The CSI Framework for Compiler-Inserted Program Instrumentation

TAO B. SCHARDL, Massachusetts Institute of Technology, USA
TYLER DENNISTON, Massachusetts Institute of Technology, USA
DAMON DOUCET, Massachusetts Institute of Technology, USA
BRADLEY C. KUSZMAUL, Massachusetts Institute of Technology, USA
I-TING ANGELINA LEE, Washington University in St. Louis, USA
CHARLES E. LEISERSON, Massachusetts Institute of Technology, USA

The CSI framework 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**;

Additional Key Words and Phrases: Program instrumentation; dynamic program analysis; compiler-inserted instrumentation

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.

Authors' addresses: T. B. Schardl and C. E. Leiserson, MIT CSAIL, 32 Vassar St., Cambridge, MA 02139; T. Denniston, (Current address) self employed; D. Doucet, (Current address) Benchling, 1122 Howard St., 3rd floor, San Francisco, CA 94103; B. C. Kuszmaul, (Current address) Oracle, Burlington, MA; I-T. A. Lee, Washington University in St. Louis, 1 Brookings Drive, Campus Box 1045, St. Louis, MO 63130.

2476-1249/2017/12-ART43 $15.00
https://doi.org/10.1145/3154502

ACM Reference Format:
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. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2, Article 43 (December 2017), 25 pages. https://doi.org/10.1145/3154502

1. Introduction

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.[1] Programmers today can avail themselves of a variety of instrumentation-based dynamic-analysis tools, such as race detectors [17, 18, 20, 46, 47, 52, 55], memory checkers [3, 29, 54], cache simulators [16, 64, 72], call-graph generators [28, 32], code-coverage analyzers [67, 70], and performance and scalability profilers [30, 53, 71]. 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 certain stack variables if the compiler analysis can prove that those stack variables never escape the function scope and thus are thread-local. Furthermore, the tool writer can take advantage of the compiler's optimizations to optimize the instrumented program. As a result, tool writers have used compiler instrumentation to write dynamic-analysis tools that produce precise results with low overhead. Google's AddressSanitizer [54], 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, creating impediments for both tool writers and tool users.

For tool writers, modifying the compiler requires compiler expertise, a skill that 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 do allow changes to be incorporated as a plugin [24, 43, 66], 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's compiler modifications also incur an ongoing maintenance problem. Whenever the compiler codebase undergoes a change that interferes with the tool writer's instrumentation, she must bear the responsibility of updating her modifications. 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.

For tool users, using a dynamic-analysis tool typically involves using a custom version of the compiler that inserts the necessary program instrumentation. The tool user must download the custom compiler and either trust the tool writer's modifications for creating the production executable for the program-under-test or use his original compiler. The problem of multiple

[1] Other strategies include ***binary instrumentation*** [6, 15, 37, 44, 48, 51, 58, 59] and ***asynchronous sampling*** [10, 28, 50]. Section 6 compares binary and compiler instrumentation.

compilers exacerbates itself if the tool user is employing several tools, each of which needs its own custom compiler. Unfortunately, it is also common that the custom compiler is out of date with respect to the latest compiler release. Who wants to risk relying on a tool that holds you back from using the latest software technology? Finally, a given tool is typically only available for a particular mainstream compiler, unless the tool writer has exerted the sizable effort to upstream the tool to multiple compiler codebases.

The CSI framework

CSI aims to simplify many aspects of writing and using tools that employ compiler instrumentation by providing comprehensive static instrumentation and removing the need for tool writers to interact directly with the compiler codebase. Specifically, 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 and making the CSI approach largely compiler independent.

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. A tool user uses the CSI framework to link the compiled CSI-tool with the 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. In order 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 CSI instrumentation of a memory operation involves calls to two hooks, one before the operation and one after. If a tool does not use these hooks, the cost of the function calls might contribute significantly and unnecessarily to runtime overhead. This paper demonstrates, however, that the overheads of generic instrumentation can be mitigated, allowing tools to be produced as libraries whose efficiency rivals that of tools produced with custom instrumentation.

CSI overcomes overheads through the use of modern compiler technology. In this paper, we shall primarily describe how CSI works using ***link-time optimization (LTO)*** [60], which is now readily available in most major compilers, including GCC [21] and LLVM [34, 42]. Conceptually, LTO allows compiler optimizations to run when the program's compiled units are statically linked together, thereby allowing the compiler to perform optimizations between these separate units. Section 5 describes how LTO enables a simple design for CSI and produces a TIX that is as efficient as a compile-time-optimized executable. In fact, in Section 4, we demonstrate that CSI's strategy for generic instrumentation incurs essentially zero overhead. We also demonstrate that Google's ThreadSanitizer, when implemented as a CSI-tool, gets similar performance compared to the original ThreadSanitizer, which uses custom compiler instrumentation.

CSI can also overcome overheads of its brute-force approach to instrumentation using ordinary compile-time optimization (CTO), instead of LTO. Although we have not implemented a complete CSI system that uses CTO, we have tested components of its design. As Section 5 describes, the design of CSI using CTO is more complicated than that using LTO, but it should provide the same capability for compiler optimizations to optimize CSI instrumentation. By using CTO, CSI need not rely on LTO for performance, allowing CSI-tools to be used on programs where compilation using LTO might be problematic [34].

The CSI framework simplifies the tasks of writing and using dynamic-analysis tools based on compiler instrumentation. With the CSI framework, tool writers can quickly and effectively implement such dynamic-analysis tools without having to modify the compiler themselves. As the compiler software evolves, the tool writer need not worry about accommodating changes to the compiler's internals, as long as the compiler continues to support CSI's API. Tool users, meanwhile, can instrument their program with any tool built using CSI's API, and they are free to use any compiler that supports CSI instrumentation. In a sense, the CSI framework supports a form of aspect-oriented programming [35] specialized for dynamic program analysis.

CSI also simplifies the code-maintenance task that compiler writers take on to support dynamic-analysis tools. Although tool writers that use CSI need not worry about maintaining their tools as the compiler's codebase evolves, compiler developers must maintain support for the CSI API in the compiler. But without CSI, compiler developers must take on an additional code-maintenance task for each dynamic-analysis tool that is upstreamed to the compiler codebase. Because CSI provides a single framework that supports a wide variety of tools, compiler developers can support all CSI tools by simply maintaining support for the CSI API, not the collection of codebases for the individual tools.

Design of the CSI API

CSI targets the ***intermediate representation (IR)*** of a mainstream compiler, such as GCC [61] or LLVM [38]. CSI instruments several ***categories*** of ***IR objects*** — memory operations, basic blocks, functions, etc. — that are commonly represented in a mainstream compiler's IR. These objects are generally familiar to programmers with introductory knowledge of compilers, which makes it easy for tool writers to understand them. By targeting common categories of IR objects, CSI provides a generic API that is largely platform independent from the point of view of tool writers and tool users.[2]

The design of the CSI API is centered around two main considerations: (1) to allow for simple library-based implementations of a wide-variety of dynamic-analysis tools, and (2) to enable tools to run fast by exploiting standard compiler optimizations and analyses for precision and efficiency. To these ends, the CSI API provides four key features.

Flat CSI ID spaces. To each IR object, CSI assigns a ***CSI ID***, an integer identifier for the IR object that is unique within its category. At runtime, these ID's are passed to hooks to identify the particular IR object or objects being instrumented. CSI assigns these ID's in such a way as to maintain a flat and compact ID space for each IR-object category, even parts of the program-under-test are loaded at runtime as dynamic libraries. As described in Section 2, these flat and compact ID's greatly simplify tool design by allowing a tool to use simple arrays, rather than hash tables, to store information for each IR object. Such arrays are simpler to maintain than hash tables, especially in multithreaded programs, and often perform more efficiently than hash tables, as discussed in Section 4.

Hooks. CSI defines two kinds of hooks: initialization hooks and IR-object hooks. The initialization hooks allow tools to perform actions immediately before `main` is invoked and when a translation unit is loaded. The IR-object hooks allow tools to perform actions before and/or after an IR object is executed at runtime. The compiler automatically elides any hooks left undefined by the tool writer.

Properties. For each IR object, CSI computes a ***property***, which is a compile-time constant that encodes the results of standard compiler analyses. The property for a memory operation, for example, might specify that the location is guaranteed to be on the stack or have a particular alignment.

[2] One place where the CSI API perhaps departs from full compiler agnosticism is that LLVM's and GCC's basic-block semantics differ from the traditional notion [1, p. 525] by allowing embedded function calls.

The CSI framework uses properties to optimize the inserted instrumentation. In particular, if a CSI-tool branches based on the value of a property, CSI can constant-fold the test and optimize the instrumentation accordingly.

Forensic tables. In addition to hooks for IR objects, the CSI API defines ***forensic*** tables which store static information created by the compiler. A CSI-tool can access these tables at runtime through abstract accessor functions. The forensic tables principally provide two kinds of functionality: (1) associating IR objects with locations in the source code; and (2) relating IR objects to each other, such as relating a basic block to the function that encloses it.

Although the CSI API is elaborate enough to support a variety of dynamic-analysis tools, an expert tool writer might want to employ a nonstandard compiler analysis that is specific to her tool. For tool writers with expertise in developing custom compiler analyses, the CSI framework provides a standard interface for compiler-inserted program instrumentation. CSI's hooks and ID's serve as generic and efficient mechanisms for managing tool initialization and identifying instrumented IR objects. Meanwhile properties and forensic tables provide flexible mechanisms to convey the results of compiler analysis. An expert tool writer can build a ***hybrid tool*** based on the CSI API that incorporates her own custom compiler analysis. By building off of the CSI API, a hybrid tool can employ the features of CSI that simplify the task of writing efficient dynamic-analysis tools without "reinventing the wheel."

Contributions

This paper 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 version 3.9 of LLVM and Clang to insert CSI instrumentation into programs-under-test and uses LTO to optimize tool-instrumented executables.
- 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 [55].
- 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.

Outline

The remainder of this paper is organized as follows. Section 2 presents a brief tutorial on CSI that highlights CSI's key features. Section 3 presents an overview of the CSI API at the time of publication. Section 4 presents our empirical evaluation of the CSI approach. Section 5 overviews the implementation of CSI:LLVM. Section 6 reviews related work. Section 7 offers some concluding remarks.

2. A brief tutorial on CSI

This section illustrates the four key features of CSI by example. Three of the features — flat CSI ID spaces, hooks, and forensic tables — are embodied in the code for CSI-cov, a code-coverage tool.

```c
static long *block_executed = NULL;
static csi_id_t num_basic_blocks = 0;

void report() {
  csi_id_t num_basic_blocks_executed = 0;
  fprintf(stderr, "CSI-cov report:\n");
  for (csi_id_t i = 0; i < num_basic_blocks; i++) {
    if (block_executed[i] > 0)
      num_basic_blocks_executed++;
    const source_loc_t *source_loc = __csi_get_bb_source_loc(i);
    if (NULL != source_loc)
      fprintf(stderr, "%s:%d-%d executed %d times\n",
              source_loc->filename, source_loc->start_line,
              source_loc->end_line, block_executed[i]);
  }
  fprintf(stderr, "Total: %ld of %ld basic blocks executed\n",
          num_basic_blocks_executed, num_basic_blocks);
  free(block_executed);
}

void __csi_init() {
  atexit(report);
}

void __csi_unit_init(const char * const name,
                     const instrumentation_counts_t counts) {
  block_executed = (long *)realloc(block_executed,
                                   (num_basic_blocks + counts.num_bb)
                                   * sizeof(long));
  memset(block_executed + num_basic_blocks, 0, counts.num_bb * sizeof(long));
  num_basic_blocks += counts.num_bb;
}

void __csi_bb_entry(const csi_id_t bb_id, const bb_prop_t prop) {
  block_executed[bb_id]++;
}
```

Fig. 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.

The final key feature — properties — is illustrated by a snippet of a race-detection tool. The CSI API is overviewed in more detail in Section 3.

CSI-cov is 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.[3] In just 36 lines, the CSI framework implements this useful compiler-based tool as a simple C library. Without the CSI framework, a code-coverage tool based compiler instrumentation would require considerably more development effort, as well as an understanding of the internals of the compiler.

[3] This code as written assumes that the program-under-test executes serially. Additional care must be taken for allocating, initializing, and accessing block_executed in order for CSI-cov to handle a multithreaded program-under-test. Thread-safety is an issue that tool writers must address explicitly.

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 basic block's CSI ID. 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` before `main` of the program-under-test is called. This hook, defined in lines 21–23, registers the function `report` to run when the program-under-test terminates. The `report` function, defined in lines 4–19, 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 10. It prints the file name, the range of source lines, and the number of times the basic block has been executed in lines 12–14. Finally, `report` prints the total number of basic blocks executed and the total number of basic blocks in the program in lines 16–17.

The CSI runtime system calls the hook `__csi_unit_init` whenever a new ***translation unit*** — a separate unit of compilation, such as a separate source file — 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 IR objects in each IR-object category that the translation unit contains. CSI-cov defines this hook in lines 25–32 to reallocate the array `block_executed` to incorporate the new basic blocks in lines 27–29, after which it updates the number `num_basic_blocks` of basic blocks in line 31.

The hook `__csi_bb_entry`, defined in lines 34–36, 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 of the key features of CSI: flat and compact CSI ID spaces, hooks, and forensic tables. Because of the flat and compact CSI ID space for basic blocks, 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 one IR-object hook, namely `__csi_bb_entry`. The other IR-object 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.[4]

The one key feature that the CSI-cov example does not illustrate is the use of properties. Every IR-object hook contains a property parameter that encodes bits of static information that can be used to optimize instrumentation. A distinct bit-field struct type is defined for every IR-object category. Since properties are compile-time constants, the compiler can fold and propagate these constants [1, p. 536] to elide unnecessary runtime instrumentation.

As an example, imagine that a tool writer builds a race detector capable of detecting races on shared variables. If she were using conventional compiler instrumentation, she could avoid instrumenting locations that could not possibly be involved in a race, such as a variable declared `const` or a variable on the stack whose address does not escape the frame. In CSI, the property argument to an IR-object hook can give her access to similar specific compile-time information concerning the memory operation being instrumented.

Figure 2 shows a snippet of the code that the writer of a race-detection tool might produce. In defining the hook `__csi_before_load`, the code safely skips the instrumentation of a load if it satisfies certain criteria: it accesses a `const` value (line 39), it accesses a memory location whose address is not captured and is thus guaranteed not to be shared (line 40), or it reads a memory

[4] Although it may seem that the forensic tables are redundant with the existing DWARF tables [19], CSI does not use DWARF for two reasons. First, obtaining the memory location associated with a line of instrumentation turns out to be problematic and is not implemented for most LLVM back ends. Second, it seemed ill-advised from a software-architecture point of view for a communication path between the architecture-independent front end and IR to go through the architecture-dependent back end.

```
37 void __csi_before_load(const csi_id_t load_id, const void *addr,
38                        const int32_t num_bytes, const load_prop_t prop) {
39   if (prop.is_const ||
40       prop.is_not_shared ||
41       prop.is_read_before_write_in_bb)
42     return;
43   check_race_on_load(addr, num_bytes);
44 }
```

Fig. 2. An example memory-load hook for a race detector that uses properties to avoid unnecessary instrumentation.

address that is written to within the same basic block without any intervening function calls (line 41). Because these conditions are known at compile time, the CSI framework can evaluate the condition when it inserts this load hook before each memory load. As a result, the CSI framework will preclude the compiler from inserting runtime calls to `__csi_before_load` before loads that satisfy these conditions.

3. The CSI API

This section overviews the CSI API at the time of publication. This section provides background on the IR features of a mainstream compiler, such as GCC [61] or LLVM [38]. This section describes the six categories of IR objects identified by the CSI API and details each of the four key features of the CSI API: CSI ID spaces, hooks, properties, and forensic tables. This section concludes with a discussion of future anticipated extensions of the API.

Background on compiler intermediate representations

CSI identifies IR objects that correspond to common features found in the intermediate representation (IR) of a mainstream compiler. Mainstream compilers, including GCC and LLVM, represent a program internally using a language called an IR. Although different compilers use different IR's, the IR's used by different mainstream compilers largely resemble an assembly language and share many common features with each other [23, 41]. In the IR of a mainstream compiler, a program is organized as a collection of functions and data. Each program function consists of a set of instructions, each of which performs a single primitive operation, such as basic arithmetic between two operands, a load of a value from memory, a store of a value into memory, a conditional or unconditional branch instruction, or a function call or return. The IR instructions in a function are organized into a ***control-flow graph (CFG)*** $G = (V, E, v_0)$, where

- the set of vertices V represents the function's ***basic blocks***: sequences of instructions where control flow can only enter through the first instruction and leave from the last instruction;
- the set of edges E model control flow between basic blocks; and
- the designated vertex $v_0 \in V$ represents the ***entry point*** of the function.

CSI ID's

CSI identifies several categories of IR objects: function entry points, function exits, basic blocks, memory loads, memory stores, and ***call sites***, the IR instructions that invoke functions. Each IR object is assigned a unique number, called a ***CSI ID***, within its category. The CSI ID's are consecutively numbered from 0 up to 1 less than the number of IR objects in the category. A CSI ID has type `csi_id_t`.

```
typedef int64_t csi_id_t;
// Value representing unknown CSI ID
#define UNKNOWN_CSI_ID ((csi_id_t) -1)

typedef struct {
  csi_id_t num_func;
  csi_id_t num_func_exit;
  csi_id_t num_callsite;
  csi_id_t num_bb;
  csi_id_t num_load;
  csi_id_t num_store;
} instrumentation_counts_t;

// Hooks to be defined by tool writer
void __csi_init();
void __csi_unit_init(const char * const file_name,
                     const instrumentation_counts_t counts);
```

Fig. 3. CSI hooks for initialization.

Initialization hooks

CSI provides two initialization hooks, shown in Figure 3. The global initialization hook `__csi_init` executes exactly once immediately before the program-under-test invokes the `main` function and before it initializes global variables. The unit-initialization hook `__csi_unit_init` is executed once whenever a translation unit is loaded into the TIX, whether statically or dynamically.

Tool writers must code their definition for the `__csi_init` hook with caution. Because the ordering of global constructors is undefined, tool writers cannot assume that static data is initialized. As is consistent with good coding style (see, for example, the section on "Static and Global Variables" in the Google style guide [27]), tool writers should ensure that objects with static storage duration (global variables, static variables, static class member variables, and function static variables) be "plain old data": only `int`'s, `char`'s, `float`'s, pointers, or arrays of plain old data. The tool writer should allocate (and initialize) any global constructable objects used by a shadow computation dynamically with `malloc` in `__csi_init` and then access them via a global static pointer.

The `__csi_unit_init` hook has two parameters. The `file_name` parameter is the name of the source file corresponding to the translation unit. The `counts` parameter is a struct that contains the number of each instrumentation category in the unit. The `counts` parameter allows a tool to prepare its data structures ahead of time (for example, an array with an element for each basic block). The `__csi_unit_init` hook is invoked once for every unit that contributes to the TIX. When multiple units contribute to the TIX, the tool writer must not assume that the invocations of `__csi_unit_init` are called in any particular order, except that if they are statically linked, they all occur after `__csi_init` and before `main`. For a dynamic library compiled with CSI, `__csi_unit_init` is invoked once per translation unit that contributes to the dynamic library at the time the library loads.

IR-object hooks

CSI provides hooks for the each of the six categories of IR objects it identifies. To provide flexibility to tool writers, CSI generally inserts a hook both just before and just after each IR object. This flexibility allows, for example, a memory location's value to be queried before a store, after a store, or both.

```
void __csi_func_entry(const csi_id_t func_id,
                      const func_prop_t prop);
void __csi_func_exit(const csi_id_t func_exit_id,
                     const csi_id_t func_id,
                     const func_exit_prop_t prop);

void __csi_before_call(const csi_id_t callsite_id,
                       const csi_id_t func_id,
                       const call_prop_t prop);
void __csi_after_call(const csi_id_t callsite_id,
                      const csi_id_t func_id,
                      const call_prop_t prop);
```

Fig. 4. CSI hooks for functions.

Figure 4 lists the four API hooks for function entries, function exits, and call sites. The first two hooks instrument functions themselves, and the second two hooks instrument call sites.

Functions are instrumented on entry and exit. The hook `__csi_func_entry` is invoked at the beginning of every instrumented function instance after the function has been entered and initialized but before any user code has run — in LLVM terminology, at the first insertion point of the entry block of the function. The `func_id` parameter identifies the function being entered or exited. Correspondingly, the hook `__csi_func_exit` is invoked just before the function returns (normally).[5] Its parameters include both a function ID `func_id` and a function-exit ID `func_exit_id`, which allows tool writers to distinguish the potentially multiple exit points from a function.

The `__csi_before_call` and `__csi_after_call` hooks instrument call sites. The `callsite_id` parameter identifies the call site, and the `func_id` parameter identifies the ***callee*** — the function being called. It is not possible at compile time to identify with certainty the callee of a call site if the callee is called indirectly via a function pointer or if the callee is not instrumented by CSI. In these cases, the `func_id` argument is set to `UNKNOWN_CSI_ID`, a macro defined by the CSI library, as shown in Figure 3.

Even though it may appear that the call-site hooks are redundant to the function-entry and function-exit hooks, they serve different purposes, since a call site may be instrumented when its callee is not and vice versa. Moreover, a tool writer should be aware that anomalies can occur when intermingling instrumented and uninstrumented code. For example, if an instrumented function `F` calls an uninstrumented function `G`, which then calls another instrumented function `H`, the call-site hook is invoked when `G` is called (with the `func_id` argument taking the value of `UNKNOWN_CSI_ID`), but not when `H` is called. Similarly, the function-entry hook is invoked when entering `H` but not when entering `G`. The tool writer must handle these situations herself if she wishes her tool to support intermingling of instrumented and uninstrumented code.

Although CSI's generic hooks for function entry, function exit, and call site are sufficient for many dynamic-analysis tools, some dynamic-analysis tools, including memory-analysis tools and multithreaded tools, need the extra capability to modify the execution of an external library function, such as `malloc` or `pthread_create` [25]. These external library functions pose two problems. First, because they are defined in an external library, the compiler has no access to their source code, and CSI therefore cannot instrument them. Second, the dynamic-analysis tools in question need to be able to read or modify the parameters or return value of these external-library functions. Because this dynamic information depends on the type of the function, CSI's generic hooks do not provide direct access to this information.

[5] We have not yet defined the API for exceptions.

```
void __csi_bb_entry(const csi_id_t bb_id,
                    const bb_prop_t prop);
void __csi_bb_exit(const csi_id_t bb_id,
                   const bb_prop_t prop);
```

Fig. 5. CSI hooks for basic blocks.

```
void __csi_before_load(const csi_id_t  load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const load_prop_t prop);
void __csi_after_load(const csi_id_t load_id,
                      const void *addr,
                      const int32_t num_bytes,
                      const load_prop_t prop);

void __csi_before_store(const csi_id_t store_id,
                        const void *addr,
                        const int32_t num_bytes,
                        const store_prop_t prop);
void __csi_after_store(const csi_id_t store_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const store_prop_t prop);
```

Fig. 6. CSI hooks for loads and stores.

To accommodate these external library functions, the tool writer can employ ***library interpositioning*** [8, Chapter 7.13] to define wrappers for these routines. By interposing an external library function, each call to that function is replaced with a call to its wrapper with the same arguments as the original call. A tool writer can call the original function from the wrapper and perform any other computation she deems fit. The tool writer must take care to ensure that the wrapper preserves the semantics of the original external library function.

Figure 5 shows the two CSI hooks for basic blocks. The hook `__csi_bb_entry` is called when control enters a basic block, and `__csi_bb_exit` is called just before control leaves the basic block. The `bb_id` parameter identifies the basic block being entered or exited.

Figure 6 shows the four CSI hooks for memory operations. The hooks `__csi_before_load` and `__csi_after_load` are called before and after memory loads, respectively, and likewise, `__csi_before_store` and `__csi_after_store` are called before and after memory stores. The argument `addr` is the location in memory, and `num_bytes` is the number of bytes loaded or stored.

Properties

CSI exports static information — the results of compiler analysis and other information known at compile time — to CSI-tools through ***properties***. Properties are compile-time constants, which allows the compiler to constant-fold and propagate them [1, p. 536] to elide unnecessary runtime instrumentation.

Every IR-object hook contains a ***property*** parameter `prop`: a bit-field struct that encodes static information for optimizing instrumentation. CSI defines a distinct bit-field struct type for every IR-object category: `func_prop_t`, `func_exit_prop_t`, `call_prop_t`, `bb_prop_t`, `load_prop_t`, `store_prop_t`. As an example, consider the `load_prop_t` type for loads, which is shown in Figure 7. The property encodes the alignment of the load (`lg_alignment`) — an integer k indicating that the

```
typedef struct {
  unsigned char lg_alignment            :  6;
  unsigned char is_volatile             :  1;
  unsigned char is_not_shared           :  1;
  unsigned char is_on_stack             :  1;
  unsigned char is_const                :  1;
  unsigned char is_vtable_access        :  1;
  unsigned char is_read_before_write_in_bb :  1;
  uint64_t _unused                      : 52;
} load_prop_t;
```

Fig. 7. Definition of the CSI property for loads.

address must have k trailing 0's — whether the load is volatile (`is_volatile`), whether the loaded location is guaranteed not to be shared (`is_not_shared`), etc.

Although properties typically export the results of standard compiler analyses, they can be extended to export results of non-standard compiler analyses. A tool writer who develops a custom compiler analysis for a hybrid tool can extend CSI properties to convey the results of her custom analysis.

Forensic tables

In addition to properties, CSI passes information known at compile time through its ***forensic tables***. The forensic tables map CSI ID's to static information associated with the instrumented IR objects. CSI encapsulates the tables with a set of accessor functions, which allow the tool writer to map IR objects to their source-code locations, such as line numbers and containing file name, and to relate IR objects to each other, such as which basic block contains a given instrumented load and which loads are contained in a basic block.

Status of the API

At the time of publication, CSI identifies the six categories of IR objects described in this section. We have observed that this API is sufficient to build a variety of dynamic-analysis tools. Section 4 describes the six demonstration CSI tools that we have built already. In addition to these demonstration tools, we have observed that the CSI API suffices to implement other tools we have found in the literature, including the Picon tool for analyzing control-flow integrity [13] and a machine-architecture-independent version of the Loca cache-simulator tool [16].

Nevertheless, we expect that future versions of CSI will identify and instrument additional categories of IR objects. For example, the CSI API does not currently instrument atomic operations or stack allocations for local variables. Such instrumentation would allow more elaborate race detectors and memory checkers to be implemented as CSI-tools. Furthermore, mainstream compiler IR's have historically grown over time to incorporate additional features. As support for new features proliferates across mainstream compiler IR's, we anticipate that the CSI API may grow to provide instrumentation facilities for these new features.

4. Evaluation

This section studies the efficacy of the CSI approach and the efficiency of CSI-tools in practice. We implemented the CSI:LLVM compiler using version 3.9 of the LLVM compiler and its link-time-optimization (LTO) capability. We implemented six demonstration CSI-tools, which perform a variety of dynamic analyses. We measured the performance of these tools on two real-world

Tool	*Description*	*Func. entry*	*Func. exit*	*Basic block*	*Call*	*Load*	*Store*	*CSI ID's*	*Prop.'s*	*Forensic tables*
CSI-null	The "null tool", which contains only empty hooks	–	–	–	–	–	–	–	–	–
CSI-cov	Code-coverage analyzer	–	–	✓	–	–	–	✓	–	✓
CSI-memop	Memory-operations counter	–	–	–	–	✓	✓	–	–	–
CSI-prof	Per-function performance profiler	✓	✓	–	–	–	–	✓	–	✓
CSI-cgg	Dynamic call-graph generator	✓	✓	–	✓	–	–	✓	✓	✓
CSI-TSan	Port of Google's ThreadSanitizer [56] race-detection tool	✓	✓	–	–	✓	✓	–	✓	–

Fig. 8. Description of the six demonstration CSI-tools. Each row corresponds to a different CSI-tool. Column *Tool* gives the name of the CSI-tool, and column *Description* gives a brief description of that CSI-tool. The remaining columns, *Func. entry* through *Forensic tables*, correspond to different features of the CSI API that a tool can use. The columns *Func. entry*, *Func. exit*, *Basic block*, *Call*, *Load*, and *Store* identify whether a tool implements IR-object hooks of the corresponding category. Column *CSI ID's* identifies whether a tool uses CSI ID's. Column *Prop.'s* identifies whether a tool uses a CSI property. Column *Forensic tables* identifies whether a tool uses a forensic table. A checkmark indicates that the CSI tool uses a particular IR-object hook or API feature in a nontrivial manner for collecting or reporting its analysis.

application benchmarks. This section presents an overview of these demonstration CSI-tools and their empirical performance using CSI:LLVM. Section 5 describes the design of CSI:LLVM.

In summary, we found that many CSI-tools exhibit low overheads, slowing down the execution of the program-under-test by less than 70%. We identified two general sources of the good performance of these CSI-tools: the simple tool designs enabled by CSI's flat ID spaces, and the ability of CSI:LLVM to optimize the program's instrumentation with standard compiler optimizations. We also compared the CSI approach to that of developing custom compiler instrumentation through a case study with Google's ThreadSanitizer tool [56]. We found that CSI-TSan achieves most of the functionality with only about 20% more overhead than ThreadSanitizer, even though ThreadSanitizer uses custom compiler instrumentation to perform tool-specific analyses and optimizations.

Figure 8 describes the six demonstration CSI-tools and documents the IR-object hooks and CSI API features that each tool uses. As Figure 8 shows, these CSI-tools perform a variety of dynamic analyses, including code-coverage analysis, performance profiling, call-graph generation, and race detection. Each tool performs its analysis using a subset of CSI's IR-object hooks and other key features. Together, these tools make use of all four of the key features of the CSI API, including all categories of IR-object hooks. The current version of the CSI API thus already supports a variety of dynamic-analysis tools, and we anticipate that this variety of supported dynamic-analysis tools will grow as future versions of the API become more comprehensive.

Experimental setup

We evaluated the performance overhead of the demonstration CSI tools by instrumenting two real-world applications: the Apache HTTP server (version 2.4.23) and the bzip2 data compressor.

The SLOCcount tool [73] counts 298,044 source lines of C code for Apache and 5,820 lines of C for bzip2. The Apache benchmark employs dynamic linking and multithreading and is largely I/O bound. We used the Apache benchmark harness from the ThreadSanitizer repository [2] to issue 300,000 connections with a concurrency level of 20 (meaning that up to 20 simultaneous requests may be issued). The benchmark harness then reports a total runtime and a mean measurement of the number of requests handled per second. The bzip2 benchmark uses static linking and is single threaded. We verified that bzip2 is primarily compute bound. The bzip2 benchmark compresses a 38MB tar archive of the Apache source tree at the highest/slowest compression setting.

To handle the Apache benchmark, all of the demonstration CSI-tools were made thread-safe. The CSI-TSan tool uses the ThreadSanitizer runtime [56] to achieve thread-safety. The remaining tools achieve thread-safety by recording and reporting all of their results on a per-thread basis. In particular, these tools use runtime library interpositioning [8, Chapter 7.13] of the `pthread_create` routine [25] and thread-local storage to record per-thread results. The thread-local storage is initialized using CSI's function-entry hook. Although CSI-tools can in principle perform more complex aggregation of thread-local results, we adopted this simple approach to making the demonstration CSI-tools thread-safe in order to measure the performance overhead of the CSI framework, as distinct from the overheads due to interthread communication and synchronization.

We compiled all benchmarks and tools using our CSI:LLVM compiler, which is based on version 3.9 of LLVM and Clang. We used the gold linker [65] from GNU Binutils [22] and LLVM's link-time optimizer [42] for all static linking of instrumented and uninstrumented programs. Each CSI-tool was compiled with `-O3` optimization. Each benchmark was compiled with and without instrumentation. To compile the uninstrumented benchmark, we only changed the benchmark's default compilation process to use our CSI:LLVM compiler and LTO, leaving all other compilation and linking parameters unchanged. To compile an instrumented version, each benchmark was compiled with CSI enabled and then statically linked with a CSI-tool.

Although compiling and linking a program with LTO often improves the running time of that program, we found that LTO had a minimal effect on the running time of these two benchmarks. The uninstrumented version of bzip2, compiled with LTO, ran 2% slower than compiling bzip2 in the ordinary fashion without LTO. Meanwhile, the uninstrumented version of Apache, compiled with LTO, ran 1% faster than Apache compiled without LTO.

For each instrumented and uninstrumented benchmark executable, we ran the executable 5 times and recorded the minimum running time.[6] All experiments were run on an Amazon AWS `c4.8xlarge` spot instance, which is a dual-socket Intel Xeon E5-2666 v3 system with a total of 60 GiB of memory. Each Xeon is a 2.9 GHz 18-core CPU with a shared 25 MiB L3-cache. Each core has a 32 KiB private L1-data-cache and a 256 KiB private L2-cache. The system was "quiesced" to permit careful measurements by turning off Turbo Boost, dvfs, hyperthreading, extraneous interrupts, etc.

Performance of the demonstration CSI-tools

Figure 9 presents the empirical performance overhead of the demonstration CSI-tools on the benchmark applications. As the figure shows, with the exception of CSI-TSan, all of the demonstration CSI-tools exhibit less than 1.7× slowdown when instrumenting the benchmark programs. Furthermore, half of the measured slowdowns incurred by CSI-tools are less than 1.2×.

CSI-TSan exhibits relatively high overheads compared to the other demonstration CSI-tools. This overhead is due in large part to the overhead of the ThreadSanitizer runtime, rather than the CSI framework itself. Figure 10 presents the performance of ThreadSanitizer and of CSI-TSan on the

[6] Because the benchmarks are deterministic, the minimum running time minimizes the effect of external system noise.

Application	CSI-null	CSI-cov	CSI-memop	CSI-prof	CSI-cgg	CSI-TSan
Apache	1.01	1.03	1.01	0.99	1.22	3.28
bzip2	1.00	1.65	1.21	1.56	1.17	23.93

Fig. 9. Performance overhead for six demonstration CSI-tools, described in Figure 8, on the bzip2 and Apache benchmark applications. Each row corresponds to a benchmark application, and each column corresponds to a CSI-tool. Each value is the ratio of the running time of the application instrumented with the specified tool divided by the running time of the application with no instrumentation. Each running time was taken as the minimum of 5 runs on the test machine, a "quiesced" Amazon AWS `c4.8xlarge` spot instance.

	Google's	CSI-TSan	CSI-TSan
Application	ThreadSanitizer	no properties	with properties
Apache	2.75	3.96	3.28
bzip2	20.68	30.97	23.93

Fig. 10. Comparison of performance overheads for Google's ThreadSanitizer [56], CSI-TSan without properties, and CSI-TSan with properties.

application benchmarks. As Figure 10 shows, CSI-TSan slows down the execution of the benchmark applications by less than 1.2 times the slow down of Google's ThreadSanitizer tool [56]. We shall compare CSI-TSan to Google's ThreadSanitizer tool more closely at the end of this section.

We identified four reasons why CSI-tools exhibit good performance overheads. One reason has to do with the simple data structures enabled by CSI flat ID spaces. The other three have to do with the ability of CSI:LLVM to apply standard compiler optimizations to optimize a CSI-tool's instrumentation when producing the TIX. We discuss each of these reasons in turn and explore their effects through case studies.

Performance impact of CSI ID's

CSI's flat and compact CSI ID spaces improve the efficiency of CSI-tools. As Figure 8 shows, three of the demonstration CSI-tools use CSI ID's in nontrivial ways. The CSI-cov tool uses CSI ID's to index a flat array of basic-block data, and both CSI-prof and CSI-cgg use CSI ID's to index flat arrays of function data. Not only do CSI ID's simplify the codebases of these three tools, but they also confer performance benefits.

We explored the performance benefits of CSI ID's through a case study involving CSI-cov. Without the ability to use a flat array indexed by a CSI ID, a natural alternative is to use a hash table, where each function or basic block is mapped to data by hashing the memory address of its entry point. We compared the performance of CSI-cov to this alternative design on the application benchmarks. For the hashing approach, we sized the hash table optimally for each benchmark and used the fast multiplicative hash function from [12, Sec. 11.3.2]. The hashing approach consistently doubled the runtime overhead of the CSI-cov tool, slowing down the runtime of the Apache benchmark from 1.03× with CSI to 1.09× with hashing, and slowing down the runtime of bzip2 from 1.65× with CSI to 3.37× with hashing. The hashing approach also lacks a mechanism for identifying which basic blocks were never executed, which CSI-cov finds trivially.

Zero-cost null hooks

The performance overhead of the CSI-null tool, presented in Figure 9, demonstrates that null hooks in a CSI-tool have essentially zero cost. (The 1% overhead recorded for the Apache benchmark falls within measurement noise.) The CSI-null tool consists entirely of ***null hooks***: each hook simply returns without examining its arguments. CSI:LLVM is robust enough to recognize null hooks as dead code and elide them all. CSI:LLVM thus produces a TIX that is as efficient as if the hooks were never inserted in the first place.

This capability of CSI:LLVM applies even when CSI instruments dynamically linked libraries. For the Apache benchmark, compiling the benchmark produces several libraries that are dynamically linked with the Apache executable. When the Apache benchmark is instrumented with a CSI-tool, all of these libraries are instrumented as well. Nevertheless, CSI:LLVM's optimization capabilities are robust enough to elide null hooks from the dynamic libraries.

Because null hooks incur zero cost in CSI-tools, the CSI-null tool plays an important role in the implementation of the CSI framework, as Section 5 describes.

CSI:LLVM's optimization of TIX's

The low overheads of the CSI-tools stems from the ability of CSI:LLVM to perform standard compiler optimizations on a TIX. For many tools, CSI:LLVM is able to inline [1, p. 903] the implementation of a hook at each point the hook is called. After inlining, CSI:LLVM can then perform the full suite of standard compiler optimizations on the instrumentation.

CSI-memop illustrates the power of CSI:LLVM to optimize instrumentation. The CSI-memop tool counts the loads and stores incurred during the execution of a program-under-test by incrementing a counter whenever a load or store occurs. Although the CSI-memop tool appears to perform an operation at each load or store instruction, CSI:LLVM often optimizes much of the tool's instrumentation. For example, if there are multiple loads or stores within the same basic block, CSI:LLVM replaces the tool's many counter-increment operations within the basic block with a single addition to the counter. Furthermore, if loads and stores occur within a loop and CSI:LLVM can infer the loop bounds, then CSI:LLVM hoists the CSI-memop operations outside of the loop using loop-invariant code motion. As a result, CSI-memop exhibits very low overheads, as Figure 9 shows.

Constant-folding of properties

CSI:LLVM is further able to optimize CSI tools by constant-folding [1, p. 536] properties into an inlined CSI instrumentation hook. We studied the performance benefits of properties by comparing two different versions of the CSI-TSan tool. The first version does not use properties and simply instruments every load and store in the program-under-test as an unaligned access to memory. The second version uses properties to refine the instrumentation to ignore loads and stores that cannot participate in a race, to differentiate memory accesses by their alignment, and to differentiate accesses to virtual table from other memory accesses.

Figure 10 presents the performance of these two versions of the CSI-TSan tool, compared to the performance of the Google's original ThreadSanitizer tool [56], which uses custom compiler instrumentation. As Figure 10 shows, the performance of CSI-TSan benefits substantially from the use of properties. In particular, the implementation of CSI-TSan that does not use properties incurs 1.4–1.5 times the slow down of Google's ThreadSanitizer tool on the application benchmarks. Meanwhile, the implementation of CSI-TSan that uses properties incurs less than 1.2 times the slow down of ThreadSanitizer. This performance improvement comes from the ability of CSI:LLVM to optimize away the checks and computation involving properties in CSI-TSan, leaving minimal, tailored instrumentation code around loads and stores.

Comparison of CSI-TSan with Google's ThreadSanitizer

We compared the CSI approach to that of implementing custom compiler instrumentation through a case study involving Google's ThreadSanitizer tool [56]. ThreadSanitizer performs race detection by instrumenting every load and store and intercepting calls to `pthread` functions, such as accesses to condition variables [25]. ThreadSanitizer also supports features beyond its core data-race-detection capability. For example, ThreadSanitizer allows the tool user to manually suppress race reports related to parts of the program-under-test, either by annotating individual functions or blacklisting entire files [11]. Furthermore, compiling a program to use ThreadSanitizer will disable some of LLVM's optimizations that can introduce apparent races into a program.

The implementation of ThreadSanitizer in LLVM employs many of the capabilities of custom compiler instrumentation. The ThreadSanitizer tool has two primary components: a compiler pass and a runtime library. The compiler pass inserts calls to the ThreadSanitizer runtime library at every memory access and at the entry and exit points of functions. The compiler pass itself uses both standard and tool-specific compiler analyses to optimize the instrumentation. To allow tool users to suppress race reports and to disable LLVM optimizations when ThreadSanitizer is being used, the compiler pass also adds tool-specific annotations throughout the IR that other LLVM optimizations have been modified to respect. The compiler modifications for ThreadSanitizer include a 500-line compiler pass plus additional changes scattered throughout LLVM's 2.5-million-line codebase. To produce race reports that relate races to the source code of the program-under-test, the ThreadSanitizer runtime library manages debugging information in a custom fashion. Specifically, the runtime library selectively ignores debugging information associated with ThreadSanitizer's compiler instrumentation.

In contrast to Google's ThreadSanitizer custom compiler modifications, CSI-TSan only uses the CSI framework to instrument a program with calls into the ThreadSanitizer runtime library. CSI-TSan is a relatively simple 100-line C library that implements CSI hooks for function entries, function exits, loads, and stores. These hooks implement calls to the ThreadSanitizer runtime. For the load and store hooks, these calls are predicated on certain property values. To satisfy the API of the ThreadSanitizer runtime library, CSI-TSan passes the addresses of functions and load and store instructions to the ThreadSanitizer runtime-library functions it calls. (We have not modified the ThreadSanitizer runtime library to use CSI ID's or forensic tables for debugging information.) CSI-TSan makes no use of tool-specific compiler analyses to optimize the instrumentation.

Despite its relatively modest implementation, CSI-TSan achieves much of the functionality of Google's ThreadSanitizer. CSI-TSan passes 90% of the 280 ThreadSanitizer regression tests. There appear to be four general reasons why CSI-TSan fails the remaining 10% of the tests.

- CSI-TSan misses races involving atomic operations that ThreadSanitizer can detect, because the CSI API does not currently instrument atomic operations.
- CSI-TSan reports races that are precluded by a happens-before relation [36], which ThreadSanitizer checks for using ThreadSanitizer-specific compiler analysis.
- CSI-Tsan does not implement ThreadSanitizer's mechanisms for manually suppressing race reports.
- The ThreadSanitizer runtime library has not been modified to ignore debugging symbols associated with CSI's instrumentation hooks.

We believe that rectifying these shortcomings would not affect the performance of CSI-TSan unduly. The approximately 20% overhead that CSI-Tsan pays compared to ThreadSanitizer seems like a reasonable price for allowing such a tool to operate as a library with no custom modifications to the compiler.

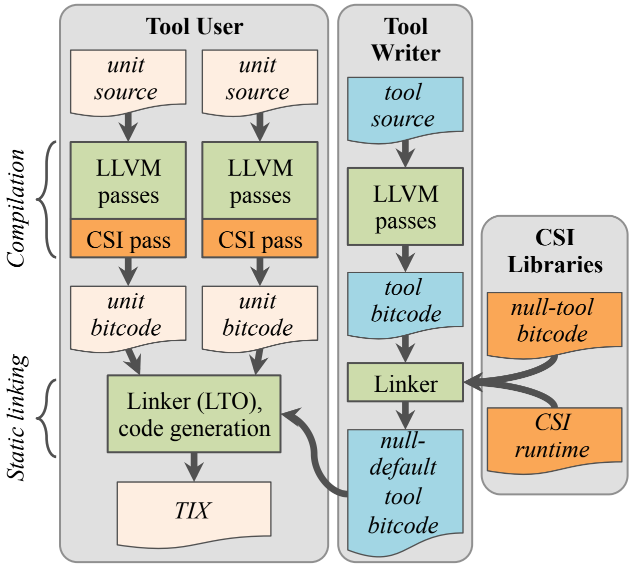

Fig. 11. The build architecture of CSI:LLVM. Rounded rectangles distinguish the concerns of the tool user, the tool writer, and the CSI-provided libraries. The boxes indicate compiler and linker actions. The “document” shapes represent code in some form.

5. CSI:LLVM

This section describes CSI:LLVM, our implementation of CSI in LLVM version 3.9 using LTO. We review how LLVM compiles a program with LTO, and we describe how CSI:LLVM employs LTO to overcome apparent overheads of the CSI’s brute-force approach to instrumentation. We describe how CSI-tools are compiled and linked and how CSI:LLVM implements CSI’s features, including the flat ID spaces for CSI ID’s. We describe an alternative design of CSI that uses ordinary compile-time optimization instead of LTO.

Build architecture

To understand how CSI:LLVM works, let us first review how LLVM compiles a program with LTO. During the ***compilation stage*** of a program, LLVM separately compiles each translation unit into LLVM IR. The compilation stage supports many platform-independent intra-unit (within a single unit) analyses and transformations over the LLVM IR. When LTO is enabled, at the end of the compilation stage, LLVM produces a ***bitcode*** file for the compiled unit — a compact on-disk binary representation interchangeable with LLVM IR — rather than a native object file. The bitcode files carry all the information and metadata produced by the compiler analyses during the compilation stage, which allows further analysis and transformation during the subsequent ***static-linking stage***, when LTO is invoked.

CSI:LLVM is implemented within the LLVM compiler as two components: a ***CSI compile pass*** and a ***CSI runtime***. Broadly, the compile pass inserts calls to the hooks, and the runtime aggregates the data necessary to establish CSI ID’s and construct the forensic tables. Figure 11 portrays the entire process of translating program and tool source into a TIX.

As Figure 11 shows, the CSI compile pass is inserted as an additional compiler pass at the end of the compilation stage, immediately before the generation of the bitcode file. Thus, instrumentation is inserted after all intra-unit compiler optimizations, meaning CSI instruments the *optimized program code*. Although this design allows CSI to instrument optimized programs, the ramifications are salient. For instance, if a function call is inlined at a particular call site, the `__csi_func_entry` and `__csi_func_exit` hooks will not be instrumented for the function instance invoked at that particular call site. Thus, CSI:LLVM should be viewed as instrumenting the compiled code, not the source code.

The CSI compile pass is implemented as an LLVM module pass [43]. For each function body, it inserts calls to the appropriate hooks at the designated points within the IR. For properties, the compiler performs any analysis necessary to provide the property argument for the appropriate hooks in the generated IR. As Section 4 describes, because property values are compile-time constants, LTO can eliminate conditional tests involving properties using its optimization pass to perform constant-folding and propagation. In addition, the CSI compile pass inserts a call to the CSI runtime's unit initialization function, `__csirt_unit_init`, by prepending the function to the unit's global constructor list, thereby ensuring that unit initialization occurs before execution of the TIX's `main` function or any other global constructors.

From the point of view of the tool writer, CSI:LLVM works as follows. The tool writer defines relevant hooks for her CSI-tool, and then she statically links her tool with the null tool. The hooks defined in the null tool are all exported as ***weak symbols*** [8, p. 680],[7] while the implemented hooks in her CSI-tool are exported as strong symbols. Linking these two tools thus produces a ***null-default CSI-tool***, in which the strong symbols defined in the tool writer's CSI-tool override the corresponding symbols in the null tool, but for hooks not defined in the tool writer's CSI-tool, the null-hook definition defaults. When a tool user statically links a null-default CSI-tool with his program-under-test, LTO automatically elides calls to null hooks during its optimization pass.

Flat ID spaces

To enable flat ID spaces, CSI must keep track of the total counts of IR objects in the program-under-test. A compile-time-only strategy does not suffice, because each unit involved in the program-under-test can be compiled separately. Furthermore, these counts cannot be determined at static-link time, because some units of the program might be dynamically linked. Thus, CSI implements its flat ID spaces using a compile pass together with a lightweight runtime library.

Conceptually, CSI assigns an interval of the ID space to each unit, but the starting point of the interval is unknown until runtime. For each unit, the CSI compile pass statically assigns unit-local ID's to IR objects and inserts a static global variable for each category that keeps track of the "base" of the ID's (i.e., the start of the interval) for that category. The CSI compile pass also provides the counts of IR objects of each category found in the unit as an argument to `__csirt_unit_init`. The runtime library aggregates these counts across units in the program-under-test. As each unit loads during execution, the runtime initializes the base variables in the unit using the aggregated counts when the unit loads, before updating its aggregated counts. The CSI ID for an IR object is thus obtained by summing the base with its unit-local ID.

For example, to assign ID's to basic blocks, the CSI compile pass statically numbers the m basic blocks in a unit with local ID values $0, 1, \ldots, m-1$, and it inserts a static global variable `bb_base` into the unit. When the unit is initialized, the CSI runtime sets `bb_base` to the number n of basic blocks in units that have already been initialized. The ID of a basic block in the unit is the sum of `bb_base` plus that basic block's local ID.

[7] Specifically, a `weak_odr` symbol in LLVM terminology [41].

Function ID's for called functions

Call-site hooks complicate the implementation of CSI ID's for functions. A call-site hook takes as a parameter the CSI ID of the callee, or `UNKNOWN_CSI_ID` if the callee cannot be determined. The standard scheme for maintaining CSI ID's does not ensure that the ID of a called function can be passed to a call-site hook. In particular, if unit A contains a call to a function in unit B and A is initialized before B, then the ID of the callee is not known when A is initialized.

To resolve this issue, the CSI compile pass generates a global weak symbol for each function defined or called in the unit. These symbols are initialized to `UNKNOWN_CSI_ID`. When the unit is initialized, `__csirt_unit_init` sets the value of the symbol for each function defined in the unit to that function's ID. Each call-site hook (for a direct call) is passed the symbol of its callee. If the callee is instrumented, then its symbol is guaranteed to be initialized when the unit defining the callee is loaded, which must occur before this call instruction calls the callee. Although multiple units might define a symbol for the same function, because these symbols are weak, only one symbol will survive for each function.

Forensic tables

To aggregate data for the forensic tables, CSI implements a strategy similar to its strategy for maintaining implementing falt ID spaces. The CSI pass generates unit-local tables for the unit being compiled. The pointers to the unit-local tables are passed as arguments to `__csirt_unit_init`. Then `__csirt_unit_init` aggregates them into a global table indexed by CSI ID's at runtime.

CSI using compile-time optimization (CTO)

An alternative design of CSI:LLVM could use ordinary compile-time optimization (CTO) instead of LTO. Although we have not implemented a complete system for CSI using CTO, we have tested components of its design and verified their ability to optimize CSI instrumentation as effectively as LTO.

A CTO-based implementation of CSI would differ from the LTO-based design of CSI:LLVM as follows. Rather than pass the null-default CSI-tool as an input to the static linker, the CSI-tool would be passed as input to each invocation of the CSI compile pass during the compilation stage. The CSI compile pass would then insert the symbols defined in the CSI-tool into the compiled unit's bitcode, in addition to inserting calls to those hooks. During this process, the CSI compile pass could avoid inserting calls to any null hooks in the CSI-tool. Further compiler optimization of the inserted instrumentation, including constant-folding of properties, could be performed in the compilation stage after the CSI compile pass finishes. As a result, a CTO-based design could perform all of CSI:LLVM's optimizations on inserted instrumentation.

One technical issue with the CTO-based approach is that the separate unit bitcodes produced by the compilation stage would each contain their own definitions of the CSI-tool's symbols. Although these definitions would be identical, the static-linking stage would observe multiple definitions of the same symbol, which could cause it to produce an error. To avoid this issue, the CSI compile pass would mark each of the CSI-tool's symbols as `weak_odr` [41] to indicate to the static-linking stage that all definitions of any particular CSI-tool symbol are identical. This design would therefore require the tool user to ensure that, when building the TIX, every invocation of the CSI pass uses the same CSI-tool and that the static-linking stage only links together units compiled with the same CSI-tool.

We chose to implement CSI:LLVM using LTO due to the simplicity of the LTO-based design. In our own tests, we have not encountered problems using LTO, and we have found that LTO is robust in its ability to optimize CSI instrumentation. Other research has discovered problems with compiling some applications using LTO [34]. Although improved implementations of LTO are being developed [34], this CTO-based design for CSI removes any reliance CSI might have on LTO to overcome overheads. Implementing a complete CSI system based on CTO is a topic of future work.

6. Related work

This section surveys related work on program instrumentation and support for compiler modifications. Much of the related work on program instrumentation focuses on binary instrumentation and bytecode instrumentation. CSI, meanwhile, addresses a problem with compiler instrumentation, an alternative to both of these. The related work on compiler modifications, meanwhile, focuses on providing support for modifying a particular open-source compiler. In contrast, from the point of view of the tool writer and tool user, the CSI API is largely compiler-agnostic, presenting hooks for IR objects and static-analysis features that are common across mainstream compiler IR's.

A popular alternative to compiler instrumentation is ***binary instrumentation*** (e.g., [6, 15, 37, 44, 48, 51, 58, 59]), which works by directly modifying the binary executable of the program-under-test to incorporate instrumentation code. To handle the task of modifying a binary, tool writers generally employ a binary-instrumentation framework, such as Pin [44], DynamoRIO [6], Valgrind [48], or DynInst [4, 9].

Binary instrumentation has several advantages and disadvantages compared to compiler instrumentation. Binary instrumentation targets a lower-level representation of the program-under-test than the compiler's IR, which can be more appropriate for some tools and less appropriate for others. For example, tools based on binary instrumentation observe all memory loads and stores, including those due to register spills, but have trouble distinguishing loads and stores due to register spills from those to local variables on the stack [54]. Furthermore, whereas compiler instrumentation requires the program-under-test to be recompiled, binary instrumentation avoids this requirement by modifying the binary directly. Binary instrumentation can thus be applied to third-party libraries, for which the tool user might only have the executable binary, whereas compiler instrumentation cannot be applied. On the other hand, inserting instrumentation into a binary often requires significant bookkeeping, for example, to relocate portions of the program-under-test and to save and restore register state around instrumentation code. As a result, binary-instrumentation-based analysis tools can slow down a program significantly [4, 68]. Furthermore, binary-instrumentation frameworks vary in their ability to analyze and optimize program instrumentation. As a result, some frameworks, such as Pin [31], rely on the tool writer to implement basic optimizations on program instrumentation by hand. In contrast, the default optimization capabilities in a mainstream compiler can readily perform these basic optimizations on program instrumentation as well as more complicated optimizations, as discussed in Section 4. Tools based on compiler instrumentation thus tend to exhibit lower performance overheads compared to their binary-instrumentation-based counterparts [54, 56]. The version of Google's ThreadSanitizer that is based on compiler instrumentation outperforms the version based on Valgrind's binary instrumentation by 1.7×–2.9× on big tests [56].

Because binary instrumentation and compiler instrumentation each come with their own advantages and disadvantages, hybrid approaches have been proposed [54]. In particular, a hybrid approach would use compiler instrumentation to instrument parts of a program-under-test that can be recompiled and then use binary instrumentation for the remaining parts of the program, including third-party libraries. A hybrid approach could thereby enjoy the benefits of both compiler instrumentation and binary instrumentation. Because CSI provides a framework for tools based on compiler instrumentation, a tool writer could similarly explore a hybrid approach to writing her tool that combines CSI with binary instrumentation. Such a combination would allow the tool writer to simplify the implementation of the compiler-instrumentation portion of the hybrid approach and to take advantage of CSI's key features without reimplementing them herself.

Many frameworks have been developed for bytecode instrumentation [5, 7, 14, 39, 45, 49, 69, 74]. These frameworks are tied to the Java Virtual Machine (JVM), even though the JVM itself is designed to be platform independent.

Meanwhile, dynamic-analysis tools that use compiler instrumentation, such as [20, 26, 28, 33, 40, 47, 53–55, 75], are generally implemented by making tool-specific modifications to the compiler. Few general compiler-instrumentation frameworks extend beyond tool-specific instrumentation. Two exceptions are SASSI [63], a low-level assembly-language instrumentation tool for GPU's, and TAU [57], which focuses on instrumenting high-level C++ language features. Both exhibit a different focus from CSI, requiring users to insert source commands to instruct the compiler what to instrument. In contrast, the CSI pass inserts instrumentation hooks throughout the compiler's IR by default.

Some work has been done to simplify the process of augmenting the functionality of a mainstream compiler. LLVM, Clang, and GCC all support a mechanism for loading modules, or plugins, into the compiler to augment its functionality [24, 43, 66]. MELT [62] builds upon GCC's plugin facility to provide a domain-specific language to simplify the development of GCC extensions. Tool writers can therefore use these plugin mechanisms to distribute custom compiler modifications. But the API for a plugin is closely tied to internals of the target compiler. As a result, to write a plugin, tool writers still require mastery of the internals of a particular compiler, and tool writers must maintain their compiler modifications through changes to the compiler codebase. Similarly, implementing an extension to GCC using MELT still requires familiarity with GCC's internal structures. In contrast, CSI allows tool writers to implement dynamic-analysis tools that use compiler instrumentation without familiarity with any particular compiler's codebase.

7. Conclusion

CSI supports the rapid development of high-performing compiler-based dynamic-analysis software tools. We have implemented CSI in the LLVM compiler and demonstrated its efficacy through six example tools. We are optimistic that CSI — or at least the concepts of CSI — will be adopted by compiler communities to enable the development of a wide range of tools. We hope that tool writers will give us feedback on how best to enrich CSI with additional features while keeping the API simple and minimal. We have made the source code for CSI:LLVM publicly available at https://github.com/csi-llvm under a liberal open-source license, so that the LLVM community can share in the evolution of the instrumentation framework.

We conclude by discussing some limitations of the current version of CSI.

The decision to instrument the IR and not the front or back end might lead to a misunderstanding of results gathered by a CSI tool. For example, loads and stores created during machine code generation (which occurs after IR generation) are not instrumented. This scenario occurs during register allocation whenever a register must be spilled to the stack using load and store machine instructions. These memory operations cannot be instrumented in the current implementation of CSI:LLVM.

CSI effectively instruments optimized program code, which is not necessarily appropriate for all tools. For example, if a compiler optimization replaces a conditional branch with an unconditional branchless implementation, then a code-coverage tool that records the execution of basic blocks cannot tell whether a program execution covered both branches of the condition. With CSI:LLVM's LTO-based design, tool-users can work around this limitation by reducing the optimization level used during the compilation stage. As long as compiler optimizations are still applied during the static-linking stage via LTO, CSI:LLVM can still optimize program instrumentation within a

TIX. More flexible solutions to this limitation, including solutions that work with a CTO-based implementation of CSI, are a topic of future work.

The existing CSI framework is limited to instrumenting a program with just one CSI-tool at a time. In our experience we have not found this limitation to be problematic because tool users typically run just one dynamic-analysis tool at a time. One reason that tool users avoid running multiple tools simultaneously is to avoid compounding the performance overheads associated with each tool, which can lead to unacceptable runtime slowdowns or significant perturbations of the analysis results. To achieve the effect today of running multiple CSI-tools simultaneously on the same program, a tool writer can write a single tool containing the instrumentations for all component tools.

Finally, the API that CSI:LLVM implements is missing some features. The API currently lacks instrumentation hooks for some IR objects, such as atomic operations and exception-handling code. The API also provides limited support for ensuring the thread-safety of a tool. Currently, tool writers must implement thread-safety themselves, using thread-local storage and function interpositioning. We plan to add support for these features in the future, as the CSI API grows more comprehensive.

ACKNOWLEDGMENTS

Many thanks to Kostya Serebryany of Google for feedback on this work, as well as for sponsoring our Google Faculty Research Grant. Thanks to James Thomas and Miriam Gershenson of MIT for their formative contributions to the CSI API and software. Thanks to the LLVM developer community for discussions regarding CSI and a CTO-based design for CSI. Thanks to Masakazu Bando of MIT for discussions on what tools can and cannot be written in the current version of the CSI API. We gratefully acknowledge grants from Amazon Web Services and Google Cloud Platform for cloud-computing resources. We thank Aaron Gember-Jacobson and the reviewers for their feedback.

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.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (second ed.).

[2] Apache Software Foundation. 2016. ab — Apache HTTP server benchmarking tool. Available at https://httpd.apache.org/docs/2.4/programs/ab.html. (2016).

[3] 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.

[4] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *PASTE*. 9–16.

[5] Walter Binder, Alex Villazón, Danilo Ansaloni, and Philippe Moret. 2009. @J: Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine. In *VMIL*. Article 4, 9 pages.

[6] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. 2001. Design and Implementation of a Dynamic Optimization Framework for Windows. In *FDDO-4*.

[7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*.

[8] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson, USA.

[9] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (Nov. 2000), 317–329.

[10] Mark Callaghan and Domas Mituzas. 2009. Poor Man's Profiler. Available at https://dom.as/2009/02/15/poor-mans-contention-profiling/. (Feb. 2009).

[11] Clang 2017. Clang 6 Documentatoin: ThreadSanitizer. http://clang.llvm.org/docs/ThreadSanitizer.html. (2017).

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

[13] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. 2015. Picon: Control Flow Integrity on LLVM IR. In *SSTIC*.

[14] Markus Dahm. 1999. *Byte Code Engineering*. 267–277.

[15] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. 2004. The Design and Implementation of FIT: A Flexible Instrumentation Toolkit. In *PASTE*. 29–34.

[16] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. In *PLDI*. 245–257.

[17] Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*. 85–96.

[18] Anne Carolyn Dinning. 1990. *Detecting Nondeterminism in Shared Memory Parallel Programs*. Ph.D. Dissertation. Department of Computer Science, New York University.

[19] DWARF Standards Committee. 2015. DWARF Debugging Information Format Version 4. (2015).

[20] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

[21] Free Software Foundation. 2009. GCC Wiki: LinkTimeOptimization. Available at https://gcc.gnu.org/wiki/LinkTimeOptimization. (Oct. 2009).

[22] Free Software Foundation. 2014. GNU Binutils. Available at https://www.gnu.org/software/binutils/. (September 2014).

[23] Free Software Foundation. 2017. *GNU Compiler Collection (GCC) Internals*.

[24] Free Software Foundation. 2017. GNU Compiler Collection (GCC) Internals: Plugins. Available at https://gcc.gnu.org/onlinedocs/gccint/Plugins.html. (April 2017).

[25] Felix Garcia and Javier Fernandez. 2000. POSIX Thread Libraries. *Linux Journal* 2000, 70es (Feb. 2000).

[26] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. *SIGPLAN Not.* 46 (June 2011), 458–469. Issue 6.

[27] Google, Inc. 2015. Google C++ Style Guide. Available at https://google.github.io/styleguide/cppguide.html. (2015).

[28] 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.

[29] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*. 125–138.

[30] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA*. 145–156.

[31] Intel Corporation. 2015. Pin 2.14 User Guide. Available at https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html. (January 2015).

[32] 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.

[33] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. 2011. Kismet: Parallel Speedup Estimates for Serial Programs. In *OOPSLA*. 519–536.

[34] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and Incremental LTO. In *CGO*. 111–121.

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP*. 327–353.

[36] Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. (July 1978), 558–565.

[37] James R. Larus and Eric Schnarr. 1995. EEL: Machine-independent Executable Editing. In *PLDI*. 291–300.

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. 75.

[39] Han Bok Lee and Benjamin Zorn. 1997. BIT: A Tool for Instrumenting Java Bytecodes. In *USENIX Symposium on Internet Technologies and Systems*. USENIX Association, 73–82.

[40] I-Ting Angelina Lee and Tao B. Schardl. 2015. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *SPAA*. 111–122.

[41] LLVM Project. 2016. LLVM Language Reference Manual. Available at http://llvm.org/docs/LangRef.html. (2016).

[42] LLVM Project. 2016. LLVM Link Time Optimization: Design and Implementation. Available at http://llvm.org/docs/LinkTimeOptimization.html. (2016).

[43] LLVM Project. 2016. Writing an LLVM Pass. Available at http://llvm.org/docs/WritingAnLLVMPass.html. (2016).

[44] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*. 190–200.

[45] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In *AOSD*. 239–250.

[46] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Supercomputing*. 24–33.

[47] John Mellor-Crummey. 1993. Compile-Time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs. In *PADD*. 129–139.

[48] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. 89–100.

[49] Oracle. 2004. JVM™Tool Interface (JVM TI). Available at http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/index.html. (2004).

[50] James Reinders. 2005. *VTune Performance Analyzer Essentials*. Intel Press.

[51] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *NT*. 1–7.

[52] 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.

[53] 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.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*.

[55] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer — Data Race Detection in Practice. In *WBIA*. 62–71.

[56] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitry Vyukov. 2011. *Dynamic Race Detection with LLVM Compiler*. Technical Report 37278. Google.

[57] Sameer Shende, Allen D. Malony, Janice Cuny, Peter Beckman, Steve Karmesin, and Kathleen Lindlan. 1998. Portable Profiling and Tracing for Parallel, Scientific Applications Using C++. In *SPDT*. 134–145.

[58] Michael D. Smith. 1991. *Tracing with pixie*. Technical Report CSL-TR-91-497. Stanford University.

[59] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *PLDI*. 196–205.

[60] 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.

[61] Richard M. Stallman and the GCC Developer Community. 2016. *Using the GNU Compiler Collection (for GCC version 6.1.0)*. Free Software Foundation.

[62] Basile Starynkevitch. 2011. MELT — A Translated Domain Specific Language Embedded in the GCC Compiler. In *DSL*.

[63] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. In *ISCA*. 185–197.

[64] Rabin A. Sugumar and Santosh G. Abraham. 1993. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. In *SIGMETRICS*. 24–35.

[65] Ian Lance Taylor. 2008. A new ELF linker. Available at https://research.google.com/pubs/archive/34417.pdf. (2008).

[66] The Clang Team. 2017. Clang Plugins — Clang 5 Documentation. Available at https://clang.llvm.org/docs/ClangPlugins.html. (2017).

[67] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. In *ISSTA*. 86–96.

[68] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. 2006. Analyzing Dynamic Binary Instrumentation Overhead.

[69] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *CC*. Lecture Notes in Computer Science, Vol. 1781. 18–34.

[70] William von Hagen. 2006. *The Definitive Guide to GCC* (second ed.). Apress, Chapter 6.

[71] David W. Wall. 1989. *Link-Time Code Modification*. Technical Report 89/17. Digital Western Research Laboratory.

[72] Josef Weidendorfer. 2008. Sequential Performance Analysis with Callgrind and KCachegrind. In *2nd International Workshop on Parallel Tools for High Performance Computing*. 93–113.

[73] David A. Wheeler. 2001. More Than a Gigabuck: Estimating GNU/Linux's Size. Available at http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html. (June 2001).

[74] Abe White. 2007. Serp. Available at http://serp.sourceforge.net/. (2007).

[75] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*. 221–234.

Received August 2017; revised October 2017; accepted December 2017