Random Testing of C Compilers Targeting Arithmetic Optimization

Eriko Nagai [1] Hironobu Awazu [2] Nagisa Ishiura [1] Naoya Takeda [3]

[1] School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] Windows Server Business Development Division, Fujitsu Ltd., Kawasaki, Kanagawa, Japan
[3] Engineering Headquarters, ITEC Hankyu Hanshin, Co., Ltd, Osaka, Japan

Abstract—This paper presents a method of testing validity of arithmetic optimization of C compilers using random programs. Compilers are tested by programs which contain randomly generated arithmetic expressions. Undefined behavior (such as zero division and signed overflow) of the C language is carefully avoided during random program generation. This is based on precise computation of expected values of the expressions which takes implementation-defined behavior (such as the size of `int` and the semantics of shift right on negative integers) into account. A method for automatic minimization of error programs is also presented which expedites the analysis of detected errors. A random test program based on our method has detected malfunctions in several compilers, which include LLVM GCC 4.2.1 shipped with the latest Mac OS X, GCC 4.4.4 for Ubuntu Linux, GCC 4.3.4 for Cygwin, and GCC 4.4.1 for h8300-elf and m32r-elf.

I. Introduction

Since compilers are infrastructure tools for developing various software including operating systems and mission critical applications, their reliability is one of the critical issues in system development.

The most common way of validating compilers is the use of test suites which are sets of programs to check the functionalities of the compilers. There are several test suites for C compilers, such as the one distributed with GCC (GNU Compiler Collection)[1], Plum Hall test suite[2], AC-TEST[3], SuperTest[4], and testgen2[5]. By repeated test suite runs and bug fixes, reliability of compilers are enhanced, to a good extent. However, it is theoretically impossible to validate a compiler completely with a finite set of test programs, and actually many bugs are reported[6] for well-used compilers such as GCC.

Random testing is a complement to test suites, which attempts to detect compiler malfunctions beyond the reach of the test suites. Compilers are tested by randomly generated programs as long as time allows. Successful examples are Quest [1], randprog [2], and Csmith [3].

[1]http://gcc.gnu.org/install/test.html.
[2]http://www.plumhall.com/suites.html.
[3]http://www.actest.co.uk/.
[4]http://www.ace.nl/compiler/supertest.html.
[5]http://ist.ksc.kwansei.ac.jp/~ishiura/pub/testgen2/.
[6]http://gcc.gnu.org/bugzilla/duplicates.cgi.

Although the overall scheme of random testing is simple, there are three subjects to be addressed: How to exclude invalid programs from random generation, how to know the "correct answers" the randomly generated programs are expected to present, and how to minimize programs which detect errors. Undefined behavior of the C language makes the first issue a challenge. The expected behavior of the programs is not always unique, for unspecified behavior and implementation-defined behavior of C may make the programs multivocal. Undefined behavior also hinders automatic minimization of error programs, which leads to painful hand minimization.

Since Quest [1] focused on passing of arguments and return values on function calls, exclusion of undefined behavior and computation of expected values were not difficult. However, targets of random testing is extended to a broader range of program constructs, the three issues become prominent. In Csmith [3], which generates programs consisting of thousands of lines, proposed variety of techniques to avoid undefined behaviors, but in general it is difficult to exclude undefined behavior completely. Differential testing [4], employed in randprog [2] and Csmith, is a workaround for the expected value issue, in which test programs are compiled by multiple compilers and the results are compared. However, it may produce false positives due to unspecified and implementation-defined behavior of the C language. Since automatic minimization of suspicious programs is extremely difficult under the existence of undefined behavior, the analysis task to distinguish the false positives and to classify the error programs takes a lot of time and efforts.

This paper presents random testing of C compilers focusing on arithmetic expressions, whose compilation is machine dependent and is susceptible to mistakes during retargeting. A distinctive feature of our method is that by focusing only on arithmetic expression the expected behavior of randomly generated programs are precisely precomputed taking implementation-defined behavior into account. As a result, undefined behavior is completely excluded during program generation and minimization of error programs becomes possible, which makes the analysis of the error programs efficient.

A random test system based on the proposed method has detected malfunctions in several compilers which include GCC 4.4.4 for Ubuntu Linux, LLVM GCC 4.2.1 for Mac OS X, etc.

II. Random Testing of C Compilers

A typical flow of random testing of compilers is shown in Fig. 1. In each iteration, a test program is randomly generated, which is compiled and executed to check the validity of the compilation. This iteration is repeated as long as time permits. The program generation may focus on a particular construct of the language (as in Quest [1]) or may target broad range of the language specification (as in Csmith [3]). If errors are detected, the programs in question are investigated. During this proess, the programs are minimized.

```
1: for (specified iterations or specified CPU time) {
2:     generate a random program p;
3:     if (compile_and_run(p) == ERROR) {
4:         keep p as an error program;
5:     }
6: }
7: analyze (minimize) error programs;
```

Fig. 1. Flow of Random Testing of Compilers.

One of the major challenges in this test scheme is how to randomly generate only valid programs. In the case of the C language, *undefined behavior* makes this issue difficult. The undefined behavior is defined as "behavior, such as might arise upon use of an erroneous program construct or of erroneous data, for which the standard imposes no requirements." For example, it is invoked by zero division, signed overflow, shift by a negative value, array reference by an out of range subscript, etc. Since possible undefined behavior of the C program ranges from ignoring the situation with unpredictable results to terminating execution, programs to test compilers must not include constructs that yields undefined behavior. However, it is difficult to detect undefined behavior in randomly generated programs for it depends on the dynamic behavior of the programs.

```
 1: #include <stdio.h>
 2:
 3: const volatile unsigned char x1 = 2U;
 4: const volatile signed long long x6 = 1476669LL;
 5: static const unsigned short x8 = 35U;
 6:
 7: int main (void)
 8: {
 9:     int rc = 0;
10:     signed long long test = 0;
11:
12:     test = (((x8*(x6<<x8))>=x1)/x6);
13:
14:     if (test == 0LL) {
15:         printf("OK, %lld\n",test);
16:     }
17:     else {
18:         rc = 1;
19:         printf("NG, %lld\n",test);
20:     }
21:     return rc;
22: }
```

Fig. 2. Example of randomly generated test program. This program actually found a bug in GCC 4.3.4 for Cygwin.

How to know the correct behavior for randomly generated programs is another challenge. In general, automatic computation of the expected values for random programs needs enormous efforts, for it is equivalent to implement an interpreter for (some subset of) the language. Instead of developing such an interpreter, differential testing [4] makes use of other compilers. Test programs are compiled by multiple compilers, or multiple versions of a compiler, or even a compiler with different optimization options. The execution results are compared, and if they disagree, the minorities would be suspects.

Although differential testing enables testing of compilers with large random programs containing many language constructs [3], a large number of false positives might be produced due to *implementation-defined behavior* and *unspecified behavior* of the C language. Unspecified and implementation-defined behavior is both "behavior for a correct program construct and correct data, that depends on the implementation," where difference of the two types of behavior is whether documentation for such behavior is required or not. Examples of implementation-defined behavior are size of `int` variables and the result of shift right operations on negative integers. Compilers of different target may produce codes of different behavior for the same program. Even if the target is the same, a program does not always compiles into the codes of the same behavior due to unspecified behavior. For example, there is no restriction on the order of evaluation for subexpressions in a expression, and optimizers in compilers make use of this freedom to generate more efficient codes.

Minimization of the error programs is also an important subject. Firstly, it must be decided if the errors are really attributed to compiler malfunctions. This is indispensable when the test generator produces false positives. Errors may be due to bugs in the test program generator or misinterpretation of the language specification. Unspecified behavior makes this decision difficult. Especially the C language allows a certain degree of freedom in the precision during evaluation of floating point arithmetic which might change the behavior of the programs. Secondly, the construct in the error program that led to compiler malfunction should be identified. Classification of the error programs should be also done, because plural error programs often detect the same type of errors. For those purposes, each error program must be minimized, namely reduced to a program which is as small as possible and still presents the same symptoms.

However, undefined behavior makes automation of the minimization task difficult. While differential testing might produce false positives, auto-minimization was not implemented in Csmith, for unguarded reductions of programs lead to undefined behavior [3].

```
1: for (;;) {
2:   randomly generate an expression tree e;
3:   for (t=0; t < T; t++) {
4:     randomly assign initial value to each variable;
5:     compute the expected value of e;
6:     if (e does not yield undefined behavior) goto FIN;
7:   }
8: }
9: FIN: generate a program from e;
```

Fig. 3. Generation of arithmetic expressions.

III. Random Testing of C Compilers Targeting Arithmetic Optimization

In this paper, random testing of C compilers targeting arithmetic optimization is presented. Our method relies on precise computation of expected values rather than differential testing. Generation of programs with undefined behavior is carefully excluded based on computation of expected values. Implementation-defined behavior is faithfully interpreted in the subroutine to compute expected values which is controlled by compiler specific parameters.

A. Test program

Fig. 2 shows an example of the test programs that are generated by our method. It contains an arithmetic expression (line 12), which consists of several operators combining variables. The structure of the expression, the kinds of the operators, the types/storage classes/type qualifiers of the variables are chosen randomly. Since the random initial value is declared to every variable (lines 3–5)[7], the expression evaluates to some constant. The result is compared with the expected value (line 14).

B. Generation of Arithmetic Expressions

The outline of the algorithm for test program generation is shown in Fig. 3. First, an expression tree e is generated using random numbers (line 2). First, a set of variables are generated. For each variable, its type, sotrage class (`static` or none), and modifier (`const`, `volatile`, `const volatile`, or none), and scope (local or global) are randomly decided. Then it is decided if the root node of e is an operation node or a variable node. For an operation node, its operation is randomly chosen from set of all the arithmetic operations, and its children are generated recursively. For an variable node, a variable is randomly selected from the set of variables. Next, random initial values are assigned to the variables (line 4). The expected value of the expression is computed (line 5), during which undefined behavior is detected. If e does not yield undefined behavior (line 6), then a program is generated from e (line 9). On the other hand, if e results in undefined behavior, new initial values for the variables are generated and e is evaluated again. If e does not pass the test within T trials, then e is discarded and regenerated.

[7]The variables can be either local or global, though this example has only global variables.

It is possible that the outer loop in Fig. 3 iterates infinitely many times. This happens if e is large and e contains shift operators, whose right operand must be less than the width of the left operand. In order to prevent this situation, the size of e is curved using two parameters p_L and p_R that define the probabilities in which each node has a left child and a right child, respectively.

The expected values for generated expressions is computed by faithfully taking the implementation-defined behavior of the compiler under test into account. More specifically, integer promotion, arithmetic conversion, wrap around on unsigned integers, shift right on negative integers as well as detection of undefined behavior invoked by overflow, underflow, and shifts with invalid right operands, are carefully computed based on user definable target parameters such as the number of bits and the minimum and maximum values for every integer and floating point types. Target specific information also includes the function to define the behavior of shift right operations on negative integers, and the command name and the options to invoke the compiler (and a simulator if it is a cross compiler).

IV. Minimization of Error Programs

Given an error program, such as the one shown in Fig. 2, the arithmetic expression in the program (in line 12) is reduced into a one that is as small as possible and that still let the compiler yield the error. Then, declaration of the variables which do not appear in the minimized expression are deleted.

A. Minimization of Arithmetic Expressions

Arithmetic expressions are minimized by applying the following basic transformations repeatedly:

1) Substitution of a variable by its initial value

   One of the variables in the expression is replaced by its initial value, as shown in Fig. 4 (a).

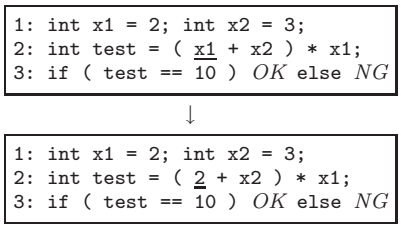
2) Evaluation of an operator

   One of the leaf subexpressions is replaced by its resulting value, as shown in Fig. 4 (b). Note that a change on the type of constant values such as the one shown in Fig. 4 (c) also takes an application of this transformation. This is to identify type related malfunctions.

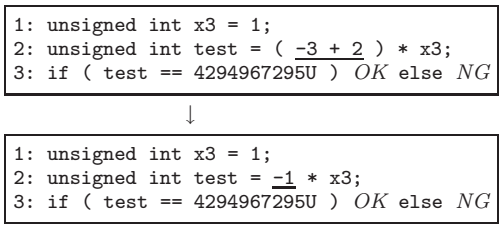
3) Choice of top operands

   One of the operands of the top operation is chosen and the other is deleted, as shown in Fig. 4 (d), where the expected value is also replaced.

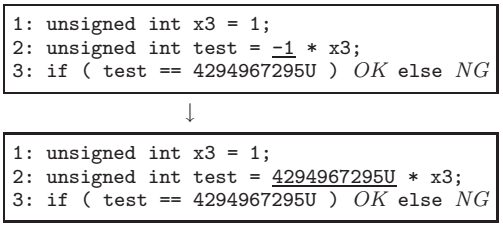
In this paper, a minimized form of an error program is defined as a program to which a single application of any basic transformations results in disappearance of the error. Note that there are multiple minimized forms for an error program that satisfy this condition. Our goal is to find one of them. Depending on compiler bugs, there are



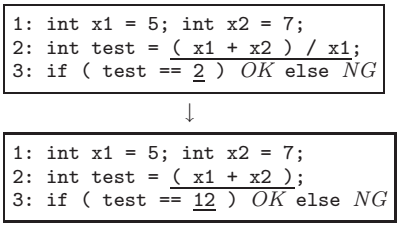
(a) Basic transformation 1 (substitution of variables)



(b) Basic transformation 2 (evaluation of leaf subexpression)



(c) Basic transformation 2 (evaluation of leaf subexpression)



(d) Basic transformation 3 (choice of subexpression)

Fig. 4. Basic transformations for minimization.

cases where a single application does not cause error but multiple application does. Such a kind of minimization is beyond the scope of this paper.

Reduction of programs using the basic transformations 1 and 2 are referred to as "top-down minimization," while reduction using the basic transformation 3 as "bottom-up minimization." Our method attempts to reduce given error programs by applying the top-down and bottom-up minimization alternately until they are not applicable any more. The both minimization procedures runs on the tree data structure representing arithmetic expressions.

B. Top-Down Minimization

The outline of our algorithm for top-down minimization is as shown in Fig. 5. It receives a tree r to represent arithmetic expression and returns a minimized tree. If r is a leaf (variable) node, it returns r, for there is no room for top-down minimization (line 3). Otherwise (if the root of r is an operator), it tests the left subtree (lines 5–8). It generates a program from the subtree and checks it for compilation and execution. If error still occurs, it recurses on the subtree. Otherwise, the same attempt is done on the right subtree (lines 9–12). Neither subtree yields an error, it returns r, which means top-down minimization is not applicable.

```
 1: tree topdown_minimization(tree r)
 2: {
 3:   if (r is a variable) { return r; }
 4:   else {
 5:     p = generate_program(r.left);
 6:     if (compile_and_run(p) == ERROR) {
 7:       return topdown_minimization(r.left);
 8:     }
 9:     p = generate_program(r.right);
10:     if (compile_and_run(p) == ERROR) {
11:       return topdown_minimization(r.right);
12:     }
13:     return r;
14:   }
15: }
```

Fig. 5. Top-down minimization.

C. Bottom-Up Minimization

Bottom-up minimization also receives an expression tree r and returns a minimized tree. Fig. 6 shows the outline of the bottom-up minimization algorithm. Set C keeps candidate nodes which can be replaced by constant nodes while set F keeps the nodes to which substitution failed (substitution has resulted in disappearance of the error). The main loop (lines 5–17) continues until there is no candidate. On each iteration, an arbitrary node x is taken out of C, which are evaluated and replaced by the resulting constant value node. If the resulting program does not yield the error (lines 9–12), this substitution is a failure; node x is restored and is kept in fail list F. On the other hand, if the error is detected (lines 13–16), this substitution is adopted; C is updated so that it includes a node which newly becomes evaluable, and the nodes in F are also included in C, for these nodes may cause the error on the modified r. Finally the procedure returns resulting tree r.

V. Implementation and Experimental Results

A. Implementation

A random test program based on the proposed method has been implemented in Perl 5. It runs on Unix systems,

TABLE I
SUMMARY OF EXPERIMENTS.

compiler (target)	generated tests	errors	error classes	false positives	OS/CPU/memory	time [h]
llvm-gcc 4.2.1 (i686-apple-darwin10)	500,000	3	3	0	Mac OS X/Core 2 Duo 2.12GHz/2GB	38.5
gcc 4.2.1 (i686-apple-darwin10)	100,000	3	3	0	Mac OS X/Core 2 Duo 2.12GHz/2GB	20.8
gcc 4.3.4 (pc-cygwin)	10,000	6	6	0	Cygwin/Core 2 Duo 1.40GHz/3.40GB	5.5
gcc 4.4.4 (i686-linux)	50,000	2	2	19	Ubuntu/Core i5 2.67GHz/4GB	5.1
gcc 4.4.1 (arm-elf)	200,000	0	0	0	Ubuntu/Core i5 2.67GHz/4GB	22.6
gcc 4.4.1 (m32r-elf)	30,000	21	5	0	Ubuntu/Core i5 2.67GHz/4GB	2.9
gcc 4.4.1 (h8300-elf)	10,000	83	4	17	Ubuntu/Core i5 2.67GHz/4GB	1.1

```
 1: tree bottomup_minimization(tree r)
 2: {
 3:   C = { all the variable nodes };
 4:   F = φ; /* to keep once failed nodes */
 5:   while( C ) {
 6:     take an arbitrary node x out of C;
 7:     evaluate node x and replace x by resulting value;
 8:     p = generate_program(r);
 9:     if (compile_and_run(p) != ERROR) {
10:       restore x;
11:       F = F ∪ {x};
12:     }
13:     else {
14:       put newly evaluable node into C;
15:       C = C ∪ F;  F = φ;
16:     }
17:   }
18:   return r;
19: }
```

Fig. 6. Bottom-up minimization.

```
 1: #include <stdio.h>
 2:
 3: const volatile signed long long x6 = 1476669LL;
 4:
 5: int main (void)
 6: {
 7:     int rc = 0;
 8:     signed long long test = 0;
 9:
10:     test = (x6<<(signed int)35);
11:
12:     if (test == 50737960496136192LL) {
13:         printf("OK, %lld\n",test);
14:     }
15:     else {
16:         rc = 1;
17:         printf("NG, %lld\n",test);
18:     }
19:     return rc;
20: }
```

Fig. 7. Program minimized from error program in Fig. 2.

including Ubuntu Linux, Mac OS X, Cygwin on Windows, etc. The target standard is C99 (ISO/IEC 9899:1999). The values of the parameters T, p_L, and p_R are set to 100, 0.45, and 0.45, respectively.

B. Error detection capability and run time

TABLE I is a summary of the experiments. The first column lists the versions and targets of the compilers tested. Column "generated tests" shows the numbers of random programs generated, "errors" the numbers of programs that detected errors. Some programs detected errors of the same type. Thus, column "error classes" shows into how many classes they are categorized. The column "false positives" lists the numbers of the programs that were reported as errors but turned out to be correct behavior by manual analysis. The last two columns are the computation environments and computation time.

With $10 \sim 40$ hours, several malfunctions for each compiler was detected, except for gcc 4.4.1 for arm-elf. All of the false positives on gcc 4.4.4 for i686-linux and gcc 4.4.1 for h8300-elf were due to the floating point precision.

C. Minimization

Fig. 7 is the reduced program obtained from the error program listed in Fig. 2 by our auto-minimizer. The expression in line 12 of Fig. 2, consisting of 5 operations, was automatically reduced to a single binary operation, from which we can see that malfunction exists in the shift left operation.

TABLE II
COMPUTATION TIME FOR MINIMIZATION.
(GCC 4.4.1 for Cygwin, Core i5 2.67GHz)

# operators		CPU
before	after	[sec]
9	1	16.85
7	1	6.55
7	1	10.53
5	2	7.31
5	2	9.42

TABLE II lists the CPU time spent for the minimization. The compiler under test was GCC 4.4.1 on Cygwin and the CPU is Core i5 2.67GHz with 4GB memory. The computation time is roughly proportional to the number of the operators in the error programs.

The automatic minimization capability was actually useful in analyzing the results in TABLE I, especially those of gcc 4.4.1 for h8300-elf. Furthermore, the minimizer had been indispensable to find bugs in the routine for computing expected values in earlier versions of our

random test system.

D. Detected Bugs

Fig. 8 shows some examples of error programs which detected compiler bugs by our method. Based on the outputs of the minimizer, the programs were further minimized by hand. In examples (a) and (d), the compilers generated codes that yielded erroneous outputs. In examples (b) and (c), the compilers crashed on such short programs.

VI. Conclusion

This paper presented a method for random testing of C compilers targeting optimization of arithmetic expressions. A method of auto-minimizing error programs is also shown. The test generator based on the methods detected several bugs in GCCs.

Currently, the ability of the test generator is limited because the size of the expression is intentionally curved to avoid undefined behavior. We are now working on the generation of longer expressions without undefined behavior. Development of algorithms to accelerate minimization is another important subjects.

Acknowledgment

Authors would like to thank Mr. Nobuyuki Hikichi (TOPS Systems Corporation) for his discussion and advices on compiler testing. We would also like to thank Mr. Shohei Yoshida (now with AXE, Inc.), Mr. Yuki Uchiyama (now with K-Opticom Corporation), and Mr. Soichiro Taga (now with Mitsubishi Electric Micro-Computer Application Software Co., Ltd.) for their discussion and help. We thank Mr. Fukumoto, Mr. Ooki, and all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion on this research.

References

[1] Christian Lindig: "Find a Compiler Bug in 5 Minutes," in *Proc. ACM International Symposium on Automated Analysis-Driven Debugging*, pp. 3–12 (Sept. 2005).

[2] Eric Eide and John Regehr: "Volatiles Are Miscompiled, and What to Do about It," in *Proc. 7th ACM International Conference on Embedded Software*, pp. 255–264 (Oct. 2008).

[3] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr: "Finding and Understanding Bugs in C Compilers," in *Proc. 2011 ACM SIGPLAN Conference on Programing Language Design and Implementation (PLDI)* pp. 283–294 (June 2011).

[4] W. M. McKeeman: "Differential Testing for Software," Digital Technical Journal, vol. 10, no. 1, pp. 100–107 (Dec. 1998).


```
 1: #include <stdio.h>
 2:
 3: unsigned int x = 0;
 4:
 5: int main (void)
 6: {
 7:    int test = ( -0.5 <= (double) x );
 8:    if (test != 1) {
 9:       printf("error (test = %d)\n", test);
10:    }
11:    return 0;
12: }
```

(a) LLVM-GCC 4.2.1 for i686-darwin10 (shipped with Mac OS 10.7 (Lion)) miscompiled this program. With -O1 option, the generated code printed `"error (test = 0)"`.

```
 1: #include <stdio.h>
 2:
 3: const volatile long long x2 = 15LL;
 4:
 5: int main (void)
 6: {
 7:    int test = 768 << ( x2 - 1LL );
 8:    if (test != 12582912) {
 9:        printf("error (test = %d)\n", test);
10:    }
11:    return 0;
12: }
```

(b) GCC 4.3.4 for x86-cygwin with -O1 option crashed on this program.

```
 1: #include <stdio.h>
 2:
 3: volatile int x = 1;
 4:
 5: int main (void)
 6: {
 8:    long test = 7L * ( 1L >> 1 / x );
 9:    if(test != 0) {
10:       printf("error (test = %l\n)", test);
11:    }
12:    return 0;
13: }
```

(c) GCC 4.4.1 for h8300-elf on Ubuntu Linux with -O1 option crashed on this program.

```
 1: #include <stdio.h>
 2:
 3: int main (void)
 4: {
 6:    volatile short x1 = -1000;
 7:    long test = 40L * x1;
 8:    if (test != -40000L) {
 9:       printf("error (test = %l)\n", test);
10:    }
11:    return 0;
12: }
```

(d) GCC 4.4.1 for M32R miscompiled this program. With -O1 option, the generated code output `"error (test = 25536)"`.

Fig. 8. Example of programs that detected bugs in compilers by our method.