Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing

Mitsuyoshi Iwatsuij 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 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 expressions 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
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 \( m \) and \( n \) 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 smaller than a threshold. Error programs are minimized for closer investigation. Then, two codes are judged to be different if

\[
\text{error1.c}
1: \text{int main (void) }
2: \{ 
3: \text{volatile signed int x = 1; }
4: \text{if } (t == 20) ;
5: \text{else _builtin_abort(); }
6: \text{return 0; }
7: \}
\]

\[
\text{error2.c}
1: \text{unsigned int x = 1;}
2: \text{int main (void) }
3: \{ 
4: \text{long long int a = -2L; }
5: \text{int t = (unsigned int)10<<x);}
6: \text{if (t != 1) }
7: \}
\]

Fig. 3. Error program (1).

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


\(^1\)https://llvm.org/bugs/ id=23673, 23672

\(^2\)https://gcc.gnu.org/bugzilla/ id=66299, 68026, 68431