Random Testing of Back-end of Compiler Infrastructure LLVM

Kenji Tanaka[1] Nagisa Ishiura[1] Masanari Nishimura[2] Akiya Fukui[2]

[1] Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] Renesas System Design, Co., Ltd., Tokyo, Japan

Abstract—This paper presents a method of directly testing back-ends of the LLVM compiler infrastructure by randomly generated LLVM IR (intermediate representation). Using LLVM, a compiler for a new target can be developed only by implementing a machine dependent back-end, then the test of the back-end becomes a focusing issue. However, there are some LLVM instructions which can never or rarely be tested by C programs. The proposed method generates random LLVM IR assembly codes intended to include such instructions. In an experiment on LLVM 3.5 for x86_64 has detected an error case which is hard to test by C programs.

I. Introduction

The compiler infrastructure LLVM [1] is widely utilized for developing compilers for newly developed embedded processors or for developing high-level synthesizers [2, 3]. Modularity is one of the main points of LLVM. In order to develop a compiler for a new target (including RTL hardware), one only has to implement a back-end dedicated to the target.

This also means that during the test phase one can focus on the back-end. However, in the later stage, where a huge set of test cases are needed to detect unexpected or latent bugs, usually test suites in the form of C programs [4] or randomly generated C programs [5] are used.

There are cases where C programs can never or rarely test some functions in the back-end, due to various transformations performed before the back-end. Integer arithmetic on the short integer is such an example, because they are promoted to that of the machine word size according to the semantics of C. However, short integers may be used in compilation of other languages without integer promotion, or in future optimization.

To address this issue, this paper proposes a random test method that directly generate test programs in the form of the LLVM IR. It can generate test cases containing all the arithmetic operations uniformly without affected by the front-ends. A test generator based on the method has successfully found an error program for LLVM 3.5 for x86_64 which is hard to test by C programs.

II. Compiler infrastructure LLVM

LLVM [1] consists of three parts; front-ends convert program texts into intermediate representation named LLVM IR, the middle-end performs target independent analysis and optimization on LLVM IR, and back-ends apply target dependent optimization and generate object codes. The optimization in the back-ends includes instruction selection, instruction scheduling, register assignment, peephole optimization, etc., which utilize deep knowledge on the instruction architecture and the microarchitecture of the target.

```
 1 @x0 = internal global i16 20, align 2
 2 @x3 = constant i16 30, align 2
 3 @.str = private unnamed_addr constant 6 x i8 c "@OK@\0A\00
    ", align 1
 4 @.str1 = private unnamed_addr constant 18 x i8 c "@NG@ (te
    st = %hhd)\0A\00", align 1
 5 ; Function Attrs: nounwind uwtable
 6 define i32 @main() #0{
 7   %def_t0 = alloca i16, align 2
 8   store i16 10, i16* %def_t0, align 2
 9   %l_t0 = load i16* %def_t0, align 2
10   %l_x0 = load i16* @x0, align 2
11   %l_x3 = load i16* @x3, align 2
12   %t0 = add i16 %l_x3, %l_x0
13   %cp_t0 = icmp eq i32 %t0, 50
14   br i1 %cp_t0, label %2, label %3
15 ; <label>:2
16   %true_pr0 = call i32 (i8*, ...)* @printf(i8* getelementptr
      inbounds (6 x i8* @.str, i32 0, i32 0))
17   br label %4
18 ; <label>:3
19   %ng_pr0 = call i32 (i8*, ...)* @printf(i8* getelementptr i
      nbounds (18 x i8* @.str1, i32 0, i32 0), i32 %t0)
20   br label %4
21 ; <label>:4
22     ret i32 0
23 }
```

Fig. 1. An example of an LLVM test program.

Most functions of the back-ends are well tested by C compiler test suites [4] or randomly generated C programs [5], but some are not. For example, 8-bit and 16-bit integer arithmetic operations never appear in the LLVM IR for 32-bit machines, due to the integer promotion rule of the C language, except when the middle-end optimizer occasionally reduces the bit widths. Another example is comparison operation. Only one of either "$<$" or "$\geq$" comparison may be generated for the convenience of optimization. Bugs regarding the rarely-used operations may survive the test, and bugs related to the never-used operations, which are never tested, may be eminent in compilation for other languages or after future improvements on optimization modules.

III. Random test generation for LLVM back-ends

The proposed method generates random LLVM IR assembly codes. Fig. 1 is an example of a test code. Lines 1, 2, 7, and 8 declare and initialize variables and lines 9–12 execute an arithmetic operation, which is on 16-bit integers. Lines 13–19 verify the result.

The type of an operation in the C language is determined by the integer promotion rule (to extend short values to the size used in machine operations) and the arithmetic conversion rule (to extend to a common type of the operands). Table I (a) summarizes the rule for the C language when `int` is 32-bit (si64 and ui64 are omitted due to space limitation). For example, an operation

TABLE I
INTEGER PROMOTION AND ARITHMETIC CONVERSION RULES.

(a) C (where int is si32)

	si8	ui8	si16	ui16	si32	ui32
si8	si32	si32	si32	si32	si32	ui32
ui8	si32	si32	si32	si32	si32	ui32
si16	si32	si32	si32	si32	si32	ui32
ui16	si32	si32	si32	si32	si32	ui32
si32	si32	si32	si32	si32	si32	ui32
ui32	ui32	ui32	ui32	ui32	ui32	ui32

(b) Proposed LLVM test

	si8	ui8	si16	ui16	si32	ui32
si8	*si8*	*ui8*	*si16*	*ui16*	si32	ui32
ui8	*ui8*	*ui8*	*si16*	*ui16*	si32	ui32
si16	*si16*	*si16*	*si16*	*ui16*	si32	ui32
ui16	*ui16*	*ui16*	*ui16*	*ui16*	si32	ui32
si32	si32	si32	si32	si32	si32	ui32
ui32	ui32	ui32	ui32	ui32	ui32	ui32

si$n$: n-bit signed integer, ui$n$: n-bit unsigned integer

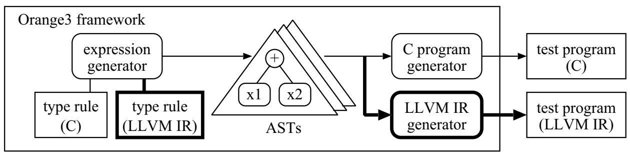

Fig. 2. Flow of test program generation.

TABLE II
FREQUENCY OF OPERATIONS IN LLVM IR.

op	C (-O0)		C (-O3)		**LLVM**	
	i8	i16	i8	i16	i8	i16
add	0	0	4	6	791	1708
and	0	0	3	2	0	0
lshr	0	0	40	35	0	0
mul	0	0	0	0	94	322
sdiv	0	0	0	0	26	143
srem	0	0	0	0	25	128
sub	0	0	0	0	77	320
udiv	0	0	18	16	76	213
urem	0	0	20	22	70	199

with operands of si8 and ui16 is promoted to that of si32. On the other hand, the proposed method only applies the arithmetic conversion as shown in Table I (b).

The program generation is achieved by extending Orange3 [5], as shown in Fig. 2. First, a set of abstract syntax trees (ASTs) is generated, each of which represents an assignment statement with an expression on the right-hand side. The ASTs are built carefully not to yield undefined behavior (such as zero division or signed overflow). From the ASTs, an LLVM IR assembly program is generated. During AST construction, the type conversion rule in Table I (b) is used instead of (a).

IV. EXPERIMENTAL RESULTS

A test program generator based on the proposed method has been implemented in Perl 5 by extending Orange3. It conforms to the LLVM 3.5.

To see how frequently each operation in LLVM IR is tested, its appearances were counted in 100 test programs each of which was designated to contain 100 operations. The target was x86_64-apple-macosx10.11.0. Table II summarizes the result. Columns "C (-O0)" and "C (-O3)" show the case where the tests were in the form of C programs and compiled with the -O0 and -O3 options, respectively, and column "LLVM" shows the case for the proposed method. "i8" and "i16" stand for the 8 and 16-bit integers, respectively. We can see that the 8 and 16-bit integer operations were never tested by the C programs with the -O0 option. Few operations of the short integers were generated with the -O3 option. On the other hand, the proposed method can generate much more tests for most of the operations.

The back-end of LLVM 3.5 for the x86_64 target was intensively tested by our system. The tests were run on a PC with Intel Core i7 1.6GHz and 16GB memory and Ubuntu 14.04 LTS. The result is summarized in Table III.

TABLE III
RESULT OF RANDOM TEST.

types	time [h]	#test	#error
i8, i16, i32, i64	120	183,520	1
i8, i16	120	296,143	1

```
 1 @.str = private unnamed_addr constant 6 x i8 c"@OK@\0A\00"
   , align 1
 2 @.str1 = private unnamed_addr constant 19 x i8 c"@NG@ (tes
   t = %hd)\0A\00", align 1
 3 ; Function Attrs: nounwind uwtable
 4 define i32 @main() #0{
 5   %1 = alloca i32, align 4
 6   store i32 0, i32* %1
 7   %def_t700 = add i16 0, 1
 8   %t700 = srem i16 -32768, -1
 9   %cp_t700 = icmp eq i16 %t700, 0
10   br i1 %cp_t700, label %2, label %3
11 ; <label>:2
12   %true_pr700 = call i32 (i8*, ...)* @printf(i8* getelement
     ptr inbounds (6 x i8* @.str, i32 0, i32 0))
13   br label %4
14 ; <label>:3
15   %ng_pr700 = call i32 (i8*, ...)* @printf(i8* getelementpt
     r inbounds (19 x i8* @.str1, i32 0, i32 0), i16 %t700)
16   br label %4
17 ; <label>:4
18   ret i32 0
19 }
20 declare i32 @printf(i8*, ...) #1
```

Fig. 3. Error program.

In the first run, the generator was configured to generate all the types i8 through i64. With 120 hours, 183,520 programs were generated out of which one detected an error. In the second run, types are restricted to i8 and i16. In 120 hours, one error was detected. Fig. 3 shows a minimized error program from the second run. This should be a valid program but its execution fails with "floating point exception." The `srem` operation in line 8 computes 16-bit signed remainder of -32768 by -1 which should yield 0. It is well-known that a C program with the 32-bit version (`-2147483648%-1`) fails for the x86 target. However, the 16-bit version is hard to test by C programs.

V. CONCLUSION

This paper has proposed a method of directly testing back-ends of LLVM by randomly generated LLVM IR, which targets bugs hard to detect by C programs. Future work includes experiments on various targets and generation of test programs which can directly test various optimization in LLVM back-ends.

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] LLVM Compiler Project (online), http://llvm.org/.

[2] A. Canis, et al.: "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. ACM FPGA '11*, pp. 33–36 (Feb.–Mar. 2011).

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang: "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. CAD*, vol. 30, no. 4, pp. 473–491 (Apr. 2011).

[4] ACE: SuperTest compiler test and validation suite (online), http://www.ace.nl/compiler/supertest.html.

[5] 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).