Random Testing of Compilers' Performance Based on Mixed Static and Dynamic Code Comparison

Kota Kitaura
Kwansei Gakuin University
Sanda, Hyogo, Japan
kota.kitaura@kwansei.ac.jp

Nagisa Ishiura
Kwansei Gakuin University
Sanda, Hyogo, Japan
nagisa.ishiura@ml.kwansei.ac.jp

ABSTRACT

This paper proposes an automated test method for detecting performance bugs in compilers. It is based on differential random testing, in which randomly generated programs are compiled by two different compilers and resulting pairs of assembly codes are compared. Our method attempts to achieve efficient and accurate detection of performance difference, by combining dynamic measurement of execution time with static assembly-level comparison and test program minimization. In the first step, discrepant pairs of code sections in the assembly codes are extracted, and then the sums of the weights of discrepant instructions in the sections are computed. If significant differences are detected, the test program is reduced to a small program that still exhibits the static difference and then the actual execution time of the codes are compared. A test system has been implemented on top of the random test system Orange4, which has successfully detected a regression in the optimizer of a development version of GCC-8.0.0 (latest as of May, 2017).

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Software performance*; *Software testing and debugging*;

KEYWORDS

random test, compiler, optimization, performance test

ACM Reference Format:
Kota Kitaura and Nagisa Ishiura. 2018. Random Testing of Compilers' Performance Based on Mixed Static and Dynamic Code Comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '18), November 5, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 7 pages.
https://doi.org/10.1145/3278186.3278192

1 INTRODUCTION

Since extremely high reliability is required for compilers, which are infrastructure tools for developing every kind of software, it is imperative that they should be tested thoroughly. Furthermore, compilers are also expected to generate superior codes in terms of execution speed, memory usage, power consumption, etc., and thus it is also important to test compilers to confirm if intended optimization is properly performed.

Validation of compilers is usually done using test suites, large sets of test programs. Test suites and benchmarks are also used to assess optimization performance of compilers. In either case, there should be bugs or unexpected performance degradation (or performance bugs) that can not be detected by the test suites, as long as they are finite.

Automated random test is one of the effective measures to reinforce validation or performance assessment of compilers by test suites. It tests compilers with randomly generated programs as long as time allows. Many random test generators [7–9, 11, 12, 14, 15] have been developed even for C compilers, and there have been some attempts to detect performance bugs using these random test generators.

Iwatsuji [6] and Barany [1] proposed differential random testing of compiler optimizers where generated programs are compiled with different compilers (or different versions of the same compiler) and generated codes are compared. Another way of performance testing is to (automatically) optimize generated program in the source code level (or in the AST level) and compare the codes generated from original and optimized programs [5]. In either approach, generated codes are *statically* compared. This may lead to cases where a detected code difference does not mean the actual performance difference.

To address this issue, we propose in this paper a random test method to detect compilers' performance bugs based on mixed static and dynamic code comparison. It is also based on the differential random testing. In a static comparison step, it tries to detect a code difference and then minimizes (or reduces) the source program to isolate the code difference. Then in a dynamic comparison step, it measures the run time of the codes to verify if there is the actual performance difference.

An automated test system for C compiler based on the proposed method has been developed in Perl5 on top of Orange4 [12]. It successfully detected regression in the optimizer of GCC-8.0.0 (the latest version at the time of experiment) and reported it to the developer. We also show that the proposed method may be used for performance assessment of C compilers such as GCC and LLVM.

2 RELATED WORK

Figure 1 is an example of performance regression on GCC presented in [6]. From the C program (test.c), GCC-6.0.0 (a development version at the point of the report) generated a longer code than the previous version. In this paper we call such a case a performance bug where a compiler under test fails to perform expected

A-TEST '18, November 5, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6053-1/18/11…$15.00
https://doi.org/10.1145/3278186.3278192

test.c

```
1: unsigned int x = 1;
2: int main (void) {
3:   long long int a = -2LL;
4:   int t = 1 <= (a/x);
5:   if (t != 1) { __builtin_abort(); }
6:   return 0;
7: }
```

gcc5.s (GCC-5.2.1 -O3)

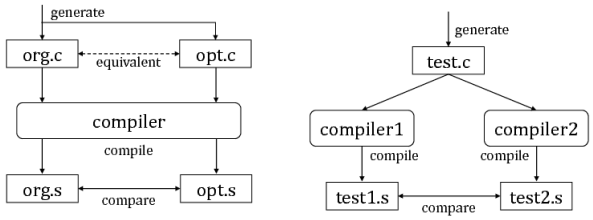
```
main:
 xorl %eax, %eax


 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
```

Reported in https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68431.

Figure 1: Example of performance regression [6]



(a) Method in [5] (b) Method in [6]

Figure 2: Random test of compilers' performance

optimization. Although the performance of compilers may refer to many aspects of generated codes, we focus on speed performance in this paper, as in most of the existing studies. Unlike the validation test where generated codes are clearly decided as correct or not according to the language standard, the boundary for the performance bugs is often ambiguous; it is hard to conclude that 10% speed-down on a particular source code is a bug. The final decision is left to the compiler writers in the end, but we can help them by finding test cases that present substantial speed-down or obvious missing of expected code transformation.

NULLSTONE [3] is a test suite to evaluate optimization performance of C compilers. It consists of about 6,500 tests targeting about 40 optimization passes. However, as long as the number of test programs are finite, its detection capability will be limited.

Randprog [4] attempts to test C compiler's miss-optimization. By randomly generated C programs, it checks if compilers do not erase codes accessing volatile variables by mistake. However, this method does not test whether performance improvement by optimization is performed as intended.

Hashimoto [5] proposed a random test based method to detect missed arithmetic optimization of C compilers. In this method, as shown in Figure 2 (a), a test program (org.c) is randomly generated, to which arithmetic optimization is performed in the AST (abstract syntax tree) level to get a reference program (opt.c). By comparing the assembly codes (org.s and opt.s) generated by a compiler under test, it is checked whether the expected optimization is performed.

Iwatsuji [6] proposed another random test method based on assembly code comparison. As shown in Figure 2 (b), a random program (test.c) is compiled with two different compilers and resulting two assembly codes (test1.s and test2.s) are compared. Regression may be detected if different versions of the same compiler are used.

While the two methods above depend on relatively simple measures for comparing assembly codes, such as the number of instructions, Barany [1] proposed much more sophisticated way of detecting the difference of two assembly codes, laying emphasis on the amounts of spill code considering loops.

Since all the three methods are based on static analysis, there are cases where detected differences do not mean actual performance differences. Monseley [10] proposed a method for comparing the performance of different versions of the same compiler by comparing their execution traces. It seems to take relatively long time to acquire and analyze the traces for large scale programs. Chen [2] directly compares the execution cycles of the codes generated by different compilers from relatively restricted programs. The both methods uses benchmark programs but not randomly generate programs.

3 RANDOM PROGRAM GENERATION AND ERROR PROGRAM MINIMIZATION

As a tool to generate random test programs, Csmith [15] was employed in Barany's method [1]. Csmith is a powerful C program generator that covers broad range of the C syntax. In this method, C-Reduce [13] is also used as a minimization tool for the error programs. C-Reduce is a general purpose error program minimizer that takes a C program and a command line to judge OK/NG of the program and outputs a program as small as possible that fails the check.

Iwatsuji's and Hashimoto's methods [5, 6] used Orange3 [11] as a random program generator. It also generates C programs but different from Csmith in a few ways. Orange3 knows the runtime values of all the variables, expressions, and subexpressions in the program at program generation time, so it can generate long and complex arithmetic expressions without undefined behavior such as zero division and signed overflow. On the other hand, it can generate a sequence of assign statements but not other control statements, so Orange3 is suitable for testing arithmetic optimization of C compilers. Orange3 has a built-in error program minimizer, which can process only the programs generated by Orange3 but is much faster than C-Reduce.

Orange4 [12], which is used in this paper, is a successor to Orange3. It employs a different generation algorithm from Orange3's but can generate complex arithmetic expressions without undefined behavior. It can also generate if-statements and for-loops. Figure 3 is an example of a test program generated by Orange4. In the main part (lines 26–35) are a for-loop, an if-statement, and assign statements. Since the runtime values of all the variables are known at program generation time, the results are compared with the correct values as in lines 37–39. Orange4 is also equipped with built-in minimizer which runs as fast as that of Orange3.

In Csmith as well as Orange3 and Orange4, the size of the random programs is specified in a configuration file or in a command

```
 1: #include <stdio.h>
 2: #define OK()
 3: #define NG(fmt,val) __builtin_abort()
 4:
 5: const volatile signed int x9 = -59;
 6: signed long long x10 = 8198LL;
 7: signed short x11 = 18332;
 8:
 9: int main (void)
10: {
11:  static unsigned long long x0 = 7LLU;
12:  static const volatile signed long x1 = 0L;
13:  signed char x2 = 22;
14:  unsigned long x3 = 0LU;
15:  static unsigned char x4 = 29U;
16:  static const signed char x5 = -1;
17:  static signed int x6 = 123621;
18:  signed int x7 = -7293637;
19:  unsigned long long t0 = 46LLU;
20:  signed long t1 = 297271L;
21:  unsigned long x8 = 102005473280LU;
22:  signed long long t2 = 0LL;
23:  signed char t3 = 3;
24:  signed int i;
25:
26:  for( i = x9*x6; i < x5+x5; i -= x7+x3 ) {
27:    t0 = x3|x0*x2-x4;
28:    t1 = x5*x5-x6+x2/x7;
29:    if( x1<<x1 ) {
30:      t2 = x8>>x4/t0+x10*x10+x11;
31:    }
32:    else {
33:      t3 = x14|x16;
34:    }
35:  }
36:
37:  if (t0 == 120) { OK(); } else { NG("%d", t6); }
38:  if (t1 == 220) { OK(); } else { NG("%d", t6); }
39:  if (t3 == 22) { OK(); } else { NG("%d", t6); }
40:  return 0;
41: }
```

Figure 3: Example of a test program generated by Orange4

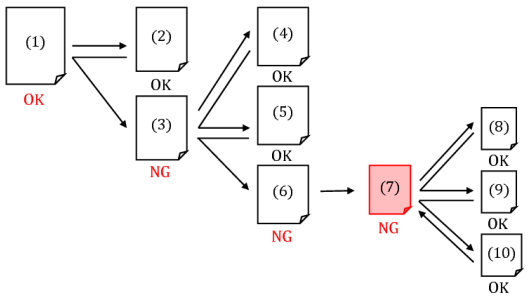


Figure 4: Minimization of error program

line. It is adjusted according to the maturity of the compiler under test, but it usually ranges from several hundred lines to several thousand lines. Test programs should not be too long because compilers suddenly slow down when the program size exceeds a certain threshold. Otherwise, the longer program is the better, because the combinations or relations among variables, operations, and branches increases exponentially with the size of the program.

Minimization of error programs is as important task as program generation, because it is impossible to debug the compiler under test with an error program consisting of thousands of lines. Figure 4 shows a general flow of error program minimization. Starting from a program that has detected some error (1), one of the transformations to reduce the size of the program is applied. The transformations includes replacing a subexpression by a constant, deletion of a statement, etc. If a transformation eliminates the error (2), then it is canceled. A transformation is also canceled if it triggers undefined behavior, which may happen in C-reduce but not in Orange3 nor Orange4. By accepting only error preserving transformations, a program (7) is reached on which any possible transformation erase the error. This is the output of the error program minimization. It is not minimum in a strict sense, but called so because it can not be reduced by any single transformation.

4 MIXED STATIC AND DYNAMIC CODE COMPARISON

4.1 Outline

Like the methods of Iwatsuji's and Barany's, we compare a pair of codes generated by different compilers or different versions of compilers from each randomly generated program.

One shortcoming of static code comparison approach is that we do not tell whether there is really performance difference even if significant differences are detected. However, simple comparison of execution time does not work. We must use long test programs in performance test as well as in validation test to increase detection capability. For such long test programs, even if performance differences existed, almost no difference would be observed as a hole. There are even cases that performance losses and gains may cancel each other.

To address this issue, we propose a comparison method consisting of two stages. In the first stage, we statically compare assembly codes. If we find a significant difference, we minimize the test program to isolate the difference. Then, in the second stage, we run the two codes generated from the minimized program to measure their actual execution time.

By this method, C programs that reveal real performance differences will be obtained. Since minimization in the first stage eliminates parts of the test programs that results in the same instruction sequences for the both compilers, the percentage of the different instruction sequences will be much increased so that it becomes easier to detect the difference in execution time.

4.2 Static Comparison of Assembly Codes

In the static comparison of the first stage, we only compare pairs of sections in which the codes disagree, ignoring the sections where codes are the same. The comparison consists of two steps.

(1) Identifying sections where the assembly codes differ.
(2) Comparing the sums of the weights of mismatched instructions in each section pair to decide if it potentially causes performance difference.

Figure 5 illustrates how the first step is performed. Two assembly codes test1.s and test2.s are being compared. By using a text level differencer on the assembly codes, the pair of code sections where more than k instructions (k is about 7) are exactly the same are eliminated. This leaves pairs of mismatched code sections (S_1, S_1'), (S_2, S_2'), and (S_3, S_3'). The pairs are further investigated in the second step. If either one of them has a meaningful difference, then test1.s and test2.s are judged to be different.

In the second step, mismatched instructions in each pair of mismatched sections are identified and the difference is evaluated based

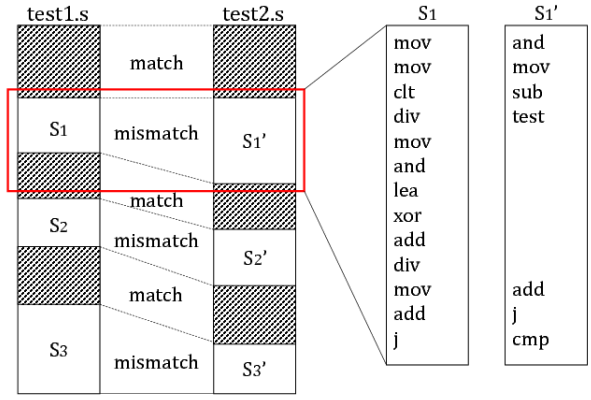


Figure 5: Extraction of mismatched section pairs in assembly codes

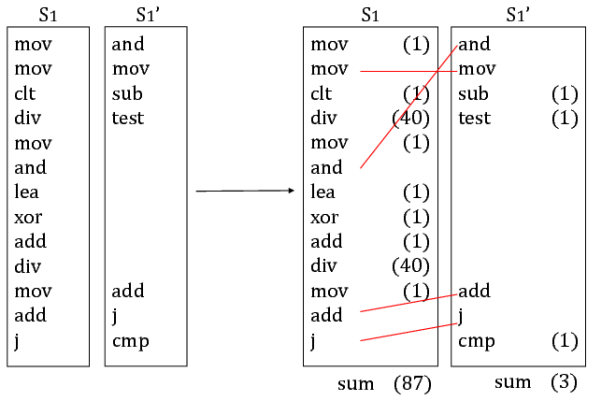


Figure 6: Mismatch instructions and their weighted sums

on the weighted sums of mismatched instructions. For example, in Figure 6, a pair of sections S_1 and S_1' is being compared. At first, instruction pairs of the same opcodes are identified, as indicated by the red lines. Then the other instructions are mismatched instructions. For each of S_1 and S_1', the weighted sum of the mismatched instructions are computed. Since it is hard to define theoretically valid performance costs to the instructions, the weights are empirically determined; large weights are give to multiplication, division, and jump instructions. If the ratio of the sums exceeds a predetermined threshold, which is also empirically decided, then S_1 and S_1' is judged to have a significant difference.

(1)After minimization	(2)For measurement of execution time
`#include <stdio.h>`	`1: #include <stdio.h>`
	`2: int main0()`
	`__attribute__((noinline));`
	`3: int main (void)`
	`4: {`
	`5: long i;`
	`6: for(i=0;i<10000000L;i++)`
	`7: {`
	`8: int rc = main0();`
	`9: }`
	`10: return 0;`
	`11: }`
`int main (void){`	`12: int main0(){`
`...`	`13: ...`
`return 0;`	`14: return 0;`
`}`	`15: }`

Figure 7: Conversion for execution time measurement

4.3 Dynamic Codes Comparison Based on Execution Time Measurement

The difference in the weighted sums in the static comparison does not necessarily mean the difference in execution time on actual machines. Longer codes are often faster because of various optimization such as loop unfolding, and function inlining.

In the method of this paper, we make final decision by measuring the execution time on actual machines.

There are several ways to measure execution time on a real machine, out of which we adopt a simple one that uses the UNIX *time* command. Since the execution time of the minimized program becomes very small, the code is looped within the test program. An example of code conversion on the test program is shown in Figure 7. (1) is an original (minimized) test program from the first stage, and (2) is an augmented program for measurement. Function `main` in (1) is renamed to `main0` in (2) and called from new `main` in (2). In order to prevent inline expansion of `main0` (and subsequent optimization), a directive to forcibly suppress inline expansion is inserted in the second line.

5 EXPERIMENTAL RESULTS

A random test system based on the method proposed in this paper has been implemented on top of Orange4. It is written in Perl5 and runs on Ubuntu 16.04 LTS environment. Currently, the static assembly comparator assumes the GAS format and only supports the x86 architecture.

Performance tests of the optimizers of GCC-8.0.0 against GCC-7.2.0, and LLVM-4.0 against LLVM-3.8 were conducted. For all the compilers, the target was x86_64 and the option tested was -O3. The execution time was measured on Core i5-6200U 2.30GHz×4 with 7.6 GB RAM. The average number of operations per a test program was set to 400. The version of Orange4 described in [12] was used in this experiment, which generated programs with only scalar variables, assign statements with arithmetic expressions, and *if* and *for* statements. The thresholds for assembly code comparison and execution time comparison was set to 60% and 50%, respectively.

The results are summarized in Table 1. In both tables (a) and (b), "#test" lists the number of test programs, "#diff ⟨assembly⟩" the number of the programs which lead to significant difference in assembly codes, and "#diff ⟨time⟩" the number of the programs which detected that the compiler of the newer version generated slow codes. For reference, we ran the same test also on Intel Xeon

Table 1: Results of regression tests

(a) GCC-8.0.0 against GCC-7.2.0 (x86_64, -O3)

	time [h]	#test	#diff ⟨assembly⟩	#diff ⟨time⟩
Proposed	12	15,348	425	308
Iwatsuji [6]	12	16,049	18	(11)

(b) LLVM-4.0.0 against LLVM-3.8.0 (x86_64, -O3)

	time [h]	#test	#diff ⟨assembly⟩	#diff ⟨time⟩
Proposed	12	11,217	23	11
Iwatsuji [6]	12	14,503	0	(0)

Core i5-6200U 2.30GHz×4, RAM 7.6GB, Ubuntu 16.04LTS
Target: x86_64, Option: -O3
Average ops per test: 400
Thres. for assembly comparison: 60%, Thres. for runtime comparison: 50%

E3−1276 v3 2.30GHz×4 with 15.6GiB RAM, but the results were exactly the same.

In table (a), we can see that the method proposed in this paper detected 425 differences by assembly code comparison, out of which 308 were confirmed to be regression errors by execution. Namely, assembly codes generated by GCC-8.0.0 were longer than those generated by GCC-7.2.0 for the 425 test programs out of 15,348, and actually slower for the 308 programs. Namely, there were 117 false positives, which accounts for 27.5% of 425 errors. For comparison, the previous (Iwatsuji's) method [6] detected only 18 assembly code differences[1]. All the 18 cases were included in the 425 cases detected by the proposed method. This means that the proposed method detected regression errors missed by the previous method. For reference, "(11)" is the result of dynamic comparison on the 18 cases. There were also false positives (38.9%) in Iwatsuji's method. In table (b), we can see that the proposed method succeeded in detecting 11 regression errors in LLVM-4.0.0, which the previous method failed to find.

Figure 8 is an example of the test programs that detected performance degradation in the latest version of GCC-8.0.0 as of May 2017 (against GCC-7.0.1). This regression was reported to the development team through GCC's Bugzilla [2] and was fixed.

Table 2 shows the result of performance comparison among six versions of LLVM by the proposed method. Every pair of the versions was tested with 1,000 random test programs. Each column lists the number of test programs for which the version generated slower codes than the other versions. For example, "6" in the 3.8's column and the 3.6's row tells that LLVM-3.8 generated slower codes than LLVM-3.6 for 6 test programs, while "77" in the lower left cell tells that LLVM-3.8 generated faster code than LLVM-3.6 for 77 test programs. Looking at the bottom row for LLVM-7.0, the numbers (191, 187, 41, ···) are decreasing as the version number increases, which indicates that performance has been improved on each version-up. Note that this also means that the proposed method of performance comparison has a certain accuracy. It is

[1] Iwatsuji's method originally used Orange3 as a random program generator, but we used Orange4 in this experiment. Thus, the proposed and Iwatsuji's methods are compared with exactly the same sequence of random programs.
[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81165

```
 1: void func() __attribute__((noinline));
 2:
 3: short x0 = 15;
 4:
 5: int main (void)
 6: {
 7:  long i;
 8:  for( i = 0; i < 100000000L; i++){
 9:    func();
10:  }
11:  return 0;
12: }
13: void func(){
14:  volatile int x1 = 1U;
15:  volatile char x2 = 0;
16:  char t0 = 0;
17:  unsigned long t1 = 2LU;
18:  int i = 0;
19:  if(1>>x2){
20:    t0 = -1;
21:    t1 = (1&(short)(x1^0U))-1;
22:  }
23:  while(i>(int)((1U>>t1)+
        (char)(128%(10*(25LU&(29%x0))))){
24:    i += (int)(12L/(1 != (int)t1));
25:  }
26:  if(t0 != -1)__builtin_abort();
27:  if(t0 != 0L)__builtin_abort();
28: }
```

(a) C program

```
gcc-7.0.1.s
main:
 subq $24, %rsp
 movl $1, %eax
 movl $1, 12(%rsp)
 movb $0, 11(%rsp)

 movzbl 11(%rsp), %ecx
 sarl %cl, %eax
 testl %eax, %eax
 jne .L12

.L2
 call abort
.L12
 movl 12(%rsp), %eax
 andl $1, %eax
 subl $1, %eax
 testl %eax, %eax
 jne .L2

 addq $24, %rsp
 ret
```

```
gcc-8.0.0.s
main:
 subq $24, %rsp

 movl $1, %edi
 movl $1, 12(%rsp)
 movb $0, 11(%rsp)
 movl %edi, %eax
 movzbl 11(%rsp), %ecx
 sarl %cl, %eax
 testl %eax, %eax
 je .L7
 movl 12(%rsp), %ecx
 movl $-1, %r9d
 andl $1, %ecx
 subl $1, %ecx
 movslq %ecx, %r8
 shrl %cl, %edi
.L2
 movswl x0(%rip), %esi
 movl $29, %eax
 cltd
 divl %esi
 movl $128, %eax
 andl $25, %edx
 leaq (%rdx,%rdx,4), %rsi
 xorl %edx, %edx
 addq %rsi, %rsi
 divq %rsi
 movsbl %dl, %edx
 addl %edi, %edx
 jns .L3
 cmpl $1, %ecx
 je .L10
.L3
 cmpb $-1, %r9b
 jne .L6
 testq %r8, %r8
 jne .L6
 addq $24, %rsp
 ret
.L10
 ud2
.L7
 movl $2, %ecx
 xorl %edi, %edi
 movl $2, %r8d
 xorl %r9d, %r9d
 jmp .L2
.L6
 call abort
```

(b) Assembly codes

Figure 8: Program detected regression in GCC-8.0.0

Table 2: Performance comparison of LLVM versions

target→ reference ↓	3.6	3.8	4.0	5.0	6.0	7.0
3.6	—	6	23	19	20	4
3.8	77	—	22	20	17	2
4.0	275	111	—	33	37	16
5.0	278	203	53	—	13	9
6.0	328	269	132	82	—	8
7.0	191	187	41	26	18	—

Target: x86_64, Option: -O3
Number of test programs: 1,000
CPU: Intel Core i7−6850K 3.60GHz with 32GB RAM

Table 3: Performance comparison between GCC and LLVM

static+dynamic (static only)

target→ reference ↓	GCC-9.0.0	LLVM-7.0.0
GCC-9.0.0	—	324 (404)
LLVM-7.0.0	232 (785)	—

Target: x86_64, Option: -O3
Number of test programs: 10,000
CPU: Intel Core i7−6850K 3.60GHz with 32GB RAM

also noted that the latest version has still a few regression from the older (or even the very old) versions.

Table 3 is a summary of the similar comparison between the latest versions of GCC and LLVM using 10,000 test programs. The figures in the parentheses (404 and 785) are the numbers of errors detected by only static comparison. After dynamic comparison, we see that GCC generated faster code than LLVM for 324 programs while LLVM did better on 232 programs. The two compilers seem to implement optimizers of different strategies but achieve comparable performance, in terms of the class of the test programs generated by Orange4. Since different compilers generates different style of code sequences aside from optimization, assembly codes tend to differ more largely than in the experiments in Table 1. This should leads to larger amount of false positives, but the actual rates of false positives were 70.4% and 19.8%. We have not yet analyzed the reason for this asymmetry.

The threshold for the runtime comparison in this experiment was determined empirically. Table 4 shows the distribution of execution time differences measured in the regression tests. The first row in (a) indicates that the execution time of the codes generated by GCC-7.2.0 was less than or equal 25% of those by GCC-8.0.0 for 59 test programs out of 425, for which GCC-7.2.0 produced shorter assembly codes. There were cases where shorter assembly codes ran slower, but separation is not very clear; some programs fell in the section between 51%∼100%.

6 CONCLUSION

We have proposed an automated test method to detect compilers' performance bugs based on mixed static and dynamic code comparison. The dynamic execution of the minimized test program successfully excluded the case where the static comparison fails to detect the actual performance differences. A test system based on our method successfully detected a performance bug in the latest version of GCC. We also expect that this method can be used for performance assessment of C compilers.

There is much room for improving the accuracy of static assembly comparison, taking memory accesses as well as instruction strength into account. The static comparison must be enhanced to deal with control flow graphs, especially if we use Csmith or extended Orange4 with more control statements. We are now also working on making the assembly comparator retargetable so that it can support other architecture than x86 and can accept other formats than GAS.

Table 4: Distribution of execution time differences

(a) GCC-8.0.0 against GCC-7.2.0 (x86_64, -O3)

execution time by GCC-7.2.0	#diff
∼ 25%	258
26%∼ 50%	50
51%∼ 75%	17
76%∼100%	78
101%∼	20

(b) LLVM-4.0.0 against LLVM-3.8.0 (x86_64, -O3)

execution time by LLVM-3.8.0	#diff
∼ 25%	5
26%∼ 50%	6
51%∼ 75%	5
76%∼100%	2
101%∼	5

ACKNOWLEDGMENTS

First of all, we would like to thank the reviewer for the valuable feedback. Authors would like to thank Mr. S. Takakura and all the members of the Ishiura Laboratory for their discussion on this research. This work was partly supported by JSPS Kakenhi Grant #25330073.

REFERENCES

[1] G. Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of 27th International Conference on Compiler Construction (CC'18)*. 82–91. https://doi.org/10.1145/3178372.3179521
[2] Y. Chen and J. Regehr. 2010. Comparing Compiler Optimization. online. Retrieved July 11, 2018 from https://blog.regehr.org/archives/320
[3] Nullstone Corporation. 2012. NULLSTONE for C. online. Retrieved June 21, 2018 from http://www.nullstone.com/
[4] E. Eide and J. Regehr. 2008. Volatiles Are Miscompiled, and What to Do about It. In *Proceedings of the 8th ACM International Conference on Embedded Software*. 255–264. https://doi.org/10.1145/1450058.1450093
[5] A. Hashimoto and N. Ishiura. 2016. Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs. In *IPSJ Trans. System LSI Design Methodology*, Vol. 9. 21–29. https://doi.org/10.2197/ipsjtsldm.9.21
[6] M. Iwatsuji, A. Hashimoto, and N. Ishiura. 2016. Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing (short paper). In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*. 2–3.
[7] V. Le, C. Sun, and Z. Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 386–399. https://doi.org/10.1145/2814270.2814319

[8] V. Le, C. Sun, and Z. Su. 2015. Randomized Stress-Testing of Link-Time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 327–337. https://doi.org/10.1145/2771783.2771785

[9] C. Lindig. 2005. Random testing of C calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05)*. 3–12. https://doi.org/10.1145/1085130.1085132

[10] T. Moseley, D. Grunwald, and R. Peri. 2009. OptiScope: Performance Accountability for Optimizing Compilers. In *Proceedings of 2009 International Symposium on Code Generation and Optimization (CGO 2009)*. 254–264. https://doi.org/10.1109/CGO.2009.26

[11] E. Nagai, A. Hashimoto, and N. Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. In *IPSJ Transactions on System LSI Design Methodology*, Vol. 7. 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

[12] K. Nakamura and N. Ishiura. 2016. Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation. In *Proceedings of 2016 IEEE Asia and Pacific Conference on Circuits and Systems (APCCAS 2016)*. 676–679. https://doi.org/10.1109/APCCAS.2016.7804063

[13] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 335–346. https://doi.org/10.1145/2254064.2254104

[14] C. Sun, V. Le, and Z. Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. 849–863. https://doi.org/10.1145/3022671.2984038

[15] X. Yang, Y. Chen, E. Eide, and J. Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 283–294. https://doi.org/10.1145/1993498.1993532