Performance Testing of VRP Optimization of C Compilers by Random Program Generation

Daiki Murakami, Toya Hamada, Toshiaki Watanabe, Yushin Kawasaki
Graduate School of Science and Technology
Kwansei Gakuin University
Sanda, Hyogo, Japan
toshiaki-watanabe@kwansei.ac.jp, yushin-kawasaki@kwansei.ac.jp

Nagisa Ishiura
School of Engineering
Kwansei Gakuin University
Sanda, Hyogo, Japan
ishiura@kwansei.ac.jp

***Abstract*—In this paper, we propose a random testing method to examine whether VRP (Value Range Propagation) optimization in C compilers is performed as intended. Constant folding, a fundamental compiler optimization, replaces a subexpression with a constant when its value is known at compile time. In contrast, VRP can replace a subexpression with a constant even when it contains variables whose values may change at run time, such as volatile variables, provided that the value of the subexpression can still be uniquely determined. Our method employs random testing to identify cases in which a C compiler fails to apply VRP optimization. We automatically generate a program containing arithmetic expressions to which VRP should be applicable, as well as another program obtained by applying VRP at the C source code level. Missed optimizations are then detected by comparing the assembly code generated from these programs. The arithmetic expressions are generated by propagating value ranges in the direction opposite to that of VRP analysis. By appropriately selecting operators and value ranges, we ensure that the generated expressions are always optimizable by VRP. We further improve testing efficiency by generating expressions in which a single expression references multiple volatile variables. We implemented the proposed method as an extension of the Orange4 random testing system. Experimental results revealed missed VRP optimization opportunities in multiple compiler versions, including GCC 8.5.0 and GCC 11.5.0.**

***Index Terms*—compiler validation, random test, tree-optimization, VRP optimization, detecting missed compiler opportunities**

I. Introduction

Compilers are fundamental tools in software development and must operate with a high degree of reliability. Therefore, extensive testing is conducted to ensure that they generate correct code [1]. In addition, modern compilers are expected to produce code with improved execution speed, reduced memory usage, and lower power consumption. Accordingly, verifying that optimizations are applied as intended is also an important aspect of compiler testing.

For testing compiler optimization performance, existing approaches include dedicated test suites [2] and trace comparison of benchmark executions [3]. However, because the number of test cases in test suites and benchmarks is limited, methods that use randomly generated programs for optimization testing have been proposed. In [4]–[6], tree optimizations such as constant folding and strength reduction are tested by extending Orange4 [10], a random testing system originally developed for checking correctness of code generation. Although these optimizations are prone to defects even in code-correctness testing [7], many defects have also been detected in optimization-performance testing [9].

VRP (Value Range Propagation) optimization is a kind of tree optimization that propagates value ranges of subexpressions in arithmetic expressions in a bottom-up manner, enabling constant folding even for expressions that include variables whose values are not fixed at compile time. Existing program-generation methods can occasionally detect missed VRP optimizations, but because those programs are not generated specifically for VRP testing, the testing is not sufficiently thorough.

In this paper, we propose a random test-program generation method specifically aimed at testing VRP optimization. By propagating value ranges in the direction opposite to VRP analysis while selecting operators and operand values of subexpressions, our method generates arithmetic expressions that become constants at compile time even when they include variables whose values are not fixed at compile time.

Experimental evaluation with a test system implemented as an extension of Orange4 [10], a random testing system for C compilers, showed that our method can detect missed VRP optimization opportunities in GCC-8.5.0 that are difficult to detect with conventional methods.

II. Testing of Compiler Optimization Performance

A. Constant Folding and VRP Optimization

Constant folding is one of the compiler optimizations for arithmetic expressions, known as tree optimization, where a subexpression is replaced with a constant when its value can be determined at compile time. In Fig. 1, the value of variable `x3` on line 5 is known to be 42 at compile time; therefore, the right-hand side of the inequality `(x3/6)` becomes `(42/6)`, which evaluates to 7. Although simple, this optimization can be highly effective in combination with other optimizations, yet it is also prone to miscompilation due to properties specific to binary-integer arithmetic.

Variables with the C volatile qualifier (variables whose values may be rewritten by external processes or devices; hereafter called *volatile variables*) and global variables may change

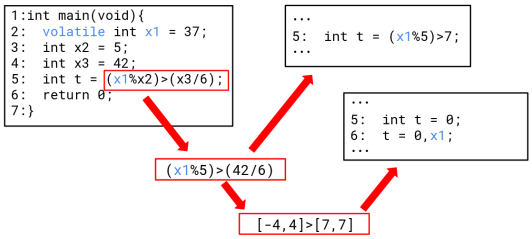

Fig. 1. Constant folding and VRP optimization

during execution. Therefore, simple constant folding cannot be applied to subexpressions that include such variables. The left-hand side of the inequality on line 5, `(x1%x2)`, can only be simplified to `(x1%5)` because `x1` is volatile, and the final optimized form of the inequality on line 5 is `(x1%5)>7`.

However, the value of a subexpression can sometimes be determined at compile time regardless of the value taken by a volatile variable. For example, in `(x1%x2)` on line 5, `x2` is 5, so regardless of `x1`, the subexpression is an integer between -4 and 4. Since 42/6 equals 7, the value of this inequality is always 0.

VRP optimization performs stronger expression simplification than constant folding in this way. In Fig. 1, $[a, b]$ indicates that the subexpression can take a value in the interval from a to b. By propagating these intervals bottom-up, constant substitution can be attempted even for subexpressions containing volatile variables.

In the example of Fig. 1, line 5 can be simplified to `t=0` by VRP optimization. `x1` is added with the comma operator because accesses to volatile variables must be preserved regardless of whether their values are used.

B. Random Testing of Compiler Optimization Performance

Optimization is one of the most complex parts of compiler processing and is therefore prone to defects. Compiler defects include those that cause generated machine code to produce incorrect results, compiler crashes, and infinite loops; in addition, there are defects where optimizations are not applied as intended. In the latter case, execution results remain correct, but runtime becomes slower than expected. In real-time systems, in particular, insufficient optimization after a compiler version upgrade can lead to failures to satisfy real-time constraints.

As random testing methods for compiler optimization performance, differential testing [4] [5] and the equivalence method [6] have been proposed. Differential testing compares assembly code generated by compiling a test program with different compilers (or different versions/options of the same compiler). The equivalence method tests whether optimization is applied as intended by comparing assembly code generated from a test program and a source-level optimized version of the same program.

Methods for comparing compilation results include extracting high-cycle-cost instructions from the differences between two codes [9], measuring differences in spill-code volume focusing on loops [4], and combining these with actual execution-time comparisons [5].

Reference [6] reports that random testing based on the equivalence method detected many missed optimization opportunities in GCC and LLVM, including missed VRP optimizations. However, because the test programs were not generated specifically to target VRP optimization, such detections were limited to accidental cases.

III. Program Generation for Testing VRP Optimization

A. Testing Strategy

In this paper, we propose a method for testing whether a compiler performs VRP optimization as intended, using the same equivalence method as in [5] [6].

Figure 2 shows the structure of test programs generated by the proposed method. We expect the optimization that transforms program (a), org.c, into program (b), opt.c, and perform testing by comparing the machine code obtained by compiling the two programs. org.c consists of variable declarations and initializations, expression evaluation and assignment, and verification of computation results. Expressions in org.c are generated so that they are reduced to constants at compile time when VRP optimization is correctly applied; in opt.c, the corresponding variables are assigned those constants.

As with Orange4 [10], the proposed method generates expressions top-down so that they evaluate to a target value. At this stage, value ranges of subexpressions are propagated in the direction opposite to VRP analysis so that VRP optimization can be applied. By appropriately selecting operators and values to control intervals, the method always generates expressions to which VRP optimization is applicable [8].

B. Generation of VRP-Optimizable Expressions

In this method, when generating expressions, we introduce two types of nodes in the expression tree: a *v-node* and an *n-node*. A v-node is defined as a node whose rooted subexpression contains volatile variables and to which VRP optimization can be applied. An n-node denotes a node whose rooted subexpression does not contain any volatile variables.

Both v-nodes and n-nodes have one value, which represents the value of the subexpression under the initial variable values. A v-node additionally has a value interval. This interval indicates that, no matter what values volatile variables in the subexpression take, the subexpression value remains within that interval.

Expression generation is performed recursively by setting the root node as a v-node with value c (a constant) and interval $[c, c]$, then selecting operators and values so that the interval expands. If we can eventually generate a v-node with interval [MIN, MAX] (where MIN and MAX are the minimum and maximum values of that node type), then even if that node is replaced with a volatile variable, the expression value can still be reduced to c by VRP optimization.

An example of expression generation is shown in Fig. 3. First, the root node is assigned value 0 and interval [0,0].

(a) org.c

```
01:#define OK()
02:#define NG(test,fmt,val)  __builtin_abort()
03:
04:int main (void)
05:{
06:  unsigned short x9 = 0U;
07:  signed char x11 = 0;
08:  unsigned char x13 = 0U;
09:  const unsigned int x16 = 0U;
10:  signed char x19 = 1;
11:  long x20 = 1L;
12:  long x21 = 1L;
13:  const long x22 = -1L;
14:  static long x23 = 1L;
15:  const volatile long x24 = 4464078639334039285L;
16:  static const volatile long x25 = 205L;
17:  static long x26 = -2L;
18:  signed char t0 = 9;
19:  long x27 = 37780795649901867 38L;
20:  static long x28 = 2L;
21:  static const long x29 = 1889039782495093369L;
22:  long x30 = 1L;
23:  long x31 = 2L;
24:  long x32 = 1L;
25:  long x33 = -1L;
26:  long long x34 = -1889039782495093369LL;
27:  const unsigned short x35 = 3U;
28:  int t1 = -51681122;

29:  t0 = ((signed char)(((long)((long long)(((long)
          ((unsigned short)(((unsigned char)x9)/x19)))*
          ((long)((short)(((long)(x24>=((long)x13)))-
          ((long)(((unsigned short)x20)^((short)
          x11)))))))))<((long)((unsigned short)(((long)
          ((long)((((int)x11)+x22)+((long)(x25\%x26)))))>
          ((long)((unsigned int)(((int)x11)>>
          ((unsigned char)x16)))))))));
30:  t1 = (((long)(((long)(((unsigned char)x28)/
          ((long long)x26)))/(((long)(x25||x32))-
          ((long)(x35^((short)x32))))))>=(((long)
          (((long)(x25||x20))-((long)(((short)x26)+
          ((short)x19)))))*((long)(x34/x22))));
31:
32:  if (t0 == 0) { OK(); } else { NG("t0", "\%d", t0); }
33:  if (t1 == 0) { OK(); } else { NG("t1", "\%d", t1); }
34:
35:  return 0;
36:}
```

(b) opt.c

```
            ...

29:  t0 = 0L, x24, x25;
30:  t1 = 0L, x25, x25;

            ...
```

Fig. 2. Pair of test programs generated by the proposed method

Next, appropriate operators and child values are randomly determined (here, $<$, 0, and -7). Then one child is made a v-node and the other an n-node (chosen randomly). The interval is propagated to the v-node child. To make the parent node's possible-value range [0,0], the interval can be set to $[-7, \mathrm{MAX}]$. Repeating the same expansion, a v-node with interval [MIN, MAX] is eventually obtained. Since the root value remains 0 regardless of what value this node takes, replacing this node with a volatile variable means we have generated an expression that can be reduced to a constant by VRP optimization.

In expression generation, if operators are chosen completely at random, the expression depth (tree depth) often exceeds the translation limit specified by the C standard before obtaining a v-node with interval [MIN, MAX]. Therefore, when the expression depth reaches a predefined threshold, we restrict the operator and value choices at v-nodes so that a v-node with interval [MIN, MAX] can always be generated within the remaining two levels. For example, for a v-node with value v and interval $[a, b]$, choose addition as the operator and set the right-child interval to include 0. Then choose logical AND as the right child's operator, and set the value of the right child of that n-node to 0; this makes it possible to obtain interval [MIN, MAX].

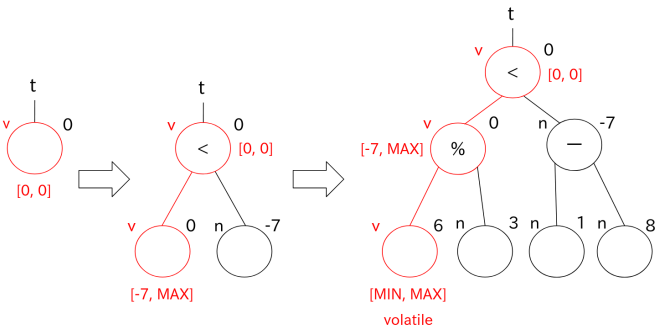

Fig. 3. Example of expression generation in the proposed method

C. Generation of Expressions with Multiple VRP-Optimizable Subexpressions

In the method of the previous subsection [8], only one child was set as a v-node, so the number of volatile variables in an expression was limited to one. In this subsection, we extend the method so that multiple volatile variables targeted by VRP optimization are included in an expression.

This is achieved by first determining one child as a v-node with its value and interval, and then determining the other child as a v-node with its own value and interval.

Figure 4 shows an example. First, following the procedure of the previous subsection, the left child is set as a v-node and its value and interval are determined (a). Next, as shown in (b), the right child is also set as a v-node. At this point, if the right-child interval is chosen as the widest interval that satisfies the subexpression for any value in the left-child interval and that includes its own value -7, the resulting interval is [MIN, -7]. This propagation branches the VRP optimization path in the syntax tree, and finally enables generation of a tree containing multiple leaves with interval [MIN, MAX].

IV. Experimental Results

We implemented a random testing system for C compiler optimization performance based on the proposed method by extending Orange4 [10]. The implementation language is Perl5, and the system runs on UNIX environments such as Ubuntu Linux. Assembly-code comparison was performed using the static method in [5].

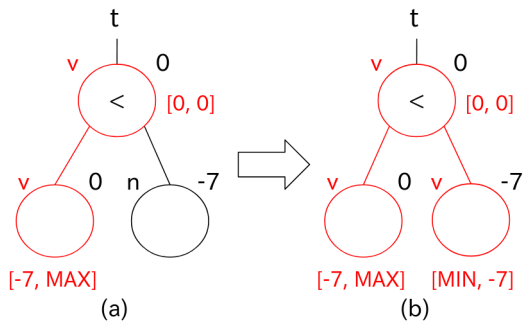

Fig. 4. Example of bidirectional propagation from a v-node

Tests were conducted on multiple GCC versions targeting x86_64. The optimization option was set to -O3, and the number of operators in each program was set to 1,000. The test environment used an Intel Core i5-1235U processor with Ubuntu 24.04.3 LTS.

Table I shows the experimental results. In the table, compiler denotes the target compiler, #test denotes the number of test programs, and time [h] denotes execution time in hours. #diff denotes the number of programs with meaningful assembly-code differences, that is, the number of test programs in which insufficient compiler optimization was detected.

Figure 5 shows a minimized program for which a difference was detected in GCC-11.5.0. Comparison of the assembly code (org.s and opt.s) indicates that VRP optimization is not properly applied to org.c. Note that the expression assigned to the variable `t` in org.c involves two volatile variables. If either `x1` or `x2` is changed to a non-volatile variable, the VRP optimization functions correctly for this test program.

TABLE I
EXPERIMENTAL RESULTS

compiler	#test	time [h]	#diff
GCC-8.5.0	1000	0.66	39
GCC-11.5.0	1000	0.64	24
GCC-13.3.0	10000	7.04	0

V. CONCLUSION

In this paper, we proposed a random program-generation method for performance testing of VRP optimization in C compilers. Experimental results showed that, although they are not the latest releases, missed VRP optimizations can be detected in multiple GCC versions. In the current version of the test system, some constraints are imposed on expression values and available operators due to implementation considerations; relaxing these constraints remains future work.

ACKNOWLEDGMENT

The authors would like to thank the members of the Ishiura Laboratory, Graduate School of Science and Technology, Kwansei Gakuin University, for their cooperation and valuable discussions related to this research.

org.c (GCC-11.5.0)

```
 1: int main()
 2: {
 3:   volatile int x1=2;
 4:   volatile int x2=0;
 5:   int t = 0;
 6:   int a = (x1%2)-1;
 7:   int b = -3;
 8:   int c = -1+(x2%2);
 9:   t = a<=(b*(c<=0));
10:   return 0;
11: }
```

opt.c (GCC-11.5.0)

```
 1: int main()
 2: {
 3:   volatile int x1=2;
 4:   volatile int x2=0;
 5:   int t = 0;
 6:
 7:
 8:
 9:   t = 0, x1, x2;
10:   return 0;
11: }
```

org.s (GCC-11.5.0)

```
 1: movl $2,8(%rsp)
 2: movl $0,12(%rsp)
 3: movl 8(%rsp),%eax
 4: movl 12(%rsp),%edx
 5: movl %edx,%ecx
 6: shrl $31,%ecx
 7: addl %ecx,%edx
 8: andl $1,%edx
 9: subl %ecx,%edx
10: movl %eax,%ecx
11: cmpl $1,%edx
12: setg %dl
13: shrl $31,%ecx
14: addl %ecx,%eax
15: movzbl %dl,%edx
16: andl $1,%eax
17: leal -3(%rdx,%rdx,2),%edx
18: subl %ecx,%eax
19: subl $1,%eax
20: cmpl %eax,%edx
21: jge .L3
22: xorl %eax,%eax
```

opt.s (GCC-11.5.0)

```
 1: movl $2,-8(%rsp)
 2: movl $0,-4(%rsp)
 3: movl -8(%rsp),%eax
 4: movl -4(%rsp),%eax
 5:
 6:
 7:
 8:
 9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22: xorl %eax,%eax
```

Fig. 5. Missed optimization opportunity detected in GCC-11.5.0

REFERENCES

[1] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang: "A Survey of Compiler Testing," ACM Comput. Surv. 53, 1, Article 4 (Feb. 2020).
[2] Nullstone Corporation: NULLSTONE for C (online), http://www.nullstone.com/ (accessed 2026-02-01).
[3] T. Moseley, D. Grunwald, and R. Peri: "OptiScope: Performance Accountability for Optimizing Compilers," in Proc. IEEE/ACM International Symposium on Code Generationand Optimization (CGO 2009), pp. 254–264 (Mar. 2009).
[4] G. Barany: "Finding Missed Compiler Optimizations by Differential Testing," in Proc. International Conference on Compiler Construction (CC 2018), pp. 82–92 (Feb. 2018).
[5] K. Kitaura and N. Ishiura: "Random Testing of Compilers' Performance Based on Mixed Static and Dynamic Code Comparison," in Proc. ACM International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST 2018), pp. 38–44 (Nov. 2018).
[6] A. Hashimoto and N. Ishiura: "Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs," IPSJ Trans. System LSI Design Methodology, vol. 9, pp. 21–29 (Feb. 2016).
[7] X. Yang, Y. Chen, E. Eide and J. Regehr: "Finding and Understanding Bugs in C Compilers," in Proc. ACM Conference on Programming Language Design and Implementation (PLDI '11), pp. 283–294 (Oct. 2011).
[8] D. Murakami and N. Ishiura: "Performance Testing of VRP Optimization of C Compilers by Random Program Generation" (in Japanese), in IEICE Technical Report, VLD2020-66 (Jan. 2021).
[9] M. Iwatsuji, A. Hashimoto, and N. Ishiura: "DetectingMissed Arithmetic Optimization in C Compilers by Differential Random Testing"(short paper), in Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016), pp. 2–3 (Oct. 2016).
[10] K. Nakamura and N. Ishiura: "Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation," in Proc. Asia and Pacific Conference on Circuits and Systems (APCCAS 2016), pp. 676–679 (Oct. 2016).