IPSJ Transactions on System LSI Design Methodology Vol.7 91–100 (Aug. 2014)

[DOI: 10.2197/ipsjtsldm.7.91]

Regular Paper

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

Eriko Nagai[1,†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 continuingly 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 suits 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 [*1] and LLVM [*2].

Random testing is a complement to the testing by those test suites, which attempts to detect compiler malfunctions by huge volumes of randomly generated programs. Several random testing systems have demonstrated their bug-finding performance. Quest [5] found bugs in calling conventions (passing of arguments and return values) of C compilers. Randprog [6] detected miscompile regarding C volatile variables. Csmith [7] achieved comprehensive testing of C compilers, covering broad range of syntax in C programs, including arrays, struts/unions, conditional and loop statements, function calls, etc. Csmith is actually one of the most successful compiler test system, which reported 79 bugs in GCCs and 202 bugs in LLVMs over three years and made great contribution to improve the reliability of those open source compilers. Mettoc [8] can also handle broad range of syntax, based on a metamorphic testing technique, though no extensive experimental results as Csmith are published. Recent powerful random test generator is CCG [*3], though it only detects compiler crashes but not miscompilation. Swarm testing [9] is a technique 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-

[1] Kwansei Gakuin University, Sanda, Hyogo 669–1337, Japan
[†1] Presently with Fujitsu Systems West Ltd.
[a)] nagisa.ishiura@ml.kwansei.ac.jp

[*1] http://gcc.gnu.org/bugzilla/ (accessed 2013-11-23).
[*2] http://www.llvm.org/bugs/ (accessed 2013-11-23).
[*3] https://github.com/Merkil/ccg/tree/d45c2231906ab9bbec7b45e8011a5b6781fec1d2 (accessed 2014-03-14).

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.

Mettoc'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]. Mettoc 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 [*4]. We concentrate on arithmetic optimization because 27.8% of the bugs Csmith found in GCC were related to arithmetic optimization [7] [*5] 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.

[*4] Preliminary version of this paper appeared in Ref. [13].

[*5] 22 bugs out of 79 were classified as in modules `fold-const`, `tree-ssa-pre` , `tree-vrp`, `tree-ssa-dce`, and `tree-ssa-reassoc`.

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.

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

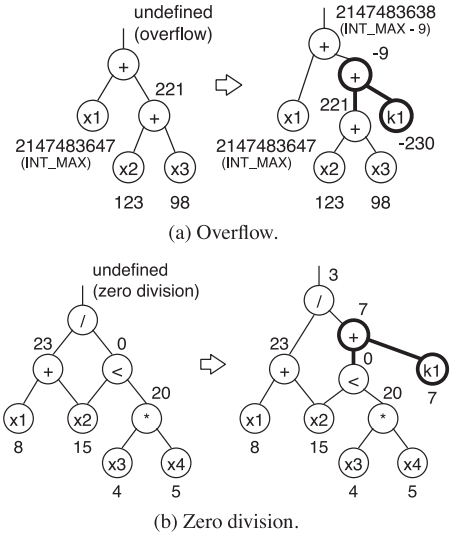
Fig. 1 Flow of compiler random testing.

```
 1: int main (void)
 2: {
 3:   int a = 60;
 4:   int b = 10;
 5:   int c = 30;
 6:   int d = 7;
 7:
 8:   int t0 = b * (a + d);  /* t0 = 670 */
 9:   int t1 = b * d - c;   /* t1 = 40 */
10:   int t2 = (a << t1) / (c > t0);
11:
12:   return 0;
13: }
```

Fig. 2 Program with undefined behavior.

```
 1: #include <stdio.h>
 2:
 3: signed long long x1 = 14766LL;
 4: static const unsigned short x2 = 3U;
 5:
 6: int main (void)
 7: {
 8:    signed int x3 = 217;
 9:    volatile unsigned char x4 = 2U;
10:
11:    int rc = 0;
12:    signed long long test = 0LL;
13:
14:    test = (((x4*(x1<<x2))>=x3)/x1);
15:
16:    if (test == 0LL) {
17:      printf("OK, %lld\n",test);
18:    }
19:    else {
20:      rc = 1;
21:      printf("NG, %lld\n",test);
22:    }
23:    return rc;
24: }
```

Fig. 3 Test program generated by the method in Ref. [12].



(a) Overflow.

(b) Zero division.

Fig. 4 Eliminating undefined behavior by operation insertion.

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.

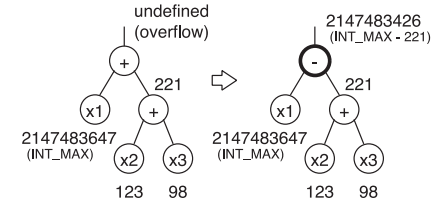
For example, suppose signed overflow is detected on a subexpression `x1+(x2+x3)` where `x1==2147483647`, `x2==123`, and `x3==98`, where we assume all the variables are of `signed int` whose maximum value (`INT_MAX`) is `2147483647`. In this case, an addition with an appropriate negative value is inserted 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.

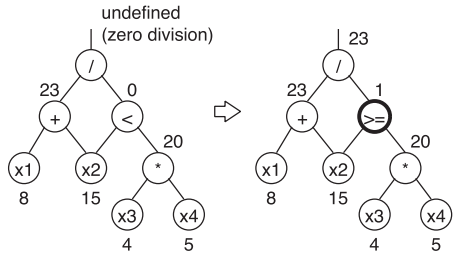
3.1.2 Eliminating Undefined Behavior by Operation Flipping

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



(a) Overflow.



(b) Zero division.

Fig. 5 Eliminating undefined behavior by operation flipping.

Table 1 Resolution of undefined behavior in integer arithmetic.

operation	condition	modification
$a+b$	signed overflow	$a-b$
$a-b$	signed overflow	$a+b$
$a*b$	signed overflow	a/b
$a/(b\ \mathtt{cmp}\ c)$	$b\ \mathtt{cmp}\ c = 0$	$a/(b\ \overline{\mathtt{cmp}}\ c)$
$a\%(b\ \mathtt{cmp}\ c)$	$b\ \mathtt{cmp}\ c = 0$	$a\%(b\ \overline{\mathtt{cmp}}\ c)$
a/b	$b = 0$	$a/(b+K)$
$a\%b$	$b = 0$	$a\%(b+K)$
$a<<b$	$\begin{cases} b<0 \\ \mathtt{width}(a)<b \end{cases}$	$a<<(b+K)$
$a<<b$	$a<0$	$(a+K)<<b$
$a>>b$	$\begin{cases} b<0 \\ \mathtt{width}(a)<b \end{cases}$	$a>>(b+K)$

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, `cmp` is a comparison operator and $\overline{\mathtt{cmp}}$ is the complement of `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.

```
node_t make_expression(n, d)
{
  if (n == 0 || d == 0) {
    return randomly chosen variable_node;
  }
  else {
    n1 = random integer ∈ [0, n − 1];
    n2 = n − 1 − n2;
    e1 = make_expression(n1, d − 1);
    e2 = make_expression(n2, d − 1);
    o = randomly chosen operator;
    return operator_node(o, e1, e2);
  }
}
```

Fig. 6 Procedure for generating expressions.

```
 1: #include <stdio.h>
 2: #define OK()  printf("@OK@")
 3: #define NG()  printf("@NG@")
 4:
 5: static unsigned long x5 = 10UL;
 6: const volatile signed long x6 = 8L;
 7: static signed int x8 = 2;
 8: unsigned long t1= 820UL;
 9:
10: int main (void)
11: {
12:   unsigned long x1 = 100UL;
13:   signed int x3 = 32;
14:   signed long t0 = 70L;
15:   unsigned long t2 = 9UL;
16:
17:   t0 = (((x8 * (x6 << x8)) >= x1) / x6);
18:   t1 = ((t0 + x3) * (x5 << x8));
19:   t2 = (((x1 + t0) - t1) * x6);
20:
21:   if (t0 == 0L) { OK(); } else{ NO(); }
22:   if (t1 == 1280) { OK(); } else{ NO(); }
23:   if (t2 == -9440L) { OK(); } else{ NO(); }
24:
25:   return 0;
26: }
```

Fig. 7 Test program with multiple expressions.

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 `t0`, `t1`, and `t2`, 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 [*6]. We assume a program to contain 1 to 10,000 expressions.

[*6] The major reason for this is that updating the same variables multiple times would increase the the complexity in minimization procedure in Section 5.

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)*(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 avoid rounding error caused by truncation on cast operation from floating point types to integer types.

(2) If the results of an addition, a subtraction, or a multiplication does not fit in the range of (1), then we apply "flipping," proposed in the previous section, to eliminate the overflow, as shown in **Fig. 9** (a).

```
 1: #include <stdio.h>
 2: int main(void)
 3: {
 4:   float c = 26;
 5:   float x1 = 1.0e9F;
 6:   float x2 = (x1 / c) * c;
 7:   int i2 = ((int) x2 % 10) * 23;
 8:    if (i2 != 0) printf("NG ($i2==%d)\n", i2);
 9:   return 0;
10: }
```

Fig. 8 An extreme example where an rounding error is amplified.

[*7] With GCC 4.8.1 for x86_64-apple-darwin12, the values of `x2` and `i2` were `1000000064.0F` and 92, respectively.

(3) 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).

(4) 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
The procedure is the same as in the previous sections but the type of each variable is chosen from `float`, `double`, and `long double` as well as integer types. For example, variables `x1` through `x4` in lines 1, 2, 5, and 6 of Fig. 10 are randomly generated variables, where two of them are of floating point types.

(2) Generate arithmetic expressions
The procedure is also the same as in the previous sections. As are commented in lines 13 and 16 of Fig. 10, the initial expressions for `t1` and `t2` are `(x3%(x2+x4))` and `(x4/x2)` and are modified in the next steps.

(3) Compute types
The type of every operation is computed in a bottom-up

```
float x1 = 4000000.0F;
float x2 = 3000000.0F;
float t = x1 + x2;
```
⇒
```
float x1 = 4000000.0F;
float x2 = 3000000.0F;
float t = x1 - x2;
```

(a) Eliminating overflow.

```
float x1 = 25.0F;
float x2 = 9.0F;

float t = x1 / x2;
```
⇒
```
float x1 = 25.0F;
float x2 = 9.0F;
float k1 = 7.0F;
float t = (x1 - k1) / x2;
```

(b) Eliminating fraction.

```
double x1 = 3239483852.0F;

int x2 = 2147483647;

int t1 = (int) x1;
int t2 = (float) x2;
```
⇒
```
double x1 = 3239483852.0F;
double k1 =-3103024134.0F;
int x2 = 2147483647;
int k2 =-2142392913;
int t1 = (int)(x1 + k1);
int t2 = (float)(x2 + k2);
```

(c) Eliminating overflow on cast.

Fig. 9 Transformation for eliminating rounding errors.

```
    ...
 1: volatile signed long x1 = -23;
 2: const float x2 = 9.0F;
 3:
 4: int main(void) {
 5:   static unsigned short x3 = 134;
 6:   double x4 = 25.0;
 7:   double k0 = 7.0;
 8:   double t0 = 9399234.0;
 9:   signed int t1 = 234;
10:
11:   t0 = (((x1+x3)*x2)-x4);
12:
13:   /* intiallly t1 = (x1%(x2+x4)); */
14:   t1 = (x1%(signed long)(x2+x4));
15:
16:   /* intiallly t2 = (x4/x2); */
17:   t2 = ((x4-k0)/x2);
      ...
```

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

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)-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 undefined behavior somewhere in the test program. In practice, a program to generator 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.

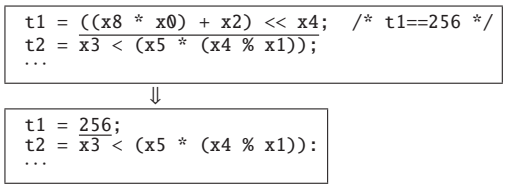


Fig. 11 Replacing expression by expected value.

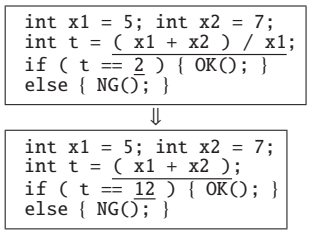
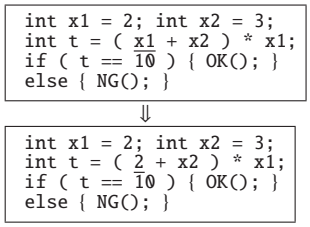
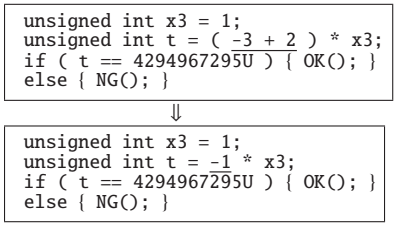


Fig. 12 Top-down minimization.



(a) Substitution.



(b) Evaluating expression.

Fig. 13 Bottom-up Minimization.

(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

```
long long x1 = 422337203685477580;
int x2 = 100;
int t = x1 + x2 << (x1 > 0);
if ( t == 422337203685477680) { OK(); }
else { NG(); }
```

⇓

```
long long x1 = 192056;
int x2 = 100;
int t = x1 + x2 << (x1 > 0);
if ( t == 192156) { OK(); }
else { NG(); }
```

(a) Value minimization.

```
long long x1 = 1;
```
⇓
```
long x1 = 1;
```

```
volatile int x2 = 4;
```
⇓
```
int x2 = 4;
```

(b) Type minimization.

Fig. 14 Minimization of types and values.

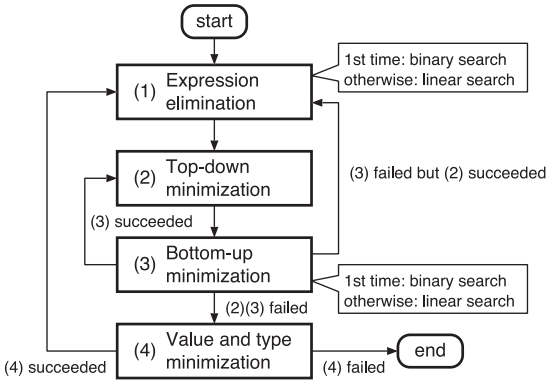


Fig. 15 Overall flow of minimization.

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 { +, -, *, /, %, <<, >>, ==, !=, <, <=, >, >= }[*8]. 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 × expr" column apply for modes (2), (3), and (4). When ops × 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 × 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

[*8] Unary and ternary operators were not supported. This was due to minor implementation reasons and there is no theoretical difficulty.

Table 2 Experimental results (effects of long and multiple expressions).

compiler	time [h]	ops × exps for (2)(3)(4)	(1) previous [12] #err	(#pat)	(2) long expr #err	(#pat)	(3) multi expr #err	(#pat)	(4) multi long expr #err	(#pat)
LLVM-GCC 4.2.1 (i686 apple)	12 *A	10,000	0	(0)	15	(11)	15	(3)	33	(13)
GCC 4.2.1 (i686 apple)	12 *A	10,000	0	(0)	1	(1)	14	(5)	3	(3)
GCC 4.4.1 (m32r linux)	6 *B	1,000	68	(4)	11	(1)	571	(6)	428	(4)
GCC 4.4.1 (arm linux)	12 *B	5,000	0	(0)	0	(0)	35	(9)	20	(8)
GCC 4.4.4 (i686 linux)	12 *B	5,000	0	(0)	2	(2)	4	(4)	21	(18)
GCC 4.5.3 (i686 cygwin)	12 *B	3,000	0	(0)	19	(19)	4	(3)	30	(29)

mode: integer only; options: {`-O0`, `-O3`};
CPU: *A Core 2 Duo 2.12 GHz, *B Core i5-2540M 2.60 GHz

Table 3 Experimental results (comparison with previous method).

compiler	time [h]	(1) previous method [12] (4 ops × 1 exp) #test	#err (#pat)	(4') proposed method (ops × exps = 1,000) #test	#err (#pat)
GCC 4.4.1 (m32r linux)	24	634,830	423 (27)	38,177	2,660 (81)
GCC 4.4.1 (arm linux)	24	613,919	5 (4)	41,468	188 (32)
GCC 4.4.4 (x86_64 linux)	24	621,881	0 (0)	43,871	96 (16)
GCC 4.5.4 (x86_64 linux)	24	616,461	0 (0)	44,924	94 (18)
GCC 4.6.3 (x86_64 linux)	24	610,167	0 (0)	45,308	99 (19)
GCC 4.6.4 (x86_64 linux)	24	620,059	0 (0)	46,447	100 (21)
GCC 4.7.0 (x86_64 linux)	24	611,526	0 (0)	44,401	171 (58)
GCC 4.8.0 (x86_64 linux)	80	1,983,077	0 (0)	151,080	6 (2)

mode: integer only; options: `-O3`; CPU: Core i7-870 2.9 GHz

Table 4 Experimental results (comparison with Csmith).

compiler	Csmith [7] #test	#err	proposed method #test	#err
GCC 4.4.4 (i686 linux)	18,257	1	6,709	74
GCC 4.5.3 (i686 cygwin)	13,253	0	6,611	198
GCC 4.5.4 (i686 linux)	24,756	0	6,686	183

mode: integer only; options: {`-O0`, `-O1`, `-O3`};
time: 24 [h]; CPU: Core i5-2540M 2.60 GHz

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

compiler	integer only #test	#err	integer & floating #test	#err (floating)
GCC 4.4.1 (arm linux)	18,118	40	16,450	19 (3)
GCC 4.4.4 (i686 linux)	20,945	54	18,608	44 (34)
GCC 4.5.4 (i686 linux)	21,342	87	18,031	43 (30)

options: `-O3`; time: 12 [h]; CPU: Core i5-2540M 2.60 GHz

Table 6 Extra operations to avoid undefined behavior.

program size (#ops)	previous [13]	proposed
10	0.38	0.22
100	5.07	3.02
1,000	49.51	30.77

the number of different patterns of the error programs after minimization, 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 for (4) in Table 3. The tests were conducted for three options `-O0`, `-O1`, and `-O3`, for Csmith needed at least three different versions or optimizing options of compilers to conduct differential testing. The run time was 24 hours for every compiler. "#test" and "#err" indicate the number of test programs generated and the number of detected errors, respectively. Our method detected much more errors than Csmith. The comparison in terms of the numbers may not be fair, for Csmith had detected many bugs in the earlier versions of GCCs which had been already fixed. However, we can at least say that the proposed method can find bugs which Csmith does not detect.

Table 5 shows the effectiveness of the integer & floating mode. The test was run for 12 hours for `-O3` option. "#Test" and "#err" again indicate the number of test programs generated and the number of detected errors, respectively, where "(float)" lists the number of the errors caused by floating point arithmetic. The integer & floating mode detected less errors within the same test time. However, it revealed floating point related errors that integer only mode can never detected.

Table 6 compares the numbers of extra operations inserted to avoid undefined behaviors by the previous method (operation insertion 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 operations 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.

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 program was further hand minimized after the automatic reduction. 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 method. The error program (b) detected "internal compiler error" in GCC 4.8.0 for x86_64 and i686 with `-O2` option (more precisely, with options `-O1 -ftree-vrp`). The LLVM SVN as of May 10, 2013 (version 3.3 under development) miscompiled the program in (c). The compiled code printed "`NG (t==1)`". (d) is

```
 1: #include <stdio.h>
 2:
 3: int main (void)
 4: {
 5:   unsigned x = 2U;
 6:   unsigned t = ((unsigned) -(x/2)) / 2;
 7:   if ( t != 2147483647 ) {
 8:     printf("NG (t==%u)\n", t );
 9:   }
10:   return 0;
11: }
```

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

```
1: int g = 0;
2: int main(void)
3: {
4:   if ( (g>>31) < -1 ) { g++; }
5:   return 0;
6: }
```

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

```
 1: #include <stdio.h>
 2:
 3: int main (void)
 4: {
 5:   volatile short x = 1;
 6:   static long k = 1L;
 7:   int a = x << ( k - 1 );   // a = 1
 8:   long t = 1L >> a ;        // t = 0
 9:   if ( t != 0L ) { printf("NG (t==%ld)\n", t); }
10:   return 0;
11: }
```

(c) LLVM (SVN as of May 10, 2013) for Linux (x86_64) with `-O1` option miscompiled this program (compiled code printed "`NG (t==1)`".

```
 1: #include <stdio.h>
 2:
 3: double x14 = 3511269280748732.0;
 4: int k132 = 1199736362;
 5: int main (void)
 6: {
 7:  volatile unsigned int x68 = 1U;
 8:  volatile int x106 = 1;
 9:  unsigned int t = 1U;
10:  t = ((((unsigned)1U-((unsigned)1U>((unsigned)1U*
     ((unsigned)2U/(((-1)+k132)<<(0<<(((int)(x14+
     (double)-3511267518266169.0))+(-1762482553)))))))
     /x106)>>((-1%(x14!=x68))/1));
11:  if (t4 != 0U) { printf("NG (t = %u)\n", t4); }
12:  return 0;
13: }
```

(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 `double`. The error disappears if `double` variable/constants are replaced by those of `long long` integer type.

We are continuingly running our random test system on the very latest versions of GCC and LLVM. Since February 2013, we have so far reported 8 bugs in GCCs (4.7.2 through 4.9.0 experimental) [*9] and 5 bugs in LLVMs (SVN) [*10]. 2 bugs out of 8 in GCCs were from more than 19 years ago but the rests were relatively recent bugs. It seems that routines for arithmetic optimization of GCCs are continuingly being updated, so we consider it important to test arithmetic optimization.

[*9] http://gcc.gnu.org/bugzilla/; bugs 56250, 56899, 56984, 57083, 57131, 57656, 57829, 58088

[*10] http://llvm.org/bugs/; bugs 15607, 15940, 15941, 15959, 16108

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 programs' expected behavior seems to have great potential to uncover bugs which are difficult by the differential testing. However, 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

[1] Plum Hall, Inc.: The Plum Hall Validation Suite for C (online), available from ⟨http://www.plumhall.com/stec.html⟩ (accessed 2013-11-23).

[2] ACE Associated Computer Experts: SuperTest compiler test and validation suite (online), available from ⟨http://www.ace.nl/compiler/supertest.html⟩ (accessed 2013-11-23).

[3] Free Software Foundation, Inc.: Installing GCC: Testing (online), available from ⟨http://gcc.gnu.org/install/test.html⟩ (accessed 2013-11-23).

[4] Fukumoto, T., Morimoto, K., and Ishiura N.: Accelerating regression test of compilers by test program merging, *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp. 42–47 (2012).

[5] Lindig, C.: Find a compiler bug in 5 minutes, *Proc. ACM Intl. Symposium on Automated Analysis-Driven Debugging*, pp.3–12 (2005).

[6] Eide, E. and Regehr J.: Volatiles are miscompiled, and what to do about it, *Proc. ACM Intl. Conf. on Embedded Software*, pp.255–264 (2008).

[7] Yang, X., Chen, Y., Eide E., and Regehr J.: Finding and understanding bugs in C compilers, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp.283–294 (2011).

[8] Tao, Q., Wu, W., Zhao, C., and Shen, W.: An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique, *Proc. IEEE 2010 Asia Pacific Software Engineering Conf.*, pp.270–279 (2010).

[9] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J.: Swarm Testing, *Proc. ACM Intel. Symposium on Software Testing and Analysis*, pp.78–88 (2012).

[10] Chen, Y., Groce, A., Zhang, C., Wong, W.-K., Fern, X., Eide, E., Regehr, J.: Taming Compiler Fuzzers, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp.197–207 (2013).

[11] McKeeman, W.M.: Differential testing for software, *Digital Technical J.*, Vol.10, No.1, pp.100–107 (1998).

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

[13] Nagai E., Atsushi H., and Ishiura N.: Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers, *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp.88–93 (2013).

[14] International Organization for Standardization: *ISO/IEC 9899:TC2: Programming Languages-C* (May 2005).

[15] Zeller, A. and Hildebrandt, R.: Simplifying and isolating failure-inducing input, *IEEE Trans. Software Engineering*, Vol.28, No.2, pp.183–200 (2002).

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*)