The CSI Framework for Compiler-Inserted Program Instrumentation

Tao B. Schardl
Massachusetts Institute of Technology
neboat@mit.edu

Tyler Denniston
Massachusetts Institute of Technology
tdennist@mit.edu

Damon Doucet
Massachusetts Institute of Technology
ddoucet@mit.edu

Bradley C. Kuszmaul
Massachusetts Institute of Technology
bradley@mit.edu

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

Charles E. Leiserson
Massachusetts Institute of Technology
cel@mit.edu

ABSTRACT

The CSI framework [15] provides comprehensive static instrumentation that a compiler can insert into a program-under-test so that dynamic-analysis tools — memory checkers, race detectors, cache simulators, performance profilers, code-coverage analyzers, etc. — can observe and investigate runtime behavior. Heretofore, tools based on compiler instrumentation would each separately modify the compiler to insert their own instrumentation. In contrast, CSI inserts a *standard* collection of instrumentation hooks into the program-under-test. Each CSI-tool is implemented as a library that defines relevant hooks, and the remaining hooks are “nulled” out and elided during either compile-time or link-time optimization, resulting in instrumented runtimes on par with custom instrumentation. CSI allows many compiler-based tools to be written as simple libraries without modifying the compiler, lowering the bar for the development of dynamic-analysis tools.

We have defined a standard API for CSI and modified LLVM to insert CSI hooks into the compiler’s internal representation (IR) of the program. The API organizes IR objects — such as functions, basic blocks, and memory accesses — into flat and compact ID spaces, which not only simplifies the building of tools, but surprisingly enables faster maintenance of IR-object data than do traditional hash tables. CSI hooks contain a “property” parameter that allows tools to customize behavior based on static information without introducing overhead. CSI provides “forensic” tables that tools can use to associate IR objects with source-code locations and to relate IR objects to each other.

To evaluate the efficacy of CSI, we implemented six demonstration CSI-tools. One of our studies shows that compiling with CSI and linking with the “null” CSI-tool produces a tool-instrumented executable that is as fast as the original uninstrumented code. Another study, using a CSI port of Google’s ThreadSanitizer, shows that the CSI-tool rivals the performance of Google’s custom compiler-based

implementation. All other demonstration CSI tools slow down the execution of the program-under-test by less than 70%.

CCS CONCEPTS

• **Software and its engineering** → **Compilers; Software testing and debugging;**

KEYWORDS

Program instrumentation; dynamic program analysis; compiler-inserted instrumentation

ACM Reference Format:

Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2018. The CSI Framework for Compiler-Inserted Program Instrumentation. In *SIGMETRICS '18 Abstracts: ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems Abstracts, June 18–22, 2018, Irvine, CA, USA*. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3219617.3219657>

1. Extended abstract

Key to understanding and improving the behavior of any system is *visibility* — the ability to know what is going on inside the system. For application and system software, *compiler-inserted program instrumentation* (or simply *compiler instrumentation*) — where the compiler inserts special code into the program-under-test to monitor its execution — has emerged as a popular way for programmers to gain visibility into how their programs are operating. Programmers today can avail themselves of a variety of instrumentation-based dynamic-analysis tools, such as race detectors [3–5, 12–14, 18], memory checkers [1, 8, 17], cache simulators [2, 20, 25], call-graph generators [7, 10], code-coverage analyzers [22, 23], and performance and scalability profilers [9, 16, 24]. These tools generally operate as *shadow computations* — executing behind the scenes while the program-under-test runs.

Compiler instrumentation allows the tool writer to take advantage of the compiler’s low-level representation of the code and the compiler’s analyses to make the instrumentation precise and “surgical,” that is, to instrument only the subset of events the tool cares about. For instance, a race detector can avoid instrumenting accesses to a stack variable if the compiler analysis can prove that the stack variable never escapes the function scope and thus is thread-local. Furthermore, the tool writer can use the compiler’s optimizations to optimize the instrumented program. As a result, tool writers have used compiler instrumentation to write dynamic-analysis tools

This research was supported in part by a Google Faculty Research Grant, in part by DoE X-Stack Grant DE-SC0008923, and in part by NSF Grants 1314547, 1533644, and 1527692. Damon Doucet was supported in part by an MIT EECS SuperUROP.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGMETRICS '18 Abstracts, June 18–22, 2018, Irvine, CA, USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5846-0/18/06.

<https://doi.org/10.1145/3219617.3219657>

that produce precise results with low overhead. Google's AddressSanitizer [17], for example, detects memory bugs at the point of occurrence while exhibiting less than 73% average slowdown.

A major disadvantage of compiler instrumentation, however, is that developing a new tool requires the compiler to be modified. Modifying the compiler requires compiler expertise, which many potential tool writers do not have. Even though they may be capable of designing and implementing the tool itself, mastering the large and complex codebase of a mainstream compiler today is daunting. Some mainstream compilers allow changes to be incorporated as a plugin [6, 11, 21], but the tool writer still requires significant expertise in the compiler's codebase to develop such a plugin. Moreover, whether implemented directly or via a plugin, the tool writer must maintain and update her compiler modifications whenever the compiler codebase undergoes a change that interferes with the tool writer's instrumentation. Finally, if the tool writer wishes her tool to be available for several compilers, it is up to her to make the modifications in each compiler she wishes to support. Some of these impediments can be mitigated by upstreaming the compiler changes into the main branch of the compiler codebase. But that process is itself onerous and may require substantially more work than was required to develop the tool itself. Regardless, it hinders the rapid prototyping and testing of new and innovative tools.

The CSI framework

CSI aims to make it easy for tool writers to produce innovative compiler-based tools without modifying the compiler. CSI provides a simple application program interface (API) consisting of functions, called *hooks*, which are automatically inserted throughout the compiled code of the program-under-test. The CSI API exposes generic instrumentation points throughout the program-under-test, thereby hiding the complexity of the compiler's codebase from tool writers. Different compilers should be able to support the same CSI API, making it possible to write compiler-independent CSI-tools.

Tool writers insert their own instrumentation into the program-under-test by writing a library that defines the semantics of relevant hooks, with unimplemented hooks defaulting to null behavior. To use a particular CSI-tool, a tool user links the compiled CSI-tool with program-under-test to produce a *tool-instrumented executable (TIX)*. When the TIX executes, the program-under-test runs normally, except that whenever a hook is invoked, the tool performs its shadow computation. To support a wide variety of tools, CSI inserts hooks to instrument many events that a tool might care about, for instance, before and after a memory access, function entry and function exit, beginning and end of a basic block, etc.

At first glance, this brute-force method of inserting hooks at every salient location in the program-under-test seems replete with overheads. For example, the function calls for the two hooks that CSI inserts around each memory operation might contribute significant unnecessary runtime overhead if a tool does not use these hooks.

The full paper [15] describes how CSI overcomes overheads through the use of modern compiler technology, such as link-time optimization (LTO) [19]. We show that CSI's strategy for generic instrumentation incurs essentially zero overhead. Furthermore, we show that Google's ThreadSanitizer, when implemented as a CSI-tool, gets similar performance compared to the original ThreadSanitizer, which uses custom compiler instrumentation.

```

01 static long *block_executed = NULL;
02 static csi_id_t num_basic_blocks = 0;
03
04 void report() {
05     csi_id_t num_basic_blocks_executed = 0;
06     fprintf(stderr, "CSI-cov report:\n");
07     for (csi_id_t i = 0; i < num_basic_blocks; i++) {
08         if (block_executed[i] > 0)
09             num_basic_blocks_executed++;
10         const source_loc_t *source_loc =
11             __csi_get_bb_source_loc(i);
12         if (NULL != source_loc)
13             fprintf(stderr, "%s:%d-%d executed %d times\n",
14                     source_loc->filename,
15                     source_loc->start_line,
16                     source_loc->end_line, block_executed[i]);
17     }
18     fprintf(stderr,
19             "Total: %ld of %ld basic blocks executed\n",
20             num_basic_blocks_executed, num_basic_blocks);
21     free(block_executed);
22 }
23
24 void __csi_init() {
25     atexit(report);
26 }
27
28 void __csi_unit_init(const char * const name,
29                     const instrumentation_counts_t counts) {
30     block_executed =
31         (long *)realloc(block_executed,
32                         (num_basic_blocks + counts.num_bb)
33                         * sizeof(long));
34     memset(block_executed + num_basic_blocks, 0,
35            counts.num_bb * sizeof(long));
36     num_basic_blocks += counts.num_bb;
37 }
38
39 void __csi_bb_entry(const csi_id_t bb_id,
40                    const bb_prop_t prop) {
41     block_executed[bb_id]++;
42 }

```

Figure 1: A serial version of the CSI-cov code-coverage tool, which reports the number of times every basic block in a TIX is executed.

A brief tutorial on CSI

To demonstrate the CSI API, we present CSI-cov, a simple tool that reports code-coverage information for an execution of the TIX of a program-under-test. As the TIX executes, CSI-cov records when each basic block runs. When the TIX terminates, CSI-cov reports how many times each block was executed, including its location in the source code.

The code for CSI-cov, shown in Figure 1 in its entirety, is both short and simple.¹ In just 42 lines, the CSI framework implements this useful compiler-based tool as a simple C library. Without the

¹This code as written assumes that the program-under-test executes serially. Tool writers must take additional care for allocating, initializing, and accessing `block_executed` in order for CSI-cov to handle a multithreaded program-under-test.

CSI framework, a code-coverage tool based on compiler instrumentation would require considerably more development effort, as well as an understanding of the internals of the compiler.

Let us walk through this code. CSI-cov maintains a table `block_executed`, defined in line 1, in which each basic block in the TIX has a unique slot indexed by the CSI ID assigned to that basic block. The number of basic blocks is stored in the variable `num_basic_blocks`, defined in line 2.

The CSI runtime system invokes the initialization hook `__csi_init` when the program-under-test is loaded. This hook, defined in lines 24–26, registers the function `report` to run when the program-under-test terminates. The `report` function, defined in lines 4–22, prints out a report when the program-under-test terminates. For each basic block, `report` retrieves its source location from CSI’s forensic tables using the accessor function `__csi_get_bb_source_loc` in line 11. It prints the file name, the range of source lines, and the number of times the basic block has been executed in lines 13–16. Finally, `report` prints the total number of basic blocks executed and the total number of basic blocks in the program in lines 19–20.

The CSI runtime system calls the hook `__csi_unit_init` whenever a new *translation unit* — a separate unit of compilation — is loaded, either statically or dynamically. The hook is called with the name of the translation unit and a structure that provides the number of instrumented functions, basic blocks, memory accesses, etc. that the translation unit contains. CSI-cov defines this hook in lines 28–37 to reallocate the array `block_executed` to incorporate the new basic blocks in lines 31–33, after which it updates the number `num_basic_blocks` of basic blocks in line 36.

The hook `__csi_bb_entry`, defined in lines 40–42, is called every time a basic block is executed. The hook has two arguments: the CSI ID of the basic block and a “property,” which CSI-cov ignores. At runtime, CSI-cov indexes `block_executed` and increments the counter for that basic block.

The CSI-cov example illustrates three key features of CSI: flat and compact CSI ID spaces, hooks, and forensic tables. Because CSI assigns ID’s to basic blocks from a flat and compact ID space, not only can CSI-cov instrument each executed block using a simple array instead of a hash table, it can easily identify which basic blocks were *not* executed. CSI-cov employs only three hooks. The other hooks need not be specified and are automatically elided, leaving the code uncluttered. The forensic tables allow the source locations of the basic blocks to be easily determined.

Contributions

The full paper on CSI [15] makes the following contributions:

- The CSI API, which defines compiler-inserted instrumentation hooks to allow dynamic-analysis tools to be built as simple libraries.
- CSI:LLVM, an implementation of CSI within the LLVM compiler, which modifies LLVM and Clang to insert CSI instrumentation into programs-under-test and uses LTO to optimize TIX’s.
- Six demonstration CSI-tools that explore CSI’s utility, ease of programming, and performance. The tools include the null tool, a code-coverage tool, a memory-operations counter, a lightweight performance profiler, a dynamic call-graph generator, and a port of Google’s ThreadSanitizer [18].

- Experiments showing that programs-under-test instrumented with a CSI-tool can run fast, incurring minimal slow down compared to an uninstrumented version of that program. In particular, all of the demonstration CSI tools other than the port of ThreadSanitizer to CSI incur less than 70% slowdown.
- Experiments showing that programs-under-test instrumented with a CSI-tool can run as fast, or nearly as fast, as comparable custom-instrumented compiler-based tools.

REFERENCES

- [1] David R. Barach, David H. Taenzer, and Robert E. Wells. 1982. A Technique for Finding Storage Allocation Errors in C-language Programs. *SIGPLAN Notices* 17, 5 (May 1982), 16–24.
- [2] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. In *PLDI*. 245–257.
- [3] Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*. 85–96.
- [4] Anne Carolyn Dinning. 1990. *Detecting Nondeterminism in Shared Memory Parallel Programs*. Ph.D. Dissertation. Department of Computer Science, New York University.
- [5] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.
- [6] Free Software Foundation. 2017. GNU Compiler Collection (GCC) Internals: Plugins. Available at <https://gcc.gnu.org/onlinedocs/gccint/Plugins.html>. (April 2017).
- [7] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 1982. gprof: A Call Graph Execution Profiler. In *SIGPLAN '82 Symposium on Compiler Construction*. 120–126.
- [8] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*. 125–138.
- [9] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA*. 145–156.
- [10] Rohit Jalan and Arun Kejariwal. 2012. Trin-Trin: Who’s calling? A Pin-Based Dynamic Call Graph Extraction Framework. *International Journal of Parallel Programming* 40, 4 (2012), 410–442.
- [11] LLVM Project. 2016. Writing an LLVM Pass. Available at <http://llvm.org/docs/WritingAnLLVMPass.html>. (2016).
- [12] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Supercomputing*. 24–33.
- [13] John Mellor-Crummey. 1993. Compile-Time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs. In *PADD*. 129–139.
- [14] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems* 15, 4 (Nov. 1997), 391–411.
- [15] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. *POMACS* 1, 2, Article 43 (Dec. 2017), 43:1–43:25 pages.
- [16] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *SPAA*. 89–100.
- [17] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*.
- [18] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer — Data Race Detection in Practice. In *WBI*. 62–71.
- [19] Amitabh Srivastava and David W. Wall. 1992. *A Practical System for Intermodule Code Optimization at Link-Time*. Technical Report 92/6. Digital Western Research Laboratory.
- [20] Rabin A. Sugumar and Santosh G. Abraham. 1993. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. In *SIGMETRICS*. 24–35.
- [21] The Clang Team. 2017. Clang Plugins — Clang 5 Documentation. Available at <https://clang.llvm.org/docs/ClangPlugins.html>. (2017).
- [22] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. In *ISSTA*. 86–96.
- [23] William von Hagen. 2006. *The Definitive Guide to GCC* (second ed.). Apress, Chapter 6.
- [24] David W. Wall. 1989. *Link-Time Code Modification*. Technical Report 89/17. Digital Western Research Laboratory.
- [25] Josef Weidendorfer. 2008. Sequential Performance Analysis with Callgrind and KCachegrind. In *2nd International Workshop on Parallel Tools for High Performance Computing*. 93–113.