Compiler Validation via Equivalence Modulo Inputs

Vu Le Mehrdad Afshari Zhendong Su

Department of Computer Science, University of California, Davis, USA

{vmle, mafshari, su}@ucdavis.edu

Abstract

We introduce *equivalence modulo inputs (EMI)*, a simple, widely applicable methodology for validating optimizing compilers. Our key insight is to exploit the close interplay between (1) dynamically executing a program on some test inputs and (2) statically compiling the program to work on all possible inputs. Indeed, the test inputs induce a natural collection of the original program's EMI variants, which can help differentially test any compiler and specifically target the difficult-to-find miscompilations.

To create a practical implementation of EMI for validating C compilers, we profile a program's test executions and stochastically prune its unexecuted code. Our extensive testing in eleven months has led to 147 confirmed, unique bug reports for GCC and LLVM alone. The majority of those bugs are miscompilations, and more than 100 have already been fixed.

Beyond testing compilers, EMI can be adapted to validate program transformation and analysis systems in general. This work opens up this exciting, new direction.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; H.3.4 [*Programming Languages*]: Processors—compilers

General Terms Algorithms, Languages, Reliability, Verification

Keywords Compiler testing, miscompilation, equivalent program variants, automated testing

1. Introduction

Compilers are among the most important, widely-used and complex software ever written. Decades of extensive research and development have led to much increased compiler performance and reliability. Perhaps less known to application programmers is that production compilers do also contain bugs, and in fact quite a few. However, compiler bugs are hard to recognize from the much more frequent bugs in applications because often they manifest only indirectly as application failures. Thus, when compiler bugs occur, they frustrate programmers and may lead to unintended application behavior and disasters, especially in safety-critical domains. Compiler verification has been an important and fruitful area for the verification grand challenge in computing research [9].

PLDI '14, June 9 – 11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594334

Besides traditional manual code review and testing, the main compiler validation techniques include testing against popular validation suites (such as Plum Hall [21] and SuperTest [1]), verification [12, 13], translation validation [20, 22], and random testing [28]. These approaches have complementary benefits. For example, CompCert [12, 13] is a formally verified optimizing compiler for a subset of C, targeting the embedded software domain. It is an ambitious project, but much work remains to have a fully verified production compiler that is correct end-to-end. Another good example is Csmith [28], a recent work that generates random C programs to stress-test compilers. To date, it has found a few hundred bugs in GCC and LLVM, and helped improve the quality of the most widely-used C compilers. Despite this incredible success, the majority of the reported bugs were compiler crashes as it is difficult to steer its random program generation to specifically exercise a compiler's most critical components—its optimization phases. We defer to Section 5 for a detailed survey of related work.

Equivalence Modulo Inputs (EMI) This paper introduces a simple, broadly applicable concept for validating compilers. Our vision is to take existing real-world code and transform it in a novel, systematic way to produce different, but equivalent variants of the original code. To this end, we introduce *equivalence modulo inputs* (EMI) for a practical, concrete realization of the vision.

The key insight behind EMI is to exploit the interplay between *dynamically executing* a program P on a subset of inputs and *statically compiling* P to work on all inputs. More concretely, given a program P and a set of input values I from its domain, the input set I induces a natural collection of programs $\mathscr{C}$ such that every program $Q \in \mathscr{C}$ is equivalent to P modulo I: $\forall i \in I, Q(i) = P(i)$. The collection $\mathscr{C}$ can then be used to perform differential testing [16] of any compiler *Comp*: If $Comp(P)(i) \neq Comp(Q)(i)$ for some $i \in I$ and $Q \in \mathscr{C}$, *Comp* has a miscompilation.

Next we provide some high-level intuition behind EMI's effectiveness (Section 2 illustrates this insight with two concrete, real examples for Clang and GCC respectively). The EMI variants can specifically target a compiler's analysis and optimization phases, and stress-test them to reveal latent compiler bugs. Indeed, although an EMI variant Q is only equivalent to P modulo the input set I, the compiler has to perform all its (static) analysis and optimizations to produce correct code for Q over *all inputs*. In addition, P's EMI variants, while semantically equivalent *w.r.t.* I, can have quite different static data- and control-flow. Since data- and control-flow information critically affects which optimizations are enabled and how they are applied, the EMI variants not only help exercise the optimizer differently, but also demand the exact same output on I from the generated code by these different optimization strategies—This is the very fact that we crucially leverage.

EMI has several unique advantages:

- It is general and easily applicable to finding bugs in compilers, analysis and transformation tools for any language.

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(a) Test `931004-11.c` from the GCC test suite; it compiles correctly by all compilers tested.

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) /* deleted */;
  if (x.d != 20) abort();
  if (x.e != 30) /* deleted */;
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) /* deleted */;
  if (z.c != 12) abort();
  if (z.d != 22) /* deleted */;
  if (z.e != 32) abort();
  if (l != 123) /* deleted */;
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(b) Test case produced by Orion by transforming the program in Figure 1a, triggering a bug in Clang.

Figure 1: Orion transforms Figure 1a to Figure 1b and uncovers a miscompilation in Clang. Note that the test cases have been reformatted for presentation. In the actual code, each "`abort();`" is on a separate line (see Section 3.2.1).

- It can directly target miscompilations and stress-test critical, complex compiler components.
- It can generate and explore test cases based on real-world code that developers actually write. This means that any detected errors are more likely to manifest in real-world code, and thus more directly impact software vendors and users.

Indeed, we believe that the EMI concept is simple and can be adapted to validate compilers, program analysis, and program transformation systems in general. Potential applications include, for example, (1) testing and validating production compilers and software analysis tools; (2) generating realistic, comprehensive test suites for validation and certification; and (3) helping software vendors detect potential compiler-induced errors in their software, which can be very desirable for safety- and mission-critical domains.

Compiler Bug Hunter: Orion Given a program P and an input set I, the space of P's EMI variants *w.r.t.* I is vast, and difficult or impossible to compute. Thus, for realistic use, we need a practical instantiation of EMI. We propose a "profile and mutate" strategy to systematically generate a subset of a program's EMI variants. In particular, given a program P and input set I, we profile executions of program P over the input set I, and derive (a subset of) P's EMI variants (*w.r.t.* I) by stochastically pruning, inserting, or modifying P's unexecuted code on I. These variants should clearly behave exactly the same on the same input set I as the original program P (assuming that P is deterministic). We then feed these variants to any given compiler. Any detected deviant behavior on I indicates a bug in the compiler.

We have implemented our "profile and mutate" strategy for C compilers and focused on *pruning unexecuted code*. We have extensively evaluated our tool, Orion[1], in testing three widely-used C compilers—namely GCC, LLVM, and ICC—with extremely positive results (Section 4). We have used Orion to generate variants for real-world projects, existing compiler test suites, and much more extensively for test cases generated by Csmith [28]. In eleven months, we have reported, for GCC and LLVM alone, 147 confirmed, *unique* bugs. More than 100 have already been fixed, and more importantly, the majority of the bugs were miscompilations (rather than compiler crashes), clearly demonstrating the ability of EMI—offered by Orion—to stress-test a compiler's optimizer. We have also found and reported numerous bugs in ICC initially, but later we only focused on the two open-source compilers, GCC and LLVM, as both use open, transparent bug-tracking systems.

We have also done less, but still considerable testing of CompCert [12, 13]. Besides a few confirmed front-end issues we found and reported, we have yet to encounter a bug in CompCert's verified components. This fact gives strong evidence of the promise and quality of verified compilers, although it is true that CompCert still supports only a subset of C and fewer optimizations than production compilers, such as GCC and LLVM.

Contributions We make the following main contributions:

- We introduce the novel, general concept of equivalence modulo inputs (EMI) for systematic compiler validation.
- We introduce the "profile and mutate" strategy to realize Orion, a practical implementation of EMI for testing C compilers.
- We report our extensive evaluation of Orion in finding numerous bugs (147 unique bugs) in GCC and LLVM.

Paper Organization The rest of the paper is structured as follows. Section 2 shows a few examples to illustrate the EMI methodology and the "profile and mutate" strategy. Section 3 formalizes EMI and describes our realization of Orion. Next, we describe our extensive evaluation and discuss our experience in using Orion

[1] Orion was a giant huntsman in Greek mythology.

to find compiler bugs (Section 4). Finally, we survey related work (Section 5) and conclude (Section 6).

2. Illustrative Examples

This section uses two concrete examples to motivate and illustrate our work: one for LLVM, and one for GCC.

In general, compiler bugs are of two main types, and they vary in severity. Some merely result in a *compiler crash*, causing minor nuisances and portability problems at times. Others, however, can cause compilers to silently miscompile a program and produce *wrong code*, subverting the programmer's intent. Miscompilations are daunting, and the following characteristics make them distinctive:

***Lead to bugs in other programs*:** Normally, a bug in a program only affects itself. Compilers generating wrong code can effectively inject bugs to programs they compile.

Hard to notice: If a miscompilation only affects a less traversed code path or certain optimization flags, it might go unnoticed during program development and only trigger in specific circumstances. Note that a rarely occurring bug can still be a severe issue. This can be especially troublesome for compilers targeting embedded platforms and micro-controllers.

Hard to track down to the compiler: Popular mainstream compilers are generally considered very reliable (indeed they are), making them often the least to suspect when client code misbehaves.

Weaken source-level analysis and verification: Correctness guarantees at the source code level may be invalidated due to a compiler bug that leads to buggy binaries, thus hindering overall system reliability.

Impact the reliability of safety-critical systems: A seemingly unimportant miscompilation bug can potentially result in a critical flaw in a safety-critical system, thus making compiler reliability critically important.

These characteristics of compiler miscompilations make their effects more similar to bugs in hardware — and in the case of popular compilers, like bugs in widely-deployed hardware — than bugs in most other programs. EMI is a powerful technique for detecting various kinds of compiler bugs, but its power is most notable in discovering miscompilations.

Our tool, Orion, detects compiler bugs by applying EMI on source programs. For instance, we took the test program in Figure 1a from the GCC test suite. It compiles and runs correctly on all compilers that we tested. We subsequently apply Orion on the test program. Clearly, none of the `abort` calls in the function `f` should execute when the program runs, and the coverage data confirms this. This allows Orion to freely alter the body of the `if` statements in the function `f` or remove them entirely without changing this program's behavior. By doing so, Orion transforms the program and produces many new test cases. One of these transformations, shown in Figure 1b, is miscompiled by Clang on the 32-bit x86 architecture when optimizations are enabled. Figure 2 shows its reduced version that we used to report the bug.

A bug in the LLVM optimizer causes this miscompilation. The developers believe that the Global Value Numbering (GVN) optimization turns the struct initialization into a single 32-bit load. Subsequently, the Scalar Replacement of Aggregates (SRoA) optimization decides that the 32-bit load is undefined behavior, as it reads past the end of the struct, and thus does not emit the correct instructions to initialize the struct. The developer who fixed the issue characterized it as

> *"... very, very concerning when I got to the root cause, and very annoying to fix."*

```
struct tiny { char c; char d; char e; };
void foo(struct tiny x) {
  if (x.c != 1) abort();
  if (x.e != 1) abort();
}
int main() {
  struct tiny s;
  s.c = 1; s.d = 1; s.e = 1;
  foo(s);
  return 0;
}
```

Figure 2: Reduced version of the code in Figure 1b for bug reporting. (`http://llvm.org/bugs/show_bug.cgi?id=14972`)

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    for (; c;)
      for (;;) {
        e = a > 2147483647 - b;
        if (d) break;
      }
  return 0;
}
```

Figure 3: GCC miscompiles this program to an infinite loop instead of immediately terminating with no output. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58731`)

The original program did not expose the bug because Clang decided not to inline the function `f` due to its size. In contrast, the pruned function `f` in the EMI variant became small enough that the Clang optimizer at -Os (and above) — when the inliner is enabled — decided to inline it. Once `f` was inlined, the incompatibility between GVN and SRoA led to the miscompilation. Indeed, using an explicit "inline" attribute on `f` in the original program also exposes this bug.

In another case, Orion derives the code in Figure 3 from a program generated by Csmith [28], which was miscompiled by GCC 4.8 and the latest trunk revision at the time when optimizations were enabled in both 32-bit and 64-bit modes. The correct execution of this program will terminate immediately, as the continuation condition of the second `for` loop will always be `false`[2], never letting its loop body execute. GCC with optimizations enabled miscompiles this program to an infinite loop. Interestingly, it does issue a bogus warning under `-O2`, but not `-O3`, which hints at the root cause of the miscompilation:

> "*cc1: warning: iteration 5u invokes undefined behavior [-Waggressive-loop-optimizations].*"

The warning implies that the sixth iteration of the outermost loop (when `b = -1`) triggers undefined behavior (*i.e.* signed integer overflow). In fact, there is no undefined behavior in this program, as the innermost loop is dead code and never executes, thus never triggering signed integer overflow at run time.

Partial Redundancy Elimination (PRE) detects the expression "`e2147483647 - b`" as loop invariant. Loop Invariant Motion (LIM) tries to move it up from the innermost loop to the body of the outermost loop. Unfortunately, this optimization is problematic, as GCC then detects a signed overflow in the program's optimized version and this (incorrect) belief of the existence of undefined behavior causes the compiler to generate non-terminating code (and the bogus warning at `-O2`).

The original program did not trigger the bug because there were other statements in the innermost loop that mutated the variable

[2] The variable `c` and other global variables are initialized to `0` in C.

b (for instance, increasing b before the assignment to e). The expression "2147483647 - b" was thus determined not to be a loop invariant and was not hoisted outside the inner loops. The program ran as expected. On the other hand, since the innermost loop was not executed, Orion could freely modify its body. It generated some variants in which all statements mutating b were removed. As explained earlier, in these variants, the expression became loop invariant, and thus was hoisted out of the loop, which effectively triggered the bug. The original program is quite complex, having 2,985 LOC; the EMI variant that exposed the bug has 2,015 LOC.

The two examples demonstrate that bugs can appear in both small, and large, complex code bases, potentially resulting in hard-to-detect errors, crashes, or security exploits, even in entirely correct, even verified, programs. They also highlight the difficulty of correctly optimizing code. Not only each optimization pass can introduce bugs directly, the interactions among different optimizations can also lead to latent bugs. EMI, being an end-to-end testing methodology, detects bugs that occur across optimization passes, as well as those that occur within an individual pass.

3. EMI and Orion's Implementation

This section introduces *equivalence modulo inputs (EMI)* and describes our realization of Orion.

3.1 Definitions and High-Level Approach

The concept of equivalence modulo inputs (EMI) that we have outlined in Section 1 is simple and intuitive. The main goal of this subsection is to provide more detailed and precise definitions.

Rather than formalizing EMI for a concrete programming language, we operate on a generic programming language $\mathscr{L}$ with *deterministic*[3] semantics $[\![\cdot]\!]$, *i.e.*, repeated executions of a program $P \in \mathscr{L}$ on the same input i always yield the same result $[\![P]\!](i)$.

3.1.1 Equivalence Modulo Inputs

Two programs $P, Q \in \mathscr{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff

$$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

We use $[\![P]\!] =_I [\![Q]\!]$ to denote that P and Q are EMI *w.r.t.* input set I.

For the degenerate case where P and Q do not take inputs (*i.e.*, they are closed programs), EMI reduces to semantic equivalence:

$$[\![P]\!] = [\![Q]\!].$$

Or more precisely, P and Q are EMI *w.r.t.* the input set {void}, where void denotes the usual "*no argument*":

$$[\![P]\!](\texttt{void}) = [\![Q]\!](\texttt{void}).$$

For example, the GCC test 931004-11.c and the output code from Orion shown respectively in Figures 1a and 1b are EMI (*w.r.t.* $I = \{\texttt{void}\}$).

Given a program $P \in \mathscr{L}$, any input set $I \subseteq \mathsf{dom}(P)$ naturally induces a collection of programs $Q \in \mathscr{L}$ that are EMI (*w.r.t.* I) to P. We call this collection P's *EMI variants*.

Definition 3.1 (EMI Variants)**.** *A program P's* EMI variants w.r.t. *an input set I is given by:*

$$\{Q \in \mathscr{L} \mid [\![P]\!] =_I [\![Q]\!]\}.$$

It is clear that EMI is a relaxed notion of semantic equivalence:

$$[\![P]\!] = [\![Q]\!] \quad \Longrightarrow \quad [\![P]\!] =_I [\![Q]\!].$$

[3] Note that we may also force a non-deterministic language to assume deterministic behavior.

3.1.2 Differential Testing with EMI Variants

At this point, it may not be clear yet what benefits our relaxed notion of equivalence can provide, which we explain next.

Differential Testing: An Alternative View Our goal is to differentially test [16] compilers. The traditional view of differential testing is simple: If two programs (in our setting, compilers or compiler versions) "act differently" on some input (*i.e.* source programs), we have found a bug in one of the compilers (maybe also in both). This is, for example, the view taken by Csmith [28] (assuming that the input programs are well-behaving, *e.g.*, they do not exhibit any undefined behavior).

We adopt an alternative view: If an oracle can generate a program P's semantic equivalent variants, these variants can stress-test any compiler *Comp* by checking whether *Comp* produces equivalent code for these variants. This view is attractive because we can (1) operate on existing code (or randomly generated, but valid code), and (2) check a single compiler in isolation (*e.g.* where competing compilers do not exist). However, we face two difficult challenges: (1) How to generate semantic equivalent variants? and (2) How to check equivalence of the produced code? Both have been long-standing challenges in software analysis and verification.

The "Profile and Mutate" Strategy Our key insight is that EMI provides a practical mechanism to realize our alternative view for differential testing of compilers. Indeed, by relaxing semantic equivalence *w.r.t.* an input set I, we reduce the second challenge to the simple task of testing against I. As for the first challenge, note that P's executions on I yield a *static slice* of P and unexecuted "dead code". One may freely mutate the "dead code" without changing P's semantics on I, thus providing a potentially enormous number of EMI variants to help stress-test compilers.

Once the EMI variants are generated, testing is straightforward. Let $\mathscr{Q} = \{Q_1, \ldots, Q_k\}$ be a set of P's EMI variants *w.r.t.* I. For each $Q_i \in \mathscr{Q}$, we verify the following:

$$\forall i \in I \;\; \mathit{Comp}(Q_i)(i) = \mathit{Comp}(P)(i).$$

Any deviant behavior indicates a miscompilation.

So far, we have not specified how to "mutate" the unexecuted "dead code" *w.r.t.* I. Obvious mutations include pruning, insertion, or modification. Our implementation, which we describe next, focuses on pruning, and we show in evaluation that even such a simple realization is extremely effective — it has detected 147 unique bugs for GCC and LLVM alone in under a year. We leave as future work to explore other mutation strategies.

3.2 Implementation of Orion

We now describe Orion, our practical realization of the EMI concept targeting C compilers via the "profile and prune" strategy. At a high level, Orion operates on a program's abstract syntax tree (AST) and contains two key steps: (1) extracting coverage information (Section 3.2.1), and (2) generating EMI variants (Section 3.2.2).

One challenge for testing C compilers is to avoid programs with undefined behavior because the C standard allows a compiler to do anything with such programs. For example, one major, painstaking contribution of Csmith is to generate valid test programs most of the time. In this regard, Orion has a strong advantage. Indeed, the EMI variants generated by Orion do not exhibit any undefined behavior if the original program has no undefined behavior (since only dead code is pruned from the original program). This advantage of Orion helps to easily generate many valid variants from a single valid seed program.

Algorithm 1 describes Orion's main process. As its first step, Orion profiles the test program P's execution on the input set I to collect (1) *coverage information* and (2) the *expected output* on each input value $i \in I$ (lines 2–3). It then generates P's EMI variants *w.r.t.*

Algorithm 1: Orion's main process for compiler validation

```
1  procedure Validate (Compiler Comp, TestProgram P, InputSet I):
2      begin
           /* Step 1: Extract coverage and output */
3          P_exe := Comp.Compile(P,"-O0")      /* without opt. */
4          C := ⋃_{i∈I} C_i, where C_i := Coverage(P_exe.Execute(i))
5          IO := {⟨i, P_exe.Execute(i)⟩ | i ∈ I}
           /* Step 2: Generate variants and verify */
6          for 1..MAX_ITER do
7              P' := GenVariant(P,C)
               /* Validate Comp's configurations */
8              foreach σ ∈ Comp.Configurations() do
9                  P'_exe := Comp.Compile(P', σ)
10                 foreach ⟨i,o⟩ ∈ IO do
11                     if P'_exe.Execute(i) ≠ o then
                           /* Found a miscompilation */
12                         ReportBug (Comp, σ, P, P', i)
```

Algorithm 2: Generate an EMI variant

```
1  function GenVariant (TestProgram P, Coverage C): Variant P':
2      begin
3          P' := P
4          foreach s ∈ P'.Statements() do
5              PruneVisit (P', s, C)
6          return P'
7  procedure PruneVisit (TestProgram P', Statement s, Coverage C):
8      begin
           /* Delete this statement when applicable */
9          if s ∉ C and FlipCoin(s) then
10             P'.Delete (s)
11             return
           /* Otherwise, traverse s's children */
12         foreach s' ∈ s.ChildStatements() do
13             PruneVisit (P', s', C)
```

I (lines 5–6), and uses them to validate each compiler configuration against the collected reference output (lines 7–11). Next, we discuss each step in detail.

3.2.1 Extracting Coverage Information

Code coverage tools compute how frequently a program's statements execute during its profiled runs on some sample inputs. We can conveniently leverage such tools to track the executed (*i.e.* "covered") and unexecuted (*i.e.* "dead") statements of our test program P under input set I. Those statements marked "dead" are candidates for pruning in generating P's EMI variants.

In particular, Orion uses `gcov` [7], a mature utility in the GNU Compiler Collection, to extract coverage information. We enable `gcov` by compiling the test program P with the following flag:

"`-O0 -coverage`"

which instruments P with additional code to collect coverage information at runtime. Orion then executes the instrumented executable on the provided input set I to obtain coverage files with information indicating how many times a source line has been executed.

Because `gcov` profiles coverage at the line level, it may produce imprecise results when multiple statements are on a single line. For example, in the example below,

```
if (false) { /* this could be removed */ }
```

`gcov` marks the entire line as executed. As a result, Orion cannot mutate it, although the statements within the curly braces could be safely removed. Note that we manually formatted the two test cases in Figure 1 for presentation. The actual code has every "`abort();`" on a separate line.

Occasionally, coverage information computed by `gcov` can also be ambiguous. For instance, in the sample snippet below (extracted from the source code of the Mathomatic[4] computer algebra system), `gcov` marks line 2613 as unexecuted (indicated by prepending the leading "`#####`"):

```
#####: 2613:  for (;; cp = skip_param(cp)) {
                      .....
    7: 2622:          break;
#####: 2623:  }
```

[4] `http://www.mathomatic.org/`

Based on this information, Orion assumes that it can remove the entire `for` loop (lines 2613–2623). This is incorrect, as the `for` loop is actually executed (indicated by the execution of its child statement `break`). What `gcov` really means is that the expression "`cp = skip_param(cp)`" is unevaluated. We remedy this coverage ambiguity by verifying that none of the children of an unexecuted statement is executed before removing it in the next step.

To avoid the aforementioned problems caused by collecting coverage statistics at line granularity, we could modify `gcov` or implement a new code coverage tool that would operate at the statement level. This can make our analysis more precise and help generate more variants. However, the practical benefits seem negligible as often there are only few such impacted statements. Our extremely positive results (Section 4) demonstrate that the use of `gcov` has been a good, well-justified decision.

3.2.2 Generating EMI Variants

Orion uses LLVM's LibTooling library [26] to parse a C program into an AST and mutate it based on the computed coverage information to generate the program's EMI variants.

The mutation process happens at the statement level in the AST. We mark a statement unexecuted if (1) the line number of its first token is marked unexecuted by `gcov`, and (2) none of its child statements in the AST is executed. When Orion decides to prune a statement, it removes all tokens in its AST subtree, including all its child statements. Thus, all variants generated by Orion are syntactically correct C programs.

Algorithm 2 describes Orion's process for generating EMI variants. The process is simple — We traverse all the statements in the original program P and randomly prune the unexecuted "dead" statements. On line 9, we use the function `FlipCoin` to decide stochastically whether an unexecuted statement s should be kept or removed. We control Orion's pruning via two parameters, P_{parent} and P_{leaf}, which specify the probabilities to prune parent or leaf statements in `FlipCoin`. One can use static probabilities for deleting statements and uniformly vary these values across different runs. An alternative is to allow dynamically adjusted probabilities for each statement. From our experience, this additional dynamic control seems quite effective. In fact, our standard setup is to randomly adjust these two parameters after each statement pruning by resetting each to an independent random probability value from 0.0 to 1.0.

In our actual implementation, Algorithm 1 is realized using shell scripts. In particular, we have a set of scripts to collect coverage and reference output information, control the outer loop, generate

EMI variants and check for possible miscompilation or compiler crashes. We have implemented Algorithm 2 in C++ using LLVM's LibTooling library [26]. Unlike random program generators such as Csmith, Orion requires significantly less engineering effort. It has approximately 500 lines of shell scripts and 1,000 lines of C++ code, while Csmith contains about 30-40K lines of C++ code.

4. Evaluation Setup and Results

This section presents our extensive evaluation of Orion to demonstrate the practical effectiveness of our EMI methodology besides its conceptual elegance and generality.

Since January 2013, we have been experimenting with and refining Orion to find new bugs in three widely-used C compilers, namely GCC, LLVM, and ICC. We have also occasionally tested CompCert [12, 13], a formally verified C compiler. In April 2013, we started our extensive testing of GCC, LLVM, and ICC. After finding and reporting numerous bugs in ICC, we stopped testing it for the lack of direct communication with its developers (although we did learn afterward by checking its later releases that many bugs we reported had been fixed). Since then, we have only focused on GCC and LLVM because both have open bug repositories, and transparent bug triaging and resolution. This section describes the results from our extensive testing effort for about eleven months.

Result Summary Orion is extremely effective:

- *Many confirmed bugs*: In eleven months, we have found and reported 147 confirmed, unique bugs in GCC and LLVM alone.
- *Many long-latent bugs*: Quite a few of the detected bugs have been latent for many years, and resisted the attacks from both earlier and contemporary tools.
- *Many have been already fixed*: So far, 110 of the bugs have already been fixed and resolved; most of the remaining ones have been triaged, assigned, or are being actively discussed.
- *Most are miscompilations*: This is perhaps the most important, clearly demonstrating the strengths of EMI for targeting the hard-to-find and more serious miscompilations (rather than compiler crashes). For example, Orion has already found about the same number (around 40) of miscompilations in GCC as Csmith did, but over several years' prior and continuing testing.

4.1 Testing Setup

Hardware and Compiler Our testing has focused on the x86-linux platform. Since late April 2013, we have performed our testing on two machines (one 18 core and one 6 core) running Ubuntu 12.04 (x86_64). For each compiler (*i.e.* GCC and LLVM), we test its latest development version (usually built once daily) under the most common configurations (*i.e.* `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`), generating code for both 32-bit (`-m32`) and 64-bit (`-m64`) environments. We did not use any finer-grained combinations of the compilers' optimization flags.

Test Programs In our testing, we have drawn from three sources of test programs to generate their EMI variants (note that none of the original test programs triggered a compiler bug):

- *Compiler Test Suites*: Each of GCC and LLVM has an already sizable and expanding regression test suite, which we can use for generating EMI variants (which in turn can be used to test any compiler). For example, the original test case shown in Figure 1 was from the GCC test suite, and one of its EMI variants helped reveal a subtle miscompilation in LLVM. We used the approximately 2,000 collected tests from the KCC [6] project, an executable formal semantics for C. Among others, this collection includes tests primarily from regression test suites of GCC and LLVM. The programs in these test suites do not take inputs, and are generally quite small. Nonetheless, we were able to find bugs by pruning them. The problem with this source is that the number of bugs revealed by their EMI variants saturated quickly, which is expected as they have few unexecuted statements.
- *Existing Open-Source Projects*: Another interesting source is the large number of open-source projects available. One challenge to use such a project is that its source code usually scatters across many different directories. Fortunately, these projects normally use the GNU build utilities (*e.g.* "`configure`" followed by "`make`") and do often come with a few test inputs (*e.g.* invoked by "`make test`" or "`make check`"), which we can leverage to generate EMI variants.

  In particular, we modify a project's build process to generate coverage information for "`make test`". To generate an EMI variant for the project, we bootstrap its build process. Before compiling a file, we invoke our EMI-gen tool that transforms the file into an EMI file, which is then compiled as usual. The output from the build process is an EMI variant of the original project. Then, we can use it to test each compiler simply by running the accompanying "`make test`" (or "`make check`"). If checking fails on the variant under a particular compiler configuration, we have discovered a compiler bug.

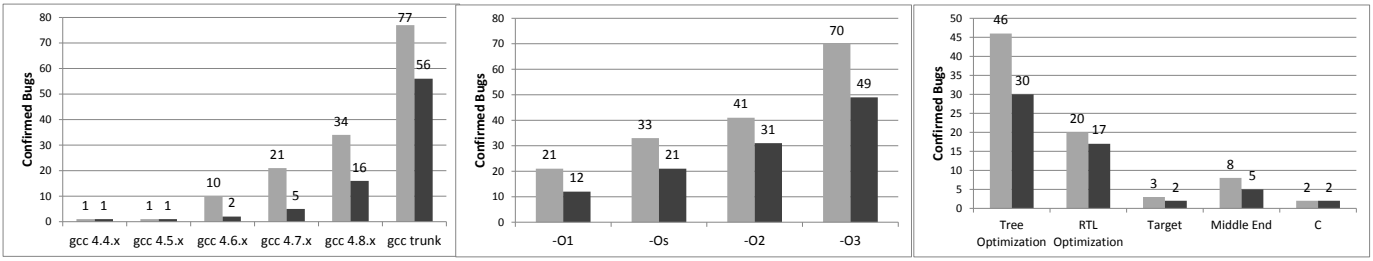
  Now, we face another challenge, that is how to reduce the bug-triggering EMI variant for bug reporting. This is a more serious challenge, particularly for miscompilations. Although we have applied Orion on a number of open-source projects—including all nine SPEC2006 integer C programs, Mathomatic and `tcc`[5]—and found many inconsistencies, we were only able to reduce one GCC crashing bug triggered by an EMI variant of `gzip`. We are yet to reduce the others, such as an interesting GCC miscompilation triggered by a variant of `tcc`.
- *Csmith-Generated Random Code*: Csmith turns out to be an excellent source for providing an enormous number of test programs for Orion. Programs generated by Csmith, which do not take inputs, are generally complex and offer quite rich opportunities for generating EMI variants. Our Csmith setup produces programs with an average size of 4,590 LOC, among which 1,327 lines on average are unexecuted. This corresponds to a vast space of EMI variants. More importantly, these Csmith-variants can often be effectively reduced using existing tools such as C-Reduce [24] and Berkeley Delta [17]. Thus, most of our testing has been on top of the random programs generated by Csmith, running in its "swarm testing" setup [8] .

Test Case Reduction Test case reduction is still a significant and time-consuming challenge. Our experience suggests that neither C-Reduce nor Berkeley Delta is the most effective on its own. We have devised an effective *meta-process* to utilize both. It is a nested fixpoint loop. First, we use Delta to repeatedly reduce the test case until a fixpoint has been reached (*i.e.* no additional reduction from Delta). Then, we run C-Reduce on the fixpoint output from Delta. We repeat this two-step process until reaching a fixpoint. This meta-process strikes a nice balance to take advantage of Delta's better efficiency and C-Reduce's stronger reduction capability.

There is another challenge: How to reject code with undefined behavior in test case reduction? We follow C-Reduce [24] and leverage (1) GCC and LLVM warnings, (2) KCC [6], and (3) static analysis tools such as Frama-C.[6] We also utilize Clang's (although imperfect) support for undefined behavior sanitization, as well as cross-checking using a few different compilers and compiler

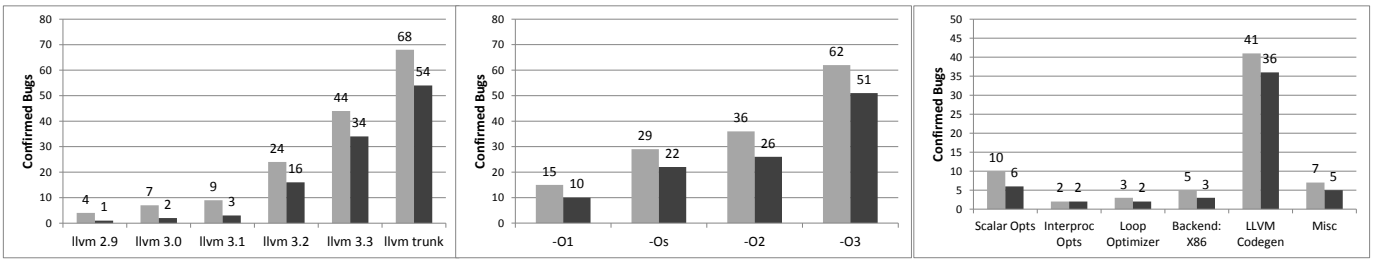
[5] The "Tiny C Compiler" (`http://bellard.org/tcc/`)

[6] `http://frama-c.com/`



(a) Affected versions (b) Affected optimization levels (c) Affected compiler components

Figure 4: Statistics for GCC (*lighter bars*: confirmed bugs; *darker bars*: fixed bugs).



(a) Affected versions (b) Affected optimization levels (c) Affected compiler components

Figure 5: Statistics for LLVM (*lighter bars*: confirmed bugs; *darker bars*: fixed bugs).

configurations to detect inconsistent behavior caused by invalid code (*i.e.* code with undefined behavior). Whenever possible, we use CompCert (and its C interpreter) for detecting and rejecting invalid code.

Number of Variants It is also important to decide how many variants to generate for each program. There is a clear trade-off in performance and bug detection. Our experience suggests that eight variants appear to strike a good balance. In earlier testing, we used a static bound, such as 8, as the number of variants to generate for each program. Later, we added a random parameter that has roughly an expected value of 8 to control the number of generated variants for each test program independently at random. This has been quite effective. For future work, we may explore white-box approaches that support less stochastic, more controlled generation of EMI variants.

We have conducted our testing over an extended period of time. We usually had multiple runs of Csmith with different configurations, especially after we learned that certain Csmith configurations may often lead to test cases with undefined behavior. One such example is the use of unions because code with unions can easily violate the strict aliasing rules, thus leading to undefined behavior. Sometimes, we also needed to restart a Csmith run due to system failures, Csmith updates, bugs in Orion, and re-seeding Csmith's random number generation. As the number of EMI variants that we generated for each test case was also stochastic, we do not have exact counts of the Csmith tests and their derived EMI variants, but both numbers were in the millions to tens of millions range.

4.2 Quantitative Results

This subsection presents various summary statistics on results from our compiler testing effort.

Bug Count We have filed a total of 195 bug reports for GCC and LLVM during our testing period: (1) three derived from compiler test suites, (2) one from existing open-source projects, and (3) the rest from Csmith tests. They can be found under "su@cs.ucdavis.edu" and "dhazeghi@yahoo.com" in GCC's and LLVM's bugzilla databases. Till March 2014, 147 have been confirmed, 110 of which have been resolved and fixed by the developers. Note that when a bug is confirmed and triaged, it corresponds to a new defect. Thus, all confirmed bugs that we reported were unique and independent.

Also note that although we always ensured that all of our reported bugs had different symptoms, some of them were actually linked to the same root cause. These bugs were later marked as duplicate by developers. The remaining 13 bugs — 4 for GCC and 9 for LLVM — have not yet been confirmed as the developers have not left any comments on these reports. One such example is LLVM bug 18447,[7] which was reported on January 11, 2014.

Table 1 classifies the reported bugs across the two tested compilers: GCC and LLVM. It is worth mentioning that we have focused more extensive testing on GCC because of the very quick responses from the GCC developers and relatively slow responses from the LLVM developers (although later we had seen much increased activities from LLVM because of its 3.4 release). This partially explains why we have reported more bugs for GCC over LLVM.

	GCC	LLVM	TOTAL
Reported	111	84	195
Marked duplicate	28	7	35
Confirmed	79	68	147
Fixed	56	54	110

Table 1: Bugs reported, marked duplicate, confirmed, and fixed.

Bug Types We distinguish two kinds of errors: (1) ones that manifest when compiling code, and (2) ones that manifest when the compiled EMI variants are executed. We further classify compile-time bugs into *compiler crashes* (*e.g.* internal compiler errors and memory-safety errors) and *performance bugs* (*e.g.* compiler hang or abnormally slow compilation).

A compile-time crash occurs when the compiler exits with a non-zero status. A runtime bug occurs when an EMI variant behaves differently from its original program. For example, it crashes or terminates abnormally, or produces a different output. We refer to such compiler errors as *wrong code* bugs. Silent wrong code bugs are the most serious, since the program surreptitiously produces wrong result.

[7] `http://llvm.org/bugs/show_bug.cgi?id=18447`

Table 2 classifies the bugs found by Orion according to the above taxonomy. Notice that Orion found many wrong code (more serious) bugs, confirming its strengths in stress-testing compiler optimizers. For example, Csmith found around 40 wrong code bugs in GCC over several years' prior and continuing testing, while Orion found about the same number of wrong code bugs in a much shorter time (and after GCC and LLVM had already fixed numerous bugs discovered by Csmith).

	GCC	LLVM	TOTAL
Wrong code	46	49	95
Crash	23	10	33
Performance	10	9	19

Table 2: Bug classification.

Importance of the Reported Bugs It is reasonable to ask whether the compiler defects triggered by randomly pruning unexecuted code matter in practice. This is difficult to answer and a question that Csmith has also faced. The discussion from the Csmith paper [28] is quite relevant here. First, most of our reported bugs have been confirmed and fixed by the developers, illustrating their relevance and importance (as it often takes substantial effort to fix a miscompilation).

Second, some of our reported bugs were later reported by others when compiling real-world programs. As a recent example, from an EMI variant of a Csmith test, we found a miscompilation in GCC and reported it as bug 59747.[8] Later, others discovered that GCC also miscompiled the movie player `mplayer`, and filed a new bug report 59824.[9] The two bugs turned out to share the same root cause, and subsequently bug 59824 was marked as duplicate.

Affected Compiler Versions We only tested the latest development trunks of GCC and LLVM. When we find a test case that reveals a bug in a compiler, we also check the compiler's stable releases against the same test case. Respectively, Figures 4a and 5a show the numbers of bugs that affect various versions of GCC and LLVM. Obviously both development trunks are the most frequently affected. However, Orion has also found a considerable number of bugs in many stable releases that had been latent for many years.

Optimization Flags and Modes Figures 4b and 5b show which optimization levels are affected by the bugs found in GCC and LLVM. In general, a bug occurs at lower optimization levels is likely to also happen at higher levels. However, we did encounter cases where a bug only affected one optimization flag. In most such cases, the flag is "`-Os`", which is quite intuitive because "`-Os`" is the only flag that optimizes for code size and is less used. Table 3 shows the number of bugs that affected code generated for 32-bit ("`-m32`") and 64-bit ("`-m64`") environments, alone or both.

	GCC	LLVM	TOTAL
`-m32` alone	15	10	25
`-m64` alone	21	18	39
Both	43	40	83

Table 3: Bugs found categorized by modes.

Affected Compiler Components Figures 4c and 5c show which compiler components in GCC and LLVM were affected by the reported bugs respectively. Most of the bugs that Orion found in GCC are optimizer bugs. As for LLVM, the developers do not (or have not) appropriately classify the bugs, so the information we extracted from the LLVM's bugzilla database may be quite skewed, where most have been classified as "LLVM Codegen" thus far.

4.3 Assorted Bug Samples Found by Orion

Orion is capable of finding bugs of diverse kinds. We have found bugs that result in issues like compiler segfaults, internal compiler errors (ICEs), performance issues at compilation, and wrong code generation, and with various levels of severity, from rejecting valid code to release-blocking miscompilations. To provide a glimpse of the diversity of the uncovered bugs, we highlight here several of the more concise GCC and LLVM bugs. The wide variety of bugs presented demonstrates EMI's power and broad applicability.

4.3.1 Miscompilations

We first discuss a few selected wrong code bugs:

Figure 6a: All versions of GCC tested (4.6 to trunk) failed to correctly compile the program shown in Figure 6a in 64-bit mode under `-O3`. The resulting code crashes with a segfault. The reason is believed to be a wrong offset computation in GCC's predictive commoning optimization. The generated code tries to access memory quite far from what it actually should access due to incorrectly generated offsets, causing a segmentation fault.

Figure 6b: When compiling the code in Figure 6b, Clang generated incorrect code, making the program return an incorrect value. The bug is caused by Clang's vectorizer.

Figure 6c: GCC trunk failed to compile the program listed in Figure 6c at `-O1` and above in both 32-bit and 64-bit modes because of a bug in its jump threading logic. The shape of the control-flow graph caused the code to handle jump threads through loop headers to fail.

Figure 6d: Clang trunk failed to compile the test case in Figure 6d and crashed with a segfault under `-O2` and above in both 32-bit and 64-bit modes. The problem was caused by GVN forgetting to add an entry to the leader table when it fabricated a "ValNum" for a dead instruction. Later on, when the compiler wants to access that table entry, it fails with a segfault as the entry is nonexistent.

Figure 6e: The test program in Figure 6e was miscompiled by Clang trunk when optimized for code size (*i.e.* at `-Os`), causing the binary to print "0" when executed where it should have printed "1". The root cause was traced to a bug in the LLVM inliner.

Figure 6f: GCC's vectorizer was not immune to Orion either. It miscompiled the program in Figure 6f, resulting in wrong output from executing the generated code.

4.3.2 Compiler Performance Issues

Orion also helps discover another category of bugs: compiler performance bugs resulting in terribly slow compilation. For instance, it took GCC minutes to compile the program in Figure 7, orders of magnitude slower than both Clang and ICC. Across different versions, GCC 4.8 was considerably faster than trunk, whereas GCC 4.6 and 4.7 were much slower. The performance issue is believed to be caused by loop unrolling while retaining a large number of debug statements ($> 500,000$) within a single basic block that will have to be traversed later.

While Clang was much faster than GCC at compiling the program in Figure 7, it had performance bugs elsewhere. Both Clang 3.3 and trunk failed to perform satisfactorily in compiling the code in Figure 8, taking minutes to compile the code under `-O3`, orders of magnitude longer than `-O2`, GCC, ICC, and even the previous Clang

[8] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=59747`

[9] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=59824`

```
int b, f, d[5][2];
unsigned int c;
int main() {
  for (c = 0; c < 2; c++)
    if (d[b + 3][c] & d[b + 4][c])
      if (f)
        break;
  return 0;
}
```

(a) http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58697: All tested GCC versions generated wrong code that crashed at run-time due to invalid memory access, when compiled at -O3 in 64-bit mode.

```
int main() {
  int a = 1;
  char b = 0;
  lbl:
    a &= 4;
    b--;
    if (b) goto lbl;
  return a;
}
```

(b) http://llvm.org/bugs/show_bug.cgi?id=17532: Clang bug affecting 3.2 and above: the vectorizer generates incorrect code affecting the program's return value in both 32-bit and 64-bit modes. The bug disappears with -fno-vectorize.

```
int a;
int main() {
  int b = a;
  for (a = 1; a > 0; a--);
lbl:
  if (b && a)
    goto lbl;
  return 0;
}
```

(c) http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58343: GCC trunk crashed at -O1 and above in both 32-bit and 64-bit modes with an Internal Compiler Error (ICE) due to the unusual shape of the control-flow graph which causes problems in the jump threading logic and leads to a failure.

```
int *a, e, f;
long long d[2];
int foo() {
  int b[1]; a = &b[0];
  return 0;
}
int bar() {
  for (f = 0; f < 2; f++) d[f] = 1;
  e = d[0] && d[1] - foo();
  if (e) return 0;
  else return foo();
}
```

(d) http://llvm.org/bugs/show_bug.cgi?id=17307: Clang trunk segfaulted when compiled at -O2 or -O3 in both 32-bit and 64-bit modes due to GVN's incorrectly updating the leader table.

```
int printf(const char *, ...);
struct S0 { int f0, f1, f2, f3, f4 }
  b = {0,0,1,0,0};
int a;
void foo(struct S0 p) {
  b.f2 = 0;
  if (p.f2) a = 1;
}
int main() {
  foo(b);
  printf("%d\n", a);
  return 0;
}
```

(e) http://llvm.org/bugs/show_bug.cgi?id=17781: Clang trunk miscompiled this program when optimized for code size (-Os) as a result of an LLVM inliner bug, generating incorrect output.

```
int printf(const char *, ...);
int a[8][8] = {{1}};
int b, c, d, e;
int main() {
  for (c = 0; c < 8; c++)
    for (b = 0; b < 2; b++)
      a[b + 4][c] = a[c][0];
  printf("%d\n", a[4][4]);
  return 0;
}
```

(f) http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58228: GCC vectorizer regression from 4.6 triggers a miscompilation affecting program output under -O3 in both 32-bit and 64-bit modes. The bug goes away with the -fno-tree-vectorize flag.

Figure 6: Example test cases uncovering a diverse array of GCC and LLVM bugs.

```
int a, b, c, d;
int *foo(int *r, short s, short t) { return &c; }
short bar(int p) {
  int t = 0;
  for (a = 0; a < 8; a++)
    for (b = 0; b < 8; b++)
      for (p = 0; p < 8; p++)
        for (d = 0; d < 8; d++) foo(&t, p, d);
  bar (0);
  return 0;
}
int main() { return 0; }
```

Figure 7: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58318: GCC retains many debug statements that will have to be traversed in a single basic block as a result of loop unrolling, causing orders of magnitude slowdown in compilation speed.

release (3.2), which took 9 seconds to compile at -O3. GCC had the fastest compilation — only 0.19 seconds at -O3.

Clang's performance issue was caused by its creation of thousands of stale lifetime marker objects within the compiler that are not properly cleaned up, drastically slowing down compilation.

4.4 Remarks

One of the reasons why Csmith has not been extended to C++ or other languages is because it requires significant engineering efforts to realize. One essentially has to rebuild a new program generator almost entirely from scratch. In contrast, Orion is much easier to be targeted for a new domain. To add C++ support to Orion, we simply need to handle a few more C++ statements and constructs in the generator EMI-gen. Although we have not yet done any substantial

```
int a = 1, b, c, *d = &c, e, f, g, k, l, x;
static int * volatile *h = &d;
static int * volatile **j = &h;
void foo(int p) { d = &p; }
void bar() {
  int i;
  foo (0);
  for (i = 0; i < 27; ++i)
    for (f = 0; f < 3; f++)
      for (g = 0; g < 3; g++) {
        for (b = 0; b < 3; b++)
          if (e) break;
        foo (0);
      }
}
static void baz() {
  for (; a >= 0; a--)
    for (k = 3; k > 0; k--)
      for (l = 0; l < 6; l++) { bar (); **j = &x; }
}
int main() { baz(); return 0; }
```

Figure 8: http://llvm.org/bugs/show_bug.cgi?id=16474: It takes Clang 3.3+ minutes to compile at -O3, compared to only 0.19 seconds with GCC 4.8.1. The performance issue is caused by the creation of a large number of dead lifetime marker objects in the compiler that are not cleaned up.

testing of Orion's C++ support, our preliminary experience has been encouraging as Orion has already uncovered potential latent compiler bugs using small existing C++ code.

Even for a completely new domain, EMI also requires much less engineering effort because it can leverage existing tools and

infrastructures. Our active ongoing work on adapting Orion to testing the JVM JIT confirms this.

Currently, Orion only focuses on integer C programs. We plan to extend the work to floating-point programs. This direction is new and exciting, and our EMI methodology offers a promising high-level approach. The key technical challenge is to define the "equivalence" of floating-point EMI variants considering the inherent inaccuracy of floating-point computation.

5. Related Work

Our work is related to the large body of research on compiler testing and verification. This section surveys some representative, closely related work, which we group into three categories: (1) compiler testing, (2) verified compilers, and (3) translation validation.

Compiler Testing The most directly related is compiler testing, which remains the dominant technique for validating production compilers in practice. One common approach is to maintain a compiler test suite. For example, each major compiler (such as GCC and LLVM) has its own regression test suite, which grows over the time of its development. In addition, there are a number of popular, commercial compiler test suites (*e.g.* Plum Hall [21] and SuperTest [1]) designed for compiler conformance checking and validation. Most of these test suites are written manually.

Random testing complements manually written test suites. Zhao *et al.* [29] develop JTT, a tool that automatically generates test programs to validate the EC++ embedded compiler. It takes as input a test specification (*e.g.* optimizations to be tested, data types to use, and statements to include), and generates random programs to meet the given specification. Recent work by Nagai *et al.* [18, 19] focuses on testing C compilers' arithmetic optimizations by carefully generating random arithmetic expressions to avoid undefined behavior. As of November 2013, they have found seven bugs each for GCC and LLVM. Another notable recent random C program generator is CCG [2], which targets only compiler crashes.

Csmith [4, 24, 28] has been the most successful random testing system for C compilers. It has helped find a few hundred compiler bugs over the last several years and contributed significantly to improving the quality of GCC and LLVM. It is based on differential testing [16] by randomly generating C programs and checking for inconsistent behavior across compilers or compiler versions. What make it stand out from other random C program generators are the many C language features it supports and its careful control to avoid generating programs with undefined behavior. Thus, in addition to compiler crashes, it is suitable for finding miscompilations. Csmith has also been applied to find bugs in static analyzers, for example, in Frama-C [5].

Orion is complementary. Different from Csmith-like tools, it does not generate random programs, but rather consumes existing code (whether real or randomly generated) and systematically modifies it. EMI variants generated from existing code, say via Orion, are likely programs that people may actually write. The EMI concept is general and can be adapted to any program analysis and transformation systems. Its simplicity makes it easy to implement for a new domain — there is no need to specifically craft a new program generator each time.

Holler *et al.*'s recent work on LangFuzz [10] is also related. It is a random generator that uses failing programs as stems for producing test programs. LangFuzz has found many bugs in the PHP and Mozilla JavaScript interpreters. The same idea may be adapted to Orion. As we have already experimented in this work, problematic programs (such as those from the GCC and LLVM test suites) can be used as seeds to generate their EMI variants. We can also incorporate them in generating other programs' EMI variants, which we plan to investigate in our future work.

Verified Compilers A decade ago, "the verifying compiler" was proposed as a grand challenge for computing research [9]. Compiler verification in particular has been a fruitful area for this grand challenge. A verified compiler ensures that the semantics of a compiled program is preserved. Each verified compiler is accompanied by a correctness proof that guarantees semantic preservation. The most notable example is CompCert [12, 13], a verified optimizing compiler for a sizable C subset. Both the compiler itself and the proof of its correctness have been developed using the Coq proof assistant. The same idea has been applied to the database domain. In particular, there is some early work toward building a verified relational database management system [14]. There is also recent work by Zhao *et al.* [30] on a proof technique to verify SSA-based optimizations in LLVM using the Coq proof assistant.

The benefits of verified compilers are clear because of their strong guarantee of semantic preservation. Despite considerable testing, neither Csmith nor Orion has uncovered a single CompCert back-end bug to date. This is a strong testimony to the promise and quality of verified compilers. However, techniques like Csmith and Orion are complementary as much work remains to build a realistic production-quality verified compiler. CompCert, for example, currently supports fewer language constructs and optimization techniques than GCC and LLVM (thus is less performant). These make verified compilers mainly suitable for safety-critical domains that may be more willing to sacrifice performance for increased correctness guarantees.

Translation Validation It is difficult to automatically verify that a compiler correctly translates every input program. However, it is often much easier to prove that a particular compilation is correct, which motivated the technique of translation validation [22]. In particular, the goal of translation validation is to verify the compiled code against the input program to find compilation errors on-the-fly. Early work on translation validation focuses on transformations among different languages. For example, Pnueli *et al.*'s seminal work [22] that introduced translation validation considers the non-optimizing compilation from SIGNAL to C.

Subsequent work by Necula [20] extends this technique to handle optimizing transformations and validates four optimizations in GCC 2.7. Extending work on super-optimization [3, 11, 15], in particular Denali [11], Tate *et al.* introduce the Peggy optimization and translation validation framework for JVM based on equality saturation [25]. Tristan *et al.* [27] adapt the work and evaluate it on validating intra-procedural optimizations in LLVM.

Although promising, translation validation is still largely impractical in practice. Current techniques focus on intra-procedural optimizations, and it is difficult to handle optimizations at the inter-procedural level. In addition, each validator is attached to a particular implementation of an optimizer, thus changes in the optimizer may require appropriate changes in the validator. Since the validator is not verified, it may also produce wrong validation results.

6. Conclusion and Future Work

We have described a new validation methodology — equivalence modulo inputs (EMI) — for testing compilers. The distinctive benefits of EMI are its simplicity and wide applicability. We have applied it to test optimizing C compilers. Our evaluation has provided strong evidence of EMI's impressive power: 147 confirmed, unique bug reports for GCC and LLVM in a short few months.

This work complements decades of extensive work on compiler validation and verification by opening up a fresh, exciting avenue of research. We are actively pursuing future work to refine the general approach and extend it to other languages (such as C++ and JVM) and settings (such as database engines, interpreters, and program analysis and transformation systems in general).

Acknowledgments

We would like to thank our shepherd, Atanas Rountev, and the anonymous PLDI reviewers for valuable feedback on earlier drafts of this paper. We thank Dara Hazeghi for his early help in setting up the experimental infrastructure, and reducing and reporting bugs. Our special thanks go to the GCC and LLVM developers for analyzing and fixing our reported bugs.

The experimental evaluation benefited tremendously from University of Utah's Csmith [28] and C-Reduce [24] tools, and the Berkeley Delta [17]. We thank John Regehr, Eric Eide, Alex Groce, Xuejun Yang, and Yang Chen for helpful comments on this work. In particular, we gratefully acknowledge John Regehr for his helpful advice on using Csmith, for pointers to Nagai *et al.*'s work [18, 19], and for his insightful blog posts on testing C compilers [23].

This research was supported in part by NSF Grants 0917392, 1117603, 1319187, and 1349528. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] ACE. SuperTest compiler test and validation suite. `http://www.ace.nl/compiler/supertest.html`.

[2] A. Balestrat. CCG: A random C code generator. `https://github.com/Merkil/ccg/`.

[3] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403, 2006.

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[5] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.

[6] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 533–544, 2012.

[7] GNU Compiler Collection. gcov. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[8] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–88, 2012.

[9] T. Hoare. The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*, pages 25–35. Springer, 2003.

[10] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[11] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314, 2002.

[12] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.

[13] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[14] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[15] H. Massalin. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[16] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[17] S. McPeak, D. S. Wilkerson, and S. Goldsmith. Berkeley Delta. `http://delta.tigris.org/`.

[18] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[19] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

[20] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.

[21] Plum Hall, Inc. The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`.

[22] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[23] J. Regehr. Embedded in academia. `http://blog.regehr.org/`.

[24] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[25] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, 2009.

[26] The Clang Team. Clang 3.4 documentation: LibTooling. `http://clang.llvm.org/docs/LibTooling.html`.

[27] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–305, 2011.

[28] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[29] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test (AST)*, pages 36–43, 2009.

[30] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186, 2013.