Random Testing of Back-end of Compiler Infrastructure LLVM

Kenji Tanaka¹ Nagisa Ishiura¹ Masanari Nishimura² Akiya Fukui²
¹Kwansei Gakuin University, Sanda, Hyogo, Japan
²Renesas 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 are used, one only has to implement a back-end dedicated to the target.

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 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.

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 “>” 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 1–8 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>
<thead>
<tr>
<th>Contents</th>
<th>Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>32</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 1. An example of an LLVM test program.
TABLE I
INTEGRAL PROMOTION AND ARITHMETIC CONVERSION RULES.
(a) C (where int is si32) (b) Proposed LLVM test

<table>
<thead>
<tr>
<th></th>
<th>si8</th>
<th>si16</th>
<th>si32</th>
<th>u8</th>
<th>u16</th>
<th>u32</th>
</tr>
</thead>
<tbody>
<tr>
<td>si8</td>
<td>si8</td>
<td>si8</td>
<td>si8</td>
<td>si8</td>
<td>si8</td>
<td>si8</td>
</tr>
<tr>
<td>si16</td>
<td>si16</td>
<td>si16</td>
<td>si16</td>
<td>si16</td>
<td>si16</td>
<td>si16</td>
</tr>
<tr>
<td>si32</td>
<td>si32</td>
<td>si32</td>
<td>si32</td>
<td>si32</td>
<td>si32</td>
<td>si32</td>
</tr>
</tbody>
</table>

TABLE II
FREQUENCY OF OPERATIONS IN LLVM IR.

<table>
<thead>
<tr>
<th>opr</th>
<th>C (-0O)</th>
<th>C (-03)</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>and</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lshr</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lsl</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>shr</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ulshr</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>shr</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 Or-
gange 3 [5], as shown in Fig. 2. First, a set of abstract
syntax trees (ASTs) is generated, each of which repre-

sents 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 over-
flow). 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 pro-
grams each of which was designated to contain 100 opera-
tions. The target was x86-64-apple-macosx10.11.0. Table
II summarizes the result. Columns “C (-0O)” and “C (-
03)” show the case where the targets were in the form of
C programs and compiled with the -00 and -03 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 -00 option. Few operations of the short integers
were generated with the -03 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.

<table>
<thead>
<tr>
<th>types</th>
<th>time [h]</th>
<th>#test</th>
<th>#error</th>
</tr>
</thead>
<tbody>
<tr>
<td>i8, i16, i32, i64</td>
<td>120</td>
<td>183,520</td>
<td>1</td>
</tr>
<tr>
<td>i8, i16</td>
<td>120</td>
<td>296,143</td>
<td>1</td>
</tr>
</tbody>
</table>

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 genera-
tion of test programs which can directly test various
optimization in LLVM back-ends.

Acknowledgement

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

REFERENCES

don testing of arithmetic optimization of C compilers by