PLDI'12, ACM, pp 335-346. 2012

Test-Case Reduction for C Compiler Bugs

John Regehr
University of Utah
regehr@cs.utah.edu

Yang Chen
University of Utah
chenyang@cs.utah.edu

Pascal Cuoq
CEA LIST
pascal.cuoq@cea.fr

Eric Eide
University of Utah
eeide@cs.utah.edu

Chucky Ellison
University of Illinois
celliso2@illinois.edu

Xuejun Yang
University of Utah
jxyang@cs.utah.edu

Abstract

To report a compiler bug, one must often find a small test case that triggers the bug. The existing approach to automated test-case reduction, delta debugging, works by removing substrings of the original input; the result is a concatenation of substrings that delta cannot remove. We have found this approach less than ideal for reducing C programs because it typically yields test cases that are too large or even invalid (relying on undefined behavior). To obtain small and valid test cases consistently, we designed and implemented three new, domain-specific test-case reducers. The best of these is based on a novel framework in which a generic fixpoint computation invokes modular transformations that perform reduction operations. This reducer produces outputs that are, on average, more than 25 times smaller than those produced by our other reducers or by the existing reducer that is most commonly used by compiler developers. We conclude that effective program reduction requires more than straightforward delta debugging.

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

Keywords compiler testing, compiler defect, automated testing, random testing, bug reporting, test-case minimization

1. Introduction

Although many compiler bugs can be demonstrated by small test cases, bugs in released compilers are more typically discovered while building large projects. Before a bug can be reported, the circumstances leading to it must be narrowed down. The most important part of this process is *test-case reduction*: the construction of a small input that triggers the compiler bug.

The importance of test-case reduction is emphasized in the GCC documentation,[1] which states that:

> Our bug reporting instructions ask for the preprocessed version of the file that triggers the bug. Often this file is very large; there are several reasons for making it as small as possible…

The instructions for submitting bug reports to the LLVM developers also highlight the importance of test-case reduction.[2] Indeed, LLVM ships with the Bugpoint tool[3] that automates reduction at the level of LLVM IR. The importance that compiler developers place on small test cases stems from the simple fact that manual test-case reduction is both difficult and time consuming. With limited time to spend fixing bugs, compiler writers require bug reporters to undertake the effort of reducing large fault-causing inputs to small ones.

Like debugging, distilling a bug-causing compiler input to its essence is often an exercise in trial and error. One must repeatedly experiment by removing or simplifying pieces of the input program, compiling and running the partially reduced program, and backtracking when a change to the input causes the compiler bug to no longer be triggered. In some cases—for example, reducing a deterministic assertion failure in the compiler—manual test-case reduction is tedious but tractable. In other cases—e.g., reducing a miscompilation bug in a large, multi-threaded application—manual test-case reduction may be so difficult as to be infeasible. Our belief is that many compiler bugs go unreported due to the high difficulty of isolating them. When confronted with a compiler bug, a reasonable compiler user might easily decide that the most economic course is to find a workaround, and not to be "sidetracked" by the significant time and effort required to produce a small test case and report the bug.

Our goal in this paper is to automate most or all of the work required to reduce bug-triggering test cases for C compilers. Our work is motivated by two problems that we have encountered in applying state-of-the-art reducers. First, these tools get stuck at local minima that are too large. This necessitates subsequent manual reduction, preventing reportable compiler bugs from being generated in an entirely automated fashion. We have developed new reducers that use domain-specific knowledge to overcome the barriers that trap previous tools. Second, existing test-case reducers often generate test cases that execute *undefined behaviors*. These test cases are useless because the C language standard guarantees

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06…$10.00

[1] http://gcc.gnu.org/bugs/minimize.html
http://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[2] http://llvm.org/docs/HowToSubmitABug.html

[3] http://llvm.org/docs/Bugpoint.html

```
int printf (const char *, ...);
union U5 {
  short f1;
  int f3:13;
};
const union U5 a = { 30155 };
int
main () {
  printf ("%d\n", a.f1);
  printf ("%d\n", a.f3);
  return 0;
}
```

Listing 1. Test case that demonstrates a bug in GCC 4.4.0

nothing about the behavior of such programs. For example, all C/C++ compiler teams we have interacted with will ignore (in the best case) any bug report that depends upon the value of an uninitialized local variable. Our new reducers confront this *test-case validity problem* directly: they avoid undefined test-case behaviors during reduction, thereby ensuring that the final reduced test cases are suitable for bug reporting.

Our work is also motivated by Csmith, a random test-case generator that has resulted in more than 400 previously unknown C compiler bugs being reported [17]. Using randomized differential testing, Csmith automates the construction of programs that trigger compiler bugs. These programs are large out of necessity: we found that bug-finding was most effective when random programs' average size was 81 KB [17, §3.3]. In this paper, we use 98 bug-inducing programs created by Csmith as the inputs to several automated program reducers, including three written by ourselves. Two of the new reducers are implemented as Csmith add-ons and can only reduce Csmith-generated programs. However, the most effective reducer (in terms of size of output) that we developed is generic: it can reduce any C program, whether generated by Csmith or not. In the great majority of cases, one or more of our reducers can distill the large (37–297 KB) test cases created by Csmith into small test cases (typically less than 0.5 KB) that preserve the bug-triggering features of the original inputs. Our reducers do this automatically and without introducing undefined behavior into the reduced test cases.

Our contributions are as follows:

1. We identify existing delta-debugging algorithms as point solutions in a more general framework. We present three new, domain-specific test-case reducers for C code that fit into this framework.
2. We identify the *test-case validity problem* as a crucial part of test-case reduction for compilers of programming languages that admit undefined and unspecified behaviors. This problem has not been acknowledged or solved in previous work. We present various solutions to the test-case validity problem.
3. We show that our best reducer produces output more than 25 times smaller than that produced by a line-based delta debugger, on average.

2. Reporting Compiler Bugs

An ideal compiler bug report is based on a small, easy-to-read, self-contained, and well-defined test program. This paper is about automatically constructing programs that meet these criteria. Other elements found in a good bug report include:

- the identification of the buggy compiler: its version, its target machine and OS, how it was built, and so on;
- instructions for reproducing the failure, including a description of how the compiler was invoked; and
- the expected and actual outputs of the compiled test program (assuming that the compiler does not fail to compile the test).

For example, the following report is informative enough that it would likely be acted upon by compiler developers:

On x86-64, the program in Listing 1 should print:

```
30155
-2613
```

However, using GCC 4.4.0 (built from scratch) and also GCC "Ubuntu 4.4.3-4ubuntu5" (the current vendor-supplied compiler on Ubuntu 10.04.3 LTS as of Nov 6 2011) on x86-64, we get:

```
$ gcc -O1 small.c
$ ./a.out
30155
30155
```

The program in Listing 1 is the verbatim output from one of our C program reducers. The underlying problem is an actual GCC bug that affects Ubuntu 10.04, a major and currently supported Linux distribution.

The program in Listing 1 is free from undefined and unspecified behaviors,[4,5] although it does involve implementation-defined behavior regarding the representations of data objects. Implementation-defined behaviors are allowed in test cases; indeed, in C, such basic things as the ranges of `int` and `char` are implementation-defined.

Although the example at hand may seem to involve the "technical minutiae" of C, it is important for compilers to get these details right. Large programs such as operating systems and virtual-machine monitors depend on subtle behaviors like those executed by Listing 1. Automated test-case reduction, when successful, allows one to obtain small test cases that cleanly present the essence of a problem.

3. Background

In this section we distinguish test-case minimization from test-case reduction. We also summarize delta debugging, an existing approach to test-case reduction.

3.1 The Test-Case Minimization Problem

Let I be the set of all valid inputs to some system under test (SUT). For $i \in I$, let $|i|$ be the size of i according to an appropriate metric such as bytes or tokens. Let $B \subseteq I$ be the set of inputs that trigger a particular failure of interest in the SUT, e.g., a crash at a particular location in the SUT. The *test-case minimization problem* for a SUT and a particular failure is to find i_{Bmin} where $i_{Bmin} \in B$ and $\forall i \in B, |i| \geq |i_{Bmin}|$. Note that i_{Bmin} may not be unique; there may be several minimum-sized inputs that result in the failure of interest. For the sake of clarity, however, we describe the search for i_{Bmin} as if it were unique.

[4] The C99 language standard [7] identifies many program behaviors that are *undefined* or *unspecified*. An example undefined behavior is dereferencing a null pointer. When a C program executes an undefined behavior, "anything can happen": the C standard places no requirements on the C implementation. Unspecified behaviors, on the other hand, permit the compiler to choose from a variety of alternatives with no requirement for consistency. An example unspecified behavior is the order in which the arguments of a function call are evaluated. Like C programs in general, good test cases for C compilers must not execute any undefined behaviors and must not depend on a particular choice for unspecified behaviors.

[5] Footnote 82 in the C99 language standard allows `a.f3` to be accessed by the second `printf()` call [7, §6.5.2.3]. This footnote, introduced in TC3, supersedes language in Annex J of the previous C99 standard which states that "the value of a union member other than the last one stored into" is unspecified. Subtle interactions between different parts of the standard—and even outright contradictions, as we see here—are not uncommon when interpreting tricky C programs.

Consider the task of finding i_{Bmin} given only an initial failing test case i_{seed}. Without knowledge of the internals of the SUT, the only information that i_{seed} provides is that it triggers the failure of interest and therefore defines a maximum value of $|i_{Bmin}|$. The reason that i_{seed} causes the failure of interest is unknown, and thus in principle, any input that is smaller than i_{seed} may also trigger the failure of interest. The only way to find i_{Bmin}, which is the absolute smallest failure-inducing input, is exhaustive search.

Let $I_{<S}$ be the set of inputs not larger than a size bound S. For realistic systems under test, one can expect that $|I_{<S}|$ is an exponential function of S. Finding i_{Bmin} requires testing increasingly large members of I until a failure-inducing input is found. Thus, it is easy to see that the test-case minimization problem is intractable unless $|i_{seed}|$ is quite small. Consequently, all test-case minimization work that we are aware of is heuristic.

An analogy can be made between test-case minimization and compiler optimization. In both situations, the problem is to find an element in a large set of programs that has a desired property while also minimizing a cost function. The major difference is that while optimization can be phrased as a relatively concise mathematical problem ("find the cheapest machine program that is semantically equivalent to the input program"), test-case minimization is conducted in the absence of an effective semantics: the minimizer lacks a model for predicting which inputs will trigger the failure of interest in the SUT. The exact program-optimization problem can be solved for small programs by encoding the meaning of the program mathematically and then passing the problem to a SAT/SMT solver [8]. In contrast, an analogous solution to the test-case minimization problem would require either a brute-force search of $I_{<|i_{seed}|}$ or else a mathematical encoding of the entire SUT.

Heuristic solutions to the test-case minimization problem start with a failure-inducing input i_{seed} and transform it. Through repeated transformation and testing, they search for ever-smaller inputs that cause the failure of interest in the SUT. We refer to this process as *test-case reduction* so as to distinguish it from absolute minimization. The goal of a reducer is to create a small test case from a large one; a reducer R_1 can be said to be better than a reducer R_2 if, over a given set of test cases, R_1 consistently produces smaller outputs than R_2 does. Our focus in this paper is on high-quality test-case reducers for C compilers, i.e., reducing C programs.

3.2 Delta Debugging

Although point solutions to the test-case reduction problem have existed for some time (see Section 8), a generic solution was not formulated until Zeller and Hildebrandt [18] developed two *delta debugging* algorithms. The *dd* algorithm seeks to minimize the difference between a failure-inducing test case and a given template; the *ddmin* algorithm is a special case of *dd* where the template is empty. Thus, *ddmin*'s goal is to minimize the size of a failure-inducing test case.

Ddmin heuristically removes contiguous regions ("chunks") of the test in order to generate a series of *variants*. *Unsuccessful variants* are those that do not trigger the sought-after behavior; all unsuccessful variants are discarded. *Successful variants*, on the other hand, are those that trigger the desired behavior. Each successful variant is used as the new basis for producing future variants; in other words, the search is greedy. When no successful variants can be generated from the current basis, the chunk size is decreased. The algorithm terminates when the chunk size cannot be further decreased and no more successful variants can be produced. The final result is the last successful variant that was produced; by the nature of the search, this is the smallest variant that produces the desired behavior. *Ddmin* can be fast even for large test cases when it is successful in removing large chunks early in its execution. If it cannot remove large input chunks, however, *ddmin* can take a long time to terminate—especially if it is expensive to discriminate between successful and unsuccessful variants.

Delta debugging was generalized to create *hierarchical delta debugging* (HDD) [13], in which chunk selection is guided by the hierarchical structure of the input. For several failure-inducing C and XSLT programs, HDD was shown to significantly reduce reduction time and also produce better results, when compared with *ddmin*.

McPeak and Wilkerson [12] implemented a variant of *ddmin*—which we refer to as "Berkeley delta" in this paper—that is both well-known and commonly used by compiler developers. Berkeley delta is generic and line-based: all variants are produced by removing one or more contiguous lines from the test input. With the help of a separate utility, called *topformflat*, Berkeley delta can be used to perform hierarchical delta debugging over program source code. The *topformflat* utility preprocesses an input test case and "flattens languages with balanced delimiters, such as most common programming languages, so that all nesting below a specified depth is on one line."[6] This is simple and effective.

4. Generalized Delta Debugging

Our observation is that algorithms such as *dd*, *ddmin*, and HDD are specific instantiations of a simple and powerful framework for reducing test cases. Most of the hard-coded aspects of these existing algorithms can be usefully generalized in order to more effectively reduce failure-inducing inputs, including the following.

Generalized transformations *ddmin* uses text-oriented chunk removal to create variants, and HDD uses an AST-oriented removal strategy. The former removes a substring with each search step, and the latter removes a subtree. However, our experience is that reducing a failure-triggering C program to a very small size requires a much richer set of transformations such as removing an argument from a function and simultaneously all its call sites, performing inline substitution of a function body, replacing an aggregate type with its constituent scalars, and so on. These transformations may operate on many scattered program points in one step, and they cannot be described as operations that simply delete a substring or subtree.

Generalized search *ddmin*'s and HDD's search strategies are greedy. This is simple and efficient but ignores a wealth of research on search that may avoid local minima by being non-greedy or improve search-termination speed by not looking at alternatives that are unlikely to be profitable. Additionally, transformations such as inlining the body of a function or splitting up a complex expression may increase code size (by adding references to temporary variables) while ultimately leading to better output. A greedy search rules out exploration of these variants, and thus tends to become stuck at a local minimum where no further substring or subtree eliminations are possible.

Addressing test-case validity *ddmin* and related algorithms ignore the *test-case validity problem*: the fact that in some use cases, some variants will be incorrect in ways that are not or cannot be detected by the system under test. This is the case when reducing C programs. For example, when a C compiler is given a program to compile, the compiler cannot in general decide if the program will dereference a null pointer or perform an out-of-bounds array access when the program is run. Both are undefined behaviors and must not happen in a valid compiler test case. Thus, for reducing compiler test cases, a delta-based reducer cannot rely on the compiler (the system under test) to detect when a variant is invalid. Sophisticated solutions are required to overcome this issue.

[6] http://delta.tigris.org/using_delta.html

Generalized fitness functions The goal of *ddmin*, HDD, and other delta debugging algorithms is to create a minimum-sized failure-inducing test. In general, however, any fitness function can be used to characterize preferable test cases. For example, since the eventual consumer for a reduced test case is a human being, it is undesirable to create tiny reduced programs if they are hard to understand. Consider these simple examples. First, almost any C program can be made smaller, but harder to read, by eliminating whitespace and playing preprocessor tricks. Second, our experience is that turning a union type into a struct can make a test case considerably easier to understand while also making it (at least) one byte larger.

In summary, *generalized delta debugging* refers to any iterative optimization strategy for simplifying the circumstances leading to program failure. A generalized delta debugger combines a search algorithm, a transformation operator, a validity-checking function, and a fitness function.

5. Test-Case Validity

Ensuring that variants are valid C programs was the most serious problem we faced doing this work. Beyond statically ill-formed C programs, *dynamically* invalid programs are those that execute an operation with undefined behavior or rely on unspecified behavior.

5.1 The Validity Problem

Consider this program:

```
int main (void) {
  int x;
  x = 2;
  return x + 1;
}
```

Assume that compiler A emits code that properly returns 3 while compiler B is buggy and generates code returning a different result. The goal of a test-case reducer is to create the smallest possible program triggering the bug in B. During reduction many variants will be produced, perhaps including this one where the line of code assigning a value to `x` has been removed:

```
int main (void) {
  int x;
  return x + 1;
}
```

This variant, however, is not a valid test case. Even if this variant exhibits the desired behavior—compilers A and B return different results—the divergence is potentially due to its reliance on undefined behavior: reading uninitialized storage. In fact, on a common Linux platform, GCC and Clang emit code returning different results for this variant, even when optimizations are disabled. Because delta debugging is an iterative optimization algorithm, once a test-case reduction goes wrong in this fashion, it is likely to remain stuck there, abandoning the original sense of the search. Compiler developers are typically most unhappy to receive a bug report whose test input relies on undefined or unspecified behavior. This is not a hypothetical problem—a web page for the Keil C compiler states that "Fewer than 1% of the bug reports we receive are actually bugs."[7]

In this specific case we might redefine the test-case criterion to be: "Compilers A and B generate code returning divergent results, and neither compiler warns about using an uninitialized variable." In this case, the solution would likely be successful. On the other hand, compiler warnings about uninitialized storage are typically unsound in the presence of function calls, arrays, and pointers. Furthermore, even after we solve this problem, the C99 standard describes 190 more kinds of undefined behavior that must be addressed to ensure that a test case is valid. As previously noted, not all of these behaviors are statically detectable by a compiler. Beyond undefined behavior, there are also many kinds of unspecified behavior that cause similar problems.

In reducing C programs, we have found the most problematic behaviors to be:

- use before initialization of function-scoped storage;
- pointer and array errors;
- integer overflow and shift past bitwidth;
- struct and union errors; and
- mismatches between `printf`'s arguments and format string.

Many languages including C, C++, C# (in unsafe code), and Scheme have some form of non-determinism or unspecified behavior, meaning the test-case validity problem is generally applicable in other languages. (There is current work toward cataloging problematic undefined and unspecified behaviors for many languages [14].) Providing a solution to the test-case validity problem for C, with its hundreds of undefined and unspecified behaviors, suggests that this can be done for other languages as well.

5.2 Solutions

There are two ways to avoid accepting variants that rely on undefined and unspecified behavior. First, the test-case reducer can avoid generating incorrect variants. We do not know how to implement such a reducer for general C programs. However, we have created two reducers that rely on the fact that Csmith already knows how to create conforming C programs [17]. Through extensions to Csmith, we can use Csmith to generate smaller versions of programs it has previously output. The second approach is to blindly generate a mix of valid and invalid variants and then use an external tool to detect invalid code. Sound, automatic static analyzers for C code have been available for some time, but we know of none that can give a large random program a "clean bill of health." To reliably avoid false positives, a semantics-checking C interpreter is needed. Two of these have recently become available.

5.2.1 KCC

KCC [6] is a semantics-based interpreter and analysis tool for C. In addition to interpretation, it is capable of debugging, catching undefined behaviors, state space search, and model checking. All of these modes are mechanically generated from a single formal semantics for C. KCC can catch many undefined behaviors by virtue of the semantics "getting stuck" when a program is not well defined. While KCC does not detect all undefined behavior, it is capable of catching the behaviors listed in Section 5.1.

No major changes were needed in KCC to make it useful in test-case reduction. However, we added English-language error messages for most of the common undefined behaviors, which made it easier to understand exactly how variants go wrong. Additionally, we added detectors for some previously-uncaught undefined behaviors to the tool because those behaviors were found in variants. Any errors reported by KCC are guaranteed to be real errors in the program, under the assumption that the underlying semantics accurately captures C.

5.2.2 Frama-C

Frama-C [5] is an open-source, extensible, static-analysis framework for C. It features a value-analysis plug-in [2]: an abstract interpreter roughly comparable to Polyspace [10] and Astrée [1]. The value analysis uses non-relational domains adapted to the C language; it soundly detects and warns about a sizable set of C's undefined

[7] `http://www.keil.com/support/bugreport.asp`

and unspecified behaviors. At the same time, it is designed to avoid warning about constructs that are technically undefined, but that programmers use intentionally, for example in embedded code. Conflicts between these objectives are resolved according to the following principle: if a compiler implementer would use the undefinedness of a construct as a reason to not fix a compiler bug, then the value analysis warns about the construct.

We were able to leverage the fact that programs generated by Csmith are *closed*—they take no external inputs—in order to avoid pessimism in Frama-C. By indefinitely deferring joins in the value analysis, Frama-C can effectively be as precise as a C interpreter, propagating a singleton abstract state all the way to the end of the program. This greatly increased precision comes at a cost of extended analysis time and memory consumption. We added an efficient interpreter mode to the value analysis, so as to diagnose terminating closed programs without the usual memory and time overhead caused by indefinitely postponed joins. Additionally, we fixed [4] several Frama-C bugs that interfered with the correct interpretation of Csmith-generated programs.

5.2.3 A Hybrid Solution

KCC and Frama-C are capable of detecting subtle dynamic violations of the C standard. They both assume their inputs to be compilable C programs; however, some of the delta debugging algorithms that we used to reduce C programs (e.g., Berkeley delta) produce variants are not even syntactically valid. Therefore, in practice we employ a hybrid solution for detecting invalid code. First, we compile a variant using any convenient C compiler, rapidly rejecting it if it fails to compile or generates warnings that reliably correspond to misbehavior. Second, we optionally examine the variant using Valgrind and/or the Clang static analyzer, again looking for specific warnings or errors that reliably indicate incorrect code. Finally, surviving variants are tested using KCC and/or Frama-C.

5.2.4 Validity for Compiler-Crash Inputs

Test cases for compiler-crash bugs—where the compiler process returns a non-zero status code to the operating system—may have weaker validity requirements than do the test cases for wrong-code bugs. As a matter of implementation quality, a compiler vendor will usually fix a segmentation fault or similar problem even if the crash-inducing test case, for example, uses a variable without initialization. Relaxing the correctness criterion permits crash-inducing inputs to be smaller and to be produced more rapidly. We take advantage of these facts when reducing C programs that cause compiler-crash bugs.

6. New Approaches to Test-Case Reduction

We implemented three new reducers for C programs. Two of them only work for programs generated by Csmith, while the third is general-purpose and would work in the reduction of any C program. (In fact, it can also be used to reduce C++ programs, though it has not yet been tuned for that purpose.)

All three of our reducers adopt Berkeley delta's convention of being parameterized by a test that determines whether a variant is successful or unsuccessful. The test determines if the variant triggers the compiler bug and also, if necessary, determines whether it is statically and dynamically valid C code.

6.1 Reduction in Csmith by Altering the Random Number Sequence

A program generated by Csmith is completely determined by a sequence of integers that describes a path through Csmith's decision tree. Usually this sequence comes from a pseudo-random number generator (PRNG), but it does not need to. Our *Seq-Reduce* test-case reducer bypasses the PRNG in order to generate variants. The result is guaranteed to be a valid C program, so no external validity-checking tool is required.

Seq-Reduce's implementation is split between a driver and two special-purpose Csmith modes. Together, these components act as a generalized delta-debugging loop in which Csmith produces variants and the driver determines whether they are successful or not.

First, the driver launches Csmith with a command telling it to dump the specification of the failure-inducing input to a file. Next, the driver repeatedly invokes Csmith using a command that tells it to load the saved sequence, modify it randomly, and then dump both the new program and its specification. If the new program constitutes a successful variant (i.e., it triggers the compiler failure and it is smaller than the previous smallest variant), this becomes the new program specification. Otherwise, Seq-Reduce rolls back to the previous successful variant. Seq-Reduce is the only randomized test-case reducer that we are aware of.

Seq-Reduce has several advantages. First, it has low implementation complexity, adding about 750 lines of code to Csmith; only 30 of these are at all entangled with Csmith's main logic. The driver is 300 lines of Perl. Second, Seq-Reduce is embarrassingly parallel. For example, for the experiments in Section 7, we ran it on four cores.

The main problem with Seq-Reduce is that it does not do a good job reducing programs in cases where the problematic code is generated late in Csmith's execution. Changes that appear near the start of a Csmith program specification tend to perturb Csmith's internal state in ways that prevent it from emitting desirable variants later on. Additionally, Seq-Reduce has no obvious termination criterion. In practice we use a timeout to terminate Seq-Reduce.

6.2 AST-Based Reduction in Csmith Using Run-Time Information

Our next reducer, *Fast-Reduce*, is also based on Csmith. It explores the idea that test-case reduction can benefit from run-time information. As suggested by its name, Fast-Reduce is intended to give fairly good results in as little time as possible. It is structured as an add-on to Csmith that can query not only the static structure of the generated program, but also its runtime behavior. Dynamic queries are implemented by instrumenting Csmith's generated code, resulting in machine-readable output when the test case is executed. Fast-Reduce supports a number of transformations; we describe three representative ones.

Dead-code elimination Programs generated by Csmith tend to contain a lot of dead code. In many cases, dead code can be removed without "breaking" a test case; Fast-Reduce attempts to do so early in its execution because this tends to give good results quickly. First, Fast-Reduce issues a static query to find which basic blocks are (conservatively) reachable from `main`. Next, it issues a dynamic query to discover which reachable blocks are actually executed. If removing dead code all at once results in an unsuccessful variant, Fast-Reduce rolls back the change and attempts to remove dead code piecewise.

Exploiting path divergence Some wrong-code bugs affect control flow: the miscompiled executable follows a different execution path than does a correctly compiled executable. Fast-Reduce uses differential testing to actively look for this kind of divergence while reducing programs because it provides a strong clue about the location of the critical piece of code in the test case that triggers miscompilation.

Effect inlining Removing the body of a large function is one of the biggest wins available to a C program reducer. Fast-Reduce can attempt to remove a function even when call sites to the function exist; its strategy is to render the function unnecessary by replacing each call with inline code that has the same dynamic effect as the

```
current = original_test_case
while (!fixpoint) {
    foreach t in transformations {
        state = t::new ()
        while (true) {
            variant = current
            result = t::transform (variant, state)
            if (result == stop)
                break
            /* variant has behavior of interest
               and meets validity criterion? */
            if (is_successful (variant))
                current = variant
            else
                state = t::advance (current, state)
        }
    }
}
```

Listing 2. The C-Reduce algorithm

function call. First, Fast-Reduce issues a static query to find all call sites for the target function. Second, it selects a call site and issues a dynamic query to record the values of global variables before and after each execution of that site. (This includes the values of pointer-typed variables.) If the call site is executed multiple times, an arbitrary final effect is selected. Fast-Reduce then replaces the call with its effect, generating a variant. This variant, and others generated by performing the same transformation at other call sites and other target functions, are all tested. When the body of a function finally becomes unreferenced, Fast-Reduce attempts to remove it.

The advantages of Fast-Reduce are that it requires no external validity-checking tool and it is very fast. Its primary disadvantage is that it does not always provide good results; sometimes it gets stuck quite early. Out of the 3,000 lines of C++ comprising Fast-Reduce, about 300 are significantly entangled with Csmith. Fast-Reduce also includes a driver component that orchestrates the reduction; it is about 1,700 lines of Perl.

6.3 A Modular Reducer

Our third new reducer, *C-Reduce*, exploits the insight that the transformations used to construct variants do not need to be hard-coded. C-Reduce invokes a collection of pluggable transformations until a global fixpoint is reached; pseudocode for C-Reduce is shown in Listing 2.

A transformation that plugs into C-Reduce is simply an iterator that walks through a test case performing source-to-source alterations. A transformation must implement three functions. The first, `new`, takes no parameters and returns a fresh transformation state object. The second, `transform`, takes a state object and a path to a test case; it modifies the test case in place and returns a status code that is either:

- `ok`, indicating that a transformation was performed; or
- `stop`, indicating that the transformation has run out of opportunities for the test case at hand.

The third, `advance`, takes a state object and a path to a test case; it advances the iterator to the next transformation opportunity.

As shown in Listing 2, C-Reduce calls `advance` only upon detecting an unsuccessful variant. A successful variant does not require advancing the iterator because it is assumed that an opportunity for transformation has been eliminated from the transformed test case. This means that for C-Reduce to terminate, it must be the case that (for any initial test case, and for a fresh transformation state) each transformation eventually returns `stop` under non-deterministic choice between (1) calling `advance` and leaving the current test case unchanged, and (2) not calling `advance` but changing the current test case to be the output of the transformation. Meeting this requirement has been natural for each of the 50+ transformations we have implemented so far. The order in which a transformation iterates through a test case is not specified by C-Reduce; in practice, each transformation uses a convenient ordering such as preorder for tree-based passes and line-order for line-based passes.

At present, C-Reduce calls five kinds of transformations. The first includes "peephole optimizations" that operate on a contiguous segment of the tokens within a test case. These include changing identifiers and integer constants to `0` or `1`, removing type qualifiers, removing a balanced pair of curly braces and all interior text, and removing an operator and one of its operands (e.g., changing `a+b` into `a` or `b`).

The second kind of transformation includes those that make localized but non-contiguous changes. Examples include removing balanced parentheses and curly braces without altering the text inside them, and replacing a ternary operator (C's `?:` construct) with the code from one of its branches.

The third is a C-Reduce module that closely follows Berkeley delta: it removes one or more contiguous lines of text from a test case. The number of lines to remove is initially the number of lines in the test case, and is successively halved until it reaches one line, at which point the test case is reformatted using *topformflat*. This transformation is 74 lines of Perl whereas Berkeley delta is 383 lines, showing that implementing a transformation is considerably easier than implementing an entire delta debugging loop.

Fourth, C-Reduce invokes external pretty-printing commands such as GNU indent. It is important that this kind of tool is called within the delta debugging loop, as opposed to being part of a pre- or post-processing step, because it is not unheard of for simple reformatting to turn a failure-inducing test case into one that does not trigger a compiler bug.

The final class of transformation was motivated by our observation that to create reduced failure-inducing C programs nearly as small as those produced by skilled humans, a collection of compiler-like transformations is needed. C-Reduce currently has 30 source-to-source transformations for C code including:

- performing scalar replacement of aggregates;
- removing a level of indirection from a pointer- or array-typed variable;
- factoring a function call out of a complex expression;
- combining multiple, same-typed variable definitions into a single compound definition;
- moving a function-scoped variable to global scope;
- removing a parameter from a function and all of its call sites, while adding a variable of the same name and type at function scope;
- removing an unused function or variable;
- giving a function, variable, or parameter a new, shorter name;
- changing a function to return void, and deleting all `return` statements in it;
- performing inline substitution of small function bodies;
- performing copy propagation; and
- turning unions into structs.

We implemented these using LLVM's Clang front end.

We have implemented only one performance optimization in C-Reduce: memoization of the (sometimes quite expensive) test for discriminating between successful and unsuccessful variants. Redundant tests occur when two or more peephole transformations produce the same output, and they also occur during the final iteration of the fixpoint computation—which, by necessity, performs some redundant tests. Several additional opportunities for speed-up exist, and we intend to implement speed improvements in future work. As shown by the results in Section 7, however, automated test-case reduction with C-Reduce already runs fast enough to be useful as part of an automated bug-reporting process.

7. Results

This section compares our reducers against each other and against Berkeley delta.

7.1 Toward a Corpus of C Programs Triggering Diverse Compiler Bugs

To thoroughly evaluate test-case reducers for C compiler bugs, one needs a number of test inputs that trigger compiler-crash bugs and wrong-code bugs. Moreover, these test cases should map to a diverse collection of underlying defects. We have observed that some compiler bugs (e.g., a crash while parsing a top-level pragma) lend themselves to easy reduction, whereas others (e.g., a wrong-code bug in an interprocedural transformation) are more difficult.

We assembled our test corpus by manufacturing a large number of bug-triggering programs and then selecting a subset that appear to map to diverse underlying bugs. To find bug-triggering programs, we ran Csmith for a few days on a collection of compilers that target x86-64 on Linux. The compilers we used were: LLVM/Clang 2.6–2.9, GCC 3.[2–4].0, GCC 4.[0–6].0, Intel CC 12.0.5, Open64 4.2.4, and Sun CC 5.11. From the set of bug-triggering inputs created by Csmith, we selected the members of our bug-triggering corpus by hand. For crash bugs we chose only one test case for each distinct "symptom"—a specific assertion violation or similar that a compiler prints while crashing. No obvious heuristic exists for figuring out which test cases triggering wrong-code bugs map to which underlying bugs, so we simply gathered no more than five wrong-code triggers for each compiler that we tested. In the end, we selected 98 test cases: 57 inputs that each trigger a different compiler crash, and 41 inputs that trigger incorrect code generation.

7.2 Evaluating Reducers

We ran Berkeley delta and our new reducers on our corpus of bug-triggering test inputs, using a machine with 16 GB of RAM, based on an Intel Core i7-2600 processor, running Ubuntu Linux 11.10 in 64-bit mode. For these experiments we disabled Linux's address space layout randomization (ASLR) feature, which is intended to thwart certain kinds of malware. ASLR has the side effect of causing some compiler bugs to occur non-deterministically. All test-case reducers that we are aware of (including our new ones) operate under the assumption that buggy executions can be detected deterministically.

The inputs to each reducer—Berkeley delta, Seq-Reduce, Fast-Reduce, and C-Reduce—are the test case that is to be reduced and a shell script that determines whether a variant is successful. Berkeley delta additionally requires a "level" parameter that specifies how much syntax-driven flattening of its input to perform. The web page for this tool suggests running the main delta script twice at level zero, twice at level one, twice at level two, and finally twice at level ten. That is what we did.

Tables 1 and 2 summarize the results of our reducer experiments. For each test case we measured:

- The size of the original test case, as emitted by Csmith.
- The size of the test case after being reduced by Berkeley delta, and the time taken by reduction. For wrong-code bugs, we evaluated Berkeley delta using both KCC and Frama-C as checkers for undefined behavior.
- The size of the test case after being reduced by Seq-Reduce with a 20-minute timeout, and the time taken by reduction. (It occasionally runs for longer than its nominal timeout when Csmith is slow.) Because Seq-Reduce is embarrassingly parallel, we ran four independent instances of it on our four-way test machine, and upon termination chose the best result.
- The size of the test case after being reduced by Fast-Reduce, and the time taken by reduction.
- The size of the test case after being reduced by C-Reduce and the time taken by reduction. For wrong-code bugs, again we used both KCC and Frama-C as checkers for undefined behavior.

```
int printf (const char *, ...);
struct {
  int f0;
  int f1;
  int f2;
}
a, b = {
  0, 0, 1
};
void
fn1 () {
  a = b;
  a = a;
}
int
main () {
  fn1 ();
  printf ("%d\n", a.f2);
  return 0;
}
```

Listing 3. The median-sized reduced test case output by C-Reduce with Frama-C for wrong-code bugs. GCC 4.3.0 for x86-64 produces incorrect code with –Os. This is W22 in Table 1.

Our choice of metrics—the size of the final output and the time taken to get it—is influenced by our belief that these are the primary metrics that people reporting compiler bugs care about. Because our results contain some significant outliers, we report both mean and median values at the bottom of each table.

7.3 Analysis of Wrong-Code Bug Results

As Table 1 shows, reducing a test case triggering a wrong-code bug is often fast, but in a few cases takes many hours.

Berkeley delta This tool generally fails to generate test cases that we would include in a compiler bug report as-is. The median run time of Berkeley delta when combined with Frama-C is four minutes; with KCC, the median is a little less than an hour.

Seq-Reduce On average, Seq-Reduce creates reduced output about twice as large as Berkeley delta's; more reduction would need to be performed before including these test cases in a compiler bug report. The primary value of Seq-Reduce lies in its simplicity. Also, it serves as a proof of concept for a generalized delta debugger based on randomized search.

Fast-Reduce Fast-Reduce is by far the fastest reducer. However, its final output is often too large to be included in compiler bug reports—its median output size is almost 3 KB—so further reduction must be performed by hand or by a different automated reducer.

C-Reduce C-Reduce is the only reducer we tested that consistently produces results that we would directly copy into a bug report. Listing 3 shows the median-sized reduced test case out of the 41 reduced wrong-code outputs produced by C-Reduce (for C-Reduce paired with Frama-C). It reasonably approximates the "typical"

			Original	Berkeley delta with Frama-C		Berkeley delta with KCC		Seq-Reduce		Fast-Reduce		C-Reduce with Frama-C		C-Reduce with KCC	
ID	**Compiler**	**Flags**	**Size**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**
W1	Clang 2.7	–O2	58,753	10,745	3	10,745	40	16,556	20	49,810	0	295	6	295	64
W2	GCC 3.2.0	–O3	54,301	15,153	4	15,153	30	15,717	20	50,200	1	179	5	179	42
W3	GCC 3.2.0	–O3	62,095	6,624	4	6,624	57	13,860	20	1,230	0	214	6	214	62
W4	GCC 3.3.0	–O3	54,301	15,153	4	15,153	31	22,410	20	50,200	0	176	5	176	39
W5	GCC 3.3.0	–O3	60,010	1,379	1	1,902	54	12,948	20	55,908	0	248	5	248	41
W6	GCC 3.3.0	–O3	89,036	2,414	2	2,399	47	9,992	20	85,944	0	248	5	248	46
W7	GCC 3.4.0	–O3	39,489	9,647	2	9,647	23	4,632	20	30,525	0	184	5	184	44
W8	GCC 4.0.0	–O3	42,516	1,995	5	2,550	91	13,953	20	2,247	1	134	11	134	67
W9	GCC 4.1.0	–O1	57,079	1,775	1	1,775	27	41,209	20	586	0	178	6	178	28
W10	GCC 4.1.0	–O1	81,067	5,789	4	4,044	113	47,841	20	5,299	0	242	9	242	215
W11	GCC 4.1.0	–O3	50,081	6,559	3	6,498	44	5,730	20	47,057	0	873	11	745	69
W12	GCC 4.1.0	–O3	57,028	11,658	3	11,658	32	5,911	20	2,124	0	202	27	202	209
W13	GCC 4.1.0	–O3	61,119	10,570	7	10,570	114	20,010	20	12,321	1	221	13	221	132
W14	GCC 4.2.0	–O0	44,078	5,208	2	5,208	21	5,615	20	2,407	0	176	5	176	32
W15	GCC 4.2.0	–O0	53,922	12,418	4	12,418	33	9,636	20	3,661	3	868	22	868	81
W16	GCC 4.2.0	–O0	56,842	15,772	7	13,585	144	7,001	20	52,377	0	971	18	971	343
W17	GCC 4.2.0	–O1	41,262	8,312	3	8,312	75	5,980	20	36,647	0	205	11	205	153
W18	GCC 4.3.0	–O0	45,298	7,816	4	7,816	63	10,652	20	6,094	0	196	7	196	79
W19	GCC 4.3.0	–O0	55,727	5,975	4	5,975	190	17,089	20	903	0	182	9	182	119
W20	GCC 4.3.0	–O2	64,349	10,233	9	10,233	88	30,228	20	2,990	0	205	10	205	72
W21	GCC 4.3.0	–O2	67,227	10,396	6	10,360	209	11,097	20	1,670	0	172	6	172	134
W22	GCC 4.3.0	–Os	96,273	10,689	7	10,814	119	37,327	20	1,961	0	192	12	199	663
W23	GCC 4.4.0	–O0	43,030	859	1	859	14	1,216	20	384	0	179	1	179	11
W24	GCC 4.4.0	–O0	52,278	1,144	1	753	130	1,103	20	393	0	182	1	182	73
W25	GCC 4.4.0	–O0	65,597	1,108	1	1,108	46	1,119	20	397	0	179	1	179	14
W26	GCC 4.4.0	–O2	40,147	4,120	2	4,120	13	6,805	20	36,060	0	755	5	755	30
W27	GCC 4.4.0	–Os	86,103	3,537	2	3,537	60	7,257	20	801	0	245	4	245	53
W28	Intel CC 12.0.5	–O2	42,510	13,983	6	13,983	260	18,181	20	33,530	1	187	81	317	2,312
W29	Intel CC 12.0.5	–Os	38,232	9,756	5	9,756	55	22,635	20	3,007	0	162	12	173	59
W30	Intel CC 12.0.5	–fast	48,803	2,967	4	2,967	28	9,242	20	2,317	0	185	7	196	31
W31	Intel CC 12.0.5	–fast	81,103	6,881	8	6,881	252	56,247	20	7,353	1	119	10	130	268
W32	Open64 4.2.4	–O2	43,176	1,428	2	1,428	28	1,259	20	376	0	218	4	218	28
W33	Open64 4.2.4	–O2	43,384	6,874	3	6,874	82	5,997	20	4,802	0	207	8	207	108
W34	Open64 4.2.4	–O2	65,732	3,976	4	3,948	77	12,189	20	316	0	216	5	216	91
W35	Open64 4.2.4	–O2	79,971	2,512	5	2,687	118	11,165	20	427	0	218	6	218	127
W36	Open64 4.2.4	–O3	96,008	30,239	11	29,956	279	17,091	20	31,320	11	160	102	160	1,129
W37	Sun CC 5.11	–xO2	41,597	1,023	1	1,023	5	1,173	20	241	0	148	4	148	11
W38	Sun CC 5.11	–xO2	43,176	6,391	3	6,391	54	6,109	20	1,767	0	144	8	144	86
W39	Sun CC 5.11	–xO2	43,384	7,090	3	7,090	77	5,744	20	2,046	0	145	7	145	113
W40	Sun CC 5.11	–xO2	69,657	6,025	4	6,025	108	28,710	20	6,340	1	204	13	204	143
W41	Sun CC 5.11	–xO2	70,793	1,322	1	1,322	18	14,302	20	248	0	145	3	145	21
		Mean	58,208	7,256	4	7,174	82	14,462	20	15,470	1	258	12	259	182
		Median	55,727	6,559	4	6,498	57	11,097	20	2,990	0	192	7	199	72

Table 1. Compiler bugs used in the "wrong code" part of our evaluation. Program sizes are in bytes, and reduction times are in minutes.

C-Reduce output that might be reported to a compiler developer. The original test case emitted by Csmith was 94 KB—clearly much too large to be included in any reasonable bug report. C-Reduce's median run time when combined with Frama-C is seven minutes; with KCC it is 72 minutes.

In some cases, Berkeley delta or C-Reduce when combined with KCC produces different final output than the same tool combined with Frama-C. This happens when KCC and Frama-C disagree about the definedness of one or more variants. Often, the disagreement is due to a tool timing out; we killed either tool whenever it took more than three seconds longer to analyze a variant than it took to analyze the original, unreduced version of the program being reduced. However, there are also real differences between the sets of programs that these tools consider to be well-defined. We investigated these issues and found three classes of root causes. First, differential testing of KCC and Frama-C revealed a small number of bugs in the tools. (These bugs did not affect the numbers that we report, because we fixed them before running our experiments.) Second, C is a language with many extensions and dialects, and sometimes these tools' differing goals caused them to differently accept non-standard inputs. For example, Frama-C aims for pragmatic compatibility with existing C code, and so it accepts GCC's extended ternary operator which permits the second operand to be omitted: e.g., `x?:y`. On the other hand, KCC aims for strict standards compliance and rejects this construct. The third class of differences we found results from genuine corner cases in the standard; we were not always able to resolve these issues even after talking to experts. For example, the C99 standard is not entirely clear about the definedness of the expression `(s.f1=0) + (s.f2=0)` when `f1` and `f2` are bitfields that may occupy the same byte of storage.

7.4 Analysis of Compiler-Crash Bug Results

The general relationships between the reducers for wrong-code bugs (Table 1) also hold for compiler-crash bugs (Table 2). In terms of size of final output, Seq-Reduce and Fast-Reduce are worst, Berkeley delta is better, and C-Reduce is best.

Relative to the reducers' performance on test cases that trigger wrong-code bugs, test-case reduction for compiler-crash bugs is faster (C-Reduce, for example, always finished in less than 20 minutes) and the reduced test cases are generally smaller. The speed-up is partly due to not having to run the expensive validity checkers for undefined behavior (Section 5.2.4). Also, when the requirement to have a complete test case (including `main`) is relaxed, test cases can be made smaller, reducing the size of the search space.

For our corpus of test cases that trigger compiler-crash bugs, Listing 4 shows the median-sized reduced test case output by C-Reduce. The reduced test case is reportable as-is, whereas the corresponding original test case (C28) was 72 KB.

PLDI'12, ACM, pp 335-346. 2012

			Original	Berkeley delta		Seq-Reduce		Fast-Reduce		C-Reduce		
ID	**Compiler**	**Flags**	**Size**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Crash String**
C1	Clang 2.6	–O0	102,012	696	0	1,320	20	98,543	0	101	1	`Assertion ‘RD->hasFlexibleArrayMember() && "Must have flexible arra...`
C2	Clang 2.6	–O2	130,931	8,334	7	40,062	20	123,293	0	103	4	`Assertion ‘isa<X>(Val) && "cast<Ty>() argument of incompatible type...`
C3	Clang 2.6	–O2	211,187	14,168	3	166,729	20	213,718	0	123	3	`Assertion ‘S < E && "Cannot create empty or backwards range"’ failed.`
C4	Clang 2.6	–O3	303,869	14,460	23	67,678	20	307,931	10	314	18	`Assertion ‘ConstantVal == V && "Marking constant with different val...`
C5	Clang 2.6	–O2	41,049	41,189	0	11,548	20	35,505	0	810	3	`Assertion ‘getTypeSizeInBits(Op->getType()) > getTypeSizeInBits(Ty)...`
C6	Clang 2.6	–O0	43,267	725	0	1,792	20	34,669	0	142	0	`Assertion ‘width > BitWidth && "Invalid APInt ZeroExtend request"’ ...`
C7	Clang 2.6	–O1	43,396	3,465	1	20,091	20	33,675	0	178	1	`Assertion ‘(isa<PHINode>(GlobalUser) || isa<SelectInst>(GlobalUser)...`
C8	Clang 2.6	–O1	65,657	954	0	1,565	20	62,804	0	104	1	`Assertion ‘NextFieldOffsetInBytes <= FieldOffsetInBytes && "Field o...`
C9	Clang 2.6	–O2	76,960	14,397	3	52,627	20	70,416	0	261	6	`Assertion ‘SortedPos == AllNodes.end() && "Topological sort incompl...`
C10	Clang 2.6	–O1	84,345	11,152	4	20,741	20	74,056	0	301	3	`Assertion ‘N->getOpcode() != ISD::DELETED_NODE && RV.getNode()->get...`
C11	Clang 2.7	–O1	43,396	3,465	1	17,410	20	33,675	0	178	1	`Assertion ‘(isa<PHINode>(GlobalUser) || isa<SelectInst>(GlobalUser)...`
C12	Clang 2.7	–O0	55,234	5,479	0	14,129	20	46,145	0	65	1	`Assertion ‘unsignedRange.NonNegative && "unsigned range includes ne...`
C13	Clang 2.7	–O2	82,690	12,682	3	38,030	20	73,032	0	119	4	`Assertion ‘(!From->hasAnyUseOfValue(i) || From->getValueType(i) == ...`
C14	GCC 3.2.0	–O1	105,197	8,942	3	13,382	20	9,757	1	101	2	`Internal compiler error in fixup_var_refs_1, at function.c:1964`
C15	GCC 3.2.0	–O3	118,289	11,063	2	16,622	20	11,036	0	339	4	`unable to find a register to spill in class ‘DREG’`
C16	GCC 3.2.0	–O3	141,198	17,409	3	19,211	20	26,818	11	146	4	`unable to find a register to spill in class ‘AREG’`
C17	GCC 3.2.0	–O2	44,047	7,371	1	6,779	20	6,584	0	97	1	`Internal compiler error in extract_insn, at recog.c:2148`
C18	GCC 3.2.0	–O1	53,948	5,828	1	6,629	20	2,501	0	55	1	`Internal compiler error in do_SUBST, at combine.c:439`
C19	GCC 3.2.0	–O3	60,878	10,539	2	11,953	20	58,479	0	177	2	`Internal compiler error in print_reg, at config/i386/i386.c:5640`
C20	GCC 3.3.0	–O3	66,580	7,930	1	26,282	20	6,659	1	233	3	`internal compiler error: in print_reg, at config/i386/i386.c:6558`
C21	GCC 3.4.0	–O3	148,886	28,554	3	30,883	20	147,895	1	125	11	`internal compiler error: in output_211, at insn-output.c:1995`
C22	GCC 3.4.0	–O3	66,580	8,131	2	26,028	20	6,198	1	210	3	`internal compiler error: in print_reg, at config/i386/i386.c:7052`
C23	GCC 4.0.0	–O2	105,843	2,965	4	85,245	20	949	0	74	2	`internal compiler error: in get_indirect_ref_operands, at tree-ssa-...`
C24	GCC 4.0.0	–O2	118,305	5,024	4	36,478	20	6,879	0	65	3	`internal compiler error: in make_decl_rtl, at varasm.c:868`
C25	GCC 4.0.0	–O0	194,529	5,629	1	17,182	20	23,291	10	71	1	`internal compiler error: in c_common_type, at c-typeck.c:529`
C26	GCC 4.0.0	–O1	253,152	28,222	10	33,443	20	254,373	1	124	10	`internal compiler error: in expand_shift, at expmed.c:2297`
C27	GCC 4.0.0	–O0	66,195	6,196	1	5,423	20	5,933	10	66	1	`internal compiler error: in c_common_type, at c-typeck.c:531`
C28	GCC 4.0.0	–O2	73,913	11,012	2	24,729	20	13,933	0	124	3	`internal compiler error: in var_ann, at tree-flow-inline.h:34`
C29	GCC 4.1.0	–O1	105,843	7,951	4	58,947	20	949	0	83	2	`internal compiler error: in get_indirect_ref_operands, at tree-ssa-...`
C30	GCC 4.1.0	–O1	137,615	18,096	7	29,013	21	141,423	1	133	4	`internal compiler error: Segmentation fault`
C31	GCC 4.1.0	–O2	190,613	16,178	6	47,255	20	186,777	0	89	5	`internal compiler error: in compare_name_with_value, at tree-vrp.c:...`
C32	GCC 4.1.0	–O1	60,756	8,758	2	31,582	20	3,811	0	95	2	`internal compiler error: in simplify_cond_and_lookup_avail_expr, at...`
C33	GCC 4.1.0	–O1	63,426	455	0	1,040	20	518	0	68	0	`internal compiler error: in reg_or_subregno, at jump.c:2011`
C34	GCC 4.1.0	–O3	70,438	6,913	3	24,780	20	10,737	0	154	3	`fatal error: internal consistency failure`
C35	GCC 4.2.0	–O1	105,843	7,983	4	85,270	20	949	0	83	2	`internal compiler error: in get_indirect_ref_operands, at tree-ssa-...`
C36	GCC 4.2.0	–O1	149,664	586	1	1,497	20	450	3	107	1	`internal compiler error: in reg_or_subregno, at jump.c:2010`
C37	GCC 4.3.0	–O1	138,798	8,843	8	103,928	20	4,460	2	164	6	`internal compiler error: in set_lattice_value, at tree-ssa-ccp.c:486`
C38	GCC 4.3.0	–O2	149,664	586	1	1,589	21	450	2	107	2	`internal compiler error: in reg_or_subregno, at jump.c:1728`
C39	GCC 4.3.0	–O3	155,051	20,084	7	18,071	20	16,732	1	128	5	`internal compiler error: in vect_update_ivs_after_vectorizer, at tr...`
C40	GCC 4.3.0	–O1	161,834	7,031	9	15,047	20	7,115	1	66	8	`internal compiler error: in get_addr_dereference_operands, at tree-...`
C41	GCC 4.3.0	–O3	218,114	38,941	22	52,361	20	219,872	0	203	19	`Segmentation fault (program cc1)`
C42	Intel CC 12.0.5	–O1	74,396	4,980	2	975	20	1,597	1	98	3	`internal error: backend signals`
C43	Open64 4.2.4	–O3	131,526	8,509	3	12,088	20	17,584	1	129	4	`Assertion failure at line 2279 of lnoutils.cxx:`
C44	Open64 4.2.4	–O3	202,966	26,531	18	44,403	20	24,116	10	64	13	`Assertion failure at line 2088 of cgemit.cxx:`
C45	Open64 4.2.4	–O3	262,005	11,513	7	68,541	28	17,417	2	57	5	`Assertion failure at line 5450 of whirl2ops.cxx:`
C46	Open64 4.2.4	–O2	52,497	8,541	2	8,585	20	48,185	0	233	4	`Signal: Segmentation fault in Global Optimization -- Dead Code Elim...`
C47	Open64 4.2.4	–O3	57,427	9,809	2	6,661	20	53,327	0	142	2	`Signal: Segmentation fault in Global Optimization -- Create AUX Sym...`
C48	Open64 4.2.4	–O3	66,890	3,721	1	9,994	20	2,873	0	161	2	`Signal: Segmentation fault in CGLOOP phase.`
C49	Open64 4.2.4	–O3	73,892	8,144	3	13,127	20	5,361	0	185	3	`Assertion failure at line 2295 of lnoutils.cxx:`
C50	Sun CC 5.11	–xO4	109,440	10,506	3	10,322	20	13,986	11	158	3	`Fatal error in /home/regehr/z/SolarisStudio12.2-linux-x86-tar-ML/so...`
C51	Sun CC 5.11	–fast	114,928	11,906	3	37,986	20	111,111	0	116	6	`(iropt) error: IR_OP is not a reduction op!`
C52	Sun CC 5.11	–xO4	130,691	8,554	2	80,198	20	125,552	0	152	2	`(iropt) error: leaf_lookup_expr: bad leaf (tag=0)`
C53	Sun CC 5.11	–fast	41,183	3,679	1	17,002	20	31,887	0	131	1	`assertion failed in function node_approx() @ forward.c:291`
C54	Sun CC 5.11	–xO0	51,049	272	0	1,579	20	1,226	0	65	0	`internal compiler error: confused cg_inflush()`
C55	Sun CC 5.11	–xO3	68,396	19,001	4	39,925	20	63,364	0	161	3	`assertion failed in function cfg_process_loops() @ cfg.c:7146`
C56	Sun CC 5.11	–xO4	71,941	13,204	2	2,340	20	6,362	0	122	2	`Assertion ‘new_childAddr.offset >= 0 && new_childAddr.offset < new_...`
C57	Sun CC 5.11	–fast	72,249	5,935	2	44,507	20	67,569	0	325	3	`assertion failed in function gra_biased_select() @ gra.c:10418`
		Mean	108,608	10,407	4	29,519	20	52,184	2	151	4	
		Median	84,345	8,509	2	19,211	20	23,291	0	124	3	

Table 2. Compiler bugs used in the “crash” part of our evaluation. Program sizes are in bytes, and reduction times are in minutes.

```
int **a[][0];
static int ***const b = &a[0][1];
void fn1 ();
int
fn2 () {
  return ***b;
  fn1 ();
}
void
fn1 () {
  **b;
}
```

Listing 4. The median-sized reduced test case output by C-Reduce for compiler-crash bugs. GCC 4.0.0 for x86-64 crashes at –O2. This is C28 in Table 2.

File	Orig. Size (Tokens)	Reduced Test-Case Size (Tokens)		
		HDD*	Berkeley delta	C-Reduce
bug.c	277	51	61	41
boom7.c	420	19	256	20
cache.c	25,011	58	124	42

Table 3. Comparing HDD*, Berkeley delta, and C-Reduce

7.5 Detecting Invalid Variants

Our claim is that naïve reducers tend to introduce problematic undefined and unspecified behaviors. In the experiments we performed with our corpus, during every wrong-code reduction run using Berkeley delta and C-Reduce, the reducers encountered variants with undefined behavior that could not be detected using compiler warnings, LLVM/Clang's static analyzer, or Valgrind. However, these undefined variants were detected by KCC and Frama-C.

We performed another experiment in which we reduced the wrong-code test cases utilizing compiler warnings, the LLVM/Clang static analyzer, and Valgrind to detect invalid variants, but *not* using KCC or Frama-C. In this experiment, 12 of the 41 reduced test cases (29%) dynamically executed an undefined behavior that would have rendered the test case inappropriate for use in a bug report. This ratio is unacceptably high. In our experience with compiler-bug reporting, we believe that a compiler team would learn to mistrust and ignore a bug submitter after fewer than 12 invalid bug reports.

7.6 Comparison with Hierarchical Delta Debugging

Although source code for HDD is available, we were not able to obtain code for its C front end. Therefore we were not able to include HDD in our experiments that compare reducers over our corpus of inputs that trigger compiler-crash bugs and wrong-code bugs. We did, however, perform a small experiment to compare C-Reduce with previously reported HDD results.

We compared Berkeley delta and C-Reduce with the results reported by Misherghi and Su [13] for reducing three test cases that cause GCC 2.95.2 to crash. (We could not find source code for a fourth test case used in their paper.) In particular, we compared Berkeley delta and C-Reduce with the results reported for HDD*, which is the particular algorithm by Misherghi and Su that yields the best results for the three test cases. As in our previous experiments with crash bugs, we did not use KCC or Frama-C to check for undefined behavior in reduced test cases. This configuration also creates a fair comparison, since HDD* itself does not check for undefined behavior in test cases.

Table 3 shows the results. For two of the cases, C-Reduce produces smaller output. For `boom7.c`, C-Reduce's output is one token larger. A previous version of C-Reduce produced an 18-token output for this case, two smaller than the current version. Our experience is that incidental phase-ordering issues in the reducer (not unlike similar issues seen in compilers) can easily cause the final reduced output to be a few tokens larger or smaller. We did not compare the execution times of C-Reduce and HDD, but it is safe to assume that the brute-force C-Reduce is considerably slower.

7.7 When Does Reduction Fail?

Non-deterministic execution of the system under test can cause test-case reduction to fail. In our experience, the most common sources of non-determinism are memory-safety bugs interacting with ASLR and resource-exhaustion conditions such as timeouts and memory limits. However, even deterministic bugs can be resistant to reduction, particularly when these bugs stem from internal resource limits in a compiler. For example, a bug in register-spilling logic may simply require a large amount of code before it is triggered.

Reduction can fail in a different sense if the original and reduced test cases trigger different bugs. For crash bugs, we robustly avoid this problem by looking for the specific error string that characterizes the compiler crash. Examining the crash string is part of the process of disambiguating successful and unsuccessful variants. In contrast to compiler-crash bugs, wrong-code bugs have no obvious fingerprint. A pragmatic approach is to fix whatever bug happens to be triggered by the reduced test case, and then to check if the original test case still triggers a miscompilation.

8. Related Work

Previous work has sought to reduce C programs in order to track down compiler defects. Caron and Darnell [3] developed Bugfind, a tool that narrows wrong-code bug triggers down to a single C file and finds the lowest optimization level that triggers miscompilation. Whalley's tool, vpoiso [15], is primarily concerned with isolating the transformation that generates wrong code. However, it also includes functionality to narrow miscompilations down to a single function in the source program. McKeeman presented a random C program generator for testing C compilers [11], including a test-case reduction tool that operates at a finer granularity than Bugfind or vpoiso. McKeeman's tool appears to be similar to C-Reduce's peephole passes. An important difference between previous program reducers and ours is that our reducers address the test-case validity problem. To the best of our knowledge, the problem of maintaining the semantic validity of test cases during reduction has not been previously identified or solved.

Our new C program reducers are instances of generalized delta debugging algorithms. Zeller and Hildebrandt [18] developed delta debugging in 2002 and instantiated it in the *dd* and *ddmin* algorithms. Their main contribution was to abstract the test-case minimization problem away from any specific domain. McPeak and Wilkerson's version of *ddmin*, "Berkeley delta" [12], reduces test cases at line granularity. For appropriately structured inputs, this technique can be more efficient than character-based *ddmin*. By allowing only whole-line deletions, Berkeley delta reduces the size of the search space; if most lines are semantically independent from each other, variants are also likely to be valid test cases. Subsequently, Misherghi and Su [13] developed Hierarchical Delta Debugging (HDD), which reduces tree-structured test cases. Berkeley delta can also reduce tree-structured inputs with the help of its *topformflat* utility, which reformats block-structured text to match Berkeley delta's line-oriented strategy for producing variants. Permitting a delta debugger to exploit the natural hierarchical structure found in some domains leads to much faster reduction times and also to better results. Our new program reducers embody new techniques for generating variants, including random perturbation of a program generator's state (Seq-Reduce), run-time feedback (Fast-Reduce), and pluggable, non-localized program transformations (C-Reduce). In contrast to previous delta-debugging algorithms, Fast-Reduce and C-Reduce implement transformations that may increase the textual size of a test case (e.g., inlining), toward the goal of enabling subsequent reductions.

Our Fast-Reduce reducer utilizes run-time feedback, whereas our Seq-Reduce and C-Reduce implement "brute-force" techniques to generate variants. A third approach to program reduction is static analysis. Leitner et al. [9] combined static backward program slicing with delta debugging to reduce test cases that were randomly generated for unit testing Eiffel classes. Program slicing is used as the first step to narrow down instructions that are responsible for reaching the failure state. This greatly improves the scalability and efficiency of subsequent delta debugging.

9. Conclusion

Our goal was to take large C programs that trigger compiler bugs and automatically reduce them to test cases that are small enough to be directly inserted into compiler bug reports. Previous program reducers based on delta debugging failed to produce sufficiently small test cases. Moreover, they frequently produced invalid test cases that rely on undefined or unspecified behavior. We developed three new reducers based on a generalized notion of delta debugging, and we evaluated them on a corpus of 98 randomly generated C programs that trigger bugs in production compilers. Over this corpus, our best reduction algorithm achieves the goal of producing reportable and valid test cases automatically.

Our future plans include the automatic production of additional elements of a compiler bug report. Besides the information described in Section 2, it would be useful to automatically determine the first broken revision of the compiler and a minimized collection of compiler flags triggering the problem. Better yet, fault-localization techniques could be used to give compiler developers an idea about where in the compiler the bug is likely to be found.

Beyond the evaluation of new reduction techniques for C, our work provides insights into the nature of test-case reduction. First, we observed that highly structured inputs make reduction difficult; we claim that navigating complex input spaces requires rich and domain-specific transformations that are beyond the capabilities of basic delta-debugging searches. Second, we showed that it can be useful to structure a test-case reducer as a collection of modular reduction transformations whose execution is orchestrated by a fixpoint computation. Finally, we identified the *test-case validity problem*, which must be addressed in testing any system that admits undefined or unspecified behaviors.

Software Our new reducers are open-source software, available for download at `http://embed.cs.utah.edu/creduce/`.

Acknowledgments

We thank Alastair Reid and the anonymous PLDI '12 reviewers for their comments on drafts of this paper. Some of our experiments were run on machines provided by the Utah Emulab testbed [16]. This research was supported, in part, by an award from DARPA's Computer Science Study Group. Pascal Cuoq was supported in part by the ANR-funded U3CAT project. Chucky Ellison was supported in part by NSA contract H98230–10–C–0294.

References

[1] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI)*, pages 196–207, San Diego, CA, June 2003.

[2] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *Proc. of the 9th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, pages 123–124, Edmonton, Alberta, Canada, September 2009.

[3] Jacqueline M. Caron and Peter A. Darnell. Bugfind: A tool for debugging optimizing compilers. *SIGPLAN Notices*, 25(1):17–22, January 1990.

[4] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *Proc. of the 4th NASA Formal Methods Symposium (NFM 2012)*, Norfolk, VA, April 2012.

[5] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In *Proc. of the 14th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*, pages 281–286, Edinburgh, Scotland, 2009.

[6] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proc. of the 39th Symp. on Principles of Programming Languages (POPL)*, pages 533–544, Philadelphia, PA, January 2012.

[7] International Organization for Standardization. *ISO/IEC 9899:TC3: Programming Languages—C*, 2007. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[8] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems*, 28(6):967–989, November 2006.

[9] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proc. of the 22nd Intl. Conf. on Automated Software Engineering (ASE)*, pages 417–420, Atlanta, GA, November 2007.

[10] MathWorks. Polyspace server 8.1 for C/C++, product brochure, 2010. `http://www.mathworks.com/products/datasheets/pdf/polyspace-server-for-c-c++.pdf`.

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

[12] Scott McPeak and Daniel S. Wilkerson. Delta, 2003. `http://delta.tigris.org/`.

[13] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proc. of the 28th Intl. Conf. on Software Engineering (ICSE)*, pages 142–151, Shanghai, China, May 2006.

[14] James W. Moore. ISO/IEC JTC 1/SC 22/WG 23: Programming language vulnerabilities. `http://grouper.ieee.org/groups/plv/`.

[15] David B. Whalley. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16(5):1648–1659, September 1994.

[16] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 255–270, Boston, MA, December 2002.

[17] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[18] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.