Formal Methods in System Design manuscript No.
(will be inserted by the editor)

INTERASPECT: Aspect-Oriented Instrumentation with GCC

Justin Seyster · Ketan Dixit · Xiaowan Huang · Radu Grosu · Klaus Havelund · Scott A. Smolka · Scott D. Stoller · Erez Zadok

Received: date / Accepted: date

Abstract We present the INTERASPECT instrumentation framework for GCC, a widely used compiler infrastructure. The addition of plug-in support in the latest release of GCC makes it an attractive platform for runtime instrumentation, as GCC plug-ins can directly add instrumentation by transforming the compiler's intermediate representation. Such transformations, however, require expert knowledge of GCC internals. INTERASPECT addresses this situation by allowing instrumentation plug-ins to be developed using the familiar vocabulary of Aspect-Oriented Programming: pointcuts, join points, and advice functions. Moreover, INTERASPECT uses specific information about each join point in a pointcut, possibly including results of static analysis, to support powerful customized instrumentation. We describe the INTERASPECT API and present several examples that illustrate its practical utility as a runtime-verification platform. We also introduce a tracecut system that uses INTERASPECT to construct program monitors that are formally specified as regular expressions.

Keywords program instrumentation, aspect-oriented programming, GCC, monitoring, tracecut

1 Introduction

GCC is a widely used compiler infrastructure that supports a variety of input languages, e.g., C, C++, Fortran, Java, and Ada, and over 30 different target machine architectures. GCC translates each of its front-end languages into a language-independent intermediate representation called GIMPLE, which then gets translated to machine code for one of GCC's many target architectures. GCC is a large software system with more than 100 developers contributing over the years and a steering committee consisting of 13 experts who strive to maintain its architectural integrity.

In earlier work [7], we extended GCC to support *plug-ins*, allowing users to add their own custom passes to GCC in a modular way without patching and recompiling the GCC

J. Seyster, K. Dixit, X. Huang, R. Grosu, S. A. Smolka, S. D. Stoller, E. Zadok
Department of Computer Science, Stony Brook University

K. Havelund
Jet Propulsion Laboratory, California Institute of Technology

source code. Released in April 2010, GCC 4.5 [16] includes plug-in support that is largely based on our design.

GCC's support for plug-ins presents an exciting opportunity for the development of practical, widely-applicable program transformation tools, including program-instrumentation tools for runtime verification. Because plug-ins operate at the level of GIMPLE, a plug-in is applicable to all of GCC's front-end languages. Transformation systems that manipulate machine code may also work for multiple programming languages, but low-level machine code is harder to analyze and lacks the detailed type information that is available in GIMPLE.

Implementing instrumentation tools as GCC plug-ins provides significant benefits but also presents a significant challenge: despite the fact that it is an intermediate representation, GIMPLE is in fact a low-level language, requiring the writing of low-level GIMPLE Abstract Syntax Tree (AST) traversal functions in order to transform one GIMPLE expression into another. Therefore, as GCC is currently configured, the writing of plug-ins is not trivial but for those intimately familiar with GIMPLE's peculiarities.

To address this challenge, we developed the INTERASPECT program-instrumentation framework, which allows instrumentation plug-ins to be developed using the familiar vocabulary of Aspect-Oriented Programming (AOP). INTERASPECT is itself implemented using the GCC plug-in API for manipulating GIMPLE, but it hides the complexity of this API from its users, presenting instead an aspect-oriented API in which instrumentation is accomplished by defining *pointcuts*. A pointcut denotes a set of program points, called *join points*, where calls to *advice functions* can be inserted by a process called *weaving*.

INTERASPECT's API allows users to customize the weaving process by defining *callback functions* that get invoked for each join point. Callback functions have access to specific information about each join point; the callbacks can use this to customize the inserted instrumentation, and to leverage static-analysis results for their customization.

We also present the INTERASPECT *Tracecut extension* to generate program monitors directly from formally specified tracecuts. A tracecut [32] matches *sequences of pointcuts* specified as a regular expression. Given a tracecut specification T, INTERASPECT Tracecut instruments a target program so that it communicates program events and event parameters directly to a monitoring engine for T. The tracecut extension adds the necessary monitoring instrumentation exclusively with the INTERASPECT API presented here.

In summary, INTERASPECT offers the following novel combination of features:

- INTERASPECT builds on top of GCC, a widely used compiler infrastructure.
- INTERASPECT exposes an API that encourages and simplifies open-source collaboration.
- INTERASPECT is versatile enough to provide instrumentation for many purposes, including monitoring a tracecut specification.
- INTERASPECT has access to GCC internals, which allows one to exploit static analysis and meta-programming during the weaving process.

The full source of the INTERASPECT framework is available from the INTERASPECT website under the GPLv3 license [19].

To illustrate INTERASPECT's practical utility, we have developed a number of program-instrumentation plug-ins that use INTERASPECT for custom instrumentation. These include a *heap visualization* plug-in designed for the analysis of JPL Mars Science Laboratory software; an *integer range analysis* plug-in that finds bugs by tracking the range of values for each integer variable; and a *code coverage* plug-in that, given a pointcut and test suite, measures the percentage of join points in the pointcut that are executed by the test suite.

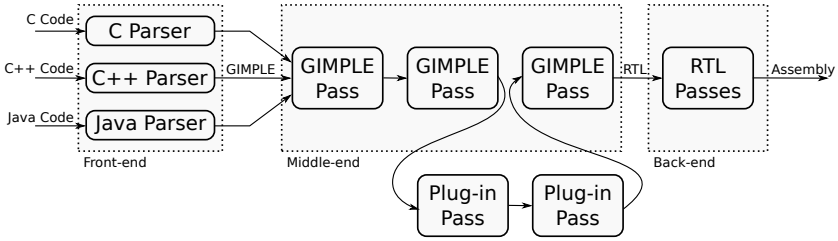

Fig. 1 A simplified view of the GCC compilation process.

The rest of the article is structured as follows. Section 2 provides an overview of GCC and the INTERASPECT framework. Section 3 introduces the INTERASPECT API. Section 4 presents the three case studies: heap visualization, integer range analysis, and code coverage. Section 5 describes how we extended INTERASPECT with a tracecut system. Section 6 summarizes related work, and Section 7 concludes the article. A preliminary version of this article, which did not consider the tracecut extension, appeared last year [29].

2 Overview of GCC and the INTERASPECT Architecture

Overview of GCC. As Figure 1 illustrates, GCC translates all of its front-end languages into the GIMPLE intermediate representation for analysis and optimization. Each transformation on GIMPLE code is split into its own *pass*. These passes, some of which may be implemented as *plug-ins*, make up GCC's *middle-end*. Moreover, a plug-in pass may be INTERASPECT-based, enabling the plug-in to add instrumentation directly into the GIMPLE code. The final middle-end passes convert the optimized and instrumented GIMPLE to the Register Transfer Language (RTL), which the *back-end* translates to assembly.

GIMPLE is a C-like three-address (3A) code. Complex expressions (possibly with side effects) are broken into simple 3A statements by introducing new, temporary variables. Similarly, complex control statements are broken into simple 3A (conditional) `goto`s by introducing new labels. Type information is preserved for every operand in each GIMPLE statement.

Figure 2 shows a C program and its corresponding GIMPLE code, which preserves source-level information such as data types and procedure calls. Although not shown in the example, GIMPLE types also include pointers and structures.

A disadvantage of working purely at the GIMPLE level is that some language-specific constructs are not visible in GIMPLE code. For example, targeting a specific kind of loop as a pointcut is not currently possible because all loops look the same in GIMPLE. INTERASPECT can be extended with language-specific pointcuts, whose implementation could hook into one of the language-specific front-end modules instead of the middle-end.

INTERASPECT *architecture.* INTERASPECT works by inserting a pass that first traverses the GIMPLE code to identify program points that are join points in a specified pointcut. For each such join point, it then calls a user-provided weaving callback function, which can insert calls to advice functions. Advice functions can be written in any language that will link with the target program, and they can access or modify the target program's state, including its global variables. Advice that needs to maintain additional state can declare static variables and global variables.

```
int main() {
    int a, b, c;
    a = 5;
    b = a + 10;
    c = b + foo(a, b);          =>
    if (a > b + c)
        c = b++ / a + (b * a);
    bar(a, b, c);
}
```

```
 1.  int main {
 2.      int a, b, c;
 3.      int T1, T2, T3, T4;
 4.      a = 5;
 5.      b = a + 10;
 6.      T1 = foo(a, b);
 7.      c = b + T1;
 8.      T2 = b + c;
 9.      if (a <= T2) goto fi;
10.          T3 = b / a;
11.          T4 = b * a;
12.          c = T3 + T4;
13.          b = b + 1;
14.  fi:
15.      bar (a, b, c);
16.  }
```

Fig. 2 Sample C program (left) and corresponding GIMPLE representation (right)

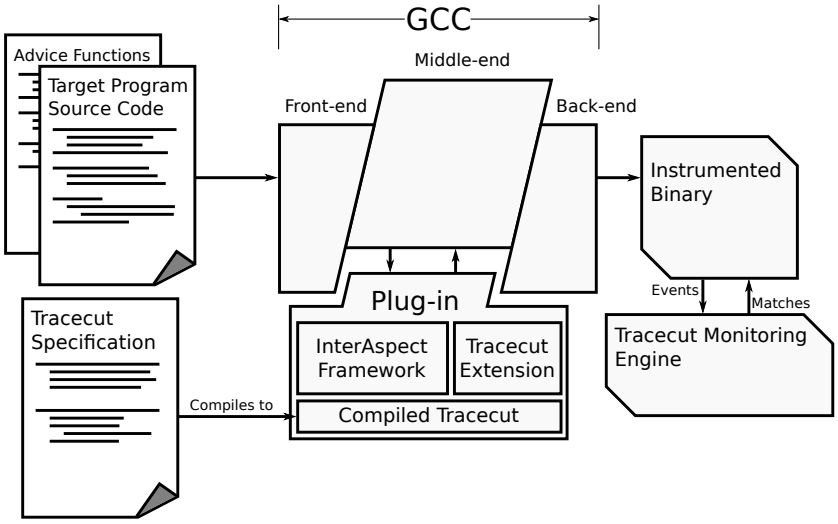

Fig. 3 Architecture of the INTERASPECT instrumentation framework with its tracecut extension. The tracecut specification is a simple C program. The tracecut extension translates events in the specification to pointcuts, and the INTERASPECT framework directly instruments the pointcuts using GCC's GIMPLE API. The instrumented binary sends events to the tracecut monitoring engine, and monitors signal matches by calling advice functions, which are compiled alongside the target program. It is also possible to specify just pointcuts, in which case the tracecut extension and monitoring engine are not necessary.

Unlike traditional AOP systems which implement a special AOP language to define pointcuts, INTERASPECT provides a C API for this purpose. We believe that this approach is well suited to open collaboration. Extending INTERASPECT with new features, such as new kinds of pointcuts, does not require agreement on new language syntax or modification to parser code. Most of the time, collaborators will only need to add new API functions.

The INTERASPECT Tracecut extension API uses INTERASPECT to generate program monitors from formally specified tracecuts. Tracecuts match sequences of pointcuts, specified as regular expressions. The instrumentation component of the extension, which is implemented in C, benefits from INTERASPECT's design as an API: it need only call API functions to define and instrument the pointcuts that are necessary to monitor the tracecut.

```
struct aop_pointcut *aop_match_function_entry(void);
```
Creates pointcut denoting every function entry point.
```
struct aop_pointcut *aop_match_function_exit(void);
```
Creates pointcut denoting every function return point.
```
struct aop_pointcut *aop_match_function_call(void);
```
Creates pointcut denoting every function call.
```
struct aop_pointcut *aop_match_assignment_by_type(struct aop_type *type);
```
Creates pointcut denoting every assignment to a variable or memory location that matches a type.

Fig. 4 *Match functions* for creating pointcuts.

Figure 3 shows the architecture of a monitor implemented with INTERASPECT Tracecut. The tracecut itself is defined in a short C program that calls the INTERASPECT Tracecut API to specify tracecut properties. Linking the compiled *tracecut program* with INTERASPECT and the tracecut extension produces a plug-in that instruments events relevant to the tracecut. A target program compiled with this plug-in will send events and event parameters to the tracecut monitoring engine, which then determines if any sequence of events matches the tracecut rule. The target program can include tracecut-handling functions so that the monitoring engine can report matches directly back to the program.

3 The INTERASPECT API

This section describes the functions in the INTERASPECT API, most of which fall naturally into one of two categories: (1) functions for creating and filtering pointcuts, and (2) functions for examining and instrumenting join points. Note that users of our framework can write plug-ins solely with calls to these API functions; it is not necessary to include any GCC header files or manipulate any GCC data structures directly.

Creating and filtering pointcuts. The first step for adding instrumentation in INTERASPECT is to create a pointcut using a *match* function. Our current implementation supports the four match functions given in Figure 4, allowing one to create four kinds of pointcuts.

Using a function entry or exit pointcut makes it possible to add instrumentation that runs with every execution of a function. These pointcuts provide a natural way to insert instrumentation at the beginning and end of a function the way one would with before-execution and an after-returning advices in a traditional AOP language. A call pointcut can instead target calls to a function. Call pointcuts can instrument calls to library functions without recompiling them. For example, in Section 4.1, a call pointcut is used to intercept all calls to `malloc`.

The assignment pointcut is useful for monitoring changes to program values. For example, we use it in Section 4.1 to track pointer values so that we can construct the heap graph. We plan to add several new pointcut types, including pointcuts for conditionals and loops. These new pointcuts will make it possible to trace the complete path of execution as a program runs, which is potentially useful for coverage analysis, profiling, and symbolic execution.

After creating a match function, a plug-in can refine it using *filter* functions. Filter functions add additional constraints to a pointcut, removing join points that do not satisfy those constraints. For example, it is possible to filter a call pointcut to include only calls that return a specific type or only calls to a certain function. Figure 5 summarizes filter functions for call pointcuts.

```
void aop_filter_call_pc_by_name(struct aop_pointcut *pc, const char *name);
```
Filter function calls with a given name.

```
void aop_filter_call_pc_by_param_type(struct aop_pointcut *pc, int n,
                                      struct aop_type *type);
```
Filter function calls that have an n^{th} parameter that matches a type.

```
void aop_filter_call_pc_by_return_type(struct aop_pointcut *pc,
                                       struct aop_type *type);
```
Filter function calls with a matching return type.

Fig. 5 *Filter functions* for refining function-call pointcuts.

```
void aop_join_on(struct aop_pointcut *pc, join_callback callback,
                 void *callback_param);
```
Call `callback` on each join point in the pointcut `pc`, passing `callback_param` each time.

Fig. 6 *Join function* for iterating over a pointcut.

```
const char *aop_capture_function_name(aop_joinpoint *jp);
```
Captures the name of the function called in the given join point.

```
struct aop_dynval *aop_capture_param(aop_joinpoint *jp, int n);
```
Captures the value of the n^{th} parameter passed in the given function join point.

```
struct aop_dynval *aop_capture_return_value(aop_joinpoint *jp);
```
Captures the value returned by the function in a given call join point.

Fig. 7 *Capture functions* for function-call join points.

Instrumenting join points. INTERASPECT plug-ins iterate over the join points of a pointcut by providing an iterator callback to the *join* function, shown in Figure 6. For an INTERASPECT plug-in to instrument some or all of the join points in a pointcut, it should join on the pointcut, providing an iterator callback that inserts a call to an *advice* function. INTERASPECT then invokes that callback for each join point.

Callback functions use *capture* functions to examine values associated with a join point. For example, given an assignment join point, a callback can examine the name of the variable being assigned. This type of information is available statically, during the weaving process, so the callback can read it directly with a capture function like `aop_capture_lhs_name`. Callbacks can also capture dynamic values, such as the value on the right-hand side of the assignment, but dynamic values are not available at weave time. Instead, when the callback calls `aop_capture_assigned_value`, it gets an `aop_dynval`, which serves as a weave-time placeholder for the runtime value. The callback cannot read a value from the placeholder, but it can specify it as a parameter to an inserted advice function. When the join point executes (at runtime), the value assigned also gets passed to the advice function. Sections 4.1 and 4.2 give more examples of capturing values from assignment join points.

Capture functions are specific to the kinds of join points they operate on. Figures 7 and 8 summarize the capture functions for function-call join points and assignment join points, respectively.

AOP systems like AspectJ [21] provide Boolean operators such as *and* and *or* to refine pointcuts. The INTERASPECT API could be extended with corresponding operators. Even in their absence, a similar result can be achieved in INTERASPECT by including the appropriate logic in the callback. For example, a plug-in can instrument calls to `malloc` *and* calls to `free` by joining on a pointcut with all function calls and using the `aop_capture_function_name` facility to add advice calls only to `malloc` and `free`. Simple cases like this can furthermore be handled by using regular expressions to match function names, which would be a straightforward addition to the framework.

```
const char *aop_capture_lhs_name(aop_joinpoint *jp);
```
Captures the name of a variable assigned to in a given assignment join point, or returns NULL if the join point does not assign to a named variable.

```
enum aop_scope aop_capture_lhs_var_scope(aop_joinpoint *jp);
```
Captures the scope of a variable assigned to in a given assignment join point. Variables can have global, file-local, and function-local scope. If the join point does not assign to a variable, this function returns `AOP_MEMORY_SCOPE`.

```
struct aop_dynval *aop_capture_lhs_addr(aop_joinpoint *jp);
```
Captures the memory address assigned to in a given assignment join point.

```
struct aop_dynval *aop_capture_assigned_value(aop_joinpoint *jp);
```
Captures the assigned value in a given assignment join point.

Fig. 8 *Capture functions* for assignment join points.

```
void aop_insert_advice(struct aop_joinpoint *jp, const char *advice_func_name,
                       enum aop_insert_location location, ...);
```
Insert an advice call, before or after a join point (depending on the value of `location`), passing any number of parameters. A plug-in obtains a join point by iterating over a pointcut with `aop_join_on`.

Fig. 9 *Insert function* for instrumenting a join point with a call to an advice function.

After capturing, a callback can add an advice-function call before or after the join point using the *insert* function of Figure 9. The `aop_insert_advice` function takes any number of parameters to be passed to the advice function at runtime, including values captured from the join point and values computed during instrumentation by the plug-in itself.

Using a callback to iterate over individual join points makes it possible to customize instrumentation at each instrumentation site. A plug-in can capture values about the join point to decide which advice function to call, which parameters to pass to it, or even whether to add advice at all. In Section 4.2, this feature is exploited to uniquely index named variables during compilation. Custom instrumentation code in Section 4.3 separately records each instrumented join point in order to track coverage information.

Function body duplication. INTERASPECT provides a *function body duplication* facility that makes it possible to toggle instrumentation at the function level. Although inserting advice at the GIMPLE level creates very efficient instrumentation, users may still wish to switch between instrumented and uninstrumented code for high-performance applications. Duplication creates two or more copies of a function body (which can later be instrumented differently) and redefines the function to call a special advice function that runs at function entry and decides which copy of the function body to execute.

When joining on a pointcut for a function with a duplicated body, the caller specifies which copy the join should apply to. By only adding instrumentation to one copy of the function body, the plug-in can create a function whose instrumentation can be turned on and off at runtime. Alternatively, a plug-in can create a function that can toggle between different kinds of instrumentation. Section 4.2 presents an example of using function body duplication to reduce overhead by sampling.

4 Applications

In this section, we present several example applications of the INTERASPECT API. The plug-ins we designed for these examples provide instrumentation that is tailored to specific problems (memory visualization, integer range analysis, code coverage). Though custom-made, the plug-ins themselves are simple to write, requiring only a small amount of code.

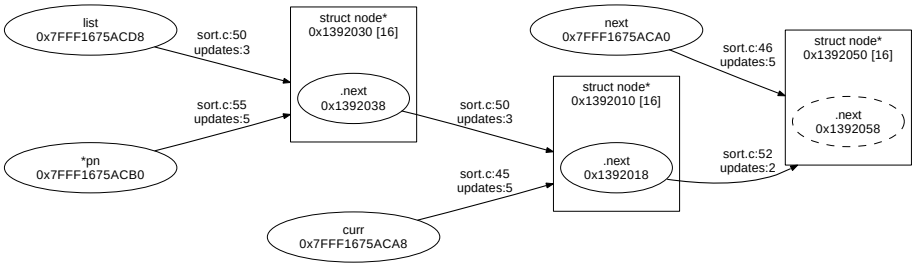

Fig. 10 Visualization of the heap during a bubble-sort operation on a linked list. Boxes represent heap-allocated `struct`s: linked list nodes in this example. Each `struct` is labeled with is size, its address in memory, and the addresses of its field. Within a `struct`, ovals represent fields that point to other heap objects. Ovals that are not in a `struct` are global and stack variables. Each field and variable has an outgoing edge to the `struct` that it points to, which is labeled with 1) the line number of the assignment that created the edge and 2) the number of assignments to the source variable that have occurred so far. Fields and variables that do not point to valid memory (such as a `NULL` pointer) have dashed borders.

4.1 Heap Visualization

The heap visualizer uses the INTERASPECT API to expose memory events that can be used to generate a graphical representation of the heap in real time during program execution. Allocated objects are represented by rectangular nodes, pointer variables and fields by oval nodes, and edges show where pointer variables and fields point.

In order to draw the graph, the heap visualizer needs to intercept object allocations and deallocations and pointer assignments that change edges in the graph. Figure 10 shows a prototype of the visualizer using Graphviz [4], an open-source graph layout tool, to draw its output. The graph shows three nodes in a linked list during a bubble-sort operation. The `list` variable is the list's head pointer, and the `curr` and `next` variables are used to traverse the list during each pass of the sorting algorithm. (The `pn` variable is used as temporary storage for swap operations.)

The INTERASPECT code for the heap visualizer instruments each allocation (call to `malloc`) with a call to the `heap_allocation` advice function, and it instruments each pointer assignment with a call to the `pointer_assign` advice function. These advice functions update the graph. Instrumentation of other allocation and deallocation functions, such as `calloc` and `free`, is handled similarly.

The INTERASPECT code in Figure 11 instruments calls to `malloc`. The API function `instrument_malloc_calls` constructs a pointcut for all calls to `malloc` and then calls `aop_join_on` to iterate over all the calls in the pointcut. Only a short main function (not shown) is needed to set GCC to invoke `instrument_malloc_calls` during compilation.

The `aop_match_function_call` function constructs an initial pointcut that includes every function call. The `filter` functions narrows the pointcut to include only calls to `malloc`. First, `aop_filter_call_pc_by_name` filters out calls to functions that are not named `malloc`. Then, `aop_filter_pc_by_param_type` and `aop_filter_pc_by_return_type` filter out calls to functions that do not match the standard `malloc` prototype, which takes an unsigned integer as the first parameter and returns a pointer value. This filtering step is necessary because a program could define its own function with the name `malloc` but a different prototype.

For each join point in the pointcut (in this case, a call to `malloc`), `aop_join_on` calls `malloc_callback`. The two `capture` calls in the callback function return `aop_dynval` ob-

```
static void instrument_malloc_calls(void)
{
  /* Construct a pointcut that matches calls to: void *malloc(unsigned int). */
  struct aop_pointcut *pc = aop_match_function_call();
  aop_filter_call_pc_by_name(pc, "malloc");
  aop_filter_call_pc_by_param_type(pc, 0, aop_t_all_unsigned());
  aop_filter_call_pc_by_return_type(pc, aop_t_all_pointer());

  /* Visit every statement in the pointcut. */
  aop_join_on(pc, malloc_callback, NULL);
}

/* The malloc_callback() function executes once for each call to malloc() in the
   target program.  It instruments each call it sees with a call to
   heap_allocation(). */
static void malloc_callback(struct aop_joinpoint *jp, void *arg)
{
  struct aop_dynval *object_size;
  struct aop_dynval *object_addr;

  /* Capture the size of the allocated object and the address it is
     allocated to. */
  object_size = aop_capture_param(jp, 0);
  object_addr = aop_capture_return_value(jp);

  /* Add a call to the advice function, passing the size and address as
     parameters.  (AOP_TERM_ARG is necessary to terminate the list of arguments
     because of the way C varargs functions work.) */
  aop_insert_advice(jp, "heap_allocation", AOP_INSERT_AFTER,
                    AOP_DYNVAL(object_size), AOP_DYNVAL(object_addr),
                    AOP_TERM_ARG);
}
```

Fig. 11 Instrumenting all memory-allocation events.

jects for the call's first parameter and return value: the size of the allocated region and its address, respectively. Recall from Section 3 that an `aop_dynval` serves as a placeholder during compilation for a value that will not be known until runtime. Finally, `aop_insert_advice` adds the call to the advice function, passing the two captured values. Note that INTERASPECT chooses types for these values based on how they were filtered. The filters used here restrict `object_size` to be an unsigned integer and `object_addr` to be some kind of pointer, so INTERASPECT assumes that the advice function `heap_allocation` has the prototype:

```
void heap_allocation(unsigned long long, void *);
```

To support this, INTERASPECT code must generally filter runtime values by type in order to capture and use them.

The INTERASPECT code in Figure 12 tracks pointer assignments, such as

```
list_node->next = new_node;
```

The `aop_match_assignment_by_type` function creates a pointcut that matches assignments, which is additionally filtered by the type of assignment. For this application, we are only interested in assignments to pointer variables.

For each assignment join point, `assignment_callback` captures `address`, the address assigned to, and `pointer`, the pointer value that was assigned. In the above examples, these would be the values of `&list_node->next` and `new_node`, respectively. The visualizer uses `address` to determine the source of a new graph edge and `pointer` to determine its destination.

The function that captures `address`, `aop_capture_lhs_addr`, does not require explicit filtering to restrict the type of the captured value because an address always has a pointer

```
static void instrument_pointer_assignments(void)
{
  /* Construct a pointcut that matches all assignments to a pointer. */
  struct aop_pointcut *pc = aop_match_assignment_by_type(aop_t_all_pointer());

  /* Visit every statement in the pointcut. */
  aop_join_on(pc, assignment_callback, NULL);
}

/* The assignment_callback function executes once for each pointer assignment.
   It instruments each assignment it sees with a call to pointer_assign(). */
static void assignment_callback(struct aop_joinpoint *jp, void *arg)
{
  struct aop_dynval *address;
  struct aop_dynval *pointer;

  /* Capture the address the pointer is assigned to, as well as the pointer
     address itself. */
  address = aop_capture_lhs_addr(jp);
  pointer = aop_capture_assigned_value(jp);

  aop_insert_advice(jp, "pointer_assign", AOP_INSERT_AFTER,
                    AOP_DYNVAL(address), AOP_DYNVAL(pointer),
                    AOP_TERM_ARG);
}
```

Fig. 12 Instrumenting all pointer assignments.

type. The value captured by `aop_capture_assigned_value` and stored in `pointer` has a void pointer type because we filtered the pointcut to include only pointer assignments. As a result, INTERASPECT assumes that the `pointer_assign` advice function has the prototype:

```
void pointer_assign(void *, void *);
```

4.2 Integer Range Analysis

Integer range analysis is a runtime tool for finding anomalies in program behavior by tracking the range of values for each integer variable [15]. A range analyzer can learn normal ranges from training runs over known good inputs. Values that fall outside of normal ranges in future runs are reported as anomalies, which can indicate errors. For example, an out-of-range value for a variable used as an array index may cause an array-bounds violation.

Our integer range analyzer uses sampling to reduce runtime overhead. Missed updates because of sampling can result in underestimating a variable's range, but this trade-off is reasonable in many cases. Sampling can be done randomly or by using a technique like Software Monitoring with Controllable Overhead [17].

INTERASPECT provides function body duplication as a means to add instrumentation that can be toggled on and off. Duplicating a function splits its body into two copies. A *distributor block* at the beginning of the function decides which copy to run. An INTERASPECT plug-in can add advice to just one of the copies, so that the distributor chooses between enabling or disabling instrumentation.

Figure 13 shows how we use INTERASPECT to instrument integer variable updates. The call to `aop_duplicate` makes a copy of each function body. The first argument specifies that there should be two copies of the function body, and the second specifies the name of a function that the distributor will call to decide which copy to execute. When the duplicated function runs, the distributor calls `distributor_func`, which must be a function that returns an integer. The duplicated function bodies are indexed from zero, and the `distributor_func` return value determines which one the distributor transfers control to.

```
static void instrument_integer_assignments(void)
{
  struct aop_pointcut *pc;

  /* Duplicate the function body so there are two copies. */
  aop_duplicate(2, "distributor_func", AOP_TERM_ARG);

  /* Construct a pointcut that matches all assignments to an integer. */
  pc = aop_match_assignment_by_type(aop_t_all_signed_integer());

  /* Visit every statement in the pointcut. */
  aop_join_on_copy(pc, 1, assignment_callback, NULL);
}

/* The assignment_callback function executes once for each integer assignment.
   It instruments each assignment it sees with a call to int_assign(). */
static void assignment_callback(struct aop_joinpoint *jp, void *arg)
{
  const char *variable_name;
  int variable_index;
  struct aop_dynval *value;
  enum aop_scope scope;

  variable_name = aop_capture_lhs_name(jp);

  if (variable_name != NULL) {
    /* Choose an index number for this variable. */
    scope = aop_capture_lhs_var_scope(jp);
    variable_index = get_index_from_name(variable_name, scope);

    aop_insert_advice(jp, "int_assign", AOP_INSERT_AFTER,
                      AOP_INT_CST(variable_index), AOP_DYNVAL(value),
                      AOP_TERM_ARG);
  }
}
```

Fig. 13 Instrumenting integer variable updates.

Using `aop_join_on_copy` instead of the usual `aop_join_on` iterates only over join points in the specified copy of duplicate code. As a result, only one copy is instrumented; the other copy remains unmodified.

The callback function itself is similar to the callbacks we used in Section 4.1. The main difference is the call to `get_index_from_name` that converts the variable name to an integer index. The `get_index_from_name` function (not shown for brevity) also takes the variable's scope so that it can assign different indices to local variables in different functions. It would be possible to directly pass the name itself (as a string) to the advice function, but the advice function would then incur the cost of looking up the variable by its name at runtime. This optimization illustrates the benefits of INTERASPECT's callback-based approach to custom instrumentation.

The `aop_capture_lhs_name` function returns a string instead of an `aop_dynval` object because variable names are known at compile time. It is necessary to check for a `NULL` return value because not all assignments are to named variables.

To better understand InterAspect's performance impact, we benchmarked this plug-in on the compute-intensive `bzip2` compression utility using trivial advice functions. The `bzip2` package is a popular tool included in most Linux distributions. It has 110 functions in about 8,000 lines of code. Our test plug-in, based on the code in Figure 13, duplicates each function body, adding an advice call to every integer assignment in one copy of the function body. Depending on the test, the distributor either returns 0 immediately, choosing the uninstrumented path, or returns 1 immediately, for the instrumented path. The integer assignment advice function only increments a counter, allowing us to measure the overhead from call-

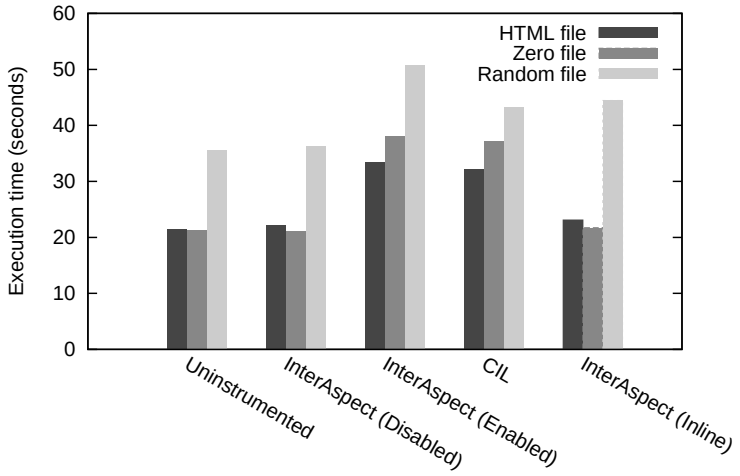

Fig. 14 Execution time for `bzip2` instrumented, using INTERASPECT or CIL, to increment a counter at every integer assignment. The programs for the three INTERASPECT configurations are instrumented with the same plug-in, which duplicates function bodies and inserts advice at every integer assignment. In the "enabled" and "inline" runs, the distributor always chooses the instrumented path; in the "disabled" run, it always chooses the uninstrumented path. For the "inline" run, the advice function was marked as inline, allowing GCC to inline it. We ran all performance tests 10 times, and the 90% confidence interval had a half width of less than 0.15 seconds for all measurements shown.

ing advice functions independently from actual monitoring overhead. All in all, the plug-in instrumented 957 assignment join points. We also compared our INTERASPECT plug-in to a similar transformation written in CIL [24] that adds an advice call to every integer assignment but does not perform function body duplication.

Figure 14 shows our results for `bzip2` with five different instrumentation configurations. We benchmarked each of these configurations with three different input files: a 161MB HTML file, a 161MB file containing random bytes, and a 1.6GB file containing zeros. The HTML file consists of a novel taken from the Project Gutenberg web site and duplicated to create a larger file.

With a distributor that maximizes overhead by always choosing the instrumented function body ("InterAspect (Enabled)"), we measured 78.7% runtime overhead in the worst case: the zero file. Function body duplication by itself contributes relatively little to this overhead; with a distributor that always chooses the uninstrumented path ("InterAspect (Disabled)"), we measured only 3.00% overhead in the worst case: the HTML file.

High overhead is expected for the integer assignment pointcut because `bzip2` performs integer assignments very frequently. To compress the HTML file, `bzip2` executed 5.34 billion join points, more than 249 million integer assignments per second. The CIL integer assignment transformation we tested ("CIL" in Figure 14) added 74.0% overhead when compressing the zero file.

Much of the overhead in our test comes from the time it takes to enter and exit advice functions. When an advice function includes only a small amount of code, as in this example, it makes sense to insert that code directly at each join point to avoid function call overhead. GCC can automatically perform this transformation, because INTERASPECT's instrumentation passes occur before GCC's function inlining pass. Marking the integer assignment

advice function with GCC's `always_inline` attribute reduced overhead to just 25.0% in the worst case: compressing the random file. We computed an overall time for each configuration by summing the average times for each of the three files. We then computed an overall overhead for each instrumented configuration. The overall overhead for the "InterAspect (Inline)" configuration was 13.4%, the lowest of all the configurations we tested.

The only memory overhead from our range analysis tool comes from duplicating every function body, which roughly doubles the size of the text segment. The instrumented `bzip2` execution image was 69KB larger, as reported by the `size` utility, an increase of 96%. Image size is small, however, compared to the total 7.2MB of heap allocations when compressing the HTML file with or without instrumentation, as reported by Valgrind.

We disabled GCC's function inlining for all configurations, because it interfered with comparisons between `bzip2` configurations with function body duplication and configurations without it. When compiling without instrumentation, GCC inlined too aggresively, actually hurting performance. But with function body duplication, GCC was more reluctant to inline functions that were now twice as large, making it appear as if function body duplication *improved* performance and unfairly masking some of the overhead in our benchmarks. Note that disabling function inlining did not prevent GCC from inlining the advice function in the "InterAspect (Inline)" configuration, because we marked the advice function with the `always_inline` attribute.

4.3 Code Coverage

A straightforward way to measure code coverage is to choose a pointcut and measure the percentage of its join points that are executed during testing. INTERASPECT's ability to iterate over each join point makes it simple to label join points and then track them at runtime.

```
static void instrument_function_entry_exit(void)
{
  struct aop_pointcut *entry_pc;
  struct aop_pointcut *exit_pc;

  /* Construct two pointcuts: one for function entry and one for function exit. */
  entry_pc = aop_match_function_entry();
  exit_pc = aop_match_function_exit();

  aop_join_on(entry_pc, entry_exit_callback, NULL);
  aop_join_on(exit_pc, entry_exit_callback, NULL);
}

/* The entry_exit_callback function assigns an index to every join
   point it sees and saves that index to disk. */
static void entry_exit_callback(struct aop_joinpoint *jp, void *arg)
{
  int index, line_number;
  const char *filename;

  index = choose_unique_index();
  filename = aop_capture_filename(jp);
  line_number = aop_capture_lineno(jp);

  save_index_to_disk(index, filename, line_number);

  aop_insert_advice(jp, "mark_as_covered", AOP_INSERT_BEFORE,
                    AOP_INT_CST(index), AOP_TERM_ARG);
}
```

Fig. 15 Instrumenting function entry and exit for code coverage.

The example in Figure 15 adds instrumentation to track coverage of function entry and exit points. To reduce runtime overhead, the `choose_unique_index` function assigns an integer index to each tracked join point, similar to the indexing of integer variables in Section 4.2. Each index is saved along with its corresponding source filename and line number by the `save_index_to_disk` function. The runtime advice needs to output only the set of covered index numbers; an offline tool uses that output to compute the percentage of join points covered or to list the filenames and line numbers of covered join points. For brevity we omit the actual implementations of `choose_unique_index` and `save_index_to_disk`.

5 Tracecuts

In this section, we present the API for the INTERASPECT Tracecut extension, and discuss the implementation of the associated tracecut monitoring engine. We also present two illustrative examples of the Tracecut extension: runtime verification of file access and GCC vectors. The architecture diagram in Figure 3 shows how this extension and its associated monitoring engine fit into the overall INTERASPECT architecture.

Our INTERASPECT Tracecut extension showcases the flexibility of INTERASPECT's API. Since one of our goals for this extension is to serve as a more powerful example of how to use INTERASPECT, its instrumentation component is built modularly on INTERASPECT: all of its access to GCC are through the published INTERASPECT interface.

Whereas pointcut advice is triggered by individual events, tracecut advice can be triggered by sequences of events matching a pattern [32]. A tracecut in our system is defined by a set symbols, each representing a possibly parameterized runtime event, and one or more rules expressed as regular expressions over these symbols. For example, a tracecut that matches a call to `exit` or `execve` after a fork would specify symbols for `fork`, `exit`, and `execve` function calls and the rule `fork` (`exit` | `execve`), where juxtaposition denotes sequencing, parentheses are used for grouping, and the vertical bar "|" separates alternatives.

Each symbol is translated to a function-call pointcut, which is instrumented with advice that sends the symbol's corresponding event to the monitoring engine. The monitoring engine signals a match whenever some suffix of the string of events matches one of the regular-expression rules.

Parameterization allows a tracecut to separately monitor multiple objects [2,8]. For example, the rule `fclose fread`, designed to catch an illegal read from a closed file, should not match an `fclose` followed by an `fread` to a different file. When these events are parameterized by the file they operate on, the monitoring engine creates a unique monitor instance for each file.

A tracecut with multiple parameters can monitor properties on sets of objects. A classic example monitors data sources that have multiple iterators associated with them. When a data source is updated, its existing iterators become invalid, and any future access to them is an error. Parameterizing events by both data source and iterator creates a monitor instance for each pair of data source and iterator.

The monitoring engine is implemented as a runtime library that creates monitor instances and forwards events to their matching monitor instances. Because rules are specified as regular expressions, each monitor instance stores a state in the equivalent finite-state machine. The user only has to link the monitoring library with the instrumented binary, and the tracecut instrumentation calls directly into the library.

```
struct tc_tracecut *tc_create_tracecut(void);
```
Create an empty tracecut.
```
enum tc_error tc_add_param(struct tc_tracecut *tc, const char *name,
                           const struct aop_type *type);
```
Add a named parameter to a tracecut.

Fig. 16 Function for initializing tracecuts.

```
enum tc_error tc_add_call_symbol(struct tc_tracecut *tc, const char *name,
                                 const char *func_name,
                                 enum aop_insert_location location);
```
Define a named event corresponding to calls to the function named by `func_name`.
```
enum tc_error tc_bind_to_call_param(struct tc_tracecut* tc,
                                    const char *param_name,
                                    const char *symbol_name, int call_param_index);
```
Bind a function call parameter from an event to one of the tracecut's named parameters.
```
enum tc_error tc_bind_to_return_value(struct tc_tracecut *tc,
                                      const char *param_name,
                                      const char *symbol_name);
```
Bind the return value of an event to one of the tracecut's named parameters.
```
enum tc_error tc_declare_call_symbol(struct tc_tracecut *tc, const char *name,
                                     const char *declaration,
                                     enum aop_insert_location location);
```
Define a named event along with all its parameter bindings with one declaration string.

Fig. 17 Functions for specifying symbols.

5.1 Tracecut API

A tracecut is specified by a C program that calls tracecut API functions. This design keeps the tracecut extension simple, eliminating the need for a custom parser but still allowing concise definitions. A tracecut specification can define any number of tracecuts, each with its own parameters, events, and rules.

Defining Parameters. The functions in Figure 16 create a new tracecut and define its parameters. Each parameter has a name and a type. The type is necessary because parameters are used to capture runtime values.

Defining Symbols. The `tc_add_call_symbol` function adds a new symbol that corresponds to an event at every call to a specified function. The `tc_bind` functions bind a tracecut parameter to one of the function call's parameters or to its return value. Figure 17 shows `tc_add_call_symbol` and the `tc_bind` functions.

The tracecut API uses the symbol and its bindings to define a pointcut. Figure 18 shows an example symbol along with the INTERASPECT API calls that Tracecut makes to create the pointcut. In a later step, Tracecut makes calls needed to capture the bound return value and pass it to an advice function.

As a convenience, the API also provides the `tc_declare_call_symbol` function (also in Figure 17), which can define a symbol and its parameter bindings with one call using a simple text declaration. The declaration is syntactically similar to the C prototype for the function that will trigger the symbol, but the function's formal parameters are replaced with tracecut parameter names or with a question mark "`?`" to indicate that a parameter should remain unbound. The code in Figure 18(c) defines the same symbol as in Figure 18(a).

```
struct tracecut *tc = tc_create_tracecut()
tc_add_param(tc, "object", aop_all_pointer());
tc_add_call_symbol(tc, "create", "create_object", AOP_INSERT_AFTER);
tc_bind_to_return_value(tc, "object", "create");
```

(a) Code to define a tracecut symbol.

```
pc = aop_match_function_call();
aop_filter_call_pc_by_name(pc, "create_object");
aop_filter_call_pc_by_return_type(pc, aop_all_pointer());
```

(b) The values that the tracecut API will pass to INTERASPECT functions to create a corresponding pointcut.

```
struct tracecut *tc = tc_create_tracecut()
tc_add_param(tc, "object", aop_all_pointer());
tc_declare_call_symbol(tc, "create", "(object)create_object()",
                       AOP_INSERT_AFTER);
```

(c) A more compact way to define the event in Figure 18(a).

Fig. 18 An example of how the tracecut API translates a tracecut symbol into a pointcut. Because the `create` symbol's return value is bound to the `object` param, the resulting pointcut is filtered to ensure that its return value matches the type of `object`.

```
enum tc_error tc_add_rule(struct tc_tracecut *tc, const char *specification);
```

Define a tracecut rule. The specification is a regular expression using symbol names as its alphabet.

Fig. 19 Function for defining tracecut rule.

Defining Rules. After symbols and their parameter bindings are defined, rules are expressed as strings containing symbol names and standard regular expression operators: `(`, `)`, `*`, `+`, and |. The function for adding a rule to a tracecut is shown in Figure 19.

5.2 Monitor Implementation

The monitoring engine maintains a list of monitor instances for each tracecut. Each instance has a value for each tracecut parameter and a monitor state. Instrumented events pass the values of their parameters to the monitoring engine, which then determines which monitor instances to update. This monitor design is based on the way properties are monitored in Tracematches [2] and MOP [8].

When a symbol is fully parameterized—it has a binding for every parameter defined in the tracecut specification—the monitoring engine updates exactly one instance. If no instance exists with matching parameter values, one is created.

For partially parameterized symbols, like `push` in Figure 23, the monitoring engine only requires the specified parameters to match. As a result, events corresponding to these symbols can update multiple monitor instances. For example, a `push` event updates one monitor for every `element_pointer` associated with the updated vector. As in the original MOP implementation, partially parameterized symbols cannot create a new monitor instance [8]. (MOP has since defined semantics for partially parameterized monitors [22].)

When any monitor instance reaches an accepting state, the monitoring engine reports a match. The default match function prints the monitor parameters to `stderr`. Developers can implement their own tracecut advice by overriding the default match function. Function overriding is possible in C using a linker feature called *weak linkage*. Placing a debugger breakpoint at the match function makes it possible to examine program state when a match occurs.

Monitoring instances get destroyed when they can no long reach an accepting state. The tracecut engine does not attempt to free instances parameterized by freed objects because it is not always possible to learn when an object is freed in C and because parameters are not required to be pointers to heap-allocated objects.

A developer can ensure that stale monitor instances do not waste memory by designing the rule to discard them. The easiest way to do this is to define a symbol for the function that deallocates an object but not to include the symbol anywhere in the tracecut's rule. Deallocating the object then generates an event that makes it impossible for the tracecut rules to match.

Figure 20 is a pseudocode representation of the monitoring logic described in this section. Note that it uses a linear search to find monitors that need to be updated. This approach makes sense when the number of monitor instances remains small throughout execution, as in the examples we discuss below. When the number of monitor instances is large, it would be more efficient to maintain a hash table index for each possible parameterization. Updating these indexes would add a constant cost to creating a new monitor instance, but events would no longer trigger an expensive $O(n)$ lookup.

```
receive_event(tracecut, monitors, event_name, param_names[], param_values[],
              num_params):
  matching_monitors := {}

  ; Find monitor instances with parameters matching this event.
  for each monitor in monitors:
    matches := true
    for i in 1 to num_params:
      ; Check that all params in the event match params in the monitor instance.
      if not monitor.get_param(param_names[i]) = param_values[i] then:
        matches := false
        break

    if matches then:
      matching_monitors.insert(monitor)

  ; Create a new monitor if necessary.
  if is_empty(matching_monitors) and is_fully_parameterized(tracecut, num_params):
    new_monitor := create_monitor(param_names, param_values)
    monitors.insert(new_monitor)
    matching_monitors.insert(new_monitor)

  ; Update the finite-state machine for each matching monitor.
  for each monitor in matching_monitors:
    monitor.update_state(event_name)

    ; Trigger advice on reaching an accepting state.
    if is_in_accepting_state(monitor)
      monitor.call_advice_function()

    ; Destroy any monitor that can no longer reach an accepting state.
    if is_in_trap_state(monitor) then:
      monitors.remove(monitor)
      destroy(monitor)

is_fully_parameterized(tracecut, num_params)
  if get_num_params(tracecut) = num_params then:
    return true
  else
    return false
```

Fig. 20 Pseudocode implementation of the INTERASPECT Tracecut monitoring logic.

5.3 Verifying File Access

As a first example of the tracecut API, we consider the runtime verification of file access. Like most resources in C, the `FILE` objects used for file I/O must be managed manually. Any access to a `FILE` object after the file has been closed is a memory error which, though dangerous, might not manifest itself as incorrect behavior during testing. Designing a tracecut to detect these errors is straightforward.

```
tc = tc_create_tracecut();

tc_add_param(tc, "file", aop_t_all_pointer());

tc_declare_call_symbol(tc, "open", "(file)fopen()", AOP_INSERT_AFTER);
tc_declare_call_symbol(tc, "read", "fread(?, ?, ?, file)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "read_char", "fgetc(file)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "close", "fclose(file)", AOP_INSERT_BEFORE);

tc_add_rule(tc, "open (read | read_char)* close (read | read_char)");
```

Fig. 21 A tracecut for catching accesses to closed files. For brevity, the tracecut only checks read operations.

The tracecut in Figure 21 defines symbols for four `FILE` operations: open, close, and two kinds of reads. The rule matches any sequence of these symbols that opens a file, closes it, and then tries to read it.

The rule matches as soon as any read is performed on a closed `FILE` object, immediately identifying the offending read. We tested this tracecut on `bzip2` (which we also use for evaluation in Section 4.2); it caught an error we planted without reporting any false positives.

5.4 Verifying GCC Vectors

We designed a tracecut to monitor a property on a vector data structure used within GCC to store an ordered list of GIMPLE statements. The list is stored in a dynamically resized array. The vector API provides an iterator function to iterate over the GIMPLE statements in a vector. Figure 22 shows how the iterator function is used. At each execution of the loop, the `element` variable points to the next statement in the vector.

```
int i;
gimple element;

/* Iterate over each element in a vector of GIMPLE statements. */
for (i = 0; VEC_gimple_base_iterate(vector1, i, &element); i++) {
  /* If condition holds, copy this element into vector2. */
  if (condition(element))
    VEC_gimple_base_quick_push(vector2, element);
}
```

Fig. 22 The standard pattern for iterating over the elements in a GCC vector of GIMPLE statements. This example copies elements matching some condition from `vector1` to `vector2`. If `vector1` and `vector2` happen to point to the same vector, this code may modify that vector while iterating over its elements.

```
tc = tc_create_tracecut();

tc_add_param(tc, "vector", aop_t_all_pointer ());
tc_add_param(tc, "element_pointer", aop_t_all_pointer ());

tc_declare_call_symbol(tc, "iterate",
                       "VEC_gimple_base_iterate(vector, ?, element_pointer)",
                       AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "push", "VEC_gimple_base_quick_push(vector, ?)",
                       AOP_INSERT_BEFORE);

tc_add_rule(tc, "iterate push iterate");
```

Fig. 23 A tracecut to monitor vectors of GIMPLE objects in GCC.

A common tracecut property for data structures with iterators checks that the data structure is not modified while it is being iterated, as can occur in Figure 22. Figure 23 specifies a tracecut that detects violations of this property.

The tracecut monitors two important vector operations: the `VEC_gimple_base_iterate` function, which is used in the guard of a for loop to advance to the next element in the list, and the `VEC_gimple_base_quick_push` function, which inserts a new element at the end of a vector. With the symbols defined, the rule itself is simple: `iterate push iterate`. Any `push` in between two `iterate` operations indicates that the vector was updated within the iterator loop.

Parameterizing the `iterate` symbol on both the vector and the `element_pointer` used to iterate makes it possible to distinguish different iterator loops over the same vector. This distinction is necessary so that a program that finishes iterating over a vector, updates that vector, and then iterates over it again does not trigger a match. Though, the tracecut monitor will observe events for the symbols `iterate push iterate`, the first and last `iterate` events (which are from different loops) will normally have different values for their `element_pointer` parameter.

When monitoring this same property in Java, usually an *iterator object* serves the purpose of parameterizing an iterator loop. In Figure 22, the `element` variable is analogous to an iterator, as it provides access to the current list element at each iteration of the loop. The `element_pointer` identifies the iterator-like variable by its address.

Keeping specifications simple is especially important in C because the language does not provide any standard data structures. A tracecut written for one program's vector type is not likely to be useful for monitoring any other program.

We applied the tracecut in Figure 23 to GCC itself, verifying that, in our tests, GCC did not update any vectors while they were being iterated. The tracecut did match a call to `VEC_gimple_base_quick_push` that we deliberately placed in an iterator loop.

Because monitored events in our INTERASPECT Tracecut examples execute less frequently than the integer assignment join points in our integer range analysis example (Section 4.2), we found overhead to be less of an issue. We measured overhead for both the file access tracecut we tested in Section 5.3 and the tracecut in this section to be less than 1%.

5.5 Verifying lighttpd Connections

We also used INTERASPECT Tracecut to check a property of connections in the lighttpd (pronounced *lighty*) HTTP server [20]. Lighttpd creates many connections, allowing us

to evaluate the performance of INTERASPECT Tracecut with many monitors. The lighttpd server maintains a `connection` object for each open connection from a client. Each `connection` object stores a TCP network socket and all state information for the client's HTTP session.

```
tc = tc_create_tracecut();

tc_add_param(tc, "connection", aop_t_all_pointer());

tc_declare_call_symbol(tc, "init", "(connection)connections_get_new_connection()",
                       AOP_INSERT_AFTER);
tc_declare_call_symbol(tc, "state", "connection_state_machine(?, connection)",
                       AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "close", "connection_close(?, connection)",
                       AOP_INSERT_BEFORE);

tc_add_rule(tc, "init state* close state");
```

Fig. 24 A tracecut for catching accesses to connections after they have been closed by the server.

The property we checked, shown in Figure 24, is that the server does not try to update the state of a connection after the connection has been closed; this is similar to the file property presented in Section 5.3. After `connection_close` is called on a `connection` object, any updates to that object via `connection_state_machine` will trigger a match, unless a call to `connections_get_new_connection` re-initializes the object first.

We found that lighttpd sometimes closes connections while they are on its list of `connection` objects that are pending a state update. The service routine for this list then updates the state of the closed connection, but this usage does not cause an error. To avoid error reports for this correct usage, we overrode the tracecut match function to ignore matches on objects in the pending connection list. The custom match function still reports other state updates on closed connections. These state updates would likely indicate an error. Our test runs did not find any such updates in the version of lighttpd we tested.

To test performance, we stressed lighttpd with the `http_load` tool, which loads an HTTP server with a large number of parallel requests and measures response times [26]. The version we used includes a patch from the lighttpd authors [20] to report errors more accurately and additional modifications to report standard deviations of response time samples, which we needed to make conclusions about statistical significance. We configured `http_load` to open HTTP requests in groups of 100 at a time, the most that lighttpd could handle on our test hardware without dropping connections. With this test workload, INTERASPECT Tracecut had to maintain at least 100 monitor instances throughout the course of the test.

Monitoring did not cause a statistically significant increase in response time, because lighttpd's operation is largely I/O-bound. The average response time was 21.9ms for the two million requests in the unmonitored and monitored runs. Monitoring did increase the server's CPU load. Running lighttpd with the connection tracecut raised CPU utilization by the lighttpd process from 36.4% to 39.7%.

5.6 Verifying Hash Table Entries

We designed a simple hash table benchmark in order to better quantify INTERASPECT Tracecut's scalability, as well as to provide another example of a useful data structure property. The benchmark performs 10M operations, either randomly inserting an element into

one of the hash tables or, with much lower probability, randomly modifying the key of an element in one of the hash tables.

Modifying an element simulates an error. Altering the element's key can change its hash value, leaving the element in the wrong bucket and violating the hash table's invariant. Figure 25 shows a tracecut designed to catch this error by matching any call to `modify_obj` that immediately follows `insert_obj`. The `htab_empty` function removes all elements from a table, and the tracecut expression is designed so that a call to `htab_empty` after inserting an object with `insert_obj` prevents a match for a subsequent call to `modify_obj` (thereby destroying the object's monitor instance). Though it was not necessary for this example, it would also be straightforward to include a symbol for a function that removes an individual object from a hash table.

```
tc = tc_create_tracecut();

tc_add_param(tc, "table", aop_t_struct_ptr("htab"));
tc_add_param(tc, "obj", aop_t_struct_ptr("obj"));

tc_declare_call_symbol(tc, "insert", "insert_obj(table, obj)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "modify", "modify_obj(obj)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "empty", "htab_empty(table)", AOP_INSERT_BEFORE);

tc_add_rule(tc, "insert modify");
```

Fig. 25 A tracecut for catching modifications to objects in hash tables.

We designed the hash table benchmark so that the tracecut monitoring framework would have to store a large number of monitor instances. The benchmark maintains 10 hash tables, which each have a maximum size. Whenever a table exceeds its maximum, the benchmark removes all its elements with the `htab_empty` function. We varied the maximum table size from 50 to 250 in increments of 25 to show how performance scales as the number of monitored objects increases.

Because the list of monitor instances is much larger than in our other benchmarks and because the hash table benchmark executes monitored operations in a tight loop, we expected the performance cost of monitoring to be high. With the maximum size set to 250, we measured 113× overhead and found that each INTERASPECT Tracecut had to search a list of 1,244 monitor instances on average for each event it monitored. Figure 26 shows these results, overhead and average number of monitor instances, for each of the maximum table sizes we tested.

In a profiled run of the benchmark, we found that the tracecut library spent more than 98% of its time in monitor instance lookup routines. The trend in Figure 26 of overhead increasing linearly with the number of monitor instances is consistent with our conclusion that these lookup routines dominate monitoring overhead. As mentioned in Section 5.2, INTERASPECT Tracecut's overhead for target programs that involve a large number of monitor instances can be greatly reduced by using an index (e.g., a hash table), instead of linear search, to find monitor instances that need to be updated. This more efficient approach is used in the MOP system [8], discussed in Section 6.

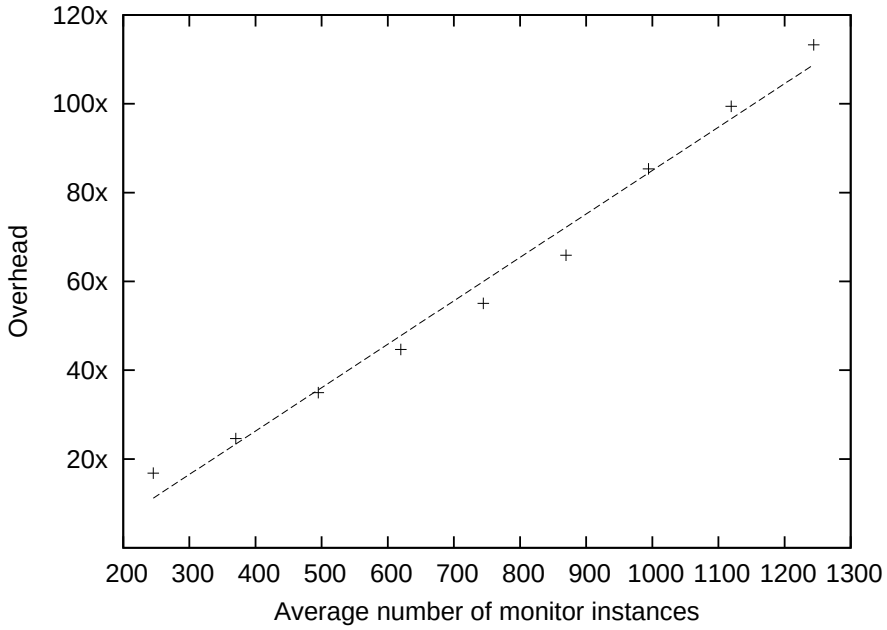

Fig. 26 Monitoring overhead for the hash table benchmark with nine different values for the maximum table size. We calculate the number of monitor instances, shown on the x-axis, as the average number of instances that exist when a tracecut event is monitored. As with our `bzip2` benchmark in Section 4.2, we obtained performance numbers by comparing the average execution time of the benchmark with and without monitoring, using ten runs for each.

6 Related Work

INTERASPECT is a framework for the aspect-oriented instrumentation of programming languages supported by GCC. Whereas the current focus has been on C, the framework should be applicable to any GCC-supported language. INTERASPECT has been extended in this paper with the Tracecut plug-in for the runtime monitoring of regular expressions. INTERASPECT Tracecut illustrates how INTERASPECT allows for such an extension. In what follows, we discuss related work in terms of instrumentation frameworks and tracecut facilities.

Concerning instrumentation frameworks, there is a great variety of them for popular programming languages, including aspect-oriented programming environments, reflection systems, and compiler frameworks. Instrumentation frameworks can be classified along four dimensions:

1. *Target language*: the language being instrumented (e.g. C, C++, and Java).
2. *Instruction language*: the language used to express instrumentation instructions. The instrumentation language can be a Domain-Specific Language (DSL) or an API.
3. *Target view*: the type of view offered by the instrumentation framework of the target language: source-code view, bytecode view, abstract syntax, etc.
4. *Infrastructure*: the prevalence of the compiler framework that the instrumentation framework is based on.

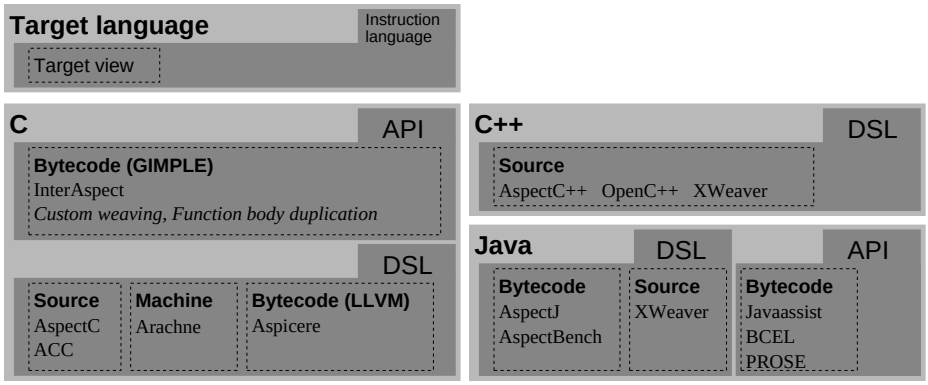

Fig. 27 A breakdown of the frameworks discussed in this section by the first three of our four classification dimensions, *target language*, *instruction language*, and *target view*. Not included in this hierarchy are CIL and Valgrind, which both provide general-purpose frameworks that are not aspect oriented.

We argue that INTERASPECT offers a unique combination of these dimensions with the target language being C (and other languages supported by GCC), the instrumentation language being an API, the target view being source code, and finally being based on the well-adopted GCC infrastructure. It is this combination of features that makes INTERASPECT unique. Being based on GCC means that INTERASPECT has a greater chance of adoption by the GCC user community and of long-term survival, because, if an instrumentation framework is part of a compiler you are already using, the barrier for usage of that instrumentation framework is significantly diminished. Being API-based means that it is flexible and permits open-source collaboration. Furthermore, the focus on C is much needed since the focus of existing instrumentation frameworks has been primarily on Java. Figure 27 shows all of the aspect-oriented frameworks that we compare INTERASPECT to in this section and organizes them according the dimensions introduced above.

In addition to filling a new role in the spectrum of instrumentation frameworks, INTERASPECT offers two novel features to the field of aspect-oriented programming. First, INTERASPECT supports the notion of callback functions, which can be applied during instrumentation. Such functions can perform customized instrumentation at each join point, a capability other AOP approaches lack. Second, function body duplication makes it possible to efficiently toggle instrumentation on and off at runtime or to switch between two different instrumentation profiles.

Aspect-oriented programming was first popularized for Java with AspectJ [13, 21]. There, weaving takes place at the bytecode level. The user is provided with a source-code view and writes instrumentation instructions in a specialized DSL supporting pointcut definitions and advice definitions. The AspectBench Compiler (abc) [5] is a more recent extensible research version of AspectJ that makes it possible to add new language constructs [6]. Similarly to INTERASPECT, it manipulates a 3A intermediate representation (Jimple) specialized to Java.

Other frameworks for Java, including Javaassist [10] and PROSE [25], offer, in a manner similar to INTERASPECT, an API for instrumenting and modifying code, and hence do not require the use of a special language. Javaassist is a class library for editing bytecode. A source-level API can be used to edit class files without knowledge of the bytecode format. PROSE has similar goals. The BCEL [3] tool provides an API for manipulating Java bytecode.

AOP for other languages such as C and C++ has had a slower uptake. AspectC [11] was one of the first AOP systems for C, based on the language constructs of AspectJ. ACC [23] is a more recent AOP system for C, also based on the language constructs of AspectJ. Both systems offer specialized DSLs for writing pointcuts and advice, just like AspectJ, providing the user with a source-code view of the code to be instrumented. They transform source code and offer their own internal compiler framework for parsing C. These are closed systems in the sense that one cannot augment them with new pointcuts or access the internal structure of a C program in order to perform static analysis.

The XWeaver system [28], with its language AspectX, represents a program in XML, making it independent of the programming language. It supports Java and C++. The choice of an XML-based representation of the base code has the advantage of partially decoupling the aspect weaver and the aspect language from the language of the base code. Aspicere [27] is an aspect language for C based on LLVM bytecode. Its pointcut language is inspired by logic programming. Adding new pointcuts amounts to defining new logic predicates. Arachne [12, 14] is a dynamic aspect language for C that uses assembler manipulation techniques to instrument a running system without pausing it.

AspectC++ [30] is targeted towards C++. It can handle C to some extent, but this does not seem to be a high priority for its developers. For example, it only handles ANSI C and not other dialects. AspectC++ operates at the source-code level and generates C++ code, which can be problematic in contexts where only C code is permitted, such as in certain embedded applications. OpenC++ [9] is a front-end library for C++ that developers can use to implement various kinds of translations in order to define new syntax and object behavior. In this sense, it attempts to provide an open compiler framework. An OpenC++ user writes a meta-program, in the form of a small number of C++ classes, which is then compiled by the OpenC++ compiler and (dynamically or statically) linked to the compiler itself as a compiler plug-in.

CIL [24] (C Intermediate Language) is an OCaml [18] API for writing source-code transformations of its own 3A code representation of C programs. CIL requires a user to be familiar with the OCaml programming language. Valgrind [31] works directly with executables and consequently targets multiple programming languages.

With respect to the INTERASPECT Tracecut plug-in, the field of runtime verification has offered many such systems, and we do not claim that our plug-in outperforms the better of these. Rather, the plug-in is an illustration of INTERASPECT, demonstrating how such an extension can be defined. Using the INTERASPECT API for our tracecut monitoring facility greatly simplified its design, which we believe makes a case for the extensibility of the INTERASPECT API.

The INTERASPECT Tracecut is informed by several tracecut systems for Java, including Declarative Event Patterns [32], which introduced the term *tracecut*, Tracematches [2], and MOP [8], the last two supporting monitoring of regular expressions. Our handling of monitor parameterization is based on the implementations in Tracematches and MOP, most specifically MOP. More concretely, in INTERASPECT Tracecut, an index is created from parameters of events to propositional state machines resulting from translation of the regular expressions. Each monitor has a set of parameters, and each event sends a value for each of those parameters. When none of the values are empty, we say that the event is “fully parameterized” and look up the (at most) one monitor instance that has matching values for all the parameters. If no monitor instance is found, we create a new one. For a partially parameterized event (some values are empty), we look for all monitor instances whose parameter values match all the non-empty parameters of the event. If there are no such instances, the event is ignored. This basically means that the first event must carry all the parameters to

create a new monitor instance. This is how initial versions of MOP worked. Subsequently, MOP has been modified so that this restriction is no longer necessary [22].

As discussed in Section 5, for a given event, an index is created from the event's parameters and used to locate the monitor instances to update by a linear search, which identifies those instances whose parameters contain the index as a subset. As also previously stated, this approach is not efficient when the number of instances is big. An efficient solution for managing a large number of monitor instances would be to (abstractly viewed) maintain a map from indexes to monitor instances. This is in fact the approach taken in the MOP system, in which each index is mapped to a monitor state. In MOP, when a monitor receives an event, it combines the events parameters with the formal parameter names associated with that event to construct the index (itself a map from parameter names to concrete values), looks up the appropriate propositional monitor state for that binding, and then applies the propositional event in that monitor state to obtain a new state. More specifically, a monitor state is updated if it is mapped to by an index that includes the index produced by the event. The complete algorithm for MOP is more sophisticated than just described here. Note, however, that the tracecut solution is not an attempt to improve on existing monitoring solutions, but rather to illustrate how such a solution can easily be built on top of InterAspcect.

Another difference between INTERASPECT Tracecut and MOP is in their approaches to destroying monitor instances. MOP destroys an instance when the garbage collector reaps the objects assigned to the instance's parameters. Because C programs are not garbage collected and because INTERASPECT Tracecut can use parameters that are not allocated objects (such as integer file handles), instances in INTERASPECT Tracecut are destroyed when they can no longer reach an accepting state.

For C, Arachne and Aspicere provide tracecut-style monitoring. Arachne can monitor pointcut *sequences* which have similar semantics to INTERASPECT Tracecut's regular expressions [12]. The cHALO extension to Aspicere adds predicates for defining sequences [1]. These predicates are designed to give developers better control over the amount of memory used to track monitor instances.

7 Conclusions

We have presented INTERASPECT, a framework for developing powerful instrumentation plug-ins for the GCC suite of production compilers. INTERASPECT-based plug-ins instrument programs compiled with GCC by modifying GCC's intermediate language, GIMPLE. The INTERASPECT API simplifies this process by offering an AOP-based interface. Plug-in developers can easily specify pointcuts to target specific program join points and then add customized instrumentation at those join points. We presented several example plug-ins that demonstrate the framework's ability to customize runtime instrumentation for specific applications. Finally, we developed a more full-featured application of our API: the INTERASPECT Tracecut extension, which monitors formally defined runtime properties. The API and the tracecut extension are available under an open-source license [19].

As future work, we plan to add pointcuts for all control flow constructs, thereby allowing instrumentation to trace a program run's exact path of execution. We also plan to investigate API support for pointcuts that depend on dynamic information, such as AspectJ's `cflow`. Dynamic pointcuts can already be implemented in INTERASPECT with advice functions that maintain and use appropriate state, or even with tracecut advice, but API support would eliminate the need to write such advice functions.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. Part of the research described herein was carried out at the Jet Propulsion Laboratory (JP), California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Research described here was supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CCF-0926190, CCF-0613913, CNS-0831298, and CNS-0509230, and ONR Grants N00014-07-1-0928 and N00014-09-1-0651.

References

1. ADAMS, B., HERZEEL, C., AND GYBELS, K. cHALO, stateful aspects in C. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software* (New York, NY, USA, 2008), ACM, pp. 1–6.
2. ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPLAN, G., AND TIBBLE, J. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)* (2005), ACM Press.
3. BCEL. `http://jakarta.apache.org/bcel`.
4. AT&T RESEARCH LABS. Graphviz, 2009. `www.graphviz.org`.
5. AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. abc: An extensible AspectJ compiler. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development* (2005), ACM Press.
6. BODDEN, E., AND HAVELUND, K. Racer: Effective race detection using AspectJ. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2008), ACM, pp. 155–165.
7. CALLANAN, S., DEAN, D. J., AND ZADOK, E. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit* (Ottawa, Canada, July 2007), pp. 31–37.
8. CHEN, F., AND ROŞU, G. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)* (2007).
9. CHIBA, S. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 1995), pp. 285–299.
10. CHIBA, S. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming, LNCS* (2000), vol. 1850, Springer Verlag, pp. 313–336.
11. COADY, Y., KICZALES, G., FEELEY, M., AND SMOLYN, G. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2001), pp. 88–98.
12. DOUENCE, R., FRITZ, T., LORIANT, N., MENAUD, J.-M., SÉGURA-DEVILLECHAISE, M., AND SÜDHOLT, M. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)* (2005), ACM Press.
13. ECLIPSE FOUNDATION, T. AspectJ. `www.eclipse.org/aspectj`.
14. Arachne. `www.emn.fr/x-info/arachne`.
15. FEI, L., AND MIDKIFF, S. P. Artemis: Practical runtime monitoring of applications for errors. Tech. Rep. TR-ECE-05-02, Electrical and Computer Engineering, Purdue University, 2005. `docs.lib.purdue.edu/ecetr/4/`.
16. GCC 4.5 release series changes, new features, and fixes. `http://gcc.gnu.org/gcc-4.5/changes.html`.
17. HUANG, X., SEYSTER, J., CALLANAN, S., DIXIT, K., GROSU, R., SMOLKA, S. A., STOLLER, S. D., AND ZADOK, E. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer (STTT) 14*, 3 (2012), 327–347.
18. Objective Caml. `http://caml.inria.fr/index.en.html`.
19. InterAspect. `www.fsl.cs.stonybrook.edu/interaspect`.
20. JAN KNESCHKE. Lighttpd, 2009. `www.lighttpd.net/`.
21. KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (2001), LNCS, Vol. 2072, pp. 327–355.

22. MEREDITH, P. O., JIN, D., GRIFFITH, D., CHEN, F., AND ROŞU, G. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer* (2011). to appear.
23. ACC. `http://research.msrg.utoronto.ca/ACC`.
24. NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), Springer-Verlag, pp. 213–228.
25. NICOARA, A., ALONSO, G., AND ROSCOE, T. Controlled, systematic, and efficient code replacement for running Java programs. In *Proceedings of the ACM EuroSys Conference* (Glasgow, Scotland, UK, April 2008).
26. POSKANZER, J. http_load, 2006. `http://www.acme.com/software/http_load/`.
27. Aspicere. `http://sailhome.cs.queensu.ca/~bram/aspicere`.
28. ROHLIK, O., PASETTI, A., CECHTICKY, V., AND BIRRER, I. Implementing adaptability in embedded software through aspect oriented programming. *IEEE Mechatronics & Robotics* (2004), 85–90.
29. SEYSTER, J., DIXIT, K., HUANG, X., GROSU, R., HAVELUND, K., SMOLKA, S. A., STOLLER, S. D., AND ZADOK, E. Aspect-oriented instrumentation with GCC. In *Proc. of the 1st International Conference on Runtime Verification (RV 2010)* (November 2010), Lecture Notes in Computer Science, Springer.
30. SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Know.-Based Syst. 20*, 7 (2007), 636–651.
31. Valgrind. `http://valgrind.org`.
32. WALKER, R., AND VIGGERS, K. Implementing protocols via declarative event patterns. In *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)* (2004), R. Taylor and M. Dwyer, Eds., ACM Press, pp. 159–169.