Regular Paper

Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions

ERIKO NAGAI$^{1,†}$ ATSUSHI HASHIMOTO$^1$ NAGISA ISHIURA$^{1,a}$

Received: December 6, 2013, Revised: March 14, 2014, Accepted: April 28, 2014, Released: August 4, 2014

Abstract: This paper presents an enhanced method of testing validity of arithmetic optimization of C compilers using randomly generated programs. Its bug detection capability is improved over an existing method by 1) generating longer arithmetic expressions and 2) accommodating multiple expressions in test programs. Undefined behavior in long expressions is successfully eliminated by modifying problematic subexpressions during computation of expected values for the expressions. A new method for including floating point operations into compiler random testing is also proposed. Furthermore, an efficient method for minimizing error inducing test programs is presented, which utilizes binary search. Experimental results show that a random test system based on our method has higher bug detection capability than existing methods; it has detected more bugs than previous method in earlier versions of GCCs and has revealed new bugs in the latest versions of GCCs and LLVMs.

Keywords: compiler validation, random testing

1. Introduction

Compilers are infrastructure tools for software development, which must be highly reliable. It is an exacting task to develop compilers of production qualities for newly developed processors. Even for well developed compilers, greatest care must be paid to keep their credibility, for various new optimization techniques are continuously implemented into them.

Correctness of compilers are tested by compiler test suites, large sets of test programs which are compiled by the compilers and resulting codes are executed to see if they behave as expected. Well-known test suites are Plum Hall[1], SuperTest[2], GCC (GNU Compiler Collection) test suite[3], and testgen2 test suite[4].

Through repeated test suite runs and subsequent bug fixes, compilers are forged to be almost perfect. However, it is theoretically impossible to completely validate a compiler with a finite set of test programs. Actually many bugs are reported for well-used compilers such as GCC$^*$ and LLVM$^*$. Random testing is a complement to the testing by those test generation engines themselves is still important to enhance the diversity of test cases (and hence the bug detection capability) of random test generators. It is not an invention of test generation algorithm itself but applicable to many random test generators. Reference [10] also presents a framework for controlling compiler random testing system efficiently. While these frameworks to utilize random test engines enhance the capability and the efficiency of compiler testing, improvement on random test generation engines themselves is still important to enhance the bug detection capabilities.

Major challenges in compiler random testing are (1) how to judge the correctness of the compiled code (how to prepare correct answers) for randomly generated program and (2) how to avoid generating test cases with undefined behavior. In such programs generated by Quest where values are just propagated from functions to functions, correct behavior is easy to predict. How-

---

$^*$ Kwansei Gakuin University, Sanda, Hyogo 669–1337, Japan
$^†$ Presently with Fujitsu Systems West Ltd.
$a$ nagisa.ishiura@m.kwansei.ac.jp

---

© 2014 Information Processing Society of Japan
ever, as programs contain the more syntax elements, preparation of expected results becomes the more difficult. If compiler crash bugs only are targeted as in CCG, those difficulties are saved, but miscompile bugs can not be detected.

Randprog and Csmith are based on a differential testing method [11], in which errors are detected by compiling test programs by different compilers (or different versions or different options of the same compiler) and by comparing the results. This method eliminates the necessity of computing expected behavior of randomly generated programs. On the other hand, some restrictions must be posed on test programs so that they do not exhibit undefined behavior, which leads to some weakness in bug detection abilities.

It is also a challenge to handle floating point operations. Since the C language standard allows the intermediate results of the floating point operations to be computed with higher precision than specified in programs, it is difficult to distinguish miscompilation and precision errors.

Metoc’s approach, in which variants are generated from correct test programs, might be promising. However, not so many transformations to generate large classes of programs to detect many errors as Csmith are not presented in Ref. [8]. Metoc also does not handle floating point operations.

Another approach is to precompute the precise expected behavior for random programs while generating them. This makes it much easier to exclude programs with undefined behavior, for pieces of program codes that cause undefined behavior are detected during program construction. Nagai [12] proposed a random test method based on this approach which targets arithmetic optimization. It avoids generating programs with undefined behavior by regenerating new expressions when it detects expressions that trigger undefined behavior. An implemented test system found some bugs in GCC 4.4.1 (i686-pc-linux), etc., but it is not necessarily effective, for no bugs were detected in GCCs of versions higher than 4.5.0. Possible reasons for this is that the generated programs were all small or that the generated program only focused on arithmetic expressions.

This paper proposes methods of enhancing the bug detection capability of the random test method in Ref. [12] focusing on arithmetic optimization. We concentrate on arithmetic optimization because 27.8% of the bugs Csmith found in GCC were arithmetic optimization [7] and we consider it one of the most important parts of the compiler to test. Reinforcement of tests are done by generating programs with many and long expressions. Generation of long expressions without undefined behavior is achieved by modifying invalid subexpressions during their expected values are computed. Furthermore, a method for incorporating floating point operations, which has not been done in Refs. [7], [8], [13] is also proposed. Besides the program generation methods, this paper also show an improved procedure for minimizing large error programs efficiently.

An implemented random test system successfully detected bugs in GCCs of versions higher than 4.5.3. For those versions of GCCs, our method found more bugs than Csmith in 12 hours. We have so far reported 8 bugs to GCC (4.7.2 through 4.9.0 experimental) and 5 bugs to LLVM (3.4 under development) which were uncovered by our test system.

2. Random Testing of Compilers Targeting Arithmetic Optimization

2.1 Random Testing of Compilers

The overall flow of compiler random testing is very simple. As shown in Fig. 1, random program generation, compile and execution, and error checking are repeated as long as time allows. If errors are detected, the programs caused the errors are saved. The analysis of the error program involves minimization (or reduction) of the programs, in which the simplest programs that still trigger the same errors are sought, automatically or manually, to make bug localization easier.

One of the most difficult issues in compiler random testing is how to avoid generating test programs with undefined behavior. The undefined behavior includes dividing by zero, dereferencing a null pointer, overflowing a signed integer etc., for which the standard imposes no requirements. A test program with any undefined behavior is of no use since any execution results are valid for such a program.

Figure 2 shows an example program with undefined behavior. Comparison (c>=t0) in the right operand of the division in line 10 evaluates to zero, since c==30 and t0==670. The shift operation in the same line also causes undefined behavior because the right operand (t1==40) exceeds the width of the left operand. These kinds of undefined behavior occur easily in randomly generated programs.

Since undefined behavior depends on run-time values of variables, it is theoretically impossible to detect the invalid behavior precisely without computing expected behavior of test programs. So, Csmith avoids generating programs with undefined behavior in a conservative way. For example, it guards divide operations as “(b!=0)?a/b:a" instead of “a/b.” However, since every arithmetic operation is always guarded, some optimizers will never be invoked and hence will not be tested. This may limit the bug detection abilities of the test programs.

```c
while (time allows) {
    randomly generate a test program r;
    compile & execute r;
    if (error) { save r; }
} analyze saved test programs;
```

Fig. 1 Flow of compiler random testing.

```c
1: int main (void) {
2:    { 
3:      int a = 60; 
4:      int b = 10; 
5:      int c = 30; 
6:      int d = 7; 
7:      int t0 = b * (a + d); /* t0 = 670 */ 
8:      int t1 = b * d - c; /* t1 = 40 */ 
9:      int t2 = (a << t1) / (c > t0); 
10:     return 0; 
11:    }
```

Fig. 2 Program with undefined behavior.
2.2 Random Testing of Arithmetic Optimization

Nagai et al. [12] proposed a compiler random testing method targeting code optimization for arithmetic expressions, which precomputes the expected behavior of test programs to provide “correct answers.” The precomputation is also useful for avoiding undefined behavior; test programs can be altered on detecting undefined behavior. Furthermore, it makes automatic minimization of error programs easier.

Figure 3 shows an example of test programs generated by this method. Lines 3, 4, 8, and 9 declare and initialize variables, then line 14 evaluates an arithmetic expression, and line 16 compares the result with the expected value. For each variable, its type, its scope (local or global), its modifier (const, volatile, const volatile, or nothing), its class specifier (static or nothing) are selected randomly. The variables are of signed or unsigned integer types (char, short, int, long, long long). Every variable is initialized with a random value at the point of declaration. The arithmetic expression consists of the variables and operators; it does not contain constants.

Undefined behavior is worked around in the following way:

1. Generate a random expression.
2. Initialize variables by random values.
3. Compute expected value of the expression.
4. If there is no undefined behavior, then return with the expression and the initial values.
5. If repetition count is less than 100, then goto 2); otherwise discard the expression and start over from 1).

Since longer expressions induce undefined behavior more probably, they have less chances to survive. 10,000 times of random program generation results in average and maximum expression size of 4.0 and 50, respectively. Moreover, each test program contains only one expression, which may limit its bug detection ability.

3. Scaling up Size and Number of Expressions

We enhance the bug detection ability of random testing method in Ref. [12] by scaling up the size and the number of the expressions generated in test programs.

3.1 Generation of Longer Expressions

Instead of regenerating variables’ initial values or expressions to avoid undefined behavior, we modify generated expressions to eliminate the undefined behavior. Given a expression and a set of initial values to the variables, we evaluate the expected value of the expression from the bottom to the top. On detecting undefined behavior of on a subexpression, we modify the subexpression so that the undefined behavior is eliminated.

3.1.1 Eliminating Undefined Behavior by Operation Insertion

We eliminate undefined behavior on an operation by inserting extra operations so that the operand causing the undefined behavior will be an appropriate value.

For example, suppose signed overflow is detected on a subexpression \( x1 = (x2 + x3) \) where \( x1 = 2147483647 \), \( x2 = 123 \), and \( x3 = 98 \), where we assume all the variables are of signed int whose maximum value (\( INT\_MAX \)) is 2147483647. In this case, an addition with an appropriate negative value is inserted into either of the operands, as shown in Fig. 4(a) so that the overflow is eliminated. The initial value of the extra variable \( k1 \) is randomly chosen within an appropriate range.

Zero division is eliminated in a similar way. As shown in Fig. 4(b), divisor is turned into non-zero by inserting an addition.

3.1.2 Eliminating Undefined Behavior by Operation Flipping

All kinds of undefined behavior in integer arithmetic expressions can be eliminated by inserting add operations with appropriate operand values. However, this makes the control over the sizes of expressions and programs difficult. We want to curve the sizes of the test programs to reduce time for overall testing, but frequent insertions of operations and new variables enlarges the programs. Moreover, the C language standard limits the nest levels of parentheses in arithmetic expressions. In order to control the nest levels as precisely as possible, insertions of extra operations should be kept as few as possible.

In order to reduce the extra operations, we propose an alternative way of avoiding undefined behavior by operation flipping, which refers to replacing operations by the complement of the...
Operations.

For example, a signed overflow on addition is eliminated by flipping the addition into subtraction, as shown in Fig. 5 (a). In the case of overflow on subtraction and multiplication, they will be changed into addition and division, respectively.

Zero division caused by comparison on the right operand, which frequently occurs, can be eliminated in a similar way. As shown in Fig. 5 (b), the divisor can be turned from zero to one by flipping the compare operator (from "<" to ">=", in this case).

Note that not all the types of undefined behavior can be cancelled by this method. We still need operation insertion to avoid zero division caused by the other operations than comparison and invalid shift amount.

Table 1 summarizes how undefined behavior in integer arithmetic is eliminated. All signed overflows (by addition, subtraction, and multiplication) are eliminated by operation flipping. Zero division caused by comparison on the second operand is eliminated by operation flipping. In Table 1, \( \text{cmp} \) is a comparison operator and \( \overline{\text{cmp}} \) is the complement of \( \text{cmp} \) (the complement of ">" is "<", for example).

All the other forms of zero division as well as invalid shift amount must be eliminated by operation insertion.

3.1.3 Controlling the Size and Depth of Expressions

Expressions with desired size and depth are generated by a procedure "make_expression(\( n, d \))" shown in Fig. 6, which generates an expression whose size and depth do not exceed \( n \) and \( d \), respectively, and returns its root node. If \( n == 0 \) or \( d == 0 \), it returns a randomly chosen variable node. Otherwise, it randomly selects positive integers \( n_1 \) and \( n_2 \), where \( n_1 + n_2 = n - 1 \) and generates two subexpressions \( e_1 \) and \( e_2 \) by recursively calling make_expression(\( n_1, d - 1 \)) and make_expression(\( n_2, d - 1 \)), respectively. Then, it randomly chooses an operator \( o \), and returns an operator node with operator \( o \) and operands \( e_1 \) and \( e_2 \). We assume the size of expressions to be 1 to 10,000.

3.2 Generating Programs with Multiple Expressions

We also try to enhance bug detection ability by putting multiple expressions into a single test program. Figure 7 shows an example of the proposed form of test programs. Multiple expressions are generated as in lines 17–19. The computed values are compared with the expected values in lines 21–23. Let us refer to the variables, such as \( t_6 \), \( t_1 \), and \( t_2 \), which appear in the left-hand sides of the statements assigning the arithmetic expressions as \( t\)-variables, and to the other variables as \( x\)-variables. All the \( t\)-variables as well as \( x\)-variables are initialized with random values at the point of their declaration. The expression may contain \( t\)-variables as well as \( x\)-variables (but not constants). Each \( t\)-variable is assigned only once. We assume a program to contain 1 to 10,000 expressions.

\[ \text{node_t make_expression}(n, d) \]
\[ \text{if} (n == 0 \text{ || } d == 0) \{ \]
\[ \text{return randomly chosen variable_node; \}
\[ \text{else} \{ \]
\[ n_1 = \text{random integer in } [0, n - 1]; \]
\[ n_2 = n - 1 - n_1; \]
\[ e_1 = \text{make_expression}(n_1, d - 1); \]
\[ e_2 = \text{make_expression}(n_2, d - 1); \]
\[ o = \text{randomly chosen operator; \}
\[ \text{return operator_node}(o, n_1, e_2); \}\]

Fig. 6 Procedure for generating expressions.

Fig. 7 Test program with multiple expressions.
4. Incorporating Floating Point Operations

Although Csmith[7] and the random testing in Ref.[13] are powerful tools in finding compiler bugs, they deal only with integer operations. In this paper, we propose a new technique to incorporating floating point operations into random testing.

4.1 Rounding Errors

The major hurdle in handling floating point operations in random testing is rounding errors. Depending on the forms of arithmetic expressions, rounding errors are amplified so that correct evaluation will be classified as invalid behavior. An extreme example is a cast operation from floating point numbers to integers. For example, in the program listed in Fig.8, the value of x2 can be slightly different from x1. This error will be amplified through the cast operation and subsequent integer operations, which results in a big difference between the value of i2 and the expected value (0)\(^7\).

We could prepare the correct expected value of floating point operations taking the rounding errors into account, by precisely computing the results to the last digit of the mantissa following the floating point number standard. However, the C language standard allows the intermediate results of the floating point operations to be computed with higher precision than specified in programs. For example, given a statement \(y = (a/b) \times (c/d)\); where all the variables are of float type, the results of \(a/b\) and \(c/d\) may be kept in the double precision and the multiplication may be computed in the double precision. So, we can not exactly predict the expected behavior of the program with floating point operations.

4.2 Eliminating Rounding Errors

We solve this problem by posing restrictions on the form of generated expressions so that all the floating point operations in the expressions are rounding error free. The concrete policies are as follows:

1. All the values of floating point types are limited to be integers in \([-2^{m-1}, 2^{m-1}]\), where \(m\) is the number of bits for the mantissa of the type. We do not allow fractions in order to incorporate floating point operations into random testing.
2. If the result of a division have a fraction, we eliminate the fraction by operation insertion. If \(x/y\) has a fraction, then it is transformed into \((x-k)/y\) where \(k = x\%y\), as shown in Fig.9 (b).
3. If overflow is detected on integer-to-float or float-to-integer cast, it is eliminated by operation insertion, as shown in Fig.9 (c).

4.3 Mixing Integer and Floating Point Operations

Based on the technique in the previous section, a procedure for generating test programs containing both integer and floating point operations is constructed as follows. Figure 10 is an illustrative code example.

1. Generate a set of variables
2. Generate arithmetic expressions
3. Compute types
4. If overflow is detected on integer-to-float or float-to-integer cast, it is eliminated by operation insertion.

(a) Eliminating overflow.

\[
\begin{align*}
\text{float x1 = 4666666.0F;} & \Rightarrow \text{float x1 = 4666666.0F;} \\
\text{float x2 = 3000000.0F;} & \Rightarrow \text{float x2 = 3000000.0F;} \\
\text{float t = x1 + x2;} & \Rightarrow \text{float t = x1 - x2;} \\
\end{align*}
\]

(b) Eliminating fraction.

\[
\begin{align*}
\text{double x1 = 3239483852.0;} & \Rightarrow \text{double x1 = 3239483852.0;} \\
\text{int x2 = 2147483647;} & \Rightarrow \text{int x2 = 2147483647;} \\
\text{int t1 = (int)x1;} & \Rightarrow \text{int t1 = (int)x1;} \\
\text{int t2 = (float)x2;} & \Rightarrow \text{int t2 = (float)x2 + k2;} \\
\end{align*}
\]

(c) Eliminating overflow on cast.

Fig. 9 Transformation for eliminating rounding errors.

![Fig. 8](image)

An extreme example where an rounding error is amplified.

\[(a) \text{Eliminating overflow.} \]

\[
\begin{align*}
\text{float x1 = 4666666.0F;} & \Rightarrow \text{float x1 = 4666666.0F;} \\
\text{float x2 = 3000000.0F;} & \Rightarrow \text{float x2 = 3000000.0F;} \\
\text{float t = x1 + x2;} & \Rightarrow \text{float t = x1 - x2;} \\
\end{align*}
\]

\[(b) \text{Eliminating fraction.} \]

\[
\begin{align*}
\text{double x1 = 3239483852.0;} & \Rightarrow \text{double x1 = 3239483852.0;} \\
\text{int x2 = 2147483647;} & \Rightarrow \text{int x2 = 2147483647;} \\
\text{int t1 = (int)x1;} & \Rightarrow \text{int t1 = (int)x1;} \\
\text{int t2 = (float)x2;} & \Rightarrow \text{int t2 = (float)x2 + k2;} \\
\end{align*}
\]

\[(c) \text{Eliminating overflow on cast.} \]

Fig. 10 Code example with mixed integer and floating point operations.

\[\text{...} \]

\[\text{1: volatile signed long x1 = -23;} \]

\[\text{2: const float x2 = 9.6F;} \]

\[\text{3: int main(void) \{} \]

\[\text{4: int main(void) \{} \]

\[\text{5: static unsigned short x3 = 134;} \]

\[\text{6: double x4 = 25.6;} \]

\[\text{7: double k0 = 7.0;} \]

\[\text{8: double k1 = 9399234.0;} \]

\[\text{9: signed int t1 = 234;} \]

\[\text{10: int t2 = (int)x1;} \]

\[\text{11: t0 = (((x1+x3)*x2)-x4);} \]

\[\text{12: int t1 = (int)(t1+x2);} \]

\[\text{13: /* initially t1 = (x1%4); */} \]

\[\text{14: t1 = (x1%signed long)x2;} \]

\[\text{15: t2 = (x4-k0)/x2;} \]

\[\text{16: t2 = (x4-k0)/x2;} \]

\[\text{17: t2 = (x4-k0)/x2;} \]

\[\text{...} \]
manner. For example, in line 11 of Fig. 10, the types of the addition \((x1+x3)\), the multiplication \((x1+x3)^{*}x2\), and the subtraction \(((x1+x3)^{*}x2)\cdot x4\) turn out to be signed long, float, and double, respectively, according to the arithmetic type conversion rule of the C language.

During the type computation, if either operand of integer operations, such as \%, <<, >>, &, and |, is of a floating point type, then an integer cast operation is inserted. For example, since the right operand of the \% operation in line 13 \((x2+x4)\) is of double type, a cast is inserted as in line 14.

4) Eliminating overflows and divisions producing fractions

The expected values of each expression is computed and at the same time floating overflows and floating divisions producing fractions, as well as integer undefined behavior, are eliminated, based on the techniques described in the previous subsection. In Fig. 10, the division in line 16 is modified into a combination of a subtraction and a division in line 17.

5. Minimization of Error Programs

Minimization of error programs is indispensable in analyzing the causes of the errors. Suppose we are given a error program of thousands of lines. Far from locating the bugs in the compiler, it is hard even to tell if the compiler is wrong or the test program is wrong; the expected values may be erroneous or there may be undefinied behavior somewhere in the test program. In practice, a program to generate valid random test programs cannot be developed without an automatic error program minimizer.

This paper proposes an error program minimization method which can efficiently handle programs with many long expressions. It is an extension of the method in Ref. [12] in four ways: 1) a transformation to handle multiple expressions is added, 2) binary search is introduced to reduce time necessary for minimizing large scale error programs, 3) a transformation to simplify values and types in error programs is added, and 4) an overall flow to control the minimization phases is redesigned.

Our minimization method is based on delta debugging [15]. If a certain transformation reducing the size of an error program preserves the occurrence of the error, the transformation is adopted, otherwise another transformation is tried. By repeating this until any of the possible transformations eliminates the error, a minimal program is obtained. Note that our method does not guarantee that the results are minimum. The results depends on the order of transformations applied, so it cannot be further reduced by any of the transformations but a smaller error program may be obtained by a different sequence of transformations.

Our method is based on the the following four transformations on error programs, where (2) and (3) are from Ref. [12] and (1) and (4) are newly introduced in this paper.

1) Expression elimination

Some of the expressions are replaced by their expected values, as illustrated in Fig. 11. If errors are detected on multiple expressions, basically only one of them is tracked. Suppose wrong results were observed on two expressions, for example. In this case, either of the expression is eliminated as long as the program yields an error or errors. However, if the errors disappear whenever either of the two expressions is eliminated, the both expressions are kept and subsequent minimization steps are applied for each expression.

2) Top-down minimization:

An expression is replaced by either of the two operands of the root operator, as shown in Fig. 12.

3) Bottom-up minimization:

A variable reference is replaced by its value, or an operation is replaced by its resulting value, as shown in Fig. 13 (a) and (b), respectively.

4) Value and type minimization:

The absolute values of constants are made smaller, as in Fig. 14 (a). Types are also made simpler; modifiers and class specifiers are removed, globals are made locals, and shorter types (short and char) and longer types (long and long long) are reduced to standard types (int), as shown in Fig. 14 (b).

The bottom-up minimization method in Ref. [12] basically reduces the operators in an expression one by one, so it took 10,000 times of compilation if an expression with 10,000 operators was reduced to a constant. In order to avoid this, we introduce binary search. First, one of the operands of the root operator of a given expression is reduced to a constant. If it succeeds (the resulting
program still produces an error), the other operand is tried. If it fails, the children of the operand are recursively attempted to be reduced.

Similarly, the expression elimination is done in a binary way, otherwise reduction of 10,000 expressions would needs 10,000 compile runs. At first, the first half of the expressions are reduced to constants. If it succeeds, the second half are tried. Otherwise, the quarters, the eighth, ... are tried in a recursive way.

Note that the effects of the four reduction strategies are not independent. For example, even if the bottom-up minimization becomes no more applicable, it often turns effective after some other minimization steps. Based on this observation, we construct the overall minimization flow as shown in Fig. 15. First, expression elimination (1) is tried until it does not eliminate any more expression. Then, top-down minimization (2) is attempted until it is not applicable, and then bottom-up minimization (3) is applied. If (3) has some effects, then (2) is tried again followed by (3). If (3) failed but (2) succeeded, then (1), (2), and (3) is repeated. When none of (2) and (3) have an effect, then value and type minimization (4) is applied. If there is any update in (4), then the whole process is repeated from the beginning. Otherwise, the procedure ends. Binary search is done in (1) and (3) only for the first time. This is because only a little reduction is observed after the first iteration, for which the binary search is less efficient than linear search.

6. Experimental Results

Random test systems based on the proposed method and the previous method in Ref. [12] have been implemented in Perl (version 5.10), which run on Windows Cygwin, Mac OSX, Ubuntu Linux, etc.

In order to evaluate the effect of longer expressions and multiple expressions, 6 versions of GCCs were tested under the four settings; (1) previous (single short expression) method [12], (2) single long expression mode, (3) multiple short expression mode, and (4) multiple and long expression mode. This version of the random test system implemented arithmetic operations \{ +, -, *, /, %, <<, >>, ==, !=, <, <=, >, >= \} \textsuperscript{a}. The test was run in “integer only” mode where generated test programs contained only integer arithmetic (did not contain floating point arithmetic) and undefined behavior was eliminated by operation insertion. The options tested were -O0 and -O3.

The results are summarized in Table 2. The figures in “ops \times expr” column apply for modes (2), (3), and (4). When ops \times expr = 10,000, for example, the target number of operations per expression in a test program for modes (2) was 10,000. The target number of operations per expression was 4 for modes (1) and (3). When ops \times expr = 10,000, a test program for mode (3) contained 2,500 expressions of target length 4. In mode (4), the numbers of the expressions and the operations per expression were randomly determined per program. So the test programs generated by (4) include both of those by (2) and (3).

The subcolumns “#err,” and “#pat” show the number of the programs that resulted in errors, and the number of different patterns of the error programs after minimization, respectively. Two programs were decided to be of the same pattern if the syntax trees of the expressions in the programs were the same; the operators and the types should match exactly but the values of the constants (the initial values of variables) might be different.

We can say that both long expressions and multiple expressions contributed to improve error detection capabilities. The effects of (2) and (3) depended on compilers, but (3) did little better than (2). However, in general, (4) exhibited stable performance, for the test programs generated by (4) include both of those generated by (2) and (3).

Table 3 shows the results of test runs for 8 versions of GCCs, 5 of which are newer than 4.5.0. The tests were conducted in integer only mode and the the optimization option examined was -O3. This version of the random test system generated logical operations \{ |, & |, || & | \} as well as the arithmetic operations, and avoids undefined behavior by operation insertion and operation flipping. Tests were run for 24 hours for the first 7 versions and 80 hours for the last version. In the previous method (1), each test program consisted of a single expressions with four operations, while in the proposed method (4) the numbers of the expressions and of the operators per expression in each program were determined randomly so that their product was 1,000. The subcolumns “#test,” “#err,” and “#pat” show the number of tests generated, the number of the programs that resulted in errors, and

\textsuperscript{a} Unary and ternary operations were not supported. This was due to minor implementation reasons and there is no theoretical difficulty.

\begin{table}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline

<table>
<thead>
<tr>
<th>Setting</th>
<th>ops \times expr</th>
<th>#test</th>
<th>#err</th>
<th>#pat</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(2)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(3)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(4)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(5)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(6)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(7)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>(8)</td>
<td>100</td>
<td>98</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
</tr>
</tbody>
</table>
\end{tabular}
\end{table}
the number of different patterns of the error programs after mini-
mimization, respectively. The proposed method found more errors
than the previous method. Especially, the new method succeeded
in finding bugs in GCCs whose versions are newer than 4.5.0.

Comparison with our random testing system and Csmith [7]
was also performed on three versions of GCCs. Table 4 shows the
result. The settings of the “proposed method” is the same as those
in Csmith Had detected many bugs in the earlier ver-
sions of GCCs which had been already fixed. However, we can
talk about the number of errors in GCCs of versions from at least 3.1.0 through 4.7.2, regardless of targets
and optimization options. This type of bugs are difficult to find
by such a method as Csmith that rely on the di-ferential testing.

Table 3 Experimental results (comparison with previous method).

<table>
<thead>
<tr>
<th>compiler</th>
<th>time [h]</th>
<th>ops × exps (for 2(3)/4)</th>
<th>(1) previous [12]</th>
<th>(2) long expr</th>
<th>(3) multi expr</th>
<th>(4) multi long expr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#err (#pat)</td>
<td>#err (#pat)</td>
<td>#err (#pat)</td>
<td>#err (#pat)</td>
</tr>
<tr>
<td>LLVM-GCC 4.2.1 (i686 apple)</td>
<td>12 *A</td>
<td>10,000</td>
<td>0 ( 0)</td>
<td>15 ( 1)</td>
<td>15 ( 3)</td>
<td>33 ( 13)</td>
</tr>
<tr>
<td>GCC 4.2.1 (i686 apple)</td>
<td>12 *A</td>
<td>10,000</td>
<td>0 ( 0)</td>
<td>1 ( 1)</td>
<td>14 ( 5)</td>
<td>3 ( 3)</td>
</tr>
<tr>
<td>GCC 4.4.1 (m32e linux)</td>
<td>6 *B</td>
<td>1,000</td>
<td>68 ( 4)</td>
<td>11 ( 1)</td>
<td>571 ( 6)</td>
<td>428 ( 4)</td>
</tr>
<tr>
<td>GCC 4.4.1 (arm linux)</td>
<td>12 *B</td>
<td>5,000</td>
<td>0 ( 0)</td>
<td>0 ( 0)</td>
<td>35 ( 9)</td>
<td>20 ( 8)</td>
</tr>
<tr>
<td>GCC 4.4.4 (i686 linux)</td>
<td>12 *B</td>
<td>5,000</td>
<td>0 ( 0)</td>
<td>2 ( 2)</td>
<td>4 ( 4)</td>
<td>21 ( 18)</td>
</tr>
<tr>
<td>GCC 4.5.3 (i686 cygwin)</td>
<td>12 *B</td>
<td>3,000</td>
<td>0 ( 0)</td>
<td>19 ( 19)</td>
<td>4 ( 3)</td>
<td>30 ( 29)</td>
</tr>
</tbody>
</table>

Table 4 Experimental results (comparison with Csmith).

<table>
<thead>
<tr>
<th>compiler</th>
<th>Ssmith [7]</th>
<th>proposed method</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#test #err</td>
<td>#test #err</td>
<td>#exp #err</td>
<td>#exp #err</td>
<td></td>
</tr>
<tr>
<td>GCC 4.4.4 (i686 linux)</td>
<td>18,257</td>
<td>1</td>
<td>6,709</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>GCC 4.5.3 (i686 cygwin)</td>
<td>13,253</td>
<td>0</td>
<td>6,611</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>GCC 4.5.4 (i686 linux)</td>
<td>24,756</td>
<td>0</td>
<td>6,686</td>
<td>183</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 Experimental results (effect of integer & floating mode).

<table>
<thead>
<tr>
<th>compiler</th>
<th>time [h]</th>
<th>ops × exps (for 1000)</th>
<th>(1) previous [13]</th>
<th>(2) proposed method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#test #err (#pat)</td>
<td>#test #err (#pat)</td>
</tr>
<tr>
<td>GCC 4.4.1 (arm linux)</td>
<td>24</td>
<td>643,840</td>
<td>423 ( 27)</td>
<td>38,177 ( 81)</td>
</tr>
<tr>
<td>GCC 4.4.4 (i686 linux)</td>
<td>24</td>
<td>613,919</td>
<td>5 ( 4)</td>
<td>41,468 ( 188)</td>
</tr>
<tr>
<td>GCC 4.5.4 (x86_64)</td>
<td>24</td>
<td>621,881</td>
<td>0 ( 0)</td>
<td>43,871 ( 96)</td>
</tr>
<tr>
<td>GCC 4.6.3 (x86_64)</td>
<td>24</td>
<td>616,461</td>
<td>0 ( 0)</td>
<td>44,924 ( 94)</td>
</tr>
<tr>
<td>GCC 4.6.5 (i386)</td>
<td>24</td>
<td>610,167</td>
<td>0 ( 0)</td>
<td>45,308 ( 99)</td>
</tr>
<tr>
<td>GCC 4.7.0 (x86_64)</td>
<td>24</td>
<td>620,059</td>
<td>0 ( 0)</td>
<td>46,447 ( 100)</td>
</tr>
<tr>
<td>GCC 4.8.0 (x86_64)</td>
<td>80</td>
<td>611,526</td>
<td>0 ( 0)</td>
<td>44,401 ( 171)</td>
</tr>
<tr>
<td>GCC 4.9.0 (x86_64)</td>
<td>90</td>
<td>1,983,077</td>
<td>0 ( 0)</td>
<td>151,080 ( 615)</td>
</tr>
</tbody>
</table>

Table 6 Extra operations to avoid undefined behavior.

<table>
<thead>
<tr>
<th>program size (#ops)</th>
<th>previous [13]</th>
<th>proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.38</td>
<td>0.22</td>
</tr>
<tr>
<td>100</td>
<td>5.07</td>
<td>0.72</td>
</tr>
<tr>
<td>1,000</td>
<td>49.51</td>
<td>30.77</td>
</tr>
</tbody>
</table>

Figure 16 shows examples of error programs that detected
bugs in the latest versions of GCCs and LLVM. (a) is one of the
three error programs for GCC 4.7.2 in Table 3. The pro-
gram was further hand minimized after the automatic reduc-
tion. It turned out that this program caused the same error on the GCCs
of versions from at least 3.1.0 through 4.7.2, regardless of targets
and optimization options. This type of bugs are difficult to find
by such a method as Csmith that rely on the differential testing.
The error program (b) detected “internal compiler error”
in GCC 4.8.0 for x86_64 and i686 with -02 option (more pre-
cisely, with options -01 -ftree-vrp). The LLVM SVN as of
May 10, 2013 (version 3.3 under development) miscompiled the
program in (c). The compiled code printed “%G (t=1)”. (d) is

avoid undefined behaviors by the previous method (operation in-
novation only) [13] and proposed method (with operation flipping).
“Program size” refers to the target number of the operations per test
program, which is the product of the number of expressions
and the number of the operations per expression in the program.
“Previous” and “proposed” show the average number of opera-
tions inserted to avoid undefined behavior by the two methods.
In the previous method, about 5% of operations had to be added.
This was reduced to about 3% in our new method.

© 2014 Information Processing Society of Japan
it important to test arithmetic optimization.
mization of GCCs are continuingly being updated, so we consider
relatively recent bugs. It seems that routines for arithmetic opti-
in GCCs were from more than 19 years ago but the rests were
experimental) *9 and 5 bugs in LLVMs (SVN)*10. 2 bugs out of 8
we have so far reported 8 bugs in GCCs (4.7.2 through 4.9.0 ex-
miscompiled this program (compiled code printed "-O1 -ftree-vrp" option crashed (internal compiler error).

(a) GCC 4.7.2 (for almost all the targets) miscompiled this program (compiled code printed "NG (t=0)").

(b) GCC 4.8.0 for Linux (x86_64 and i686) and Mac OS X (x86_64) with "-O1 -fleece-vrp" option crashed (internal compiler error).

c) LLVM (SVN as of May 10, 2013) for Linux (x86_64) with -01 option miscompiled this program (compiled code printed "NG (t=1)").

(d) LLVM-3.3 for Linux (x86_64) with -O3 option crashed (internal compiler error).

Fig. 16 Examples of error programs.

an example of error programs which was detected by integer &
floating mode. It contains operations of type
Examples of error programs.

7. Conclusion

An enhanced method of testing validity of arithmetic optimization
of C compilers using random programs has been presented
in this paper. The compiler testing system is able to detect bugs
which cannot be found by the existing methods, and has revealed
several bugs in the very latest versions of GCCs and LLVMs.

Compiler random testing based on precomputation of pro-
grams’ expected behavior seems to have great potential to un-
cover bugs which are difficult by the differential testing. How-
ever, our random program generator currently covers only small
portion of the C language as compared with Csmith. We are now
trying to extend our method to handle pointers, arrays, structs/unions, as well as loop and conditional statements.

Acknowledgments Authors would like to thank Mr. Masatoshi Nakahashi of Kwansei Gakuin University who helped
us in conducting the experiments. We would also like to thank
Mr. Ryo Nakamura who helped us refactoring the test generation
program, Mr. Kazuhiro Nakamura and all the members of Ishiura
Lab. of Kwansei Gakuin University for their discussion and
advices on this research.

References


© 2014 Information Processing Society of Japan
Eriko Nagai received her B.E. and M.E. degrees from School of Science and Technology, Kwansei Gakuin University in 2011 and 2013, respectively, and was engaged in the research on testing of system software including compilers. Since April 2013, she has been with Fujitsu Systems West Ltd., Japan.

Atsushi Hashimoto received his B.E. degree from Kwansei Gakuin University in 2013, and now is pursuing his M.E. degree at Graduate School of Science and Technology, Kwansei Gakuin University. His current interest includes testing and performance enhancement of compilers.

Nagisa Ishiura received his B.E., M.E., and Ph.D. degrees in Information Science from Kyoto University, Kyoto, Japan, in 1984, 1986, and 1991, respectively. In 1987, he joined the Department of Information Science, Kyoto University, where he was Instructor until April 1991. He joined the Department of Information Systems Engineering, Osaka University, Osaka, Japan, as Lecturer where he was promoted to Associate Professor in December 1994. Since 2002, he has been Professor at School of Science and Technology, Kwansei Gakuin University, Hyogo, Japan. His current research interests include compilers for embedded processors, hardware/software codesign, and high-level synthesis. He is a member of IEEE, ACM, and IEICE.

(Recommended by Associate Editor: Toshinori Hosokawa)