkpointing from the TM library interface. Second, since a transaction's lambda executes in a different stack frame from its enclosing scope, it is easier to identify transaction-local variables. This removes runtime overhead, by avoiding checks to determine situations in which transaction-local writes can avoid instrumentation.

3 Forbidding Explicit Rollback

In the TMTS, a transaction can only call "safe" functions, i.e., functions whose type makes clear that they do not touch `atomic` variables, make system calls, perform in-line assembly, or call non-safe functions. These functions are instrumented at compile-time, so that their memory accesses can be undone. With separate compilation, it becomes difficult for programmers to start using transactions: every function must be marked safe before the program will even compile [3]. By removing explicit rollback, exception semantics become cleaner (exceptions that escape a transaction cause it to commit, whereas in the TMTS they may cause a commit or an abort, depending on the flavor of transaction being used).

In studying transactional applications, we found that most functions called by a transaction (especially in programs that use templates) are visible to the compiler and can be instrumented *without requiring programmer involvement*. Thus "viral" changes to function types are no longer required: a transaction simply serializes when it encounters code that has not been instrumented. During unit testing, these serialization points are easy to detect, and C++ annotations suffice to then coax the compiler into producing instrumentation.

4 Evaluation

For as trivial as the above changes seem, their impact is profound. An elegant solution, of only 2000 commented lines of LLVM plugin code, is able to achieve the same performance as GCC's implementation of the TMTS. Our code is decoupled from the rest of the compiler, and the semantics of the executor are cleanly defined even when the compiler does nothing (in which case the executor can simply run lambdas one at a time).

To support this claim, we measured the performance of the STAMP transactional benchmark suite [2], as well as the memcached [3], x265 [4], and PBZip2 [4] applications. Experiments were conducted on a Dell PowerEdge R640 with two Intel Xeon Platinum 8160 CPUs at 2.1GHz and 196 GB of RAM. Experiments are the average of five trials; to avoid NUMA effects, we limited execution to a single CPU Socket.

In Figure 2, we analyze the instrumentation overhead of our system for single-threaded code, using a no-op TM library. The three bars correspond to the baseline code, our LLVM plugin, the addition of link-time optimization to the

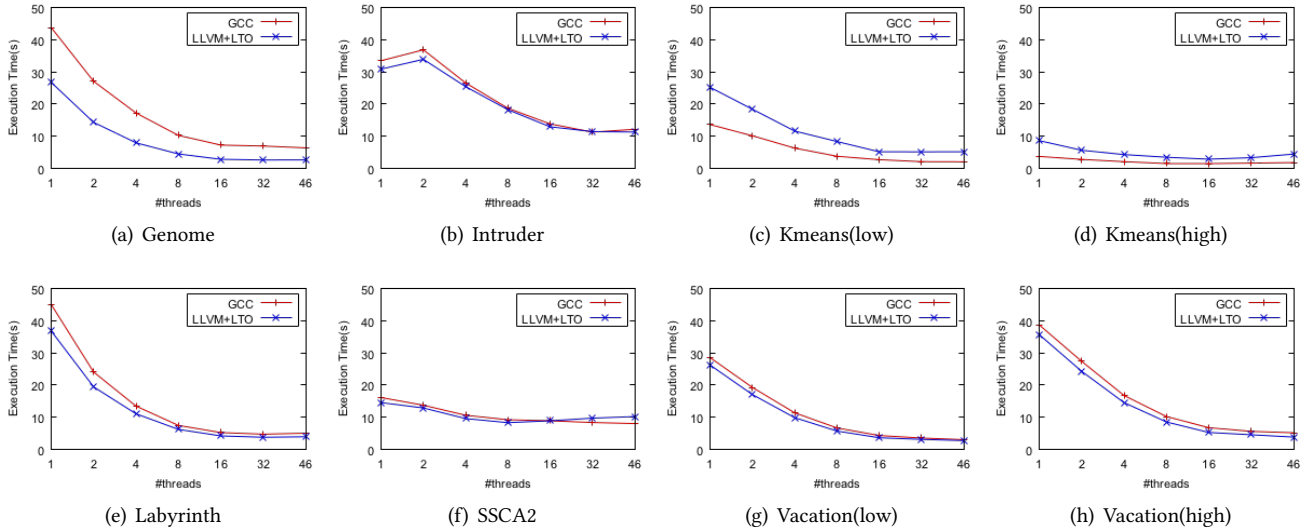

Figure 1. Stamp Benchmark

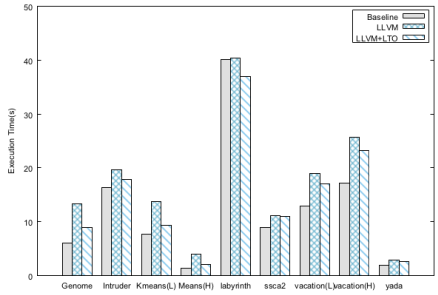

Figure 2. Single-thread execution overhead of the LLVM plugin

use of our plugin. The main take-away is that the fundamental overhead of our instrumentation, relative to the original code, is small. Despite its simpler design, our plugin does not introduce high fundamental overheads. Most importantly, LTO alone suffices to reduce most of the costs, and the remaining costs, which are primarily due to lambdas, are of interest to all uses of executors.

Due to space constraints, Figure 1 restricts to investigating performance on the STAMP benchmarks. Here, the main takeaway is that our approach is competitive with GCC's implementation: across STAMP and the other benchmarks, we are able to achieve the same performance, or sometimes a constant factor better, while supporting the same features and implementing the same TM algorithms.

While our results are limited to an apples-to-apples comparison of the same software TM algorithm in GCC and our system, we have successfully implemented many more software TM algorithms in our system, and we found the process to be much simpler than when we did the same in GCC. Anecdotally, this suggests that our approach simplifies TM-related programming tasks at all levels, from compiler to library to application.

5 Conclusions

We believe that this work establishes a clear path forward for the standardization of TM in C++. The simpler "executor" interface, coupled with the elimination of self-abort, makes for a fast, powerful approach to TM that can be implemented in a compiler and run-time library in a piecemeal fashion, and is able to handle the most advanced TM applications available. Our work also establishes a need for more effort in optimizing lambda support in C++ compilers not only for TM, but for any executor framework.

References

[1] 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

[2] 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.

[3] 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.

[4] 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.