IPSJ Transactions on System LSI Design Methodology Vol.9 21–29 (Feb. 2016)

[DOI: 10.2197/ipsjtsldm.9.21]

Regular Paper

Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs

Atsushi Hashimoto[1,a)] Nagisa Ishiura[2]

Received: June 3, 2015, Revised: September 4, 2015,
Accepted: October 21, 2015

Abstract: This paper presents new methods of detecting missed arithmetic optimization opportunities for C compilers by random testing. For each iteration of random testing, two equivalent programs are generated, where the arithmetic expressions in the second program are more optimized in the C program level. By comparing the two assembly codes compiled from the two C programs, lack of optimization on either of the programs is detected. This method is further extended for detecting erroneous or insufficient optimization involving volatile variables. Two random programs differing only on the initial values for volatile variables are generated, and the resulting assembly codes are compared. Random test systems implemented based on the proposed methods have detected missed optimization opportunities on several compilers, including the latest development versions of GCC-5.0.0 and LLVM/Clang-3.6.

Keywords: compiler validation, random testing, arithmetic optimization, optimization opportunities

1. Introduction

Compilers must be highly reliable, for they are infrastructure tools for software development. If a compiler bug should result in malfunctions of the application programs, it would be a very hard and time consuming task to track down the cause, so the validity of the compilers is a crucial issue. In application domains where performance is critical, compiler reliability refers also to the performance of the generated codes in terms of the execution speed or the memory usage. Thus, compilers must be also tested if they perform intended optimization.

编译器测试的重要性

There have been many methods of validating compilers. Compiler test suites, such as Plum Hall [1], SuperTest [2], GCC (GNU Compiler Collection) test suite [3], testgen2 test suite [4], are large sets of programs to test the correctness of compilers. Although they are powerful and essential tools for compiler development, it is theoretically impossible to validate a compiler completely with a finite set of test programs, and many bugs are reported for well-developed compilers such as GCC [*1] and LLVM [*2].

接着介绍了编译器测试套件的优点和限制

Random testing is a complement to these test suites. It attempts to detect compiler malfunctions by huge volumes of randomly generated programs. Several random test systems have demonstrated their bug-finding performance on C compilers. CCG [5] is a C code generator which attempts to search for compiler crashes. Quest [6] found bugs in calling conventions (passing of arguments and return values) of C compilers. Csmith [7] achieved comprehensive testing of C compilers, which detected 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. Orange3 [8], [9] is a random test generator targeting arithmetic optimization which has reported 8 bugs and 5 bugs in the latest versions of GCCs and LLVMs, respectively.

随机程序生成器

All the above methods test the correctness of the compilers by executing generated codes and checking if they produce expected results. So they do not examine other functions which do not affect the execution results directly. For example, the compilers pass the tests even if they perform extra computation which cause unintended memory accesses or performance degradation.

There have been several attempts to detect such bugs lying under the surface. NULLSTONE [10] is a test suite targeting C compilers' optimization, which consists of about 6,500 test programs to evaluate the effects of optimizers. Since it is a test suite with a finite number of test cases, it is inevitable that its bug detection ability is limited. Randprog [11] is a random test system which detects invalid deletion of memory accesses for volatile variables by comparing the memory access traces of the two codes generated with and without an optimizing option. It detects *over-optimization* but not *under-optimization*.

关注编译器优化问题的工作以及其缺陷

This paper newly proposes a method of detecting missed opportunities for arithmetic optimization (i.e., under-optimization) in C compilers by randomly generated programs. A pair of equivalent programs, one is unoptimized and the other is optimized in the C program level, are generated and compiled assembly codes are compared. An extended version of this method is also proposed to test arithmetic optimization regarding volatile variables. A pair of programs which differ only on the initial values of volatile variables are generated and the resulting assembly codes are compared to examine if optimization is performed as

本文的方法简述

[1] Nomura Research Institute, Ltd., Chiyoda, Tokyo 100–0005, Japan
[2] Kwansei Gakuin University, Sanda, Hyogo 669–1337, Japan
[a)] atsushi.hashimoto@kwansei.ac.jp

[*1] http://gcc.gnu.org/bugzilla/ (accessed 2015-05-06).
[*2] http://www.llvm.org/bugs/ (accessed 2015-05-06).

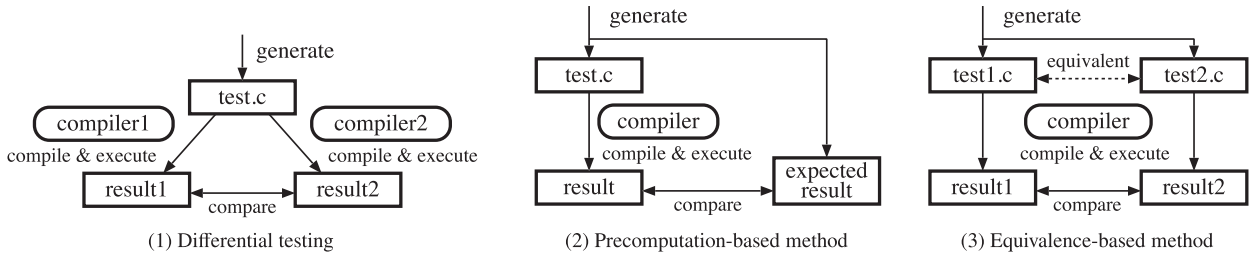


Fig. 1 Three methods for compiler random testing.

intended. Random test systems implemented based on the proposed methods have detected missed optimization opportunities on several compilers, including the latest development versions of GCC and LLVM/Clang.

2. Related Work

2.1 Random Testing of Compilers

The overall flow of compiler random testing is simple; random test program generation, compile and execution, and error checking are repeated as long as time allows. If errors are detected, error programs (programs caused the errors) are minimized (or reduced); smaller programs that still trigger the same errors are searched, automatically or manually, to make bug localization easier.

If we are interested in crash test like CCG [5], any test programs conforming to the language syntax, or even any random character string will work as the test programs. However, if we want to test the correctness of code generation, it involves two major challenges: (1) how to decide the correctness of the compiled codes for randomly generated programs, and (2) how to avoid generating test programs with *undefined behavior*. As the test programs grow larger and contain more syntax features, it becomes more difficult to tell the correct answers that the programs should produce. The undefined behavior includes zero division, overflowing a signed integer, dereferencing a null pointer, out of bounds array accesses, etc., for which the standard [12] imposes no requirements to the computation results. A test program with any undefined behavior is of no use, since any execution results are valid for such a program. It is very difficult to generate large scale random programs without undefined behavior, for it depends on dynamic behavior of the programs.

The existing methods for compiler random testing are classified into three categories; (1) differential testing methods, (2) precomputation based methods, and (3) equivalence-based methods.

(1) Differential testing

Differential testing [13] tries to test a compiler by comparing the execution result with that obtained by the other compiler (or other version of the same compiler or with different compiler option), as shown in **Fig. 1** (1). More than two compilers may be used to decide the correctness by voting. Differential testing eliminates the necessity of preparing expected results for the test programs and thus solves the first challenge above. Based on this approach, Csmith [7] achieved comprehensive testing of C compilers. It detected 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. However, this approach does not resolve the second issue. In Csmith, undefined behavior is eliminated in a conservative way. For example, it guards every divide operation as "`(b!=0)?(a/b):(a)`" instead of "`a/b`." Since every arithmetic operation is always guarded, some optimizers will never be invoked and will not be tested. This may limit the bug detection abilities of the test programs.

(2) Precomputation-based methods

This approach tries to overcome the both of the two challenges by computing the expected behavior, including every intermediate computation result, of a test program while it is generated. If undefined behavior is detected, the program is modified so that the behavior is well defined. Orange3 [8], [9] is based on this approach. For example, if zero division is detected during test program generation on a subexpression `a/(b<c)` where `b` is known to be smaller than `c`, the expression is altered into `a/(c<b)`. Similarly, signed overflow on integer addition, subtraction, and multiplication are eliminated by replacing the operations with subtraction, addition, and division, respectively. Out of range on the right operand of shifts is resolved by inserting addition or subtraction to fit the operand in a proper range. This approach enables generation of more sophisticated test cases than the differential approach, but it is applicable to limited classes of C programs. Orange3 detected bugs in the latest versions of GCC and LLVM which had not been detected by Csmith, but it can test only arithmetic optimization.

(3) Equivalence based methods

This approach gives two equivalent programs to a compiler and check if the compiled codes yield equivalent execution results. This resolves the first of the two challenges. In Mettoc's method [15], variants are generated from correct test programs, though not so many transformations to generate large classes of programs to detect many errors as Csmith are not presented. Le et al. [16] proposed a method of generating variants from a test program which are equivalent with respect to the same inputs. Orion, a test tool based on this method has reported 147 unique bugs for GCC and LLVM. The second of the two challenges still remains unresolved, but proved test cases can be used as seeds for the variants.

Some types of compiler bugs can not be detected by simply comparing the final values computed by the the compiled codes. Even though the outputs of the codes are correct, intended optimization might have not been applied, or necessary memory accesses regarding volatile variables might be eliminated.

There have been some efforts to detect such incompleteness. NULLSTONE [10] is a test suite targeting compiler optimization. It consists of about 6,500 test programs that evaluate the effects of

Table 1 Transformations for error program minimization in Orange3. (1) Expression elimination, (2) Top-down minimization, (3) Bottom-up minimization, and (4) Value and type minimization.

	before	after

(1)	`t1 = ((x8*x0)+x2);`
	`t2 = x5*(x4%x1);`
	`...`
		`t1 = 256;`
		`t2 = x5*(x4%x1):`
		`...`

(2)	`int x1 = 5;`
	`int x2 = 7;`
	`int t = (x1+x2)/x1;`
	`if ( t==2 ) OK();`
	`else NG();`
		`int x1 = 5;`
		`int x2 = 7;`
		`int t = (x1+x2);`
		`if ( t==12 ) OK();`
		`else NG();`

(3)	`int x1 = 2; int x2 = 3;`
	`int t = (x1+x2)*x1;`
	`if ( t==10 ) OK();`
	`else NG();`
		`int x1 = 2; int x2 = 3;`
		`int t = (2+x2)*x1;`
		`if ( t==10 ) OK();`
		`else NG();`

	`int x3 = 1;`
	`int t = (-3+2)*x3;`
	`if ( t==-1 ) OK();`
	`else NG();`
		`int x3 = 1;`
		`int t = -1*x3;`
		`if ( t==-1 ) OK();`
		`else NG();`

(4)	`long x1 = 42233720;`
	`int x2 = 100;`
	`int t = (x1+x2)<<(x1<3);`
	`if ( t==42233820 ) OK();`
	`else NG();`
		`long x1 = 28;`
		`int x2 = 100;`
		`int t = (x1+x2)<<(x1<3);`
		`if ( t==128 ) OK();`
		`else NG();`

	`long long x1 = 1;`	`long x1 = 1;`
	`volatile int x2 = 4;`	`int x2 = 4;`

more than 40 optimizing transformations of the compilers. Since it is a test suit consisting of a finite number of test cases, it is inevitable that its bug detection ability is limited. Randprog [11] is a random test generator which tries to detect invalid optimization regarding volatile variables (which must be accessed exactly as written in the source codes). It compiles and executes test programs involving volatile variables both with and without optimizing options and compares the two memory access traces. If the numbers of loads and stores on each volatile variable are different, then miscompile (over-optimization) is detected. However, this method is not able to find under-optimization where volatile variables may block necessary optimization.

2.2 Minimization of Error Programs

Minimization of error programs is another important issue, for the test programs in random testing can be of thousands of lines and it is virtually impossible to locate the causes of the errors without boiling down the error programs. The most popular minimization method is delta debugging [17], in which transformations to reduce the size of error programs are applied repeatedly as long as the programs produce the error.

C-Reduce [14] is a general minimizer which takes a C program triggering an error as an input and outputs a minimized C program. It is based on transformations to reduce the size of C programs and static analysis to avoid undefined behavior.

Orange3 [9] implements its own minimizer which can handle only programs that Orange3 generates but runs much faster. **Table 1** summarizes the transformations used in Orange3; (1) expression elimination replaces some of the expressions by their expected values; (2) top-down minimization substitutes an expression by one of the operands of the root operator (the expected values of the expressions are recomputed, accordingly); (3) bottom-up minimization replaces a variable reference or an operation by its expected value; and (4) value and type minimization makes the

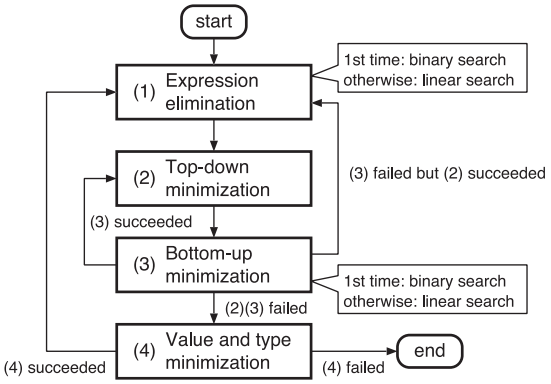


Fig. 2 Overall flow of error program minimization in Orange3 [9].

absolute values of constants smaller and types simpler. **Figure 2** is the overall flow of minimization. Basically, each of (1) through (4) is applied until it has no more effect. If some transformations are adopted, preceding steps are retried. The repetition terminates when none of the transformations is applicable.

Note that this kind of minimization procedure does not guarantee that the results are *minimum*. The results are dependent on the order of transformations, so a smaller error program might be obtained by a different sequence of transformations. However, the results are minimal in the sense that the programs cannot be made smaller by any of the transformations without eliminating the errors.

3. Random Testing for Detecting Arithmetic Optimization Opportunities

3.1 Overview

In this section, a random testing method for detecting arithmetic optimization opportunities missed by C compilers is proposed. It is based on the equivalence based method shown in Fig. 1 (c). For each test iteration, two equivalent programs, one is unoptimized and the other is optimized in the C language level, are given to the compiler under test. The compiler's optimization ability is evaluated by comparing the resulting assembly codes, instead of executing the codes.

Figure 3 shows the dataflow of the test in each iteration. First, a set of abstract syntax trees (ASTs) representing random arithmetic expressions is constructed. While a C program (`org.c`) containing the arithmetic expressions is generated from the ASTs, another set of trees (ASTs') is derived from the original ASTs by applying tree-level optimization (such as constant propagation and constant folding), from which another C program (`opt.c`) is generated. The two C programs are compiled and the resulting assembly codes (`org.s` and `opt.s`) are compared.

If the compiler is not able to perform an equivalent level of optimization as performed on the ASTs, there should be some difference on the two assembly codes. Moreover, the test may detect cases where stronger optimization is performed on `opt.c` but not on `org.c` when such optimization is blocked by some redundant dataflow in `org.c` or some other earlier transformation on `org.c` eliminate the conditions to fire the stronger optimization.

3.2 Test Program Generation

The original (unoptimized) ASTs are constructed in the same

way as in Orange3 [9]. First, a set of variables with randomly determined types and initial values are generated. Then, binary trees of random shapes are generated, and operators and variables are randomly assigned to the internal nodes and the leaf nodes, respectively. The correct value of every subtree in the trees is computed in a bottom-up manner. If a subtree results in undefined behavior, the subtree is modified so that the undefined behavior will be eliminated.

Then, optimized ASTs are created by reducing each of the original ASTs. Any type of tree optimization is applicable, but in this paper we focus on constant propagation (substitution of variables by their values) and constant folding (replacement of subexpressions by their expected values). These optimizing transformations are performed by replacing each variable or operator node by a constant node. However, the reduction must not be done on nodes that depend on volatile variables, whose values might be updated from outside at any time during execution.

For this purpose, all the nodes on the paths from the volatile variable nodes to the root node are marked as volatile and the other nodes as normal. Then, the tree is traversed in a depth first manner and the subtrees whose root nodes are marked as normal

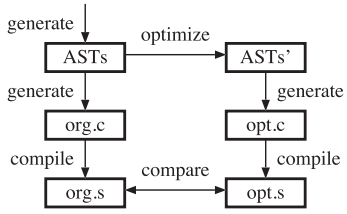


Fig. 3 Dataflow of testing for detecting optimization opportunities.

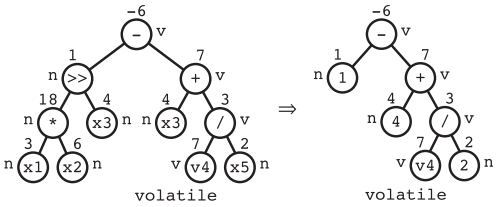


Fig. 4 Optimization of AST.

are replaced by nodes with the corresponding constants. **Figure 4** shows an example, where `v4` is a volatile variable. All the ancestor nodes of `v4` are marked as volatile ("v") and all the other nodes as normal ("n"). Every time a normal node is encountered during depth first tree traversal, the subtree rooted by the node is replaced by a constant node. In our method, variables declared as `const volatile` are treated in the same way as `volatile` variables, though compilers may perform stronger optimization on `const volatile` variables than `volatile` variables.

Figure 5 is an example pair of the test programs (`org.c` and `opt.c`) generated by our method. The both programs have assign statements with arithmetic expressions in lines 21–23, where possible constant propagation and constant folding have been performed in `opt.c`. Some variables are declared global (in lines 5–9) and others local (in lines 12–19). Although nonvolatile variables are not used in `opt.c`, all the variables declared in `org.c` are also declared in `opt.c`. This is to avoid unnecessary changes on the assembly code. In spite that the compiled codes are compared but not executed, the programs compare the results with the expected values in lines 25–28. If not, optimizers would eliminate all the assign statements in lines 21–23 whose left hand size variables are never referenced. These programs were generated with a parameter $\#op = 10$ which specifies the target number of the operations per test program. We assume that the value to this parameter is set between 1 to about 10,000, depending on the qualities of the compilers under test. The number of assign statement (which is smaller than $\#op$) is determined randomly per program.

3.3 Comparison of Assembly Codes

Even if the compiler under test performs desired optimization on both test programs `org.c` and `opt.c`, the resulting assembly codes `org.s` and `opt.s` are not always identical. The same set of instructions may appear in different order. Instructions of the same operations but of different data sizes or of different addressing modes may be used. Actually, it is not a simple task to judge if codes are under-optimized and it might only be concluded by compiler designers. Thus, our goal here is to enumerate suspi-

org.c

```
01: #include <stdio.h>
02: #define OK() printf("@OK@\n")
03: #define NG(fmt,val) printf("@NG@ (t = "fmt")\n", val)
04:
05: volatile int x0 = -9;
06: static signed long x2 = -1L;
07: signed long t0 = -2314269L;
08: unsigned long long t2 = 22682838128721517LLU;
09:
10: int main (void)
11: {
12:   static signed short x5 = -451;
13:   volatile unsigned int x8 = 876U;
14:   unsigned long long x9 = 8614LLU;
15:   signed short x11 = 7932;
16:   const volatile int x12 = -3732;
17:   unsigned long long t1 = 10LLU;
18:   static volatile signed long k13 = -3717L;
19:   signed long k14 = 319664938L;
20:
21:   t0 = (x2/((x2+k14)>>((x12^x2)+k13)));
22:   t1 = ((x9*(x8+x5))/x11);
23:   t2 = ((x12/x12)|(x0|t1));
24:
25:   if (t0 == 0L) { OK(); } else { NG("%ld", t0); }
26:   if (t1 == 461LLU) { OK(); } else { NG("%llu", t1); }
27:   if (t2 == 18446744073709551615LLU) { OK(); }
28:   else { NG("%llu", t2); }
29:
30:   return 0;
31: }
```

opt.c

```
01: #include <stdio.h>
02: #define OK() printf("@OK@\n")
03: #define NG(fmt,val) printf("@NG@ (t = "fmt")\n", val)
04:
05: volatile int x0 = -9;
06: static signed long x2 = -1L;
07: signed long t0 = -2314269L;
08: unsigned long long t2 = 22682838128721517LLU;
09:
10: int main (void)
11: {
12:   static signed short x5 = -451;
13:   volatile unsigned int x8 = 876U;
14:   unsigned long long x9 = 8614LLU;
15:   signed short x11 = 7932;
16:   const volatile int x12 = -3732;
17:   unsigned long long t1 = 10LLU;
18:   static volatile signed long k13 = -3717L;
19:   signed long k14 = 319664938L;
20:
21:   t0 = (-1L/(319664937L>>((x12^-1L)+-3717L)));
22:   t1 = ((8614LLU*(x8+-451))/7932);
23:   t2 = ((x12/x12)|(-9|t1));
24:
25:   if (t0 == 0L) { OK(); } else { NG("%ld", t0); }
26:   if (t1 == 461LLU) { OK(); } else { NG("%llu", t1); }
27:   if (t2 == 18446744073709551615LLU) { OK(); }
28:   else { NG("%llu", t2); }
29:
30:   return 0;
31: }
```

Fig. 5 Example pair of test programs.

cious cases which will be examined later. For this purpose, two empirical measures are used.

If the numbers of the instructions in the two assembly codes are different, it is possible that one of them (in most cases `org.s`) is under-optimized. Let n and n' be the numbers of the instructions in `org.s` and `opt.s`, respectively. Then the first measure is defined as:

$$r_1 = \frac{n'}{n}. \tag{1}$$

Smaller r_1 means that `opt.s` has fewer instructions than `org.s`, so it is likely that `org.s` under-optimized when r_1 is smaller than a threshold.

Miss of optimization opportunities may also be predicted by examining how different the instructions used in the two assembly codes are. This is seized by counting the the numbers of the instructions of the same operations. For example, x86 instructions `addb` (8 bit), `addw` (16 bit), `addl` (32 bit), and `addq` (64 bit) are all classified as add instructions, in spite that their data sizes and addressing modes are different. Let n_o and n'_o be the numbers of the instructions of operation o in `org.s` and `opt.s`, respectively. Let m and m' be the numbers of the operations of the instructions used in `org.s` and `opt.s`, respectively, and let m_u be the number of the operations o where $n_o = n'_o$ holds. For example, the instruction counts in **Table 2** results in $m = 7$, $m' = 6$, and $m_u = 2$ (for $n_{srl} = n'_{srl}$ and $n_{xor} = n'_{xor}$). Then, the second measure is defined as:

$$r_2 = \frac{m_u}{\left(\frac{m+m'}{2}\right)}. \tag{2}$$

Smaller r_2 means more different operations are used in the two files, so there should be under-optimization on `org.s` when r_2 is smaller than a proper threshold.

In our method, the geometric mean of the r_1 and r_2 is used as the overall measure.

$$r = \sqrt{r_1 \cdot r_2}\,. \tag{3}$$

Namely, the test case is classified as potentially under-optimized when r is smaller than a threshold.

3.4 Minimization of Error Programs

The test cases that detected potential under-optimization are minimized for close examination. The same minimization strategy as Orange3 can be used, with slight modifications, to minimize a pair of programs simultaneously.

Table 2 Example of instruction counts.

operation	#instructions	
	`org.s`	`opt.s`
`add`	350	235
`sub`	100	98
`imul`	56	23
`idiv`	8	-
`shl`	32	38
`srl`	25	25
`xor`	12	12

This distribution results in $m = 7$, $m' = 6$, $m_u = 2$, and $r_2 = \frac{2}{\left(\frac{7+6}{2}\right)} \approx 0.31$.

Every time one of the minimizing transformations listed in Table 1 is attempted on unoptimized ASTs, a pair of C programs `org.c` and `opt.c` are generated in the same way described in Section 3.2. If the reduced test case still resulted in r smaller than the threshold, then the transformation is adopted, otherwise it is cancelled. A single threshold is used throughout the minimization procedure. The procedure terminates when none of the transformations is applicable any more.

4. Random Testing of Optimization Involving Volatile Variables

Equivalence-based random test is further extended in this paper to test over- or under-optimization regarding volatile variables.

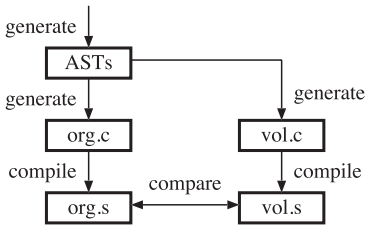
Volatile variables are the variables that might be read or written by other processes or hardware devices outside of the program in which the variables are declared, and that must be loaded or stored exactly as described in the program. Thus,

(1) compilers must not perform optimization to delete loads and stores of volatile variables, but

(2) compilers should perform optimization wherever volatile variables are irrelevant.

Our second method presented in this paper uses a pair of programs, which differ only on initial values of volatile variables, as a test case and compares the compiled code to examine (1) and (2) above. **Figure 6** (a) shows the dataflow of test by this method. Random ASTs are constructed from which `org.c` is generated, as is in Orange3. Then, the other program `vol.c` is generated from the same ASTs, where different initial values are given to volatile variables (as shown in (b)). The two resulting assembly codes `org.s` and `vol.s` are compared by the same criteria as in the previous section.

Since compilers must not perform optimization using the knowledge on the values of the volatile variables, the assembly codes `org.s` and `vol.s` must be the same except for the part that initialize the variables. By comparing the two assembly codes, over-optimization (deletion of some instructions that must not be deleted) on either of the codes will be detected. At the same time,



(a) Dataflow of each test iteration.

```
org.c
01: int main (void)
02: {
03:  volatile int t1 = 13;
04:  int x1 = 12;
05:  volatile int x2 = 3;
06:  volatile int x3 = 2;
07:  t1 = (x1 + x2) >> x3;
08:  if (t1 == 3) OK();
09:  else NG();
10:  return 0;
11: }
```

```
vol.c
01: int main (void)
02: {
03:  volatile int t1 = 12;
04:  int x1 = 12;
05:  volatile int x2 = 2;
06:  volatile int x3 = 1;
07:  t1 = (x1 + x2) >> x3;
08:  if (t1 == 3) OK();
09:  else NG();
10:  return 0;
11: }
```

(b) Example pair of test programs.

Fig. 6 Outline of random testing of optimization involving volatile variables.

under-optimization due to some confusion involving volatile variables will be also detected. Any initial values will do for the volatile variables in `vol.c`. Note that the initial values to `const volatile` variables must be same in the both programs (line 16), for different initial values to those variables lead to different assembly codes. Since we do not run the code generated from `vol.c`, the expected values in line 8 of `vol.c` are not recomputed.

Comparison and minimization are performed in the same way as described in Sections 3.3 and 3.4, respectively.

5. Implementation and Experimental Results

5.1 Implementation

A random test system based on the proposed methods has been implemented using the libraries of Orange3 [*3]. It is written in Perl 5.20 and runs on Windows Cygwin, Max OS X, Ubuntu Linux, etc.

Table 3 summarizes the types and the operators used in random program generation. Signed and unsigned integer types were used but floating point types were not. This is because they would compile into dedicated extended instructions such as SSE by which small changes on source code would result in large difference in the assembly codes and yet it would be difficult to decide which code sequences ran faster. Most of the binary arithmetic operators were used, but the modulo operator ("%") was excluded from the test program, for it lead to many move instructions and makes the error judgement difficult.

5.2 Result of Testing for Arithmetic Optimization Opportunities

The test was run on 10 versions of the compilers including GCC, LLVM/Clang, SunCC, and IntelCC. The parameter to control the number of operations per test program (#*op*) was set to 500, taking into account that 1,000 had been enough for miscompile detection in Orange3 for GCCs of version 4.4 through 4.8 [9] and that one of the compilers under test this time (SunCC) was tested for more time consuming optimization options. The test was run for 12 hours for each version where the CPU was Intel(R) Core(TM) i7–4930K 3.40 GHz with 15.6 GiB RAM [*4]. The threshold of r was set to 80%. This value was determined empirically: smaller threshold than 70% detected almost no differences while almost all test cases were decided as positive with threshold larger than 90%.

Table 3 Types and operators used in test program generation.

types	signed, unsigned char, short, int, long, long long
scopes	local, global
classes	none (auto), static
modifiers	none, const, volatile, const volatile
operators	arithmetic (+, -, *, /) logical (&&, ||) comparison/relational (==, !=, <, >, <=, >=) bitwise (<<, >>, &, |, ^) type-conversion

[*3] https://github.com/ishiura-compiler/Orange3 (accessed 2015-05-30).

[*4] We used the same parameters for the both experiments in Table 4 and Table 6 for comparison, though smaller run time was enough for the experiment of Table 4.

The result is summarized in **Table 4**. Column "opt" shows the tested optimization options, "#test" the numbers of pairs tested within the run time of 12 hours, and "#diff" the numbers of test cases that detected differences on the compiled codes with r smaller than the threshold.

Large number of test cases detected differences on the assembly codes for all the compilers under test. As will be described later, those actually included cases that detected under-optimization. However, we could not conclude that all the cases were due to under-optimization. This is partly because all the test cases were not well minimized, and partly because we could not tell the pairs of assembly codes had really performance differences for all the minimized cases. Thus, #diff in this experiment does not necessary serve as a measure of the strength of compiler optimizers, though we can see that the newer versions of the same compiler series have the smaller difference counts.

Table 5 summarizes the result of minimization where the first 100 test cases (as indicated in column "#diff") that had detected differences in the experiment of Table 4 for each of GCC-4.8.2 and LLVM/Clang-3.3. The CPU was Intel(R) Core(TM) i7–5500U 2.40 GHz with 7.7 GiB RAM. Column "ave time" shows the average run time per test case, and subcolumns "before" and "after" under "ave #op" are the numbers of operators in the test programs (`org.c` in Fig. 3) before and after minimization, respectively. "#min" indicates the number of well minimized test cases which contained less than 10 operators.

The minimizer reduced the size of test programs in several seconds on average. The longest runtime among 200 cases were 47.3 seconds. Not all but more than 80% of the test programs were minimized to the size of less than 10 operators, of which code inspection or further manual minimization was possible. The maximum number of the operators in the minimized program among 200 cases were 31. Out of the 94 and 82 minimized test cases for GCC-4.8.2 and LLVM/Clang-3.3, respectively, we

Table 4 Result of test for arithmetic optimization opportunities.

compiler (target)	opt	#test	#diff
GCC-4.4.7 (A)	-O3	53,374	30,852
GCC-4.6.4 (A)	-O3	53,113	27,416
GCC-4.7.3 (A)	-O3	53,237	24,222
GCC-4.8.2 (A)	-O3	54,000	20,119
GCC-5.0.0* (B)	-O3	49,574	17,925
LLVM/Clang-2.8 (B)	-O3	61,008	1,110
LLVM/Clang-3.3 (B)	-O3	60,495	1,105
LLVM/Clang-3.6** (B)	-O3	55,804	434
SunCC-5.12 (C)	-O5	33,552	3,994
IntelCC-15.0.1 (D)	-O3	45,847	22,187

* version 5.0.0 20141010 (experimental)
** version 3.6.0 (trunk 217856)

time: 12 (h), size: 500, r: 80 (%)
CPU: Intel(R) Core(TM) i7–4930K 3.40 GHz, RAM: 15.6 GiB
A: x86_64-pc-linux-gnu, B: x86_64-unknown-linux-gnu,
C: linux-i386, D:Intel(R)-64

Table 5 Result of minimization for optimization opportunity test.

compiler	#diff	ave time [sec]	ave #op		#min (#op< 10)
			before	after	
GCC-4.8.2	100	1.62	529.9	3.14	94
LLVM/Clang-3.3	100	3.19	524.9	5.90	82

CPU: Intel(R) Core(TM) i7–5500U 2.40 GHz, RAM: 7.7 GiB

org.c

```
01: int main (void)
02: {
03:  static   int a = 1;
04:  volatile int b = 1;
05:
06:  int c = (a/(0+b))>=2;
07:
08:  if (c == 0);
09:  else __builtin_abort();
10:
11:  return 0;
12: }
```

opt.c

```
01: int main (void)
02: {
03:  static   int a = 1;
04:  volatile int b = 1;
05:
06:  int c = (1/(0+b))>=2;
07:
08:  if (c == 0);
09:  else __builtin_abort();
10:
11:  return 0;
12: }
```

(a) Minimized pair of test programs.

org.s (LLVM/Clang-3.6 -O3)

```
01:   .text
02:   .file   "org.c"
03:   .globl  main
04:   .type   main,@function
05: main:           # @main
06:   .cfi_startproc
07: # BB#0:          # %entry
08:   pushq   %rax
09: .Ltmp0:
10:   .cfi_def_cfa_offset 16
11:   movl    $1, 4(%rsp)
12:   movl    4(%rsp), %eax
13:   leal    1(%rax), %ecx
14:   cmpl    $2, %ecx
15:   ja      .LBB0_2
16: # BB#1:          # %entry
17:   cmpl    $2, %eax
18:   jge     .LBB0_3
19: .LBB0_2:          # %if.end
20:   xorl    %eax, %eax
21:   popq    %rdx
22:   retq
23: .LBB0_3:          # %if.else
24:   callq   abort
25: .Ltmp1:
26:   .size   main,.Ltmp1-main
27:   .cfi_endproc
```

opt.s (LLVM/Clang-3.6 -O3)

```
01:   .text
02:   .file   "opt.c"
03:   .globl  main
04:   .type   main,@function
05: main:           # @main
06:   .cfi_startproc
07: # BB#0:          # %entry
08:   movl    $1, -4(%rsp)
09:   movl    -4(%rsp), %eax

10:   xorl    %eax, %eax

12:   retq
13: .Ltmp0:
14:   .size   main,.Ltmp0-main

15:   .cfi_endproc
```

(b) Assembly codes generated from (a).

org-gcc.s (GCC-4.8.2 -O3)

```
01:   .file    "org.c"
02:   .section .text.startup, "ax",@progbits
03:   .p2align 4,,15
04:   .globl   main
05:   .type main, @function
06: main:
07: .LFB0:
08:   .cfi_startproc
09:   movl     $1, -4(%rsp)
10:   movl     -4(%rsp), %eax
11:   xorl     %eax, %eax
12:   ret
13:   .cfi_endproc
14: .LFE0:
15:   .size    main, .-main
16:   .ident   "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
17:   .section .note.GNU-stack, "",@progbits
```

(c) Assembly code for (a) generated by GCC.

Fig. 7 Detected optimization opportunity for LLVM/Clang-3.6.

judged by assembly code inspection that 38 and 49 cases, respectively, had detected under-optimization. For the other cases, we could not decide if each pair of codes were of different performance, though the codes looked different. Interestingly, it turned out that `opt.c` compiled to less optimized code than `org.c` in 1 out of the 38 cases for GCC-4.8.2 and in 34 out of the 49 cases for LLVM/Clang-3.3. As is shown in the following paragraphs, detected under-optimization was not always of constant propagation nor constant folding. We guess this is because small changes on input C programs might have big impact on how chains of optimizers process intermediate representation.

Figure 7 shows the result of minimization of a test case on LLVM/Clang-3.6 [*5] (with `-O3` option), which originally consisted of 851 lines. The C codes `org.c` and `opt.c` differ only on line 6, where variable `a` in `org.c` is replaced by constant 1, which should compile into the same assembly codes. However, the resulting assembly codes shown in (b) are very different; `org.s` contains

[*5] clang version 3.6.0 (trunk 217334) (x86-unknown-linux-gnu)

org.c

```
01: int main (void)
02: {
03:  int a = -1;
04:  volatile unsigned b = 1U;
05:  int c =  1;
06:
07:  c = (a+972195718)>>(1LU<=b);
08:
09:  if (c == 486097858);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

opt.c

```
01: int main (void)
02: {
03:  int a = -1;
04:  volatile unsigned b = 1U;
05:  int c =  1;
06:
07:  c = 972195717>>(1LU<=b);
08:
09:  if (c == 486097858);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

(a) Minimized pair of test programs.

org.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   subq    $24, %rsp
05:   .cfi_def_cfa_offset 32
06:   movl    $1, 12(%rsp)
07:   movl    12(%rsp), %eax
08:   testl   %eax, %eax
09:   movl    $972195717, %eax
10:   setne   %cl
11:   sarl    %cl, %eax
12:   cmpl    $486097858, %eax
13:   jne     .L5
14:   xorl    %eax, %eax
15:   addq    $24, %rsp
16:   .cfi_remember_state
17:   .cfi_def_cfa_offset 8
18:   ret
19: .L5:
20:   .cfi_restore_state
21:   call    abort
22:   .cfi_endproc
        .
        .
        .
```

opt.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   subq    $24, %rsp
05:   .cfi_def_cfa_offset 32
06:   movl    $1, 12(%rsp)
07:   movl    12(%rsp), %eax
08:   testl   %eax, %eax

09:   jne     .L2
10:   call    abort
11: .L2:
12:   xorl    %eax, %eax
13:   addq    $24, %rsp
14:   .cfi_def_cfa_offset 8
15:   ret

16:   .cfi_endproc
        .
        .
        .
```

(b) Assembly codes generated from (a).

Fig. 8 Detected optimization opportunity for GCC-4.8.2 (-O3).

a redundant code sequence. For comparison, `org.c` was compiled by GCC-4.8.2 (with `-O3` option) to get the code `org-gcc.s` in (c), which is equal to `org.s` in the essential part. Thus, it was concluded that the optimizer of this version of LLVM/Clang had room for improvement. This case was reported to the bug database of LLVM/Clang [*6] and modification was made in response. Note that this test case did not detect missing of the simple optimization to propagate constant 1 into variable `a` (as expressed in `org.c` to `opt.c`). It revealed instead that much stronger optimization, which should eliminates all the computations regarding the expression on line 06 in `opt.c`, was blocked when constant propagation was not explicitly expressed in the C program `org.c`. Thus, the proposed method may detect missing of more sophisticated optimization, such as variable range propagation, other than those performed on ASTs during test case generation.

Figure 8 shows a minimized test case for GCC-4.8.2 with `-O3` option. The two C programs in (a) differ only on line 07 and expected to compile to the same code. However, assembly codes (b) are different; computation of the branch condition is not folded away in `org.s`. This case was reported to the bug database of GCC [*7] after confirming the same syndrome still appeared on the development version of GCC-4.8.4 [*8]. Modification on the compiler has not been made but the cause of under-optimization was discussed.

In our current minimization procedure, succinct programs as Figs. 7 and 8 were not obtained from all of the test cases listed in

[*6] http://llvm.org/bugs/ bug #20916
[*7] http://gcc.gnu.org/bugzilla/ bug #61839
[*8] gcc version 4.8.4 20140622 (prerelease) (x86-unknown-linux-gnu)

Table 6 Result of test for optimization involving volatile variables.

compiler (target)	opt	#test	#diff
GCC-4.4.7 (A)	-O3	45,795	5
GCC-4.6.4 (A)	-O3	45,204	15
GCC-4.7.3 (A)	-O3	44,157	13
GCC-4.8.2 (A)	-O3	44,986	12
GCC-5.0.0* (B)	-O3	40,510	10
GCC-5.0.0* (B)	-Os	49,220	12
LLVM/Clang-2.8 (A)	-O3	47,582	0
LLVM/Clang-3.3 (A)	-O3	46,843	0
LLVM/Clang-3.6** (B)	-O3	42,973	3
SunCC-5.12 (C)	-O5	32,260	253
IntelCC-15.0.1 (D)	-O3	36,363	16,952

CPU, parameters, and targets are same as Table 4

the column "#diff" in Table 4. There were cases where assembly codes differ for large C programs but not after applying any minimizing transformation. There should be a lot of room for improvement in the method of comparing assembly codes, especially in the criteria of deciding the codes are different.

It should be also noted that our test detected the lack of other optimization than constant folding and constant propagation which was implemented in the test generator. We consider that small changes on test programs make big difference on how chains of optimizers are invoked, and thus our method may be effective on compilers with sophisticated optimization passes. Missing of the stronger optimization such as variable range propagation might be easily detected by implementing the same optimization in AST level in the test generator. Other tree optimization such as strength reduction may be worth implementing in the test generator to enhance effectiveness of the test.

5.3 Result of Testing of Optimization Involving Volatile Variables

Test of volatile variable related optimization is performed with the same settings of the compilers, the CPU, and the parameters as in Section 5.2. The result is summarized in **Table 6**. Column "opt" shows the tested optimization options, "#test" the numbers of pairs tested within the run time of 12 hours, "#diff" the numbers of test cases that detected potential over- or under-optimization.

A fewer differences were detected than those in Section 5.2. We examined all the "#diff" cases except for those for IntelCC. All the test programs were successfully auto-minimized to the size of less than 10 operators, and it was confirmed that all the differences were due to under-optimization (namely, no over-optimization was detected).

Figure 9 shows a minimized test case for GCC-5.0.0 [*9] (with `-Os` option; original C program was of 691 lines). The two source programs in (a) differ only on the initial values of the volatile variable `c` (line 5). The resulting assembly codes should be different only on the initialization of the variables, but the two codes in (b) are different; `org.s` contains a redundant code sequence which should be optimized away. This test case was reported to the bug database of GCC [*10], and modification was made in response.

[*9] gcc version 5.0.0 20141215 (experimental) (x86-unknown-linux-gnu)
[*10] http://gcc.gnu.org/bugzilla/ bug #64322

org.c

```
01: int main (void)
02: {
03:           long a = -1L;
04:  volatile long b =  0L;
05:  volatile long c =
      0x100000000L;
06:
07:  a = (((1+b)>>63)<<1) /
      0x100000000L;
08:
09:  if (a == 0L);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

vol.c

```
01: int main (void)
02: {
03:           long a = -1L;
04:  volatile long b =  0L;
05:  volatile long c = 0L;
06:
07:  a = (((1+b)>>63)<<1) /
      0x100000000L;
08:
09:  if (a == 0L);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

(a) Minimized pair of test programs.

org.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:    .cfi_startproc
04:    subq   $24, %rsp
05:    .cfi_def_cfa_offset 32
06:    movabsq $4294967296, %rcx
07:    movq   $0, (%rsp)
08:    movq   %rcx, 8(%rsp)
09:    movq   (%rsp), %rax
10:    incq   %rax
11:    sarq   $63, %rax
12:    addq   %rax, %rax
13:    cqto
14:    idivq  %rcx
15:    testq  %rax, %rax
16:    je  .L2
17:    call   abort
18: .L2:
19:    xorl   %eax, %eax
20:    addq   $24, %rsp
21:    .cfi_def_cfa_offset 8
22:    ret
23:    .cfi_endproc
24: .LHOTE0:
        .
        .
        .
```

vol.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:    .cfi_startproc
04:    movq   $0, -24(%rsp)
05:    movq   $0, -16(%rsp)
06:    movq   -24(%rsp), %rax
07:    xorl   %eax, %eax
08:    ret
09:    .cfi_endproc
10: .LHOTE0:
        .
        .
        .
```

(b) Assembly codes generated from (a).

Fig. 9 Detected optimization opportunity for GCC-5.0.0 (-Os).

6. Conclusion

New methods of detecting missed arithmetic optimization opportunities for C compilers based on random testing have been proposed. The effectiveness of the methods were shown through experiments, in which under-optimization cases for the latest versions of GCC and LLVM/Clang were detected.

However, not all the test cases which detected potential under-optimization were not properly minimized. Further research should be done on the minimization procedure or the criteria of comparing assembly codes. Incorporating optimizing transformations on abstract syntax trees other than constant folding and constant propagation would be future work to improve the performance of the proposed method.

Acknowledgments Authors would like to thank Mr. Mitsuyoshi Iwatsusji who helped us conduct experiments. We would also thank all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion and advices on this research. This work was partly supported by JSPS KAKENHI Grant Number 25330073.

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] Balestrat, A.: CCG: A random C code generator, available from ⟨https://github.com/Merkil/ccg/⟩ (accessed 2014-03-14).
[6] Lindig, C.: Find a compiler bug in 5 minutes, *Proc. ACM Intl. Symposium on Automated Analysis-Driven Debugging*, pp.3–12 (2005).
[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] Nakamura, K. and Ishiura, N.: Introducing Loop Statements in Random Testing of C Compilers Based on Expected Value Calculation, in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies* (*SASIMI 2015*), pp.226–227 (2015).
[9] Nagai, E., Hashimoto, A. and Ishiura, N.: Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions, *IPSJ Trans. System LSI Design Methodology*, Vol.7, pp.91–100 (2014).
[10] Nullstone Corporation: NULLSTONE for C (online), available from ⟨http://www.nullstone.com/⟩ (accessed 2014-12-10).
[11] 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).
[12] International Organization for Standardization: *ISO/IEC 9899:TC2: Programming Languages-C* (May 2005).
[13] McKeeman, W.M.: Differential testing for software, *Digital Technical J.*, Vol.10, No.1, pp.100–107 (1998).
[14] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C. and Yang, X.: Test-Case Reduction for C Compiler Bugs, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp.335–346 (2012).
[15] 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).
[16] Le, V., Afshari, M. and Su, Z.: Compiler Validation via Equivalence Modulo Inputs, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp.216–226 (2014).
[17] Zeller, A. and Hildebrandt, R.: Simplifying and isolating failure-inducing input, *IEEE Trans. Software Engineering*, Vol.28, No.2, pp.183–200 (2002).

Atsushi Hashimoto received his B.E. and M.E. degrees from School of Science and Technology, Kwansei Gakuin University in 2013 and 2015, respectively, and was engaged in the research on testing of system software including compilers. Since April 2015, he has been with Nomura Research Institute, Ltd., Japan.

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: *Keiji Kimura*)