CSI: LLVM Comprehensive Static Instrumentation for Dynamic-Analysis Tools

6.S898: Advanced Performance Engineering for Multicore Applications
April 5, 2017

Adapted from slides by I-Ting Angelina Lee.

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

What Are Dynamic-Analysis Tools?

Dynamic-analysis tools examine the execution of a program under test to analyze its behavior and find bugs.

Example tools you might have used:

- Cilksan
- Valgrind, Callgrind, Cachegrind
- Address Sanitizer
- Linux perf tools
- gprof

Dynamic-Analysis Tools

A wide variety of dynamic-analysis tools exist today for analyzing and debugging programs.

- Memory checkers
- Race detectors
- Security tools
- Undefined behavior checkers
- Performance profilers
- Cache simulators
- Data-flow analyzers
- Code-coverage analyzers

Goals for Today

- Examine modern compiler and linker technology that enables fast dynamic-analysis tools.
- Learn how to write your own dynamic-analysis tool.

Outline

- Technology behind dynamic-analysis tools
- Introduction to CSI:LLVM
- The compiler and linker technology behind CSI

How Do Dynamic-Analysis Tools Work?

There are three mechanisms that tools typically use to examine the program's dynamic execution.

- **Asynchronous sampling**: Once in a while, pause the executing program and examine its state.
- **Instrumentation:** Modify the program to perform analysis as it executes.
  - **Binary:** Rewrite the executable binary directly.
  - **Compiler:** Inject the tool's code at compile time.

Asynchronous Sampling

The tool periodically interrupts the execution of the program under test — typically using an OS signal, such as `SIGALRM` or `SIGPROF` — to examine the program's dynamic state.

- Example tools: `perf record`, Google's profiling tools, HPCToolkit, poor man's profiler
- Pro: Tool exhibits low overhead, based on sampling rate.
- Pro: Tool perturbs the program's execution minimally.
- Con: Tool gathers coarse-grained, imprecise measurements.

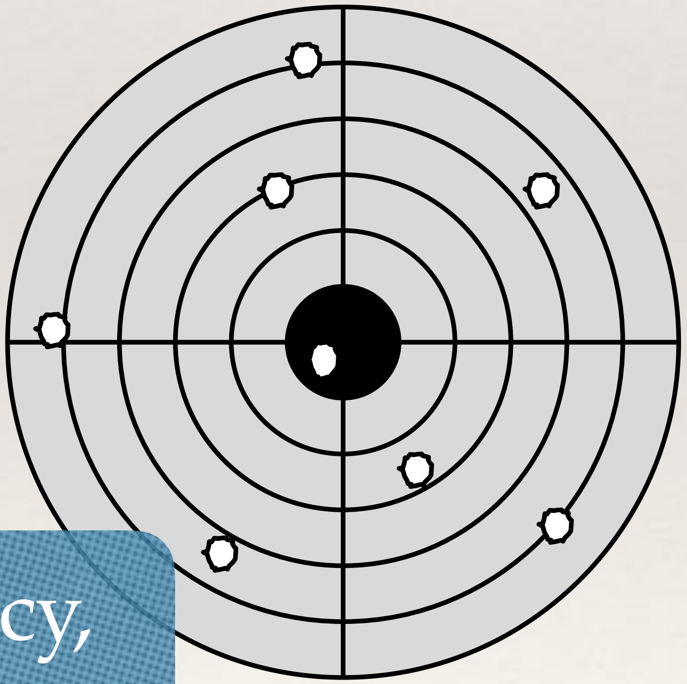
Instrumentation

The tool's code is injected into the program under test, such that the tool's analysis runs as the program runs.

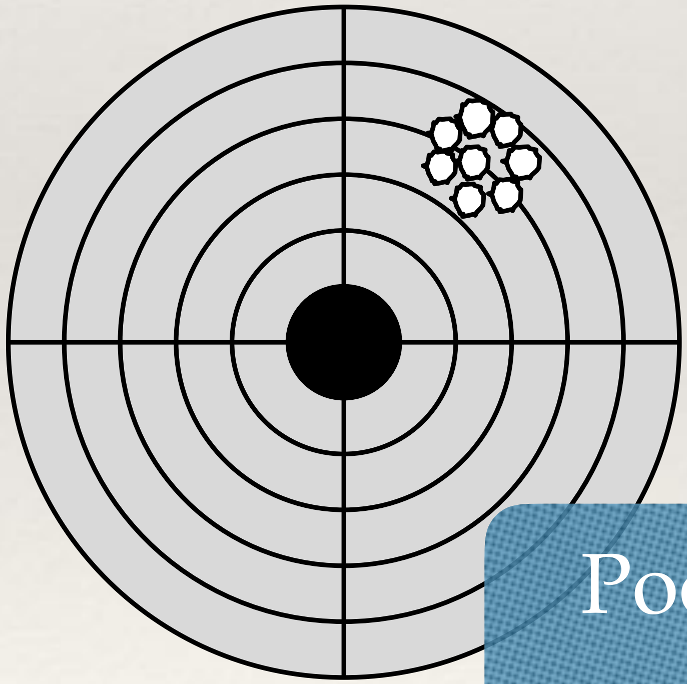
- Example tools: Cilksan, Google's Sanitizer tools, the Valgrind tool suite
- Pro: Tool can perform precise, detailed analysis.
- Con: Tool perturbs the execution of the program.
- Con: Tool can exhibit high overheads.

Accuracy vs. Precision

Question [Taguchi]**:** Suppose you are assembling an Olympic pistol team. You receive these two targets from two candidates for the team. Which candidate do you choose?



Good accuracy, poor precision



Poor accuracy, good precision

Instrumentation Example: Profiling

Conceptually, instrumenting a program simply involves adding the tool's code at all relevant program points.

Original program

```
void mmbase(double *C, double *A,
            double *B, int size) {
  // …
}

void mmdac(double *C, double *A,
           double *B, int size, int n) {
  if (size <= THRESHOLD) {
    mmbase(C, A, B, size);
  } else { /* … */ }
}

int main(int argc, const char *argv[]) {
  // …
  mmdac(C, A, B, n);
  // …
  return 0;
}
```

Program instrumented for profiling

```
void mmbase(double *C, double *A,
            double *B, int size) {
  start_timer("mmbase");
  // …
  stop_timer("mmbase");
}

void mmdac(double *C, double *A,
           double *B, int size, int n) {
  start_timer("mmdac");
  if (size <= THRESHOLD) {
    mmbase(C, A, B, size);
  } else { /* … */ }
  stop_timer("mmdac");
}

int main(int argc, const char *argv[]) {
  start_timer("main");
  // …
  mmdac(C, A, B, n);
  // …
  stop_timer("main");
  return 0;
}
```

Instrumentation Techniques

There are two general techniques for instrumenting a program automatically.

- **Binary instrumentation:** Rewrite the binary of the program to incorporate the tool’s code.
- **Compiler instrumentation:** Make the compiler inject the tool’s code into the program at compile time.

Binary Instrumentation

- Instrumentation points are identified in the executable binary of the program.
- Instrumentation is performed using a binary-instrumentation framework, such as Pin, Valgrind, or DynamoRIO, which handles modifying the binary.
- Pro: Tool does not require program recompilation.
- Pro: Tool can instrument third-party libraries.
- Con: Limited ability to optimize the tool's code.
- Con: Instrumentation imparts significant overhead.

Compiler Instrumentation

- Instrumentation points are identified in the compiler's intermediate representation (IR) of the program.
- Instrumentation is performed as a separate code-transformation pass in the compiler.
- Con: Tool requires program recompilation.
- Con: Tool cannot instrument third-party libraries.
- Pro: Tool can employ standard and custom compiler optimizations to improve its efficiency.
- Pro: Instrumentation imparts substantially less overhead than binary instrumentation.

High Level Summary

Asynchronous sampling	Binary instrumentation	Compiler instrumentation
Mechanism incurs practically no overhead.	Mechanism incurs substantial overhead.	Mechanism incurs little overhead.
Mechanism does not admit precise results.	Mechanism admits precise results on all code.	Mechanism admits precise results on recompiled code.
Tools are written by writing an OS signal handler.	Tools are written within a special framework.	Tools are written by hacking a compiler.

Outline

- Technology behind dynamic-analysis tools
- Introduction to CSI:LLVM
- The compiler and linker technology behind CSI

What is CSI?

- CSI is an instrumentation framework that provides comprehensive static instrumentation through the compiler.
- CSI enables programmers to write a new dynamic-analysis tools that use compiler instrumentation **without** having to modify the compiler.
- CSI supports most of the benefits of compiler instrumentation while simplifying the task of writing tools.

CSI was developed by Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson.

Example: Code Coverage in CSI

Using CSI, writing a dynamic-analysis tool amounts to writing a C library.

CSI-cov: A code-coverage CSI tool

```c
csi_id_t num_basic_blocks = 0;
int *block_executed = NULL;

void report() {
  fprintf(stderr,
          "Basic blocks not executed:\n");
  for (csi_id_t i = 0;
       i < num_basic_blocks; i++)
    if (source_loc_t *const source_loc =
        __csi_bb_get_source_loc(i))
      fprintf(stderr,
              "%s:%d executed %d times\n",
              source_loc->filename,
              source_loc->line,
              block_executed[i]);
  free(block_executed);
}
```

```c
void __csi_init() { atexit(report); }

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

Overview of CSI System

- By default, CSI inserts **instrumentation hooks** for all program points in the compiler's intermediate representation of the program.
- CSI publishes static and dynamic information for each instrumented program points.
- The tool writer implements the hooks her tool requires.
- The CSI system removes any unused hooks and optimizes the existing hooks using standard compiler optimizations.

The CSI API

The CSI:LLVM prototype currently inserts instrumentation for the following program points:

- Program initialization
- Six categories of IR objects: functions, function exits, basic blocks, loads, stores, and call sites

CSI Program-Initialization Hooks

CSI provides two hooks that run at the very start of the program, before `main()` executes.

```
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);
```

Counts of IR objects within a program "unit."

The very first thing that runs.

Run when each unit in the program is loaded into memory.

CSI Hooks for IR Objects

Function entry / exit hooks

```
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);
```

Basic block entry / exit hooks

```
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);
```

Call-site hooks

```
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);
```

Memory load and store hooks

```
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);
```

More hooks forthcoming!

Parameters of a CSI IR-Object Hook

Example CSI hook

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

CSI ID: A unique integer identifier for this IR object.

Dynamic parameters of the target IR object.

Property field: A bit-field struct that conveys results of static analysis.

CSI's Flat and Compact ID's

IR objects are numbered consecutively in a flat and compact ID space.

```
csi_id_t num_basic_blocks = 0;
int *block_executed = NULL;

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

The CSI-cov tool can use a simple array, indexed by CSI ID, to represent the set of basic blocks in the program.

CSI Forensic Tables

CSI also provides **forensic tables** for accessing source information for different IR objects.

```c
typedef struct {
  char * name;
  int32_t line_number;
  char * filename;
} source_loc_t;

// Accessors for various CSI Forensic tables.
// Return NULL when given an invalid ID.
source_loc_t const * __csi_get_func_source_loc(const csi_id_t func_id);
source_loc_t const * __csi_get_func_exit_source_loc(const csi_id_t func_exit_id);
source_loc_t const * __csi_get_bb_source_loc(const csi_id_t bb_id);
source_loc_t const * __csi_get_call_source_loc(const csi_id_t call_id);
source_loc_t const * __csi_get_load_source_loc(const csi_id_t load_id);
source_loc_t const * __csi_get_store_source_loc(const csi_id_t store_id);
```

CSI Property Fields

CSI exports bits of information from existing compiler analyses to hooks through property fields.

Code snippet from CSI-TSan

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  if (!prop.load_read_before_write_in_bb)
    UnalignedMemoryAccess(cur_thread(), CALLERPC,
                          (uptr)addr, num_bytes,
                          false, false);
}
```

The CSI port of Google's Thread Sanitizer uses a property to remove unnecessary checks for data races.

Optimization Using Properties

The CSI system uses properties to optimize away unnecessary instrumentation.

Configuration of Apache benchmark	Min runtime (s)	Max runtime (s)	Runtime overhead
Uninstrumented	5.33	9.48	—
Thread Sanitizer	13.64	21.12	1.43x—3.96x
CSI-TSan, no properties	28.02	43.73	2.95x—8.20x
CSI-TSan, one property	19.80	33.65	2.08x—6.31x

Efficiency of CSI

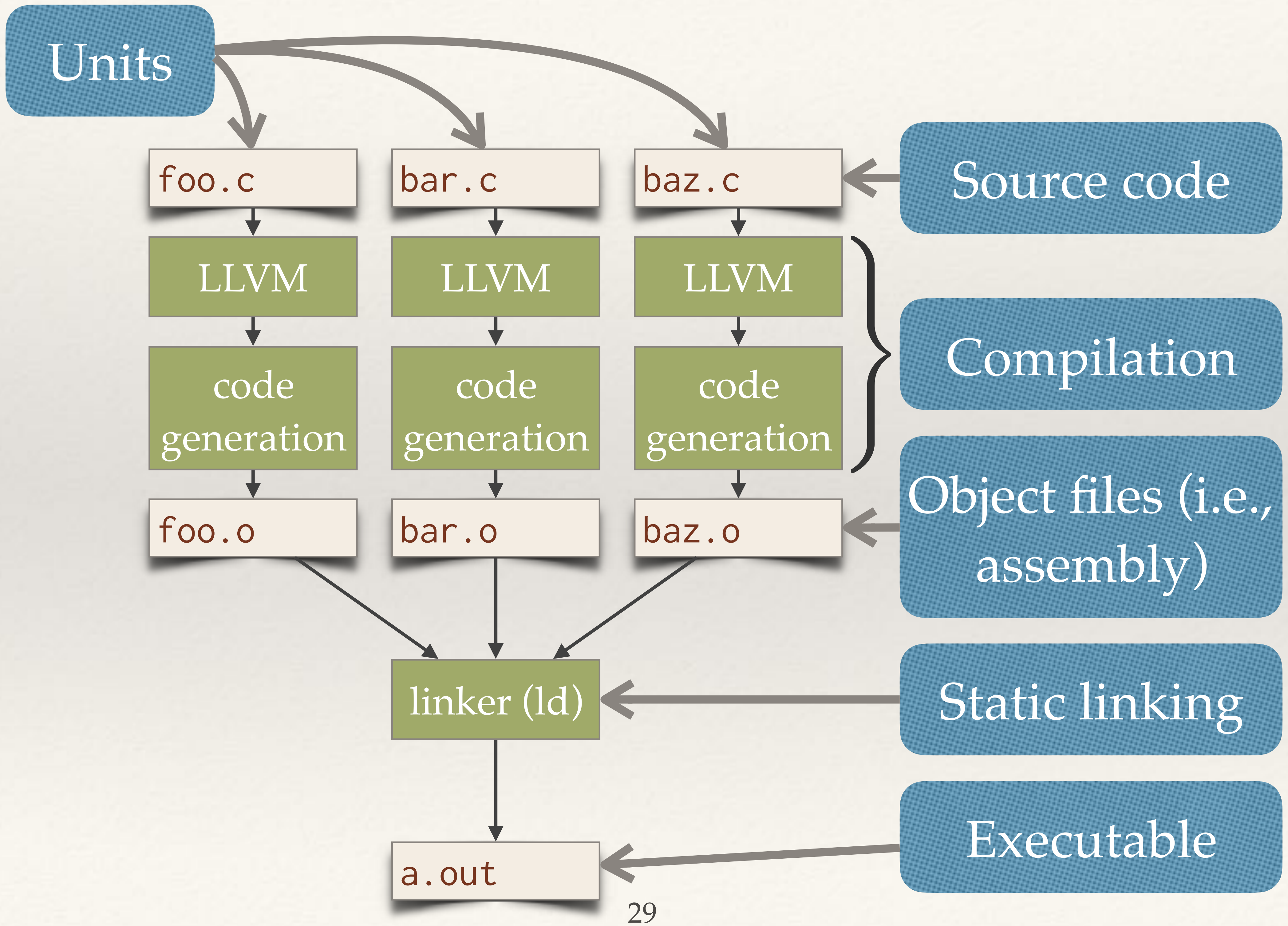
The CSI system also completely elides all unused instrumentation hooks.

Configuration	Min runtime (s)	Max runtime (s)	Runtime slowdown
bzip2, uninstrumented	13.06	13.25	—
bzip2 with CSI-null tool	13.04	13.24	0.98x—1.01x
Apache, uninstrumented	17.41	18.11	—
Apache with CSI-null tool	17.77	18.17	0.98x—1.04x

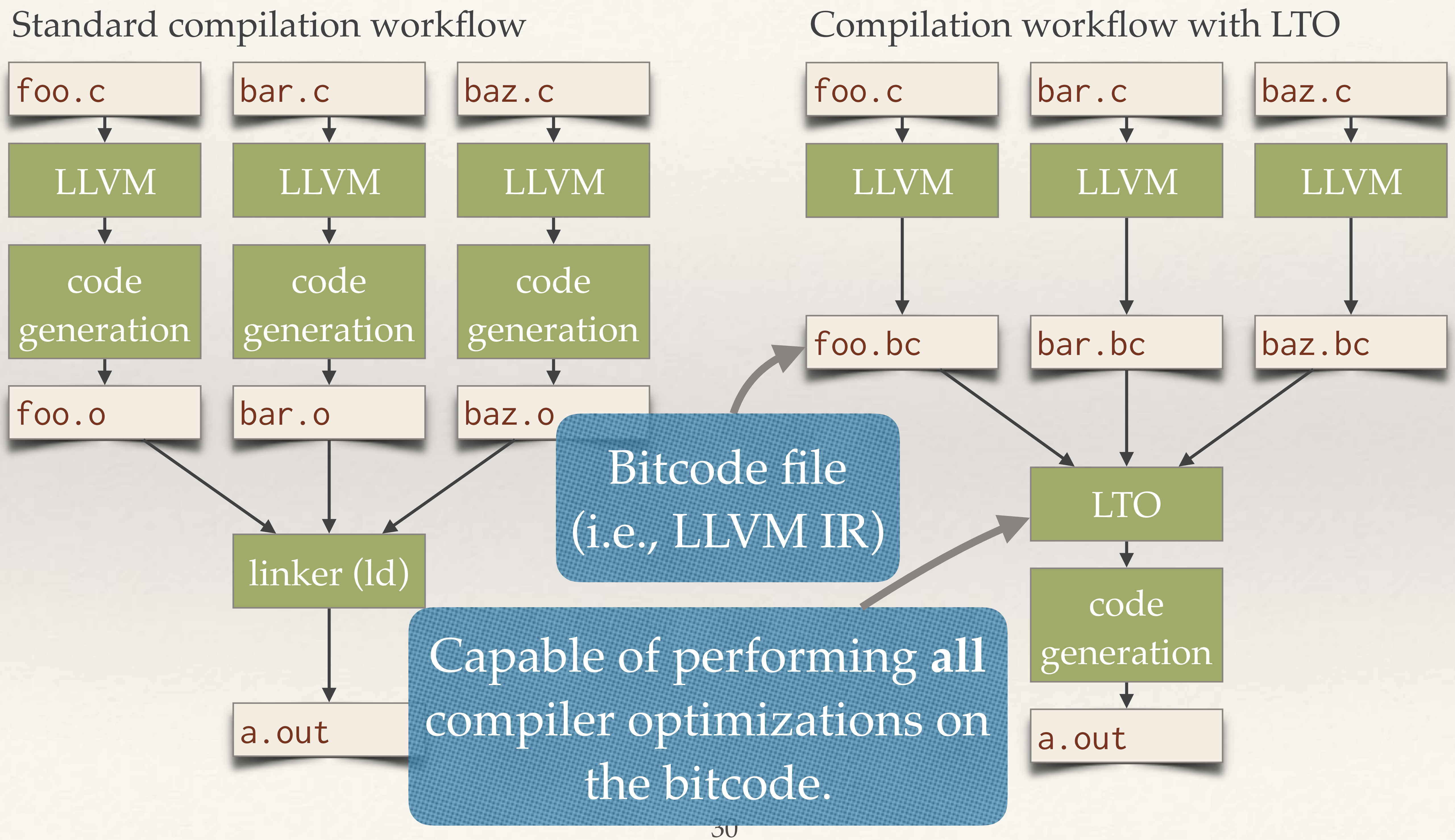
Outline

- Technology behind dynamic-analysis tools
- Introduction to CSI:LLVM
- The compiler and linker technology behind CSI

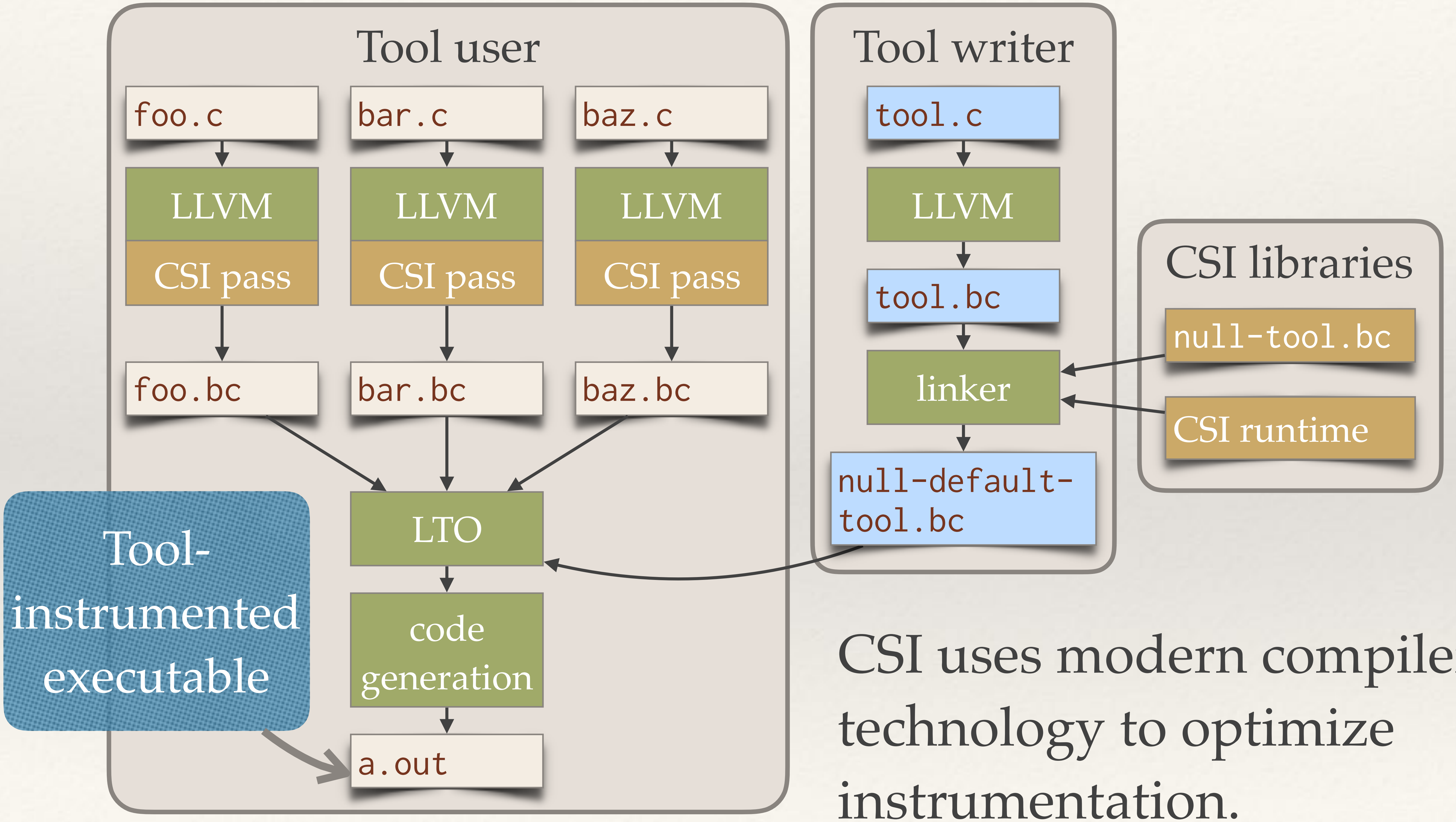
Standard Compilation and Linking



Link-Time Optimization (LTO)

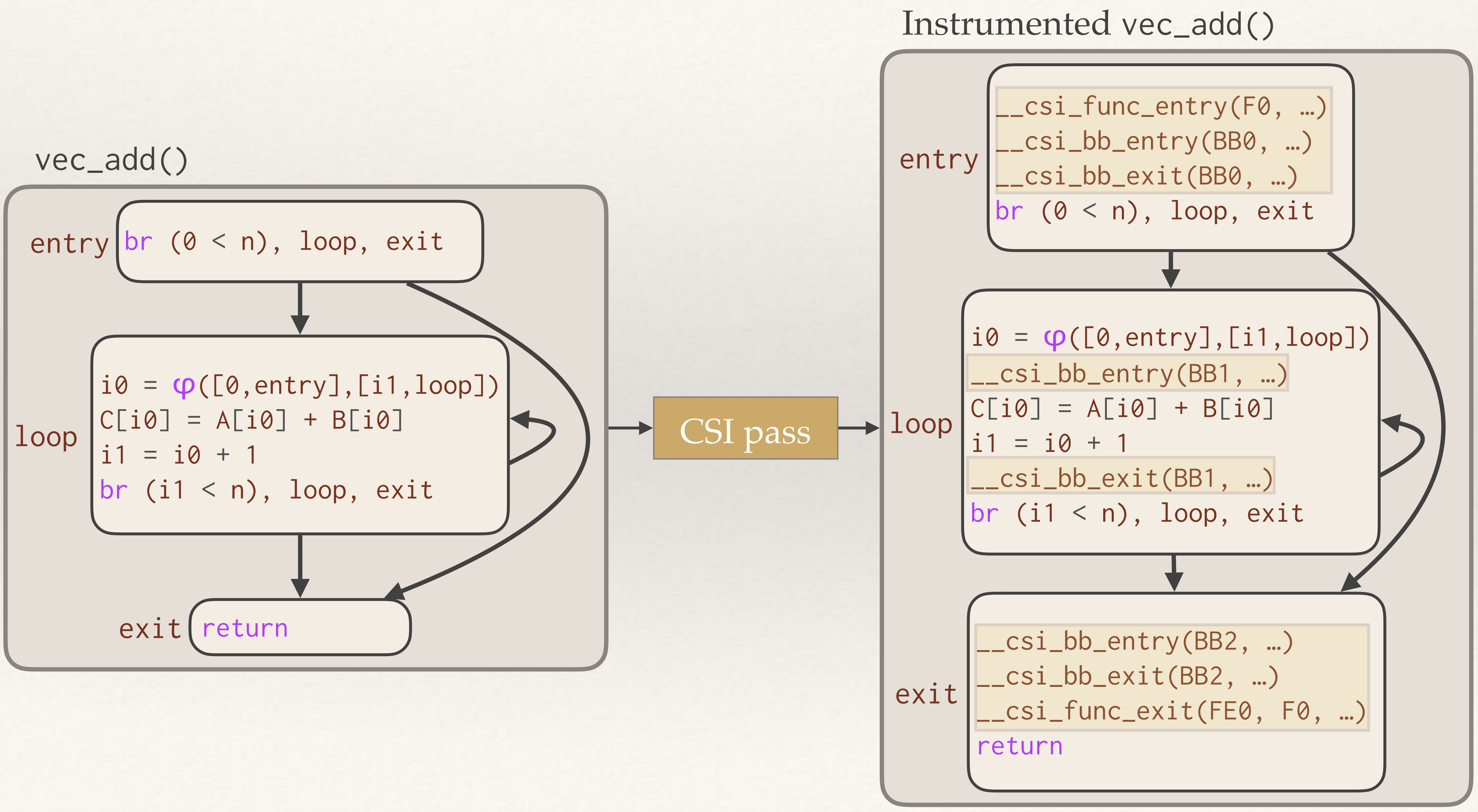


Architecture of CSI



CSI uses modern compiler technology to optimize instrumentation.

Simplified Example Output of CSI Pass



vec_add()

```
entry   br (0 < n), loop, exit

loop    i0 = φ([0,entry],[i1,loop])
        C[i0] = A[i0] + B[i0]
        i1 = i0 + 1
        br (i1 < n), loop, exit

exit    return
```

Instrumented vec_add()

```
entry   __csi_func_entry(F0, …)
        __csi_bb_entry(BB0, …)
        __csi_bb_exit(BB0, …)
        br (0 < n), loop, exit

loop    i0 = φ([0,entry],[i1,loop])
        __csi_bb_entry(BB1, …)
        C[i0] = A[i0] + B[i0]
        i1 = i0 + 1
        __csi_bb_exit(BB1, …)
        br (i1 < n), loop, exit

exit    __csi_bb_entry(BB2, …)
        __csi_bb_exit(BB2, …)
        __csi_func_exit(FE0, F0, …)
        return
```

Strong and Weak Symbols

CSI provides a null tool that contains default, empty implementations of all hooks as weak symbols.

CSI-cov tool

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

CSI null tool

```
__attribute__((weak))
void __csi_func_entry(const csi_id_t func_id,
                      const csi_prop_t prop) {}

__attribute__((weak))
void __csi_func_exit(const csi_id_t func_exit_id,
                     const csi_id_t func_id,
                     const csi_prop_t prop) {}

__attribute__((weak))
void __csi_bb_entry(const csi_id_t bb_id,
                    const csi_prop_t prop) {}

__attribute__((weak))
void __csi_bb_exit(const csi_id_t bb_id,
                   const csi_prop_t prop) {}
```

When statically linked, the strong `__csi_bb_entry` symbol in CSI-cov will override its weak counterpart.

Eliding Unused Instrumentation

Using LTO, ordinary **dead-code elimination** removes null hooks.

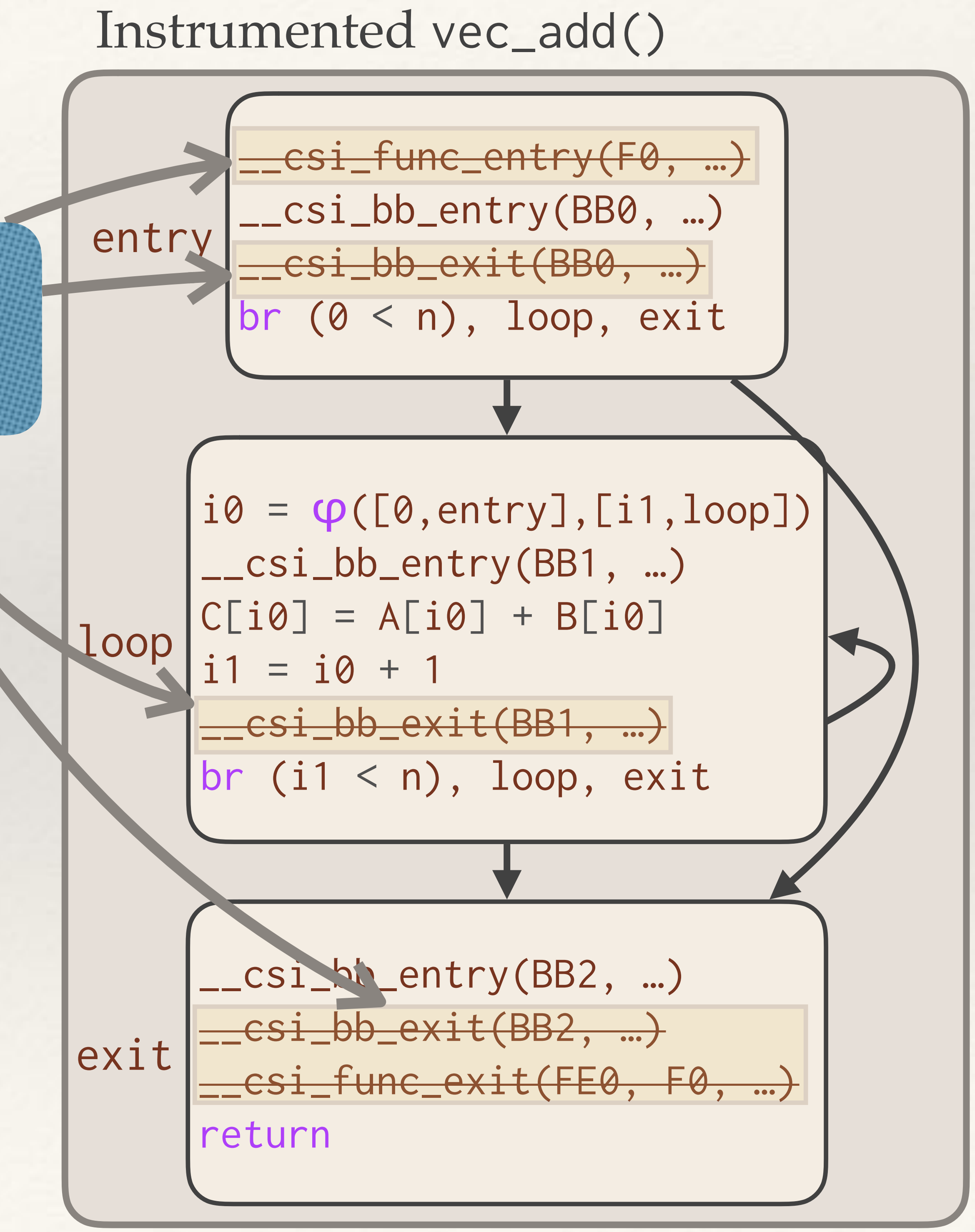
Null-default CSI-cov tool

```
__attribute__((weak))
void __csi_func_entry(const csi_id_t func_id,
                      const csi_prop_t prop) {}

__attribute__((weak))
void __csi_func_exit(const csi_id_t func_exit_id,
                     const csi_id_t func_id,
                     const csi_prop_t prop) {}

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

__attribute__((weak))
void __csi_bb_exit(const csi_id_t bb_id,
                   const csi_prop_t prop) {}
```



entry:
```
~~__csi_func_entry(F0, …)~~
__csi_bb_entry(BB0, …)
~~__csi_bb_exit(BB0, …)~~
br (0 < n), loop, exit
```

loop:
```
i0 = φ([0,entry],[i1,loop])
__csi_bb_entry(BB1, …)
C[i0] = A[i0] + B[i0]
i1 = i0 + 1
~~__csi_bb_exit(BB1, …)~~
br (i1 < n), loop, exit
```

exit:
```
__csi_bb_entry(BB2, …)
~~__csi_bb_exit(BB2, …)~~
~~__csi_func_exit(FE0, F0, …)~~
return
```

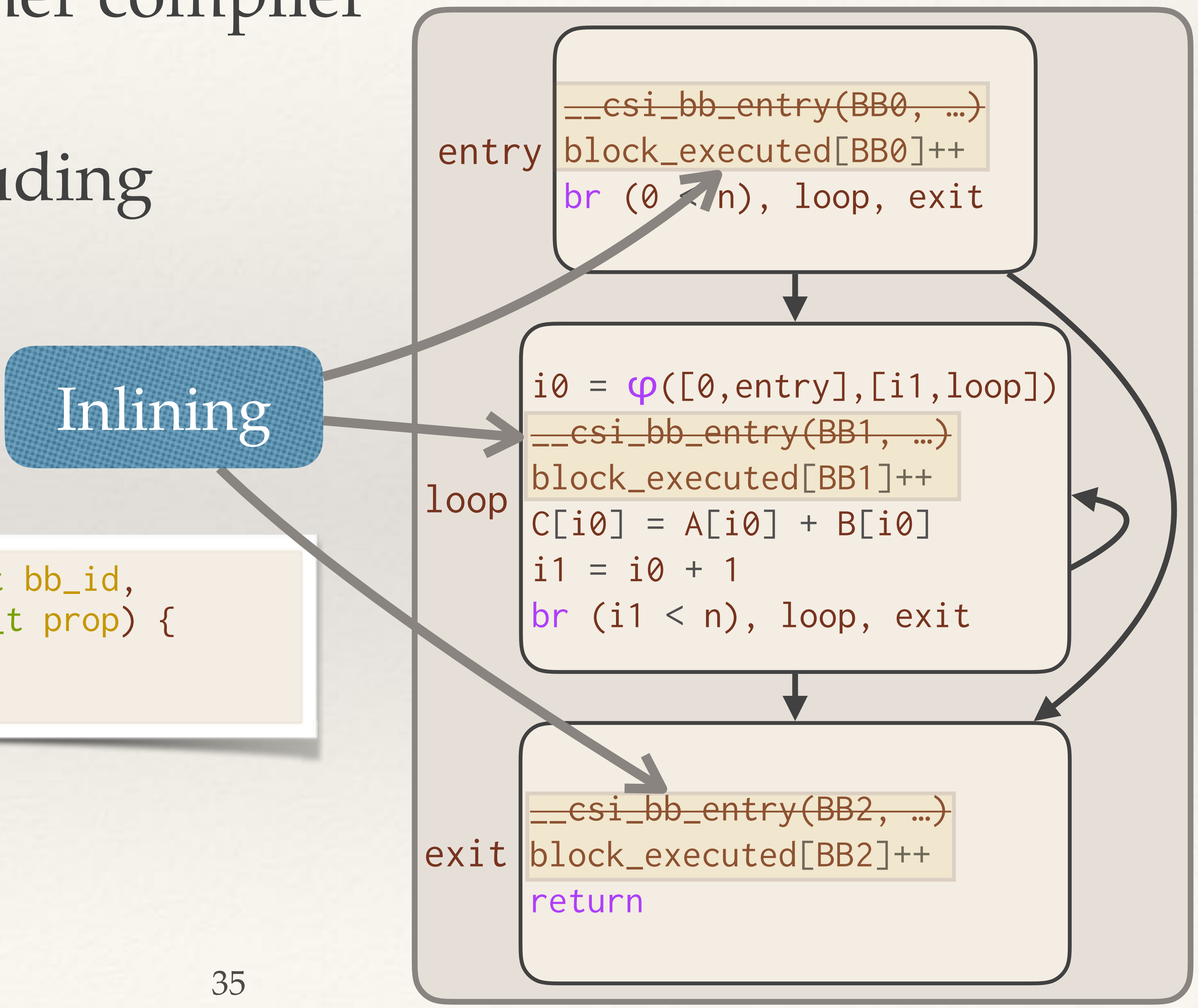
Inlining Instrumentation

LTO also performs other compiler optimizations on the instrumentation, including function inlining.

CSI-cov tool

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

Instrumented vec_add()



Other Instrumentation Optimization

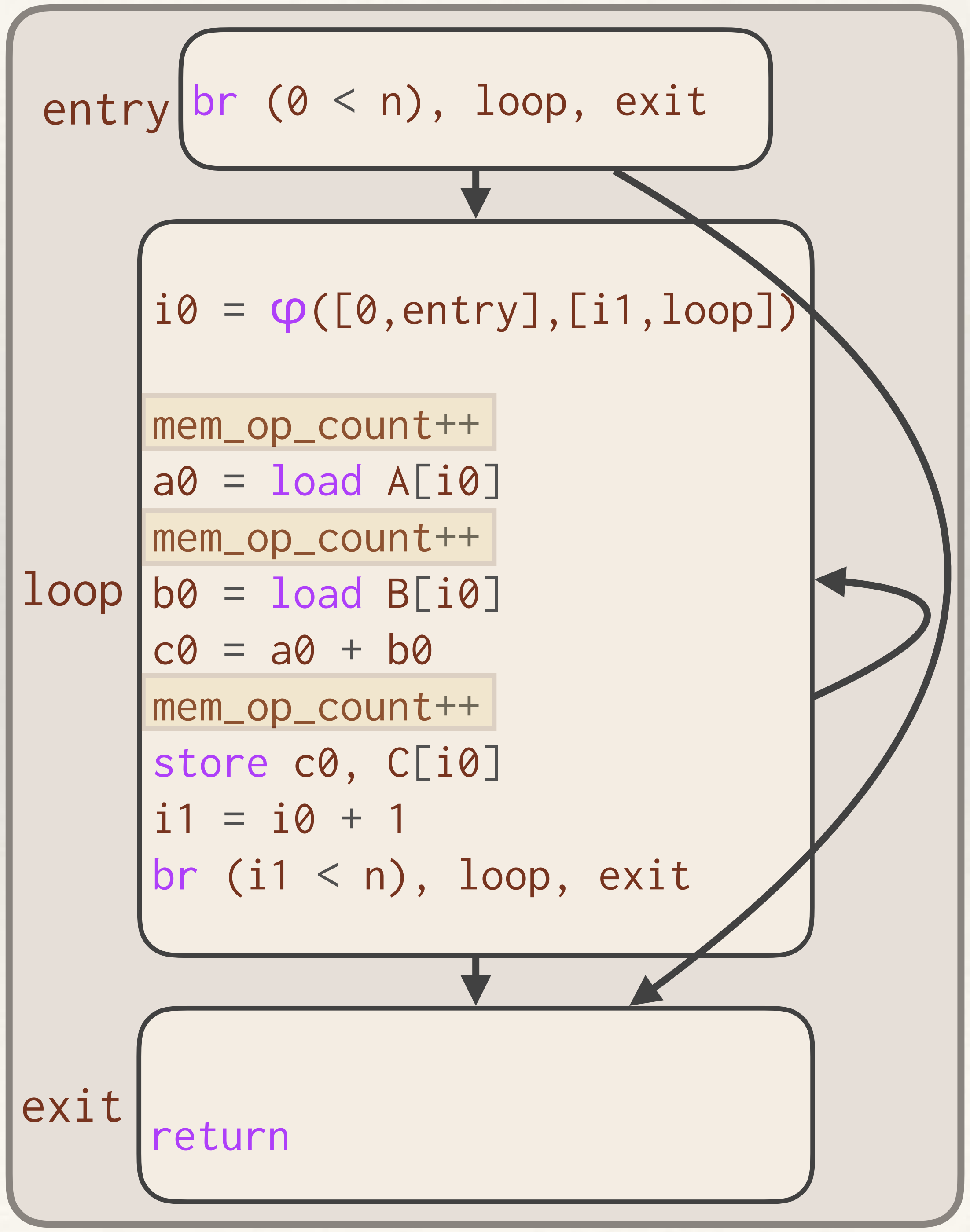
LTO can further simplify instrumentation with standard compiler optimizations.

CSI memory-operation counter

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  mem_op_count++;
}

void __csi_before_store(const csi_id_t store_id,
                        const void *addr,
                        const int32_t num_bytes,
                        const csi_prop_t prop) {
  mem_op_count++;
}
```

Instrumented vec_add()



```
entry:
br (0 < n), loop, exit

loop:
i0 = φ([0,entry],[i1,loop])
mem_op_count++
a0 = load A[i0]
mem_op_count++
b0 = load B[i0]
c0 = a0 + b0
mem_op_count++
store c0, C[i0]
i1 = i0 + 1
br (i1 < n), loop, exit

exit:
return
```

Other Instrumentation Optimization

LTO can further simplify instrumentation with standard compiler optimizations.

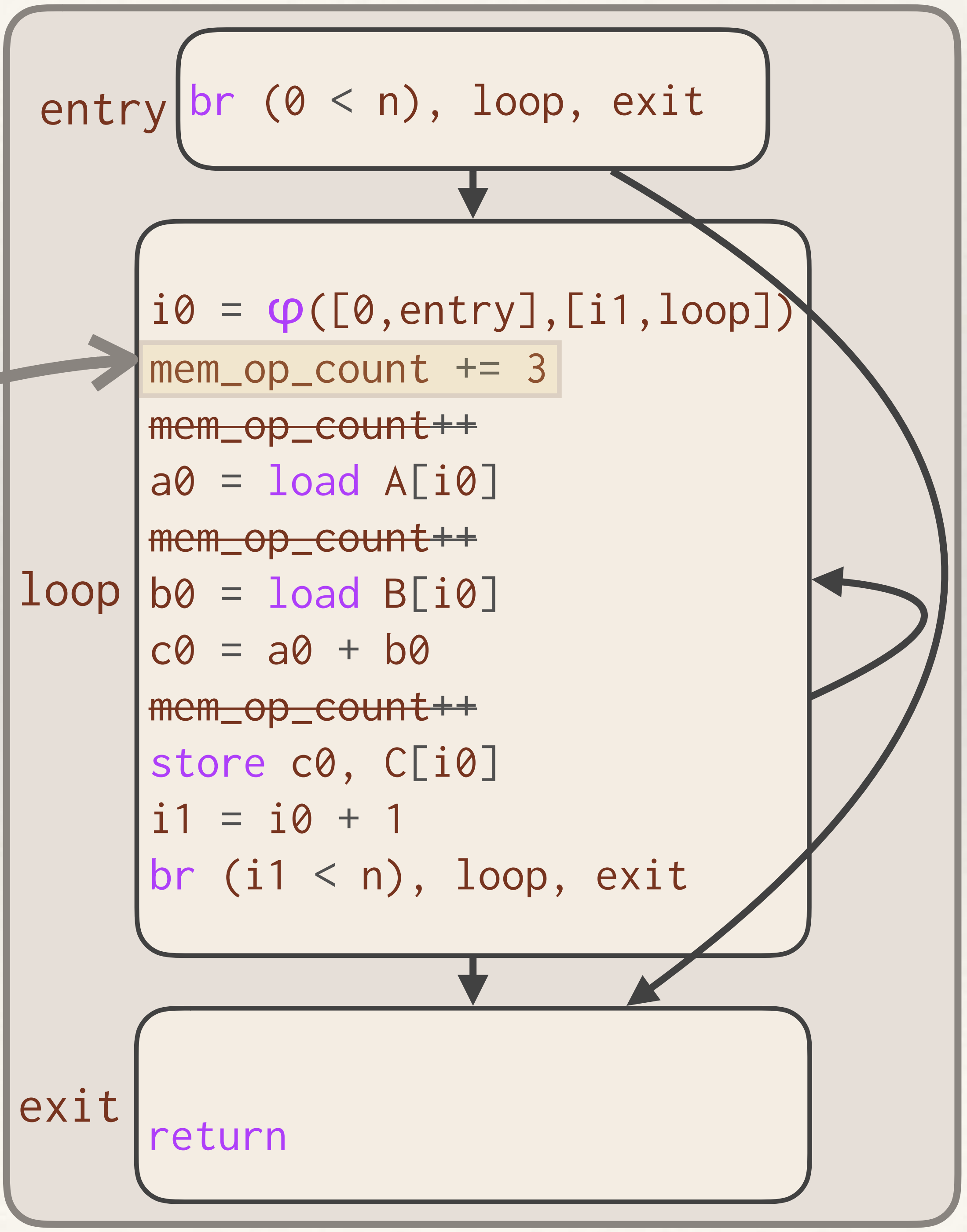
CSI memory-operation counter

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  mem_op_count++;
}

void __csi_before_store(const csi_id_t store_id,
                        const void *addr,
                        const int32_t num_bytes,
                        const csi_prop_t prop) {
  mem_op_count++;
}
```

Algebraic simplification of counter updates

Instrumented vec_add()



```
entry:
br (0 < n), loop, exit

loop:
i0 = φ([0,entry],[i1,loop])
mem_op_count += 3
~~mem_op_count++~~
a0 = load A[i0]
~~mem_op_count++~~
b0 = load B[i0]
c0 = a0 + b0
~~mem_op_count++~~
store c0, C[i0]
i1 = i0 + 1
br (i1 < n), loop, exit

exit:
return
```

Other Instrumentation Optimization

LTO can further simplify instrumentation with standard compiler optimizations.

CSI memory-operation counter

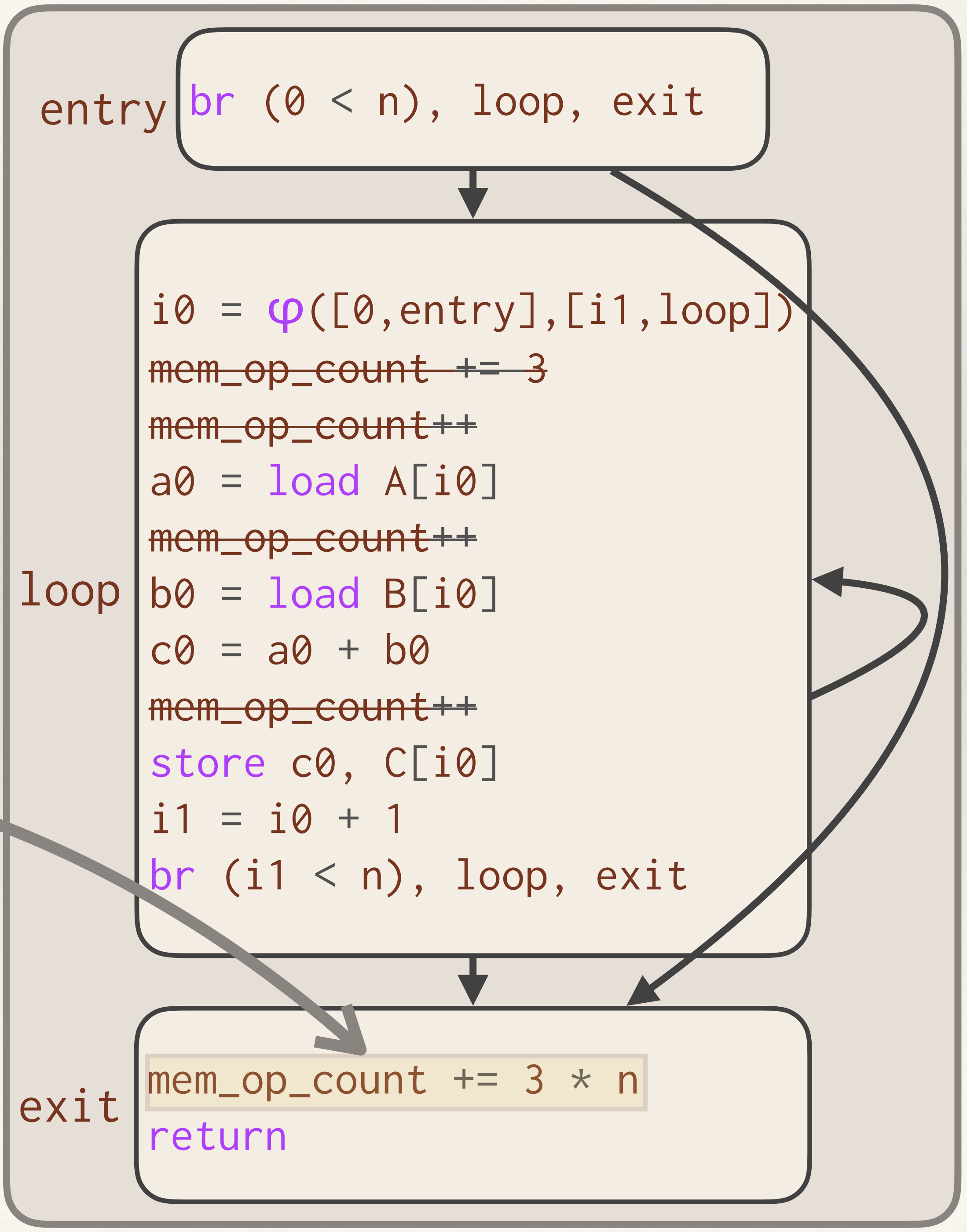
```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,

                       const csi_prop_t prop) {
  mem_op_count++;
}

void __csi_before_store(const csi_id_t store_id,
                        const void *addr,
                        const int32_t num_bytes,
                        const csi_prop_t prop) {
  mem_op_count++;
}
```

Code motion of counter updates out of loop

Instrumented vec_add()



```
entry  br (0 < n), loop, exit

loop   i0 = φ([0,entry],[i1,loop])
       ~~mem_op_count += 3~~
       ~~mem_op_count++~~
       a0 = load A[i0]
       ~~mem_op_count++~~
       b0 = load B[i0]
       c0 = a0 + b0
       ~~mem_op_count++~~
       store c0, C[i0]
       i1 = i0 + 1
       br (i1 < n), loop, exit

exit   mem_op_count += 3 * n
       return
```

Optimizing Based On Properties

Because CSI properties are constants at compile time, ordinary constant propagation can optimize instrumentation based on properties.

Instrumented basic block

```
…
__csi_before_load(7, A[i0], 4,
    { LOAD_BEFORE_STORE_IN_BB })

a0 = load A[i0]
__csi_before_load(8, B[i0], 4, 0)

b0 = load B[i0]
a1 = a0 + b0
__csi_before_store(3, A[i0], 4, 0)
store a1, A[i0]
…
```

Code snippet from CSI-TSan

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  if (!prop.load_read_before_write_in_bb)
    UnalignedMemoryAccess(cur_thread(), CALLERPC,
                          (uptr)addr, num_bytes,
                          false, false);
}
```

Optimizing Based On Properties

Because CSI properties are constants at compile time, ordinary constant propagation can optimize instrumentation based on properties.

Instrumented basic block

```
…
~~__csi_before_load(7, A[i0], 4,~~
~~    { LOAD_BEFORE_STORE_IN_BB })~~
if (!true) UnalignedMemoryAccess(…)
a0 = load A[i0]
~~__csi_before_load(8, B[i0], 4, 0)~~
if (!false) UnalignedMemoryAccess(…)
b0 = load B[i0]
a1 = a0 + b0
__csi_before_store(3, A[i0], 4, 0)
store a1, A[i0]
…
```

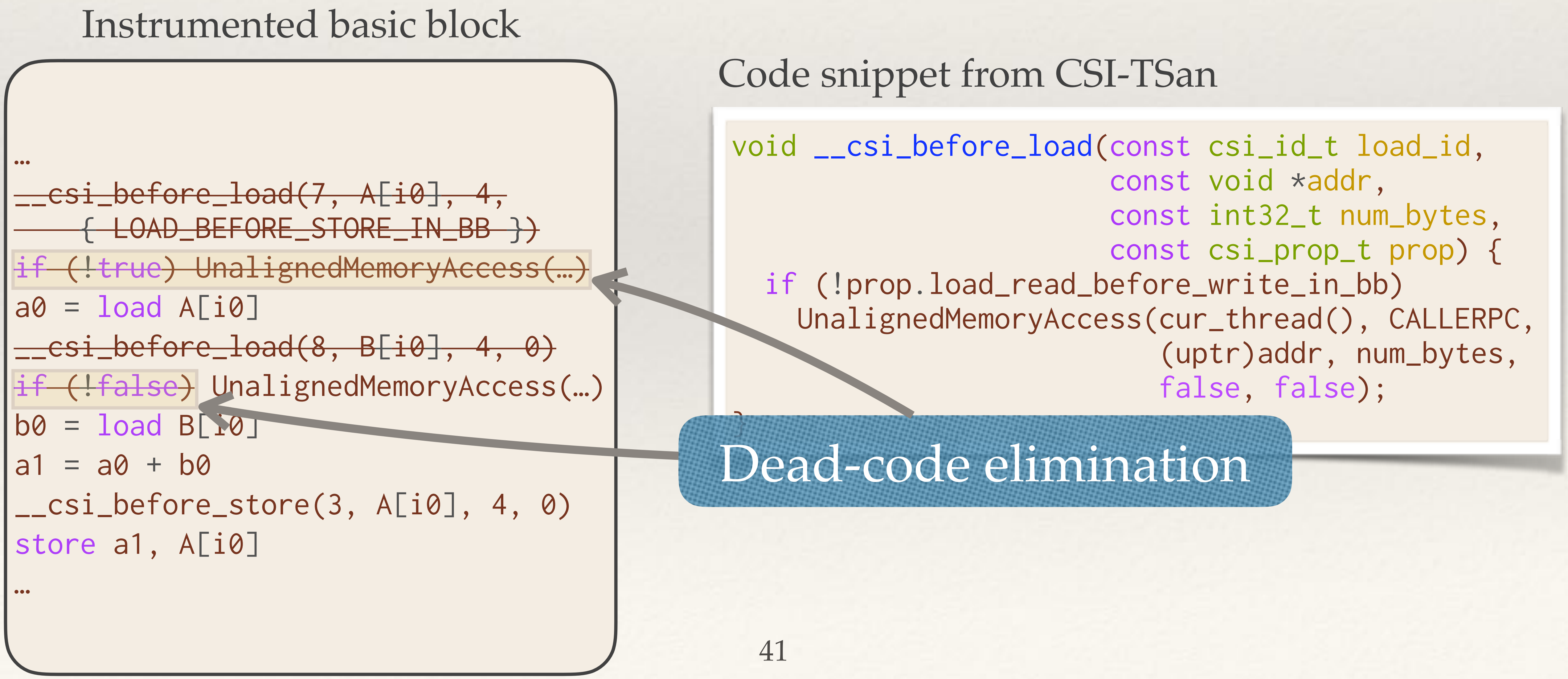
Code snippet from CSI-TSan

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  if (!prop.load_read_before_write_in_bb)
    UnalignedMemoryAccess(cur_thread(), CALLERPC,
                          (uptr)addr, num_bytes,
                          false, false);
}
```

Inlining and constant propagation

Optimizations Based On Properties

Because CSI properties are constants at compile time, ordinary constant propagation can optimize instrumentation based on properties.



Instrumented basic block

```
…
~~__csi_before_load(7, A[i0], 4,~~
~~    { LOAD_BEFORE_STORE_IN_BB })~~
~~if (!true) UnalignedMemoryAccess(…)~~
a0 = load A[i0]
~~__csi_before_load(8, B[i0], 4, 0)~~
~~if (!false)~~ UnalignedMemoryAccess(…)
b0 = load B[i0]
a1 = a0 + b0
__csi_before_store(3, A[i0], 4, 0)
store a1, A[i0]
…
```

Code snippet from CSI-TSan

```
void __csi_before_load(const csi_id_t load_id,
                       const void *addr,
                       const int32_t num_bytes,
                       const csi_prop_t prop) {
  if (!prop.load_read_before_write_in_bb)
    UnalignedMemoryAccess(cur_thread(), CALLERPC,
                          (uptr)addr, num_bytes,
                          false, false);
}
```

Dead-code elimination

Hands-On: Write Your Own CSI Tool

Try writing a simple profiling tool using CSI!

Here's the plan:

- Log in to 6898aws0.csail.mit.edu or 6898aws1.csail.mit.edu.
- Write a simple tool that counts the number of times each function executes.
- Compile your tool with a **serial** program, and run the instrumented program.

Hands-On: Writing the Tool

- Copy the contents of `/efs/share/csi-demo`.
- Fill in the missing code in `prof-tool.c`.

Hands-On: Compiling the Tool

- Compile the null tool:
```
$ clang -o null-tool.bc -emit-llvm -c null-tool.c -O3
```
- Compile your CSI tool:
```
$ clang -o prof-tool.bc -emit-llvm -c prof-tool.c -O3
```
- Compile your program with CSI and LTO:
```
$ clang -o fib.bc -c fib.c -fcsi -flto -O3 -g
```
- Link everything together with LTO:
```
$ clang -o fib fib.bc prof-tool.bc null-tool.bc \
> -flto -fuse-ld=gold -O3 -lclang_rt.csi-x86_64
```

Hands-On: Try the Tool on bzip2

The `bzip2-1.0.6` directory contains a copy of the bzip2 source and a Makefile that has been modified to compile bzip2 with LTO and, optionally, with a CSI tool.

- Compile bzip2 with LTO:

```
$ cd bzip2-1.0.6
$ make test
```

- Compile bzip2 with LTO and CSI:

```
$ make clean
$ make test CSI="../prof-tool.bc ../null-tool.bc"
```

Function Interpositioning

Problem: Suppose your tool wants a slightly modified version of a common library routine, e.g., a version of `malloc()` that guarantees all allocations are 8-byte aligned.

- Modern programming technology supports **function interpositioning** for replacing individual routines.
- Function interpositioning allows the replacement function to call the original function.
- There are three kinds of function interpositioning: compile-time, link-time, and run-time.

Compile-Time Interpositioning

Compile-time interpositioning involves using the C preprocessor to interpose a target function.

mymalloc.c

```
#include <malloc.h>
#include <stdio.h>

void *mymalloc(size_t s)
{
  printf("Calling my malloc\n”);
  return malloc(s);
}
```

malloc.h

```
#define malloc(size) mymalloc(size)
void *mymalloc(size_t size)
```

Included in program files

Compiled separately

Link-Time Interpositioning

Link-time interpositioning uses the static linker to rename the target function.

mymalloc.c

```c
#include <stdio.h>

// __real_malloc() is used to call
// actual library malloc()
void *__real_malloc(size_t size);

// User defined wrapper for malloc()
void *__wrap_malloc(size_t size)
{
  printf("Calling my malloc\n");
  return __real_malloc(size);
}
```

Compiled separately

Pass the next flag to the linker

Compiling foo to use the custom malloc:

```
$ clang -o foo -Wl,--wrap=malloc \
> foo.o mymalloc.o
```

Replace calls to malloc with calls to __wrap_malloc, and let __real_malloc refer to the original malloc.

Run-Time Interpositioning

Run-time interpositioning uses the dynamic linker to replace calls to the target function.

mymalloc.c

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

typedef void*(*malloc_t)(size_t);
static malloc_t real_malloc = NULL;

void *malloc(size_t size) {
  char buffer[50];
  if (!real_malloc) {
    real_malloc =
      (malloc_t)dlsym(RTLD_NEXT, "malloc");
    char *error = dlerror();
    if (error != NULL) {
      fputs(error, stderr);
      exit(1);
    }
  }
  sprintf(buffer, "Calling my malloc\n");
  write(1, buffer, strlen(buffer));
  return real_malloc(size);
}
```

- Compile mymalloc.c as a dynamic library:

```
$ clang -ldl —shared -fPIC \
-o mymalloc.so mymalloc.c
```

- Use the LD_PRELOAD environment variable to load mymalloc.so first:

```
$ LD_PRELOAD=./mymalloc.so \
> ./foo
```

Happy Tool-Writing!