Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing

Mitsuyoshi Iwatsuji [1] Atsushi Hashimoto [2] Nagisa Ishiura [1]

[1] Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] Nomura Research Institute, Ltd., Tokyo, Japan

Abstract— This paper presents a method of detecting missed optimization opportunities in C compilers by differential random testing. By compiling randomly generated test programs by two different compilers, or different versions of the same compiler, and comparing the resulting pairs of assembly codes, lack of optimization is detected. Comparison is based on the instruction count. An experimental test system has successfully detected under-optimization in the latest versions of GCC and Clang/LLVM.

I. Introduction

Compilers must be highly reliable, for they are infrastructure tools for software development. In addition, the performance of generated codes is also a critical issue. Thus, compilers must be tested for their performance as well as for their correctness.

There have been a few attempts to detect performance bugs. Nullstone [1] is a test suite for C compilers' optimization consisting of about 6,500 test programs. As it is a finite set of test cases, it is inevitable that its detection ability is limited. Randprog [2] tries to find invalid deletion of memory accesses for volatile variables by random testing. It detects *over-optimization* but not *under-optimization*. [3] proposed a method of detecting under-optimization in C compilers by random testing. A pair of equivalent programs, one is unoptimized and the other is optimized in the C language level, are compiled and assembly codes are compared. However, in principle it can not find lack of stronger optimization than that performed in the C language level.

To address this issue, this paper presents yet another method of detecting missed optimization opportunities in C compilers by differential random testing [4]. Randomly generated test programs are compiled by different compilers and the resulting codes are compared to detect missed optimization. A test system based on our method has detected several under-optimization in the latest versions of GCC and Clang/LLVM.

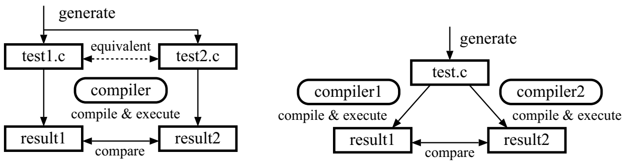
II. Random testing of C compilers

After validation using test suites, compilers are often tested by randomly generated programs to detect potential bugs. Csmith [5] and Orange [6] are examples of successful random test systems for C compilers.

In this paper, we employ Orange3 [6] as a random test generator. Fig. 1 shows an example of a test program. It verifies (in lines 14–15) the resulting values of the arithmetic expressions (in lines 12–13). The expres-

```
 1: #include <stdio.h>
 2: #define OK() printf("@OK@\n")
 3: #define NG(fmt,val) printf("@NG@ (test="fmt")\n",val)
 4: const signed int k8 = 144011145;
 5: int main (void) {
 6:   signed short x1 = 6;
 7:   static unsigned short x2 = 1U;
 8:   static signed long x4 = 4542636934L;
 9:   volatile signed short x7 = -1;
10:   signed long t0 = -1261917469307207940L;
11:   signed long t1 = -42079927921L;
12:   t0 = (x2^(x4+((x7+k8)>>x2)));
13:   t1 = ((x4<<(x4/t0))/x1);
14:   if (t0 == 4614642507L) {OK();} else {NG("%d", t0);}
15:   if (t1 == 757106155L) {OK();} else {NG("%lld", t1);}
16:   return 0;
17: }
```

Fig. 1. Test program generated by Orange3.



(a) Method in [3]. (b) Proposed method.

Fig. 2. Approaches for optimization test.

sions are constructed carefully not to yield undefined behavior (such as zero division or signed overflow). The size of the programs can be adjusted to the compiler under test, which typically ranges from 500 to 10,000 lines.

Once an error is detected, the program must be minimized (reduced to an error reproducing program as small as possible) for closer investigation. This is achieved by applying various size reducing program transformations as long as the error is reproduced. Orange3 has a built-in minimizer along with the random program generator.

Although the random testing is a powerful tool to detect compilers' crash or wrong code generation, lack of optimization is hard to detect by the same method, for the compiled codes yield the same results even though intended optimization was not performed.

III. Detecting missed optimization by differential testing

This paper proposes a random test method to detect missed compiler optimization by differential testing [4]. While [3] compared the assembly codes generated from a pair of equivalent C programs (as in Fig. 2 (a)), we compile a randomly generated program (test.c) by two different compilers, or different versions of the same compiler, and compare the resulting two assembly codes (test1.s and test2.s), as illustrated in Fig. 2 (b). The test program that yields a significant difference on the assembly

TABLE I
Experimental results.

compiler (target) option	time [h]	#test	#diff
GCC-5.0.0 (x86_64) -O3 Clang-3.7.0 (x86_64) -O3	24 *1	62,672	593
GCC-6.0.0 (x86_64) -O3 GCC-5.3.0 (x86_64) -O3	24 *2	60,536	185

*1 Core i5-4200U 1.60GHz, RAM 7.7GB, Ubuntu 14.10
*2 Core i7-5500U 2.40GHz, RAM 7.7GB, Ubuntu 14.04

```
error1.c
int main (void) {
    volatile signed int x = 1;
    unsigned int t = ((unsigned int)1U<<x);
    if (t == 2U) ;
    else __builtin_abort();
    return 0;
}
```

```
gcc.s (GCC-5.0.0 -O3)
main:
    subq   $24, %rsp
    movl   $1, %eax
    movl   $1, 12(%rsp)
    movl   12(%rsp), %ecx
    sall   %cl, %eax
    cmpl   $2, %eax
    jne    .L5
    xorl   %eax, %eax
    addq   $24, %rsp
    ret
.L5:
    call   abort
```

```
clang.s (Clang-3.7.0 -O3)
main:
    pushq %rax
    movl  $1, 4(%rsp)
    movl  4(%rsp), %eax
    cmpl  $1, %eax
    jne   .LBB0_2


    xorl  %eax, %eax
    popq  %rdx
    retq
LBB0_2:
    callq abort
```

Fig. 3. Error program (1).

codes is called an error program, which may detect under-optimization in one of the assembly codes. This method can detect lack of optimizing transformations in one of the compilers, which was not always possible in [3]. The proposed method can be also used for regression test; if an older version produces better codes than the latest version, there must be some degeneration.

There are many possible ways of comparing assembly codes. In this paper, we adopt a simple method based on the instruction count. This is based on our observation that the assembly codes from the same C program are usually very similar. Let n and m be the numbers of the instructions in the two assembly codes where $n < m$. Then, two codes are judged to be different if n/m is smaller than a threshold.

Error programs are minimized for closer investigation. The same set of program transformations as Orange3 is used. For example, if `x1` and `x2` are known to evaluate to 3 and 5, respectively, expression `x1+x2` is reduced to `x1+5`, then to `3+5` and `8`. Reduction is repeated automatically as long as the resulting assembly codes reproduce differences.

IV. Experimental results

A test system based on the proposed method has been developed in Perl5 and runs on Linux. Results of experiments are summarized in Table I. The first test was run on GCC-5.0.0 and Clang-3.7.0 with the -O3 option. In 24 hours, 62,672 cases were tested and 593 programs detected differences in the assembly codes. The other test was on different versions of GCC. In 24 hours, 185 out of 60,536 programs detected differences. The thresholds for comparison were set to 0.6 and 0.7 in the first and the second runs, respectively. All the error programs were manually inspected after minimization. Among them, there were cases where we could not tell if they detected lack of optimization, but some revealed defects.

```
error2.c
unsigned int x = 1;
int main (void) {
    long long int a = -2LL;
    int t = 1 <= (a/x);
    if (t != 1) { __builtin_abort(); }
    return 0;
}
```

```
gcc5.s (GCC-5.2.1 -O3)
main:
    xorl %eax, %eax






    ret
```

```
gcc6.s (GCC-6.0.0 -O3)
main:
    movl x(%rip), %ecx
    movq $-2, %rax
    cqto
    idivq %rcx
    testq %rax, %rax
    jg .L7
    xorl %eax, %eax
    ret
.L7
    pushq %rax
    call abort
```

Fig. 4. Error program (2).

Fig. 3 shows one of the error programs in the first test. Code gcc.s faithfully computes the sequence specified in source code error1.c, for `x` is a volatile variable. On the other hand, clang.s just tests if `x==1` and omits all the other computation. We can conclude that GCC-5.0.0 has missed an opportunity for this optimization. Fig. 4 is an error program from the second test run. Code gcc5.s is apparently succinct than gcc6.s, which means that there has been some regression in GCC-6.0.0. We have so far reported 2 bugs to Clang/LLVM[1], and 3 bugs to GCC[2] detected by our test system.

V. Conclusion

The paper has presented a new random test method to detect lack of arithmetic optimization in C compilers. The current system generates a considerable number of false positive error programs, so we are now working on improvement on the method of assembly code comparison.

Acknowledgement

Authors would like to thank all the members of Ishiura Lab. of Kwansei Gakuin University. This work was partly supported by JSPS KAKENHI Grant Number 25330073.

References

[1] Nullstone Corporation: Nullstone for C (online), `http://www.nullstone.com/`.

[2] E. Eide and J. Regehr: "Volatiles are miscompiled, and what to do about it," in *Proc. EMSOFT*, pp. 255–264 (Oct. 2008).

[3] A. Hashimoto and N. Ishiura: "Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs," *IPSJ Trans. SLDM*, vol. 9, pp. 21–29 (Feb. 2016).

[4] R. B. Evans and A. Savoia: "Differential testing: A new approach to change detection," in *Proc. ACM ESEC-FSE*, pp. 549–552 (Sept. 2007).

[5] X. Yang, Y. Chen, E. Eide, and J. Regehr: "Finding and understanding bugs in C compilers," in *Proc. ACM PLDI*, pp. 283–294 (June 2011).

[6] E. Nagai, A. Hashimoto, and N. Ishiura: "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Trans. SLDM*, vol. 7, pp. 91–100 (Aug. 2014).

[1] `https://llvm.org/bugs/` id=23673, 23672
[2] `https://gcc.gnu.org/bugzilla/` id=66299, 68026, 68431