Universal Support for Scoped Memory Access Instrumentation

Pantea Zardoshti
Lehigh University
paz215@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

1 Introduction

Modern microprocessors are increasingly relying on the memory system to deliver value. However, this value is no longer transparent. Gone are the days where deeper memory hierarchies, larger caches, and smarter prefetching could reliably improve the performance of existing code. Instead, new memory features like persistent memory [2, 13], scratchpads [1], secure enclaves [5], and transactional memory [3, 4] all require explicit programmer intervention.

These features introduce several new requirements for programmers. Each feature introduces nuanced semantics governing how certain parts of memory operate. Most require fall-back code to handle when the feature is unavailable or capacity-constrained [10, 11]. Some features do not (yet) compose with each other. More broadly, the features remain experimental, and lack rich language and compiler support. Some, such as transactional memory, require changes to the typing rules for functions [6]. Others, such as persistent memory, require the addition of assembly "fence" instructions on loads and stores, and the transformation of pointers to self-referential offsets.

At the same time as hardware vendors are turning to advanced memory features as a source of performance and programmer productivity, researchers have increasingly used combinations of static and dynamic memory instrumentation for such features as race detection, profiling for locality, and taint analysis.

We present a system for memory instrumentation of programmer-annotated code regions. Our system is implemented within the LLVM compiler framework [8], and supports C and C++ programs. Through careful use of features of the LLVM compiler (such as user-defined function annotations) and of C++11 (such as lambdas), our system does not require any changes to the language or parser: all analysis and transformation is performed as a pass over the intermediate representation of the code. In addition, our system allows for incremental instrumentation of legacy programs, supports separate compilation, and provides a hybrid static/-dynamic linking mechanism, through which programmers can intercept and redirect calls to un-instrumented libraries. Individual memory features (currently TM and a memory profiler) are implemented as separate libraries, which are optimized at link-time to keep overheads to a minimum while providing a flexible framework for adding arbitrary memory instrumentation to C and C++ programs.

2 An Annotation-Based API

In this work, we are focused on instrumenting the memory accesses made within marked regions of code. Thus we do not require special annotations on data; instead, data accesses will be instrumented only when they are made within a marked region.

We provide two mechanisms for identifying the start of a region requiring memory instrumentation. In C++ programs, the statement `EXECUTE(λ,F)` indicates that the code represented by λ should have its memory accesses instrumented according to feature `F`. In effect, when `EXECUTE` is reached, memory instrumentation for `F` will be enabled, and when `EXECUTE` returns, memory instrumentation for `F` will be disabled. For C programs, we use a similar approach: `EXECUTE_C(function, args, F)` takes the name of a function, an opaque pointer, and a feature specifier `F`. When reached, memory instrumentation will be enabled, the function will be executed (with `args` as its parameter), and then when the function returns, memory instrumentation will be disabled. Nesting of features is allowed, but once a feature `F` is enabled, a region nested within it cannot disable `F`.

When an instrumented region calls a function that is defined in a separate source file, we require that function to be annotated with `INSTRUMENTED`. Failure to mark such functions results in a run-time warning or error, depending on the memory feature. We also provide the annotation `NOT_INSTRUMENTED`, to indicate that all memory operations performed by a function at a particular call site, as well as those functions reached from it, do not require instrumentation. This is useful for math and other third-party libraries. Finally, the `RENAME_INSTRUMENT(x)` annotation indicates that a function should serve as a substitute for function `x` in situations where `x` would other require its memory accesses to be instrumented.

There are two corner cases for which the API requires special thought. First, LLVM does not allow user-defined annotations on constructors. To indicate that a constructor is reached from an instrumented region, we add the `INSTRUMENT_CONSTRUCTOR()` function, placed in the constructor body. This stub function does not result in any additional run-time constructor code, but informs our system that the constructor's memory accesses (and those of any functions reachable from the constructor) will be instrumented. Second, the use of C++ lambdas can introduce subtle changes

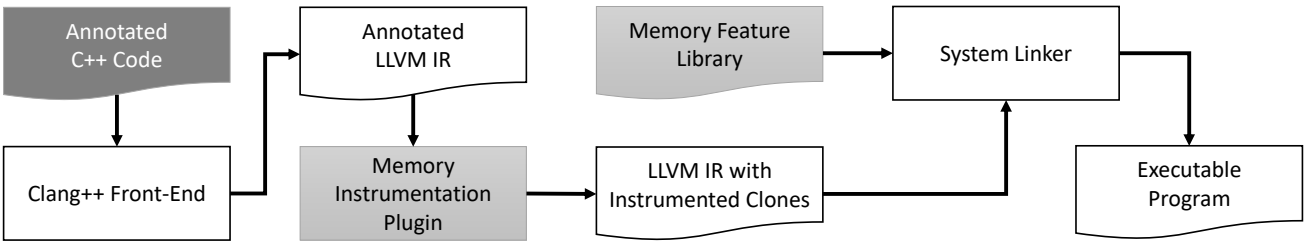

Figure 1. Overall system design. Programmer involvement is limited to annotating the source code (dark gray). New components in the compilation toolchain are shaded light gray.

in control flow. When a program is affected by this nuance, there is a trivial two-line edit to the source code.

3 System Workflow

The annotations in Section 2 cause LLVM's compilers for C and C++ (`clang` and `clang++`) to add function calls and/or annotations to the intermediate representation (IR) that it produces. Our system applies a custom LLVM plugin to this annotated IR to transform it so that every interaction with memory from an annotated code region becomes a function call to a library that provides the appropriate memory instrumentation, and so that the boundaries of each instrumented memory region entail function calls to the library for initializing and finalizing instrumentation. The overall system architecture appears in Figure 1.

The behavior of our memory instrumentation plugin appears in Figure 2. There are five steps, as discussed below:

Discovery: The annotations establish a root set of functions that may be called from an instrumented region. We seed a worklist with these functions, and then analyze each to find all functions reachable from the root set. This operation is a single pass over the IR, and takes linear time. Its output is a set of all functions within the IR that require memory access instrumentation.

Cloning and Instrumentation: Functions reachable from annotated regions may also be reached from regular code. To handle this possibility, we create a clone of each function identified by the discovery phase, and instrument the clone. The names of these clones are generated in a manner compatible with C++ name mangling. For each cloned function, our plugin replaces all loads and stores with calls to the library. These calls are parameterized based on the primitive type being accessed, and any qualifiers (e.g., `volatile` or `atomic`). For each function call within the clone, we replace it with either (a) a direct call to the clone of the function, in situations where the called function is defined in the same source file, or (b) an indirect call to the library, so that the appropriate instrumented clone can be found at run-time. Special functions and compiler intrinsic functions, such as `malloc` and `memcpy`, are replaced with calls to the library. A pair consisting of the original function and its clone are also saved in a set, to be used in the Clone Map Generation phase. Functions annotated with `NOT_INSTRUMENTED` are not cloned or instrumented, but are added to the set. Functions annotated with `RENAME_INSTRUMENT` are instrumented but not cloned, and are also added to the set. As with Discovery, this is a single linear-time pass over the IR.

Boundary Instrumentation: The boundary instrumentation step modifies calls to `EXECUTE` and `EXECUTE_C`, so that instrumented functions are called instead of the originals. As it does so, it alters the call signature so that both the original and instrumented version of the function are passed to the corresponding library `EXECUTE` call. This makes it possible for certain language features, such as TM, to use uninstrumented code for those settings where hardware can fully provide the desired memory behavior on the original code. This transformation is linear in the number of `EXECUTE` statements in the IR.

Optimization: There are two types of optimization that we consider. The first is the set of optimizations that are general to all forms of memory instrumentation. Examples include optimizing reads that are known to be post-dominated by a write to the same location, caching redundant dynamic lookup of function clones, and removing other redundant library calls. The second set of optimizations are feature-specific optimizations. While supported, such optimizations, which may have arbitrary asymptotic complexity, are outside of the scope of this abstract.

Clone Map Generation: When an instrumented region calls a 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 that comprise a program. During Cloning and Instrumentation, our system produces a set of these pairs. During the Clone Map Generation step, we create a static initializer for the object file. The initializer we generate performs a set of calls to the library, which register these function/clone pairs. In this manner, the library will have the union of all pairs, from all of its source

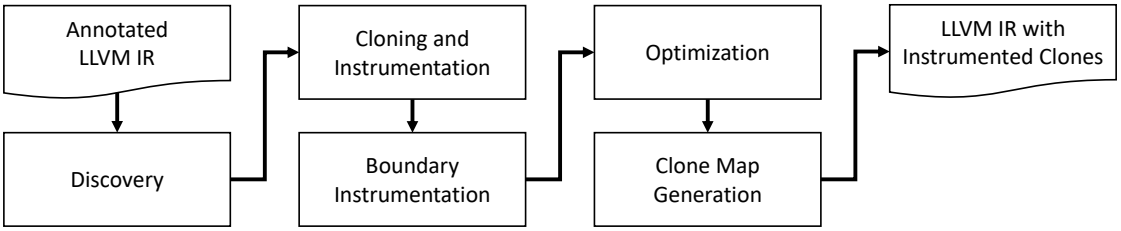

Figure 2. Code Region Plugin Design

files, available at run time. The creation of this initializer is linear in the number of instrumented functions in the IR.

4 Implementation

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

In terms of memory features, we currently have libraries to support transactional memory and memory access profiling. Our basic profiler is implemented in 300 lines of library code, and generates files that can be analyzed off-line to understand memory behaviors as well as instrumentation errors in programs. For TM, we ported 10 software TM libraries from the latest version of the RSTM suite [7]. Each took under 600 lines of code to implement in full. Libraries for encrypted memory and persistent memory are under active development.

Beyond unit testing, we have also tested our implementation on a set of existing transactional benchmarks and real-world programs: STAMP [9], memcached [12], the pbzip2 file compressor [14], and the x265 video codec [14]. Broadly, performance is on par with the best existing TM implementations, such as those in GCC, and the programmer effort to use our system was much less than the effort it took to use the existing approach for TM [6]. Based on these findings, we expect our system to offer broad value to several communities, including those interested in concurrency, persistence, and security.

References

[1] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *Proc. of CODES+ISSS*. 73–78.

[2] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, CA.

[3] 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.

[4] Intel Corporation. 2012. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). (Feb. 2012).

[5] Intel Corporation. 2018. Intel Software Guard Extensions (Intel SGX). (March 2018). https://software.intel.com/en-us/sgx

[6] 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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2008. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[12] 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.

[13] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

[14] 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.