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

Eriko Nagai [1] Atsushi Hashimoto [2] Nagisa Ishiura [2]

[1] Solution Business Group, Business Solution Unit, Fujitsu Systems West Ltd., Chuo-ku, Osaka, Japan
[2] School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan

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 avoided by modifying problematic subexpressions during computation of expected values for the expressions. An efficient method for minimizing error inducing test programs is also 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.

I. 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. Csmith is based on a differential testing method, 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 minimize error programs for bug localization.

In contrast, the random testing method in [9] precomputes the precise expected behavior for randomly generated programs to decide the validity of compilers under test. 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. Moreover, minimization of error detecting programs becomes easier.

The random testing system based on this method found bugs in GCC 4.2.1 (apple-darwin10), GCC 4.3.4 (i686-pc-cygwin), GCC 4.4.1 (i686-pc-linux), etc. However, it was not so effective to the newer versions of the GCC; no bugs were detected in GCCs of versions higher than 4.5.0. Possible reasons for this is that the generated programs are all small. Each program contains only one arithmetic expression. Moreover, the expression tend to be short in order to avoid undefined behavior.

This paper proposes methods of substantially enhancing the bug detection capability of the random test system based on [9]. Much longer expressions are generated while avoiding undefined behavior. This is achieved by

[1] http://gcc.gnu.org/bugzilla/
[2] http://www.llvm.org/bugs/

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

Fig. 1: Overall flow of compiler random testing.

modifying invalid subexpressions as their expected values are computed. Test programs are also fortified by accommodating multiple expressions. Besides the program generation methods, this paper also presents an improved procedure for efficiently minimize large error programs.

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

II. Random testing of compilers targeting arithmetic optimization

A. 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.

A major challenge 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.

Fig. 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

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

Fig. 3: Test program generated by the method in [9].

will never be invoked and hence will not be tested. This may limit the bug detection abilities of the test programs.

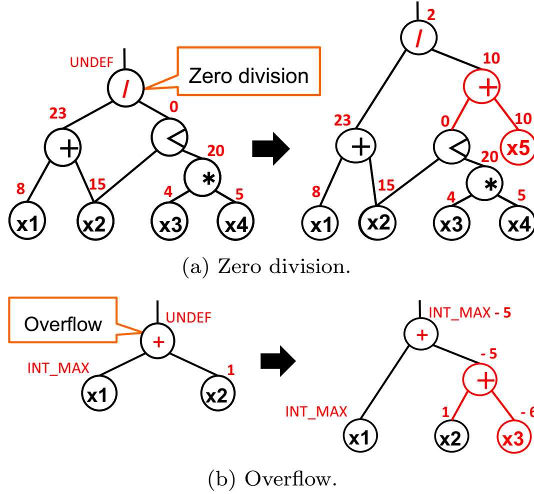
B. Random testing of arithmetic optimization

Nagai et al. [9] 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.

Fig. 3 shows an example of test programs generated by this method. Lines 3–10 declare and initialize variables, line 12 gives an arithmetic expression, and line 14 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), and its initial value are selected randomly.

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); oth-



(a) Zero division.

(b) Overflow.

Fig. 4: Avoiding undefined behavior.

erwise 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.

III. Scaling up Size and Number of Expressions

We enhance the bug detection ability of random testing method in [9] by scaling up the size and the number of the expressions generated in test programs.

A. *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. Every time we detect undefined behavior during computing the expected values of a randomly generated expression, we insert a new operator to eliminate the undefined behavior.

(1) Avoiding zero division

If the right operand of division or modulo turns out to be zero, extra addition is inserted to make the operand non-zero. For example, in Fig. 4 (a), the expression yields undefined behavior due to zero division. In this case, addition is inserted into the right operand of the division so that it will be non-zero. The non-zero value for the new variable `x5` is chosen randomly.

(2) Avoiding signed overflow and invalid shift amount

If signed overflow is detected, it is eliminated by adding an arbitrary value to either of the operands. In Fig. 4, for example, overflow (assume `x1` and `x2` are both signed integers) is avoided by adding a negative value to one of the operands. Invalid shift amounts (less than 0 or more than the width of the left operands) are corrected in the same way.

TABLE I: Undefined behavior in integer arithmetic.

operator	condition
/	right operand is 0
%	right operand is 0
<<	right operand is negative
	right operand is larger than left operand width
	left operand is signed type and negative
>>	right operand is negative
	right operand is larger than left operand width
+, -, , /,	overflow on signed operation

```
 1: #include <stdio.h>
 2: #define OK()  printf("@OK@")
 3: #define NG()  printf("@NG@")
 4:
 5: static signed int x5 = 10;
 6: const volatile signed long x6 = 8L;
 7: static signed int x8 = 2;
 8: signed int t1= 820;
 9:
10: int main (void)
11: {
12:   signed long x1 = 100L;
13:   signed int x3 = 32;
14:   signed long t0 = 70L;
15:   signed long t2 = 9;
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. 5: Test program with multiple expressions.

There are eight kinds of undefined behavior in integer arithmetic operations in C, as summarized in Table I. For all those cases, undefined behavior can be eliminated in the same way.

Expressions with desired size and depth are generated by a procedure "make_expression(n,d)" 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, 1) it randomly chooses an operator o, 2) randomly generate positive integers n_1 and n_2 where $n_1 + n_2 = n - 1$, 3) generate two subexpressions e_1 and e_2 by calling make_expression(n_1,$d-1$) and make_expression(n_2,$d-1$), respectively, and 4) returns an operand node with operator o and operands e_1 and e_2. We assume the size of expressions to be 1 to 10000.

B. *Generating programs with multiple expressions*

We also try to enhance bug detection ability by putting multiple expressions into a single test program. Fig. 5 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. We assume a program to contain 1 to 10000 expressions.

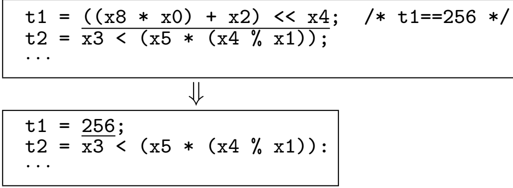


Fig. 6: Replacing expression by expected value.

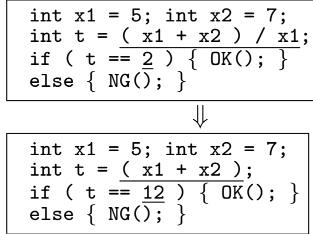


Fig. 7: Top-down minimization.

IV. 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 [9] 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.

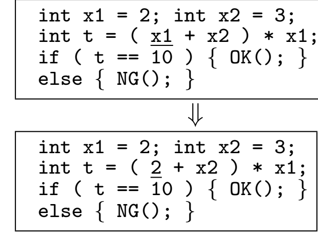
Our minimization method is based on delta debugging [11]. If a certain transformation reducing the size of an error program preserves the occurrence of the error, the transformation is adopted, otherwise another transformation is tried. By repeating this until any of the possible transformations eliminates 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 [9] and (1) and (4) are newly introduced in this paper.

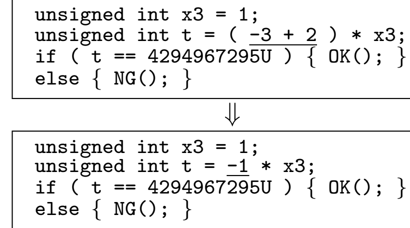
(1) Expression elimination

   Some of the expressions are replaced by their expected values, as illustrated in Fig. 6.

(2) Top-down minimization:

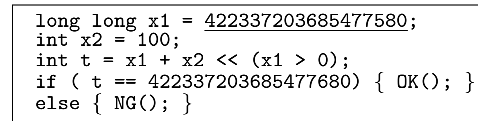


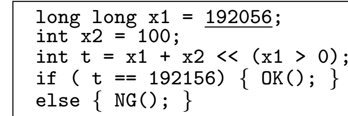
(a) Substitution



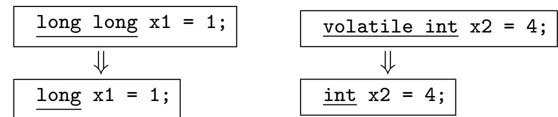
(b) Evaluating expression

Fig. 8: Bottom-up Minimization





(a) Value minimization



(b) Type minimization

Fig. 9: Minimization of types and values

   An expression is replaced by either of the two operands of the root operator, as shown in Fig. 7.

(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. 8 (a) and (b), respectively.

(4) Value and type minimization:

   The absolute values of constants are made smaller, as in Fig. 9 (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. 9 (b).

The bottom-up minimization method in [9] basically reduces the operators in an expression one by one, so it took 10000 times of compilation if an expression with 10000 operators was reduced to a constant. In order to avoid this, we introduce binary search. First, one of the operands

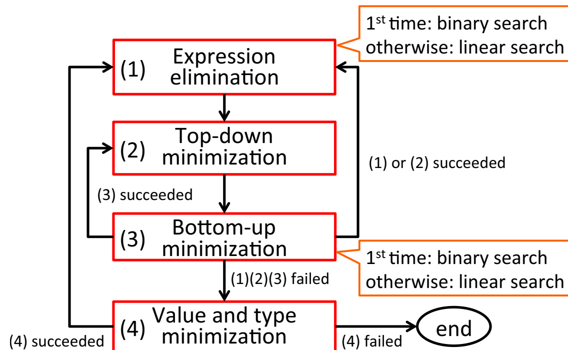


Fig. 10: Overall flow of minimization.

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 binary way, unless reduction of 10000 expressions would needs 10000 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. 10. (2) and (3) are repeated after (1), because (2) alone has little effect on (1). If the expression under test is reduced by (4), the whole process is repeated until no gain is obtained. Note that 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.

V. Implementation and experimental results

A. Implementation

Random test systems based on the proposed method and the previous method in [9] have been implemented in Perl (version 5.10), which run on Windows Cygwin, Mac OSX, Ubuntu Linux, etc.

B. Experimental results

Eight versions of GCCs were tested by the both methods. Compiler options examined were `-O0` and `-O3`. Table II summarizes the result. Column "CPU" refers to the machines on which the tests were run. "Size" lists the products of the sizes and the numbers of the expressions in a test program. "Time" and "#test" shows total execution times and the number of tests generated, respectively. "#err" indicates the number of the programs that resulted in errors and "#pat" the number of different patterns of the error programs. For the GCCs of the first two rows, the proposed method found more or the same number of error patterns within much smaller execution time. The new method succeeded in finding bugs in the latter five (newer) versions of GCCs, in which the previous method detected no errors. GCC 4.7.2 is the latest version at the point of this experiment and at least one of the three errors are due to a bug which had never been discovered.

Comparison with our random testing system and Csmith [7] was also performed on three versions of GCCs. Table III shows the result. Tests are run for 12 hours for three options, none, `-O0`, and `-O3`. The proposed method detected much more bugs 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 fixed. However, we can at least say that the proposed method can find bugs which Csmith does not detect.

Fig. 11 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 II. 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)`".

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 eight bugs in GCC (4.7.2 through 4.9.0 experimental) [3] and five bugs in LLVM (SVN) [4].

VI. 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

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

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

TABLE II: Experimental results (comparison with the previous method)

compiler (target)	CPU	size	previous method [9]				proposed method			
			time [h]	#test	#err	(#pat)	time [h]	#test	#err	(#pat)
LLVM-GCC 4.2.1 (apple-darwin10)	A	10,000	106.8	200,000	4	(3)	12.0	3,383	33	(13)
GCC 4.2.1 (apple-darwin10)	A	10,000	105.5	200,000	4	(3)	12.0	2,772	3	(3)
GCC 4.4.1 (m32r-elf)	B	1,000	6.0	47,836	68	(4)	6.0	5,038	428	(4)
GCC 4.4.1 (arm-elf)	B	5,000	12.0	24,632	0	(0)	12.0	1,718	20	(8)
GCC 4.4.4 (i686-pc-linux)	B	3,000	12.0	38,981	0	(0)	12.0	4,505	21	(18)
GCC 4.5.3 (i686-pc-cygwin)	B	3,000	12.0	26,047	0	(0)	12.0	4,296	30	(29)
GCC 4.5.4 (i686-pc-linux)	B	5,000	12.0	23,674	0	(0)	12.0	1,977	26	(26)
GCC 4.7.2 (x86_64-apple-darwin10)	A	1,000	144.0	713,851	0	(0)	68.1	50,000	3	(3)

Tested options: `-O0`, `-O3` CPU: A Core 2 Duo 2.12GHz
B Core i5-2540M 2.60GHz

TABLE III: Experimental results (comparison with Csmith [7])

compiler (target)	time [h]	Csmith [7]			proposed method		
		#test	#err	(#pat)	#test	#err	(#pat)
GCC 4.4.4 (i686-pc-linux)	12.0	9,291	1	(1)	4,179	21	(18)
GCC 4.5.3 (i686-pc-cygwin)	12.0	10,643	0	(0)	4,172	29	(28)
GCC 4.5.4 (i686-pc-linux)	12.0	12,389	1	(1)	2,236	30	(30)

Tested options: NULL, `-O0`, `-O3` CPU Core i5-2540M 2.60GHz

compared with Csmith. We are now trying to extend our method to handle pointers, arrays, structs/unions, as well as loop and conditional statements.

Acknowledgements

Authors would like to thank Mr. Masatoshi Nakahashi and all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion and advices on this research.

References

[1] http://www.plumhall.com/stec.html.

[2] http://www.ace.nl/compiler/supertest.html.

[3] http://gcc.gnu.org/install/test.html.

[4] http://ist.ksc.kwansei.ac.jp/~ishiura/pub/testgen2/.

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

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

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

[8] W. M. McKeeman: "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107 (Dec. 1998).

[9] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda: "Random testing of C compilers targeting arithmetic optimization ," in *Proc. SASIMI 2012*, pp. 48–53 (Mar. 2012).

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

[11] Andreas Zeller and Ralf Hildebrandt: "Simplifying and isolating failure-inducing input," *IEEE Trans. on Software Engineering*, vol. 28, no. 2, pp. 183–200 (Feb. 2002).


```
 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)`".

Fig. 11: Examples of error programs.