Aspect-Oriented Instrumentation with GCC

Appears in the Proceedings of the First International Conference on Runtime Verification (RV 2010)

Justin Seyster[1], Ketan Dixit[1], Xiaowan Huang[1], Radu Grosu[1], Klaus Havelund[2], Scott A. Smolka[1], Scott D. Stoller[1], and Erez Zadok[1]

[1] Department of Computer Science, Stony Brook University
[2] Jet Propulsion Laboratory, California Institute of Technology

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. InterAspect also supports powerful customized instrumentation, where specific information about each join point in a pointcut, as well as results of static analysis, can be used to customize the inserted instrumentation. We introduce the InterAspect API and present several examples that illustrate how it can be applied to useful runtime verification problems.

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 very large software system with over 100 developers contributing over the years and a steering committee consisting of 13 experts who strive to maintain its architectural integrity.

In earlier work [5], 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 source code. Released in April 2010, GCC 4.5 [14] 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 for everyone but rather only 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.

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

– InterAspect builds on top of GCC, a compiler infrastructure having a large and dedicated following.
– InterAspect exposes an API, which encourages and simplifies open-source collaboration.
– InterAspect has access to GCC internals, which allows one to exploit static analysis and meta-programming during the weaving process.

To illustrate the practical utility of the InterAspect framework, we have developed a number of program-instrumentation plug-ins that use InterAspect for custom instrumentation. These include a *heap visualization* plug-in for anticipated use by the JPL Mars Science Laboratory software development team; 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.

The rest of the paper is structured as follows. Section 2 provides an overview of GCC and the InterAspect framework architecture. Section 3 introduces the InterAspect API. Section 4 presents the three applications: heap visualization, integer range analysis, and code coverage. Section 5 summarizes related work, and Section 6 concludes the paper.

2 Overview of GCC and the InterAspect Architecture

As Fig. 1 illustrates, GCC translates all of its front-end languages into the `GIMPLE` intermediate representation for analysis and optimization. Each transformation

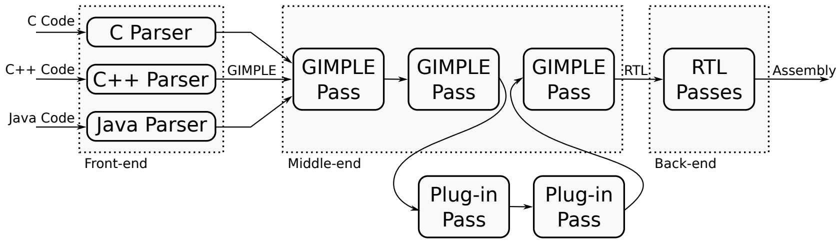

Fig. 1. A simplified view of the GCC compilation process.

on `GIMPLE` code is split into its own *pass*. These passes, some of which may be *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 lower 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.

Fig. 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.

```
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); }
```

=>

```
int main {
    int a, b, c;
    int T1, T2, T3, T4;
    a = 5;
    b = a + 10;
    T1 = foo(a, b);
    c = b + T1;
    T2 = b + c;
    if (a <= T2) goto fi;
       T3 = b / a;
       T4 = b * a;
       c = T3 + T4;
       b = b + 1;
fi: bar (a, b, c); }
```

Fig. 2. Sample C program and corresponding `GIMPLE` representation.

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 would examine the AST.

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

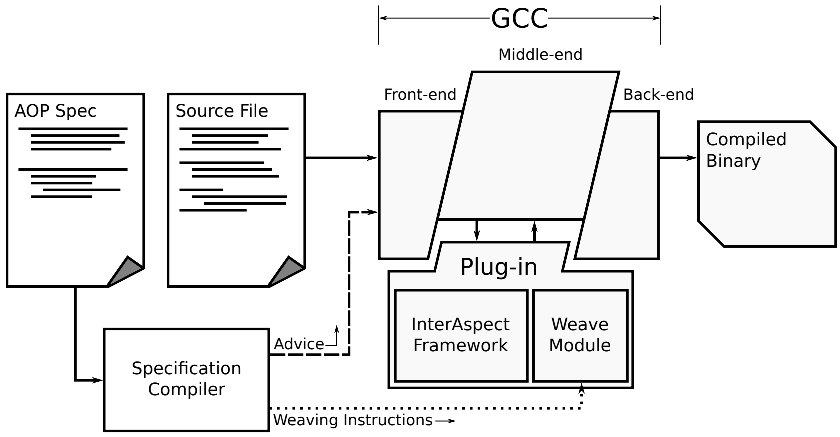

Fig. 3. Architecture of the InterAspect instrumentation framework for GCC.

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.

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.

As Fig. 3 illustrates, InterAspect can further serve as the instrumentation back-end for a traditional AOP specification language. The specification compiler's job is to split an AOP specification into pointcut definitions, associated weaving instructions, and advice code. The first two are sent to an InterAspect-based *weave module* for evaluation during the instrumentation plug-in pass, whereas the advice code is sent to GCC for compilation.

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 Table 1, allowing one to create four kinds of pointcuts.

`struct aop_pointcut *aop_match_function_entry();`	
Creates pointcut denoting every function entry point.	
`struct aop_pointcut *aop_match_function_exit();`	
Creates pointcut denoting every function return point.	
`struct aop_pointcut *aop_match_function_call();`	
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.	

Table 1. *Match functions* for creating 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 put 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. Table 2 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.	

Table 2. *Filter functions* for refining function-call pointcuts.

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 Table 3. InterAspect then calls the iterator callback for each join point so that it can instrument the join point with a call to an *advice* function.

Callback functions use *capture* functions to examine values associated with a join point. Capture functions expose two kinds of values: static values that are known at compile time and runtime values that will not be known until program

`void aop_join_on(struct aop_pointcut *pc, join_callback callback, void *callback_param);`
Supply `callback` function with any data structure as `callback_param`.

Table 3. *Join function* for iterating over a pointcut.

execution time. Static values, such as the name of the variable assigned by an assignment statement, are directly readable in the callback itself. The callback cannot access runtime values, such as the values assigned by an assignment statement, but it can pass them as parameters to advice functions, so that they are available to instrumentation code at runtime. These runtime values are represented in the callback function as special `aop_dynval` objects. Capture functions are specific to the kinds of join points they operate on. Tables 4 and 5 summarize the capture functions for function-call join points and assignment join points, respectively.

`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.

Table 4. *Capture functions* for function-call join points.

`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.

Table 5. *Capture functions* for assignment join points.

AOP systems like AspectJ [17] provide Boolean operators, such as *and* and *or*, to refine pointcuts. The InterAspect API could be extended with corresponding operations. Even without them, 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 will be added to the framework.

After capturing, a callback can add an advice function call before or after the join point using the *insert* function of Table 6. 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.

```
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`.

Table 6. *Insert function* for instrumenting a join point with a call to an advice function.

Function duplication. InterAspect provides a *function 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 duplication to reduce overhead by sampling.

4 Applications

To demonstrate InterAspect's flexibility, we present several example applications of the 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.

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. Fig. 4 shows a prototype of the visualizer using Graphviz [2], an open-source graph layout tool, to draw its output. The graph shows three nodes in a linked list during a bubble-sort operation. Each node is labeled with its size, its address in memory, and the addresses of its fields. Variables that point to `NULL` or to an invalid memory location are drawn with a dashed border. Edges are labeled with the line number of the assignment that created the edge, as well as the number of assignments to the source variable that have occurred so far.

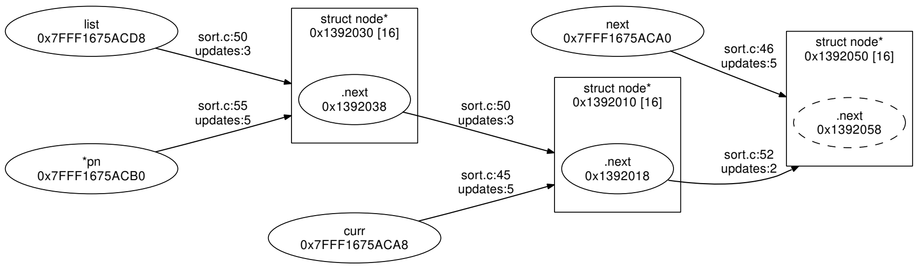

Fig. 4. A visualization of the heap during a bubble sort operation on a linked list.

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 Fig. 5 instruments calls to `malloc`. The 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. Additional `filter` functions narrow down 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 statement that calls `malloc`), `aop_join_on` calls `malloc_callback`. The two `capture` calls in the call-

```
static void instrument_malloc_calls()
{
  /* 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. 5. Instrumenting all memory allocation events.

back function return `aop_dynval` objects 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 Fig. 6 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`,

```
static void instrument_pointer_assignments()
{
  /* 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. 6. Instrumenting all pointer assignments.

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 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 [12]. 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 Controlled Overhead [15].

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.

```
static void instrument_integer_assignments()
{
  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. 7. Instrumenting integer variable updates.

Fig. 7 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.

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 empty advice. The instrumented `bzip2` contains advice calls at every integer variable assignment, but the advice functions themselves do nothing, allowing us to measure the overhead from calling advice functions independently from actual monitoring overhead. With a distributor that maximizes overhead by always choosing the instrumented function body, we measured 24% runtime overhead. Function duplication by itself contributes very little to this overhead; when the distributor always chooses the uninstrumented path, the overhead from instrumentation was statistically insignificant.

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.

The example in Fig. 8 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 Related Work

Aspect-oriented programming was first introduced for Java with AspectJ [10, 17]. There, weaving takes place at the bytecode level. The AspectBench Compiler (abc) [3] is a more recent extensible research version of AspectJ that makes it possible to add new language constructs (see for example [4]). Similarly to InterAspect, it manipulates a `3A` intermediate representation (Jimple) specialized to Java.

```
static void instrument_function_entry_exit()
{
  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. 8. Instrumenting function entry and exit for code coverage.

Other frameworks for Java, including Javaassist [7] and PROSE [19], offer 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.

AOP for other languages such as C and C++ has had a slower uptake. AspectC [8] was one of the first AOP systems for C, based on the language constructs of AspectJ. ACC [18] is a more recent AOP system for C, also based on the language constructs of AspectJ. It transforms source code and offers its own internal compiler framework for parsing C. It is a closed system in the sense that one cannot augment it with new pointcuts or access the internal structure of a C program in order to perform static analysis.

The XWeaver system [21], with its language AspectX, represents a program in XML (srcML, to be specific), making it language-independent. It supports Java and C++ . A user, however, has to be XML-aware. Aspicere [20] 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 [9, 11] is a dynamic aspect language for C that uses assembler manipulation techniques to instrument a running system without pausing it.

AspectC++ [22] 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++ [6] 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. CIL [13] (C Intermediate Language) is an OCaml [16] API for writing source-code transformations of its own `3A` code representation of C programs. CIL requires a user to be familiar with the less-often-used yet powerful OCaml language.

Additionally, various low-level but mature tools exist for code analysis and instrumentation. These include the BCEL [1] bytecode-instrumentation tool for Java, and Valgrind [23], which works directly with executables and consequently targets multiple programming languages.

6 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.

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`, by introducing filters that are evaluated at run-time. Dynamic pointcuts can already be implemented in InterAspect with advice functions that maintain and use appropriate state, but API support would eliminate the need to write those 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, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Research supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CCF-0926190, CCF-0613913, and CNS-0831298, and ONR Grants N00014-07-1-0928 and N00014-09-1-0651.

References

1. BCEL. `http://jakarta.apache.org/bcel`.
2. AT&T Research Labs. Graphviz, 2009. `http://www.graphviz.org`.
3. 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.

4. Bodden, E., and Havelund, K. Racer: Effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis, Seattle, WA* (2008), ACM, pp. 155–165.
5. 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).
6. 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.
7. 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.
8. 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* (2001), pp. 88–98.
9. 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* (2005), ACM Press.
10. AspectJ. `http://www.eclipse.org/aspectj`.
11. Arachne. `http://www.emn.fr/x-info/arachne`.
12. 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/`.
13. G. C. Necula and S. McPeak and S. P. Rahul and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction* (London, England, 2002), Springer-Verlag, pp. 213–228.
14. GCC 4.5 release series changes, new features, and fixes. `http://gcc.gnu.org/gcc-4.5/changes.html`.
15. 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)* (2010). Accepted for publication.
16. Objective Caml. `http://caml.inria.fr/index.en.html`.
17. 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.
18. ACC. `http://research.msrg.utoronto.ca/ACC`.
19. 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).
20. Aspicere. `http://sailhome.cs.queensu.ca/~bram/aspicere`.
21. Rohlik, O., Pasetti, A., Cechticky, V., and Birrer, I. Implementing adaptability in embedded software through aspect oriented programming. *IEEE Mechatronics & Robotics* (2004), 85–90.
22. Spinczyk, O., and Lohmann, D. The design and implementation of AspectC++. *Know.-Based Syst. 20*, 7 (2007), 636–651.
23. Valgrind. `http://valgrind.org`.