Performance Engineering of Software Systems
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 12

Homework 7: Dynamic-Analysis Tools

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

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. In your previous assignments, you have already availed yourself of a variety of dynamic-analysis tools, such as the Cilksan race detector, Valgrind, AddressSanitizer, Cachegrind, and perf. These tools generally operate as ***shadow computations*** — executing behind the scenes while the program-under-test runs. In this homework, you will investigate how to build your own dynamic-analysis tool using compiler instrumentation.

Traditionally, to write a dynamic-analysis tool that uses compiler instrumentation, a tool writer must modify the source code of the compiler. But modifying a compiler requires substantial mastery of the compiler's source code that is beyond the scope of this course. In this homework, you will use ***CSI***, a prototype framework supported by the Tapir/LLVM compiler for writing dynamic-analysis tools.

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.

Using CSI, tool writers insert their own instrumentation into the program-under-test by writing a library, called a ***CSI-tool***, that defines the semantics of relevant hooks. All unimplemented hooks in the CSI-tool default 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.

Figure 1 presents CSI-cov, an example tool written using the CSI framework. CSI-cov reports code-coverage information for an execution of the TIX of a program-under-test. As the TIX executes, CSI-cov records when each basic block runs. When the TIX terminates, CSI-cov reports how many times each block was executed, including its location in the source code. The code for CSI-cov, shown in Figure 1 in its entirety, is both short and simple.[1] In just 40 lines, the CSI framework allows this useful compiler-based tool to be implemented as a simple C library.

CSI targets program points that are visible in LLVM's intermediate representation (IR). In particular, CSI instruments six ***categories*** of ***IR objects***: memory loads, memory stores, basic blocks, function entry points, function exits, and call instructions.

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 quickly by exploiting standard compiler optimizations and analyses for precision and efficiency. To these ends, the CSI API provides four key features: hooks, flat and compact CSI ID spaces, forensic tables, and properties. We overview each of the features in turn.

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 when ***translation units*** — components of a program that are separately compiled — are dynamically loaded. 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 they often perform more efficiently than hash tables.

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.

[1] 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <csi.h>

long *block_executed = NULL;
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->line_number,
              source_loc->column_number, 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]++;
}
```

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

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 prototype CSI implementation in Tapir/LLVM provides forensic tables that associate IR objects with locations in the source code, as well as tables that specify the number of LLVM IR instructions in each basic block.

The code of CSI-cov in Figure 1 demonstrates that CSI ID's, hooks, and forensic tables can be used in a tool. CSI-cov maintains a table `block_executed`, defined in line 5, 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 6.

The initialization hook `__csi_init` is invoked before `main` of the program-under-test is called. This hook, defined in lines 25–27, registers the function `report`, defined in lines 8–23, to run when the program terminates. The `report` function prints out a report summarizing the results of the tool's analysis. 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 14. It prints the file name, the initial source line, and the number of times the basic block has been executed in lines 16–18. Finally, `report` prints the total number of basic blocks executed and the total number of basic blocks in the program in lines 20–21.

The hook `__csi_unit_init` is invoked whenever a new translation unit 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 29–36 to reallocate the array `block_executed` to incorporate the new basic blocks in lines 31–33, after which it updates the number `num_basic_blocks` of basic blocks in line 35.

The hook `__csi_bb_entry`, defined in lines 38–40, 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.

Properties. The one key feature of CSI that the CSI-cov example does not illustrate is the use of 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, specifies that the location is guaranteed to be on the stack and has a particular alignment. 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.

As an example of how a CSI tool might use properties, 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 45), it accesses a memory location

```
41 void __csi_before_load(const csi_id_t load_id,
42                         const void *addr,
43                         const int32_t num_bytes,
44                         const load_prop_t prop) {
45   if (prop.is_const ||
46       prop.is_not_shared ||
47       prop.is_read_before_write_in_bb)
48     return;
49   check_race_on_load(addr, num_bytes);
50 }
```

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

whose address is not captured and is thus guaranteed not to be shared (line 46), or it reads a memory address that is written to within the same basic block without any intervening function calls (line 47). 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. The function is analyzed at compile time and executed at runtime only when necessary. If a given property holds at link time and a tool branches depending on that condition, CSI can constant-fold the test and optimize the instrumentation.

Architecture of CSI

At first glance, CSI's 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 can contribute significantly and unnecessarily to runtime overhead.

The CSI prototype overcomes overheads through the use of modern compiler technology, specifically, through the use of ***link-time optimization (LTO)***. 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. Because all unimplemented hooks in a CSI-tool default to a null behavior, LTO can remove calls to these hooks as dead code. Furthermore, LTO can can constant-fold the test and optimize the instrumentation based on property values and perform a variety of optimizations on the instrumentation code itself.

From the point of view of the tool writer, CSI works as follows. The tool writer defines relevant hooks for her CSI-tool, and then she statically links her tool with a ***null tool***, which provides a no-op implementation of every hook. Linking these two tools thus produces a ***null-default CSI-tool***, where the functions defined in the tool writer's CSI-tool override the corresponding functions 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 a

program-under-test, LTO automatically elides calls to null hooks during its optimization pass.

2 Recitation: Analyzing function executions

In recitation, you will use CSI to develop tools to analyze the execution of functions in a serial program. You will first develop a simple tool to measure ***function coverage***, that is, to count how many times each function in the program executes. Time permitting, you will then extend the function-coverage tool to compute a ***profile*** for the program-under-test, which specifies how much of the total execution time of the program-under-test is spent in each function. For this recitation assignment, refer to Section 4 for documentation on the CSI API.

The file `func-analysis.c` provides a boiler-plate for the function-analysis CSI-tool. The check-off items in this homework will walk you through completing the implementation of this CSI-tool.

Checkoff Item 1: Add global variables to `func-analysis.c` for recording the number of times each function in the program executes. Add code to the `__csi_init` and `__csi_unit_init` instrumentation hooks to initialize these global variables.

Checkoff Item 2: Add a function `report` to `func-analysis.c` for outputting the results of the analysis. In particular, `report` should use CSI forensic tables to output the name of each function in the program and how many times that function was executed. Register the `report` function to run at program termination by adding the statement `atexit(report);` to `__csi_init`.

Checkoff Item 3: Which CSI IR-object hooks should be implemented to observe when each function is executed? Implement those IR-object hooks in `func-analysis.c` to record the number of times each function executes.

Checkoff Item 4: Compile the function-analysis CSI-tool and null tool and instrument `fib_serial` with the function-analysis tool as follows:

```
51 $ make CSI=1 FUNCANALYSIS=1 fib_serial
```

Finally, run the instrumented `fib_serial` executable.

Checkoff Item 5: Once you have confirmed that function-coverage CSI-tool is working, use it to instrument and analyze `bzip2`. We have already modified the `Makefile` for `bzip2` to optionally make use of a specified CSI-tool. Hence, you can instrument `bzip2` with your function-coverage CSI-tool and test the executable as follows:

```
52 $ cd bzip2-1.0.6
53 $ make test CSI=../func-analysis.o
54 $ ./bzip2 -z -9 -c ../bzip2-1.0.6.tar > /dev/null
```

Checkoff: Demonstrate to a TA that the function-coverage tool correctly counts the number of times each function in `bzip2` is executed.

Checkoff: *(Time permitting.)* Modify the function-coverage tool to profile each function, that is, to record the amount of the running time of the program is spent within each function. Try out the tool on `fib` and on `bzip2`. Make sure the tool doesn't over-count the time spent in recursive functions! Demonstrate to a TA that the profiling tool works.

3 Homework: Linear-regression analysis

In this homework, you will write some simple CSI-tools to perform ***linear-regression analysis***. Linear regression is a linear approach to modelling the relationship between a scalar ***dependent variable*** (in this case runtime) and one or more ***independent variables***. The idea behind linear-regression analysis is to develop a simple, predictive linear model to describe the performance of a program. Such a ***performance model*** provides guidance for how one can tune the performance of a program. There are many possible independent variables in a program that can affect it's running time, including:

- Lines of code. Some lines of code are more expensive than others.
- C language primitives. Some primitives are more expensive. Vectorizable instructions can make things cheaper.
- Machine instructions. Some instructions are more expensive than others e.g. square root compared to increment.
- Machine instructions weighted by their cost. Some instructions vary in cost e.g. a load with a cache hit vs. a cache miss.
- Cache Misses. TLB misses.

In this homework, we will guess that we can model the time T to run a program as following equation:

$$T = a \cdot I + b \cdot C \ ,$$

where I is the number of instructions, C is the number of cache misses, and a and b are constant values. Linear-regression analysis involves solving for the constants a and b in this model in two steps. First, one accumulates a variety of measurements of the program's running-time and corresponding instruction and cache-miss counts. Given those measurements, one can use a program such as gnuplot or Excel to perform a linear-regression fit of this model to the measurements, thereby solving for the constants a and b in the model. The constant a describes the cost of an instruction, and the constant b describes the cost of a cache miss in the program.

This homework uses linear-regression analysis to measure the cost of two short-running program events. First, you will measure the cost of a function call. Second, you will measure the cost of a `cilk_spawn`. You will study to programs, `fib_serial` and `fib_cilk`, to perform the linear-regression analyses in this homework. You can compile these programs without instrumentation as follows:

```
55 $ make fib_serial fib_cilk
```

As with the recitation assignment, refer to the documentation of the CSI API in Section 4 when writing CSI-tools for this homework.

Measuring the cost of a function call

To estimate the cost of a single function call, let us consider the serial recursive exponential-time `fib` function in the file `fib_serial.c`. This execution of this `fib` function is comprised of function calls and integer arithmetic and logic. Hence, we can model the execution time T of this code using the following linear-regression model:

$$T = a \cdot I + b \cdot F \,, \tag{1}$$

where I is the number of (non-function-call) instructions, F is the number of function calls, and a and b are constants. Using this model, we can estimate the cost of a function call by performing a linear-regression analysis to solve for the constants a and b.

To perform this analysis, we need to observe several measurements of the running-time T of different executions of `fib` that each perform different, uncorrelated numbers of instructions I and function calls F. To control the number of instructions and function calls that an execution of `fib` performs, the program `fib_serial.c` takes two parameters: the nth Fibonacci number to compute, and a ***base-case size***, which must be at least 2. The program then computes the nth Fibonacci number using an exponential-time recursive routine until `fib` is recursively called on an input value n smaller than the base-case size. At that point, the routine uses a simple loop to compute the requested Fibonacci number.

Write-up 1: Implement a CSI-tool in `linreg.c` to count the number of LLVM IR instructions and function calls that a program execution performs. You can get the number of LLVM IR instructions in a basic block using `__csi_get_bb_sizeinfo`. Use the `non-empty-size` value for the LLVM IR instruction count for a basic block. Submit your CSI-tool for this write-up.

Note: The prototype CSI implementation currently contains a bug that affects its instrumentation on simple recursive functions, such as `fib`. To get around this bug, declare all counter variables in your tool as `volatile`.

One issue with compiler instrumentation is that the instrumentation code can perturb the execution of the program-under-test. In particular, a program instrumented with the CSI-tool from Write-up 1 will perform more operations than the original uninstrumented program. As a result, the instrumented program might run more slowly than the uninstrumented program.

Write-up 2: Describe your strategy for collecting the necessary measurements to perform this linear-regression analysis using timers (e.g., calls to `clock_gettime`) and the CSI-tool from Write-up 1. Your strategy should address the following questions:

- What inputs will you use to run the program such that the counts of instructions and function calls are not highly correlated?
- How will you measure the running time of the program? What timers will you use? How many times will you run the program on each input? How will you aggregate your measurements to ensure you have reliable timings (e.g., using min, mean, or median)?
- How will you accommodate the fact that program instrumentation can perturb the running time of the program?

Now use your CSI-tool from Write-ups 1 and 2 to perform the linear-regression analysis.

Write-up 3: Following the strategy you outlined in Write-up 2, use timers and the CSI-tool from Write-up 1 to collect the running-time measurements and instruction and function-call counts for the `fib_serial.c` program. You can compile the linear-regression CSI-tool and instrument `fib_serial` with it as follows:

```
56 $ make CSI=1 LINREG=1 fib_serial
```

Use these measurements to perform the linear-regression analysis to solve for constants a and b in (1). You can use any data-analysis program you prefer (e.g., gnuplot or Excel) to perform the linear-regression fit on the collected measurements. What values does your analysis find for the constants a and b? How correlated are the counts of instructions and function calls in your measurements?

Measuring the cost of a Cilk spawn

Now you will use linear-regression analysis to measure the cost of a `cilk_spawn` statement. To perform this analysis, consider the Cilk program in `fib_cilk.c`, which computes Fibonacci numbers in exponential time. The serial running time of this program T_1 can be modeled using the following linear-regression model:

$$T_1 = a \cdot I + b \cdot S, \tag{2}$$

where I is the number of instructions (including serial function calls), S is the number of `cilk_spawn`'s, and a and b are constants. Using this model, one can estimate the cost of a `cilk_spawn` by performing a linear-regression analysis to solve for the constants a and b. For this analysis, we are primarily interested in solving for the constant b.

To solve (2) for the constant b, the `fib_cilk.c` program can be run using different numbers of `cilk_spawn` statements to compute a particular Fibonacci number. The `fib` routine in `fib_cilk.c` takes a base-case size along with the value n specifying which Fibonacci number to compute. When `fib` is called on a value of n larger than the base-case size, then the first recursive call to `fib` is spawned using a `cilk_spawn`. Otherwise this recursive call is called serially.

Write-up 4: Modify the CSI-tool you implemented in Write-up 1 to additionally count the number of `cilk_spawns` executed by a program-under-test. The prototype CSI framework in the Tapir/LLVM compiler inserts the following hook at the point in LLVM's IR corresponding to a `cilk_spawn`:

```
void __csi_detach(const csi_id_t detach_id);
```

To simplify the design of your tool, you can assume that you will run the program-under-test with `CILK_NWORKERS=1`. Submit your CSI-tool for this write-up.

Write-up 5: Mirroring the steps outlined in Write-ups 2 and 3, use the CSI-tool developed in Write-up 4 to perform the linear-regression analysis to estimate the cost of a `cilk_spawn`. You can compile the linear-regression CSI-tool and instrument `fib_cilk` with it as follows:

```
03 $ make CSI=1 LINREG=1 fib_cilk
```

When taking measurements for this analysis, run the `fib_cilk` executable using `CILK_NWORKERS=1`. Describe your strategy for taking the necessary timing measurements and instruction and `cilk_spawn` counts. What values does your analysis compute for the constant b? How correlated are the counts of instructions and spawns in your

```
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);
```

Figure 3: CSI hooks for initialization.

measurements? Based on the linear-regression analysis from Write-up 3, what is the cost of a `cilk_spawn` in terms of function calls?

4 Reference: The CSI API

This section overviews the CSI API implemented in the Tapir/LLVM compiler. The CSI API consists of four key features: CSI ID spaces, hooks, properties, and forensic tables. The API employs flat and compact CSI ID spaces to identify IR objects. The API provides hooks that are invoked for IR objects, as well as hooks for initializing a CSI tool. The API exports results of standard compiler analysis through property parameters, allowing the tool writer to avoid unnecessary instrumentation. Forensic tables associate IR objects with locations in the source code and relate IR objects to each other. This section describes each of these four key features.

CSI ID's

CSI identifies six categories of IR objects: function entries, function exits, basic blocks, call sites, loads, and stores. Each IR object is assigned a unique number, called a ***CSI ID***, within each 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`.

```
21 void __csi_func_entry(const csi_id_t func_id,
22                        const func_prop_t prop);
23 void __csi_func_exit(const csi_id_t func_exit_id,
24                       const csi_id_t func_id,
25                       const func_exit_prop_t prop);
26
27 void __csi_before_call(const csi_id_t callsite_id,
28                         const csi_id_t func_id,
29                         const call_prop_t prop);
30 void __csi_after_call(const csi_id_t callsite_id,
31                        const csi_id_t func_id,
32                        const call_prop_t prop);
```

Figure 4: CSI hooks for functions.

```
33 void __csi_bb_entry(const csi_id_t bb_id,
34                      const bb_prop_t prop);
35 void __csi_bb_exit(const csi_id_t bb_id,
36                     const bb_prop_t prop);
```

Figure 5: CSI hooks for basic blocks.

Tool initialization

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.

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.

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).[2] 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.

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

```
37 void __csi_before_load(const csi_id_t  load_id,
38                         const void *addr,
39                         const int32_t num_bytes,
40                         const load_prop_t prop);
41 void __csi_after_load(const csi_id_t load_id,
42                        const void *addr,
43                        const int32_t num_bytes,
44                        const load_prop_t prop);
45
46 void __csi_before_store(const csi_id_t store_id,
47                         const void *addr,
48                         const int32_t num_bytes,
49                         const store_prop_t prop);
50 void __csi_after_store(const csi_id_t store_id,
51                        const void *addr,
52                        const int32_t num_bytes,
53                        const store_prop_t prop);
```

Figure 6: CSI hooks for loads and stores.

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.

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

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`. In the prototype CSI implementation in Tapir/LLVM, Figures 7 and 8 presents the property struct types that have been implemented in the existing CSI prototype. As an example, consider the `load_prop_t` type for loads, which is shown in Figure 8. The property encodes the alignment of the load (`alignment`) — an integer k indicating that the address must be a multiple of k — whether the load is volatile (`is_volatile`), whether the loaded location is guaranteed not to be shared (`is_not_shared`), etc.

```
54 typedef struct {
55   // The call is indirect.
56   unsigned may_spawn : 1;
57   // Pad struct to 64 total bits.
58   uint64_t _unused : 63;
59 } func_prop_t;
60
61 typedef struct {
62   // The function might spawn.
63   unsigned may_spawn : 1;
64   // Pad struct to 64 total bits.
65   uint64_t _unused : 63;
66 } func_exit_prop_t;
67
68 typedef struct {
69   // The function might spawn.
70   unsigned is_indirect : 1;
71   // Pad struct to 64 total bits.
72   uint64_t _unused : 63;
73 } call_prop_t;
```

Figure 7: Definition of the CSI property for function entry points, function exits, and call sites.

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 get instruction counts associated with each basic block. Figure 9 presents the accessors for the forensic tables implemented in the CSI prototype implementation in Tapir/LLVM.

```
74 typedef struct {
75   // The alignment of the load.
76   unsigned alignment : 8;
77   // The loaded address is in a vtable.
78   unsigned is_vtable_access : 1;
79   // The loaded address points to constant data.
80   unsigned is_constant : 1;
81   // The loaded address is on the stack.
82   unsigned is_on_stack : 1;
83   // The loaded address cannot be captured.
84   unsigned may_be_captured : 1;
85   // The loaded address is read before it is written in the same basic block.
86   unsigned is_read_before_write_in_bb : 1;
87   // Pad struct to 64 total bits.
88   uint64_t _unused : 51;
89 } load_prop_t;
90
91 typedef struct {
92   // The alignment of the store.
93   unsigned alignment : 8;
94   // The stored address is in a vtable.
95   unsigned is_vtable_access : 1;
96   // The stored address points to constant data.
97   unsigned is_constant : 1;
98   // The stored address is on the stack.
99   unsigned is_on_stack : 1;
100  // The stored address cannot be captured.
101  unsigned may_be_captured : 1;
102  // Pad struct to 64 total bits.
103  uint64_t _unused : 52;
104 } store_prop_t;
```

Figure 8: Definition of the CSI property for memory loads and memory stores.

```
105 // Structure for the source-location information of an instrumented IR
106 // object.
107 typedef struct {
108   // The name of the instrumented IR object, such as the name of the
109   // function.  This field is NULL if no name is found.
110   char * name;
111   int32_t line_number;
112   int32_t column_number;
113   char * filename;
114 } source_loc_t;
115
116 // Accessors for various CSI forensic tables that associate IR objects
117 // with locations in the source code.  These accessors return NULL
118 // when given an invalid ID.
119 const source_loc_t *__csi_get_func_source_loc(const csi_id_t func_id);
120 const source_loc_t *__csi_get_func_exit_source_loc(const csi_id_t func_exit_id);
121 const source_loc_t *__csi_get_bb_source_loc(const csi_id_t bb_id);
122 const source_loc_t *__csi_get_call_source_loc(const csi_id_t call_id);
123 const source_loc_t *__csi_get_load_source_loc(const csi_id_t load_id);
124 const source_loc_t *__csi_get_store_source_loc(const csi_id_t store_id);
125
126 // Structure for the count of LLVM IR instructions within a basic
127 // block.
128 typedef struct {
129   // Total LLVM IR instructions.
130   int32_t full_ir_size;
131   // Number of LLVM IR instructions that are implemented with at least
132   // one instruction in a machine architecture, e.g., no debug
133   // intrinsic function calls or phi instructions.
134   int32_t non_empty_size;
135 } sizeinfo_t;
136
137 const sizeinfo_t *__csi_get_bb_sizeinfo(const csi_id_t bb_id);
```

Figure 9: CSI accessor functions for reading forensic tables for source location information and for reading the number of LLVM IR instructions in each basic block.