Native Code Level Test of Optimizing Performance of Android Compilers

Naoki YOSHIDA [†,*] Toya HAMADA [†] Nagisa ISHIURA [††]
[†] Graduate School of Science and Technology [††] School of Engineering
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330, JAPAN

Abstract— In this paper, we introduce a technique for assessing the optimization performance of the Android DEX compiler at the level of native code. This method is designed to detect missed optimization in native codes generated by the Android runtime environment through random generation of Java programs. The detection of optimization deficiencies is performed using both differential method and equivalent program methods. In the differential method tests, we attempt to identify missed optimization by comparing the native code produced by newer and older versions of DEX compilers. In the equivalent program method, we aim to identify missed optimizations by comparing the native codes generated by a DEX compiler from both optimized and unoptimized source programs. The Random Java programs are generated from a modified version of Orange4, which were originally developted for generating C programs. The test systems, employing the proposed methods, effectively identified insufficient optimization in x86_64 native code generated by the d8 DEX compiler.

I. Introduction

Android[1] is an open-source operating system designed for mobile devices, and its utilization has been steadily on the rise in recent years. Android finds extensive application across various portable devices, including smartphones, which underscores the pressing need for improving its performance.

To tackle this challenge, continuous enhancements are being made to Android's compilers and runtime systems. As an alternative to the long-standing Dalvik virtual machine, Android adopted ART (Android Runtime) [1]. ART is based on Ahead-Of-Time (AOT) compilation, which precompiles parts of the code into native code. Consequently, it delivers superior runtime performance compared to Dalvik.

Alongside these developments, ensuring the reliability of these components through comprehensive testing is also a crucial task. In the context of testing Android's runtime, [2, 3, 4] have conducted testing of ART by generating random bytecodes. However, these tests primarily focus on error detection within the runtime systems and do not specifically address performance testing.

In contrast, this paper introduces a novel approach for evaluating the performance of Android's compilers and runtime. Our method aims to identify optimization shortcomings in both the DEX compiler and the AOT compiler in Android Runtime (ART). Identification of optimization deficiencies is carried out based on both differential testing [5, 6] and equivalence testing [7]. Class files compiled from randomely generated Java programs are used as test inputs. We detect the missed optimization by by analyzing the native code generated by the AOT compiler.

The experimental results with two test systems based on the proposed method have identified optimization deficiencies in the x86_64 native code produced by the DEX compiler d8 and ART.

II. Random testing of android processors and compiler optimizations

A. DEX compiler and Android virtual machine ART

The Android processing system is depicted in Fig. 1. First, a program undergoes compilation using either the Java or Kotlin compiler, resulting in the creation of a class file. Subsequently, the DEX compiler (dx or d8) processes the class file, producing a DEX file that encapsulates bytecode for Android's runtime environment, referred to as ART [1].

The virtual machine (VM) in ART executes bytecode through interpretation, but for performance improvement, a portion of the code is converted into native code by a backend compiler and executed directly by the processor. This compilation process is performed ahead of time (AOT) during app installation, not Just-In-Time (JIT).

Both DEX compilers, dx and d8, perform bytecode-level optimizations. Among them, d8 is the latest compiler, and it is known to exhibit equivalent or better runtime performance compared to dx. Additionally, d8 has shorter compilation times and generates smaller DEX files compared to dx.

ART takes the received DEX files and converts them into executable ELF format files called OAT files using dex2oat. Before compilation, dex2oat verifies the validity of the DEX files through verification and then extracts portions for compilation from bytecode through filtering.

In other words, dex2oat compiles only those methods that meet specific conditions, such as a certain method size (e.g., the number of instructions or maximum register numbers).

The backend compiler transforms the bytecode's selected portions into intermediate representation (IR) for optimization, and from IR, it generates native code. Finally, dex2oat adds the native code to the original DEX file's contents to create the OAT file.

[*] Currently with NSW Inc., Japan.
[1] `https://source.android.com` (accessed 2023-9-20)

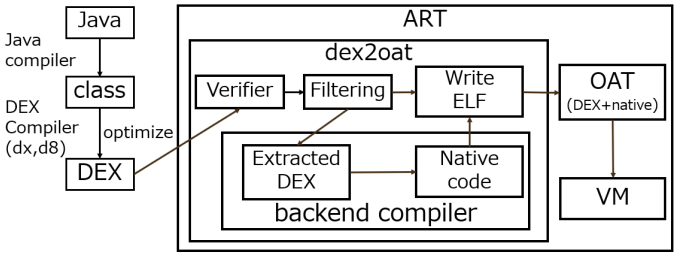
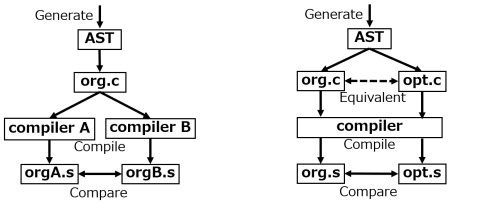


Fig. 1. DEX compiler and Android run time environment [4]



(a) Differential method (b) Equivalent program method

Fig. 2. Methods for testing compiler optimization [5]

```
01: #include <stdio.h>
02: #include <stdarg.h>
03: #define OK()  printf("@OK@\n")
04: #define NG(test,fmt,val)  printf("@NG@ (test = " fmt ")\n",val)
05:
06: static volatile unsigned short x86 = 3U;
07: const signed short x88 = 386;
08: static volatile unsigned short t0 = 2428U;
09: volatile unsigned long long t1 = 2LLU;
10: static volatile signed char t3 = 9;
11: volatile signed char x92 = 115;
12:
13: int main (void)
14: {
15:   static const unsigned char x33 = 1U;
16:   static const volatile signed long long x41 = -1LL;
17:   volatile unsigned char x52 = 5U;
18:   const volatile unsigned char x65 = 38U;
19:   static signed long long x77 = 0LL;
20:   volatile signed int x87 = 511;
21:   const volatile signed long x89 = 2029215198009L;
22:   const volatile unsigned int x90 = 140U;
23:   volatile unsigned short t2 = 6570U;
24:   const unsigned long long x91 = 1255744988028LLU;
25:   static volatile unsigned long t4 = 1404136965112782LU;
26:   static volatile signed char t5 = 17;
27:
28:   t0 = ((unsigned short)(((signed int)x87)&((signed short)x88)));
29:   t1 = ((unsigned long)(((signed long)x89)*((signed long long)x33));
30:   t2 = ((unsigned short)(((unsigned int)x90)<<(((signed char)x77))));
31:   if(((signed char)(((unsigned short)x86)*((signed int)x41)))) {
32:     t3 = ((signed char)(((unsigned long)x77)<<((signed long)x52)));
33:     t4 = ((unsigned long)(((signed char)t3)|((unsigned long)x91)));
34:   }
35:   t5 = ((signed char)(((signed char)x92)%((signed char)x65)));
36:
37:   if (t0 == 386U) { OK(); } else { NG("t0", "%hu", t0); }
38:   if (t1 == 2029215198009LLU) { OK(); } else { NG("t1", "%llu", t1);}
39:   if (t2 == 140U) { OK(); } else { NG("t2", "%hu", t2); }
40:   if (t3 == 0) { OK(); } else { NG("t3", "%hhd", t3); }
41:   if (t4 == 1255744988028LU) { OK(); } else { NG("t4", "%lu", t4); }
42:   if (t5 == 1) { OK(); } else { NG("t5", "%hhd", t5); }
43:
44:   return 0;
45: }
```

Fig. 3. Test program generated by Orange4 [9]

B. *Random testing of compiler optimization*

There are two distinct methods to assess the optimization performance of compilers: the differential method [5, 6] and the equivalent program method [7].

The differential method attempts to find optimization deficiencies in compilers by contrasting the outputs of two different compilers. The test procedure is illustrated in Fig. 2 (a). A test program `org.c` is compiled by distinct compilers (or different versions of the same compiler), compilerA and compilerB. The generated assembly codes denoted as `orgA.s` and `orgB.s` are then compared to identify optimization shortcomings, if any, in either of the compilers.

On the other hand, the equivalent program method tests a compiler using a pair of test programs which are functionally equivalent. In Fig. 2 (b), `org.c` and `opt.c` are the test programs, where `opt.c` is derived from `org.c` by applying anticipated optimization at the source code level. By comparing the assembly codes `org.s` and `opt.s` generated from them, optimization shortcomings are identified.

In terms of test programs for performance assessment, it is common to use benchmark programs or manually crafted test programs. However, for comprehensive testing, it is essential to utilize a large volume of test programs obtained through automatic generation.

In the context of automatic random generation of programs for compiler testing, a crucial challenge is how to avoid undefined behavior, such as division by zero, overflow, and infinite loops, all of which are tied to the dynamic behavior of these programs.

In the case of C program generation, tools like Csmith [8] achieve this by inserting guards to avoid undefined behavior. This approach facilitates the creation of programs that span a broad spectrum of C language syntax, but the forms of expressions or statements are limited.

In contrast, Orange4, as described in [9], guarantees the elimination of undefined behavior through the utilization of data structures that consistently track values for all variables and expressions within the generated program. While this approach may impose certain restrictions on program syntax, it empowers Orange4 to generate a wide array of expressions, characterized by both diversity and length.

Fig. 3 provides an example of a test program generated by Orange4. Lines 6–26 declare variables, lines 28–35 contain statements with arithmetic expressions and control flow, and lines 37–42 involve statements for checking computed results.

III. Random testing of Android compiler optimization based on native code comparison

A. *Outline*

This paper proposes novel random testing techniques designed to assess the optimization performance of the Android DEX compiler. The evaluation of the DEX compiler's optimization performance is based on the quality of the native code generated within dex2oat. Please note note that this evaluation takes place at the native code level, as the ultimate performance outcome is dictated by the execution of native code within the runtime environment, rather than the bytecode generated by the DEX compiler.

We present two distinct methods for performance testing, which are based on the differential method and the equivalent program method. The differential method involves comparing the native code outputs derived from the same Java program using two different DEX compilers. On the other hand, the equivalent program method revolves around the comparison of native code generated by a single compiler from two Java programs that are functionally equivalent.

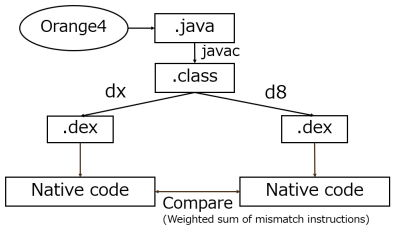


Fig. 4. Proposed flow of differential testing

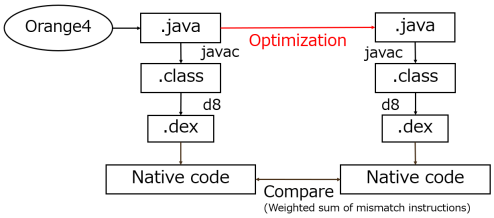


Fig. 5. Proposed flow of equivalence testing

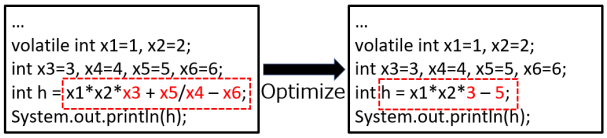


Fig. 6. Arithmetic optimization

The Java programs for the tests are generated by a modified version of Orange4 tailored for Java. Care is taken to generate only those those Java programs that meet the conditions for native code generation by dex2oat.

B. *Differential method*

In the differential testing approach, two versions of DEX compilers are used. Typically, performance of a new version is tested against an old version, where the goal is to check whether the code generated by the newer version is not inferior to that generated by the older one.

The flow of the differential testing is illustrated in Fig. 4. Randomly generated Java programs are compiled, resulting in class files, which are then compiled using the two versions of DEX compilers, denoted as “dx” and “d8” in the figure, to generate two DEX files. Native codes generated by ART from each of these DEX files are extracted and compared to detect performance differences.

C. *Equivalent program method*

In the equivalent program method, it is tested if a combination of a DEX compiler and AOT compilers perform intended optimization to generate native codes. The workflow of this method is illustrated in Fig. 5. In this approach, equivalent programs are generated by applying the expected optimizations at the source code level to randomly generated test programs. By compiling these programs using the d8 compiler and comparing the resulting two native codes, the goal is to detect optimization deficiencies.

In this paper, our specific focus lies in arithmetic optimization, which simplifies expressions during compile time. While this optimization is considered fundamental, it frequently exposes compiler bugs. Fig. 6 provides an illustrative example of arithmetic optimization in action. Initial constant values are assigned to the variables `x3`, `x4`, `x5`, and `x6`, and subexpressions are evaluated during the compilation process. Note that variables that variables `x1` and `x2` cannot be substituted with constants due to their declaration as `volatile`, indicating the potential for external updates to their values during program execution.

D. *Native code comparison*

The native code segments to be compared are derived from the OAT files generated by ART from DEX files. A textual dump of an OAT file encompasses various pieces of information, from which only the lines containing instructions and operands relevant to the target processor are isolated. An example of the extracted native code is illustrated in Fig. 7.

The comparison of native code is performed following the method described in [5], as shown in Fig. 8.

Initially, code pairs are examined from the start to the end, and contiguous sections where two codes exhibit matching instructions and operands for a minimum of k instructions (with k typically around 7) are designated as matching sections, while the remaining segments are classified as non-matching sections. Within a non-matching section, pairs featuring identical instructions are filtered out. The sum of the weights assigned to the remaining instructions (with higher weights assigned to instructions presumed to consume more execution cycles, such as multiplication/division and branch instructions) is assessed, and if the ratio surpasses a predefined threshold, it signifies a significant performance disparity in the code.

E. *Generation of random Java programs*

We adapted the random C program generator of Orange4 for generating Java programs. Major modifications include the following three points.

1. Variable modifiers

   The generator has been adjusted to generate variable modifiers in accordance with the Java language specification. The modifiers are `volatile`, `final`, `static`, `private`, `protected`, `public`, and `transient`. Specifically, the `final` modifier is applied to variables that undergo a single assignment, while the `static` modifier is assigned to all class variables but not to local variables.

2. Statements

   The statements generated in the test programs are `if`, `for`, and `switch`. `while` statements are omitted due to a minor technical issue with Orange4.

3. Explicit type conversion between integer and boolean types

```
 1:0x00001010:                C3     ret
 2:0x00001020:     85842400E0FFFF     test eax, [esp + -8192]
 3:0x00001027:                55     push ebp
 4:0x00001028:             83EC18     sub esp, 24
 5:0x0000102b:             890424     mov [esp], eax
 6:0x0000102e: 6466833D0000000000     cmpw fs:[0x0], 0  ; state_and_flags
 7:0x00001037:       0F8584000000     jnz/ne +132 (0x000010c1)
 8:0x0000103d:             8B28     mov ebp, [eax]
 9:0x0000103f:   64833D6003000000     cmp fs:[0x360], 0  ; pReadBarrierMarkReg05
10:0x00001047:       0F8580000000     jnz/ne +128 (0x000010cd)
11:0x0000104d:          F6450710     test [ebp + 7], 16
12:0x00001051:       8B85AC000000     mov eax, [ebp + 172]
13:0x00001057:       0F857C000000     jnz/ne +124 (0x000010d9)
14:0x0000105d:             837808     cmp [eax + 8], 0
15:0x00001061:       0F867E000000     jbe/na +126 (0x000010e5)
16:0x00001067:          F6400710     test [eax + 7], 16
17:0x0000106b:             8B500C     mov edx, [eax + 12]
18:0x0000106e:       0F857D000000     jnz/ne +125 (0x000010f1)
19:0x00001074:       8B85B8000000     mov eax, [ebp + 184]
20:0x0000107a:             85C0     test eax, eax
21:0x0000107c:       0F847B000000     jz/eq +123 (0x000010fd)
22:0x00001082:             837A0800     cmp [edx + 8], 0
23:0x00001086:       0F8678000000     jbe/na +120 (0x00001104)
24:0x0000108c:          0FBF420C     movsxw eax, [edx + 12]
25:0x00001090:                99     cdq
26:0x00001091:             89C3     mov ebx, eax
27:0x00001093:             0BDA     or ebx, edx
28:0x00001095:       0F8475000000     jz/eq +117 (0x00001110)
29:0x0000109b:             89D3     mov ebx, edx
30:0x0000109d:             89C2     mov edx, eax
31:0x0000109f:         B817855D9D     mov eax, -1654815465
32:0x000010a4:         B999381D00     mov ecx, 1915033
33:0x000010a9:     64FF1594020000     call fs:[0x294]  ; pLdiv
34:0x000010b0:       8985B0000000     mov [ebp + 176], eax
35:0x000010b6:       8995B4000000     mov [ebp + 180], edx
36:0x000010bc:             83C418     add esp, 24
37:0x000010bf:                5D     pop ebp
38:0x000010c0:                C3     ret
39:0x000010c1:     64FF15E0020000     call fs:[0x2e0]  ; pTestSuspend
40:0x000010c8:         E970FFFFFF     jmp -144 (0x0000103d)
41:0x000010cd:     64FF1560030000     call fs:[0x360]  ; pReadBarrierMarkReg05
42:0x000010d4:         E974FFFFFF     jmp -140 (0x0000104d)
43:0x000010d9:     64FF154C030000     call fs:[0x34c]  ; pReadBarrierMarkReg00
44:0x000010e0:         E978FFFFFF     jmp -136 (0x0000105d)
45:0x000010e5:             8B4808     mov ecx, [eax + 8]
46:0x000010e8:             33C0     xor eax, eax
47:0x000010ea:     64FF15E8020000     call fs:[0x2e8]  ; pThrowArrayBounds
48:0x000010f1:     64FF1554030000     call fs:[0x354]  ; pReadBarrierMarkReg02
49:0x000010f8:         E977FFFFFF     jmp -137 (0x00001074)
50:0x000010fd:     64FF15EC020000     call fs:[0x2ec]  ; pThrowDivZero
51:0x00001104:             8B4A08     mov ecx, [edx + 8]
52:0x00001107:             33C0     xor eax, eax
53:0x00001109:     64FF15E8020000     call fs:[0x2e8]  ; pThrowArrayBounds
54:0x00001110:     64FF15EC020000     call fs:[0x2ec]  ; pThrowDivZero
```

Fig. 7. Native code extracted form OAT file

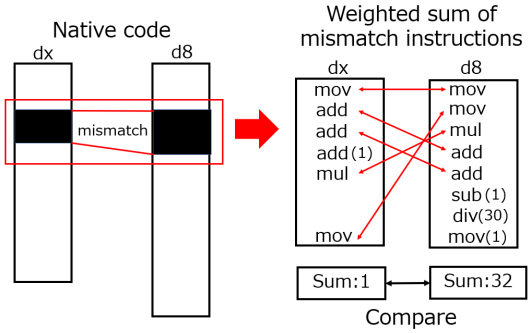


Fig. 8. Comparison based on weighted sum [5]

TABLE I
TYPE CONVERSION BETWEEN INTEGER AND BOOLEAN

C	→	Java
`t1=(x1<x2)+x3;`	→	`t1=((x1<x2)?1:0)+x3;`
`t1=(x1+x2)&&(x3<x4);`	→	`t1=((x1+x2)!=0)&&(x3<x4);`
`if(x1*x2){...};`	→	`if((x1*x2)!=0){...};`

Integer values and boolean values may be used interchangablly in C but they are strictly distinguished in Java. So the expression generator is modified to insert subexpressions for appropriate type conversion. Some examples of the conversion are provided in Table I.

With the exception of these issues, the framework for program generation from Orange4 remains unchanged.

Test programs encompass scalars and arrays of data types `byte`, `short`, `int`, `long`, and expressions are composed with

```
 1:class test {
 2:static volatile long []x12 =  { 8097786081625503911L } ;
 3:static long [][]t2 =
     { {-426199267453973891L,310236433L,4098038826782914 3L,0L} } ;
 4:static short x21 = 1;
 5:static volatile byte x22 = 0;
 6:static volatile long x26 = -9223372036740299116L;
 7:static long x30 = -9223372036854775807L;
 8:static long x31 = 1L;
 9:static volatile short x34 = -125;
10:static volatile short x37 = 9;
11:static transient int t7 = -89;
12:static volatile int x39 = 0;
13:
14:public static void main (String args[]){
15: byte x9 = 0;byte x9 = 0;
16: short []x10 =  { 1 } ;
17: int []x11 =  { 0,0,0,23251,45021,-27,-1243,-10,3121 } ;
18: final short [][]x14 =  { {-19,1,5,-12,547,133,0,-8912,1736},
                             {-149,-599,-360,-1,1,-9906,8836,13,1} } ;
19: int x18 = 1;
20: short x19 = 1;
21: final int x20 = 1;
22: short x23 = 0;
23: byte x24 = -1;
24: long x25 = -9223372036854775808L;
25: long x32 = -1L;
26: short x35 = 1;
27: short x36 = -14;
28: final int x40 = -1;
29: final int x41 = 0;
30: if( (((( int)(((( byte)(x23>>(( int)x22)))
      *(( int)(( short)(x20*(( short)(( int)
      ((( long)x23)-x24)))))))|x10[(( int)x22)]))
      *(( short)(x20*x21)))!= 0 ?true:false){;}
31: t7 = ( int)(( int)(((x30/x31)-(( long)x19))
      -((x26/(( long)(x35*(( short)x32))))
      /(( short)(( byte)(x36*x37))%x34)))));
32: t2[x11[2]][x41] = ( long)(x12[(( int)x9)]/
                      (( long)(x14[(( int)(( byte)(x25*x11[1])))]
                      [(x39/x40)]/(( short)x14[x11[0]][x18])));
33: int h = 0;
34: h += t7;
35: h += t2[0][0];
36:
37: System.out.println(h);
38: }
39:}
```

Fig. 9. Java program generated by modified Orange4

arithmetic operators(+, -, *, /, %), shift operators(<<, >>), bitwise operators(&, |, ^), comparison operators(<, >, <=, >=, ==, !=), and logical operators(&&, ||).

To prevent compiler optimizers from eliminating the entire code within test programs, each method is compelled to return the hash value of the resultant variables.

An example of a test program generated by this method is shown in Fig. 9. Lines 2–29 are variable declarations, lines 30–32 contain statements with arithmetic expressions and control flow, and lines 33–35 calculates the hash value.

To ensure ART to convert byte codes to native codes, it is crucial to pay special attention to the size parameters of the test programs. Each class should contain exactly one method. In addition, the maximum number of operators per program should be limited (typically, to as much as 25).

IV. EXPERIMENTAL RESULTS

We implemented two test systems using the proposed methods, namely the differential method and the equivalent program method, in Perl 5. The systems run on Ubuntu Linux. The test generation component running on Ubuntu produces DEX files, which are transmitted to an Android emulator also running on the Ubuntu system via the Android Debug Bridge (adb). The version os the ART was 2.1.0.

Two test systems based on the proposed methods, differential method and equivalent program method, were implemented in Perl 5. The systems run on Ubuntu Linux. DEX files, generated by the test generation component running on Ubuntu, are sent to Android emulator running on the same Ubuntu system through the Android Debug Bridge (adb). The

TABLE II
RESULT BY DIFFERENTIAL TESTING

(a) x86_64

target→ reference ↓	dx	d8
dx	—	7
d8	62	—

3,000 tests

(b) ARM

target→ reference ↓	dx	d8
dx	—	0
d8	13	—

3,000 tests

test1.java

```
 1: class test
 2: {
 3:   static int [][] t0 = {{1}};
 4:   static int x1 = 1;
 5:
 6:   public static void main (String args[])
 7:   {
 8:     int t1 = 1;
 9:     t1 = 1/t0[0][0&(1/x1)];
10:  }
11: }
```

test1.s (dx)

```
        ...
 1: mov eax, [edx + 176]
 2: test eax, eax
 3: jz/eq +73 (0x000010ca)
 4:
 5:
 6:
 7:
 8:
 9: cmp [ebx + 8], 0
10: jbe/na +70 (0x000010d1)
11: mov eax, [ebx + 12]
12: test eax, eax
13: jz/eq +71 (0x000010dd)
14: add esp, 28
15: ret
        ...
```

test1.s (d8)

```
        ...
 1: mov ecx, [ecx + 176]
 2: test ecx, ecx
 3: jz/eq +90 (0x000010db)
 4: mov eax, 1
 5: cmp ecx, -1
 6: jz/eq +83 (0x000010e2)
 7: cdq
 8: idiv edx:eax, edx:eax / ecx
 9: cmp [ebx + 8], 0
10: jbe/na +74 (0x000010e6)
11: mov eax, [ebx + 12]
12: test eax, eax
13: jz/eq +75 (0x000010f2)
14: add esp, 12
15: ret
        ...
```

Fig. 10. Test program detecting under-optimization of d8

test2.java

```
 1: class test
 2: {
 3:   static int x0 = 1;
 4:
 5:   public static void main (String args[])
 6:   {
 7:     int t0 = 1;
 8:     int []x1 = {1} ;
 9:     long [][]x2 = {{1,1}} ;
10:     t0 = (int)(x2[0&(int)(x0*x2[0][0])][(1/x1[0])]);
11:   }
12: }
```

test2.s (dx)

```
        ...
 1: mov ecx, [ecx + 172]
 2: xor eax, ecx
 3: xor ecx, eax
 4: xor eax, ecx
 5: cdq
 6: mov eax, [ecx + 8]
 7: mov edx, [ecx + 16]
 8: mov ebx, [ecx + 20]
 9: cmp [ebx + 8], 0
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23: mov eax, 1
24: mov eax, eax
25: jnb/ae/nc +73 (0x00001114)
26: add esp, 20
27: pop ebp
28: pop esi
29:
30: ret
        ...
```

test2.s (d8)

```
        ...
 1: mov ecx, [ecx + 172]
 2: xor eax, ecx
 3: xor ecx, eax
 4: xor eax, ecx
 5: cdq
 6: mov ebx, [ecx + 8]
 7: mov esi, [ecx + 16]
 8: mov edi, [ecx + 20]
 9: cmp [ebx + 8], 0
10: mov [esp + 4], esi
11: mov [esp + 8], edi
12: mov esi, eax
13: mov edi, edx
14: mov eax, [esp + 8]
15: imul eax, esi
16: imul edi, [esp + 4]
17: add edi, eax
18: mov eax, esi
19: mul edx:eax, eax * [esp + 4]
20: add edi, edx
21: mov esi, eax
22: mov eax, esi
23: mov eax, 1
24: mov eax, eax
25: jnb/ae/nc +77 (0x0000113d)
26: add esp, 32
27: pop ebp
28: pop esi
29: pop edi
30: ret
        ...
```

Fig. 11. Test program detecting under-optimization of d8

version of the ART subjected to testing was 2.1.0.

A. *Differential method*

The results of the differential testing are presented in Table II. The target ISAs for testing were x86_64 and ARM. In total, we executed 3,000 test programs, with the testing duration being approximately 3 hours for x86_64 and around 6 hours for ARM.

In the table, "target" represents the compiler under evaluation, while "reference" refers to the compiler serving as a benchmark. The numerical values represent the quantity of programs for which the target compiler exhibited optimization shortcomings in comparison to the reference compiler. For example, in Table II (a), 62 programs identified optimization shortcomings in dx when contrasted with d8, and conversely, 7 programs identified optimization deficiencies in d8 when compared to dx.

Assuming that d8 is newer than dx, it is reasonable that dx exhibits higher count of optimization shortcomings than d8 in both (a) and (b). No optimization deficiencies were detected in d8 for ARM. On the other hand, 7 programs in x86_64 detected optimization deficiencies in d8, indicating a regression in performance during the version upgrade. The 7 programs could be classified into two types.

A program of the the first type has been minimized, which is shown in Fig. 10. "test1.java" is a test program in Java and "test1.s (dx)" and "test1.s (d8)" are native codes generated by dx and d8, respectively. The native codes are identical except for lines 4–8, where d8 introduced generated an instruction sequence including division operation (idiv). The results of the idiv instruction are stored in the registers edx and eax, but they remain unused subsequently: at line 11 eax is reassigned another value at line 11 (it is also not used in the jump destination at line 10). This observation implies that d8 has failed to eliminate unnecessary codes in lines 4–8.

TABLE III
RESULT BY EQUIVALENCE TESTING

Test programs	Under-optimization	Time [s]
3,000	558	13,229

A minimized test program of the second type is shown in Fig. 11. The sequence of instructions d8 generated in lines 10–22 including multiplication operations (imul and mul) has no effect to the subsequent instructions and should have been removed through optimization.

org.java

```
 1: class test {
 2:   static volatile int x1 = 1;
 3:   public static void
 4:   main(String args[]){
 5:     int x2 = 1;
 6:     int x3 = 0;
 7:     int h = (x3<<x2)*(1/x1);
 8:     System.out.println(h);
 9:   }
10: }
```

opt.java

```
 1: class test {
 2:   static volatile int x1 = 1;
 3:   public static void
 4:   main(String args[]){
 5:     int x2 = 1;
 6:     int x3 = 0;
 7:     int h = 0*(1/x1);
 8:     System.out.println(h);
 9:   }
10: }
```

org.s

```
                ...
 1: jz/eq +92 (0x000010bc)
 2: mov eax, 1
 3: cmp ecx, -1
 4: jz/eq +86 (0x000010c4)
 5: cdg
 6: idiv edx:eax, edx:eax / ecx
 7: mov eax, [RIP + 0xf89]
 8: cmpb [rax + 115], -16
 9: jb/nae/c +71 (0x000010c8)
10: test [rax + 7], 16
11: mov esi, [rax + 244]
                ...
```

opt.s

```
                ...
 1: jz/eq +75 (0x000010ab)
 2: mov eax, [RIP + 0xf9a]
 3: cmpb [rax + 115], -16
 4:
 5:
 6:
 7:
 8:
 9: jb/nae/c + 67 (0x000010b3)
10: test [rax + 7], 16
11: mov esi, [rax + 244]
                ...
```

Fig. 12. Test programs detecting under-optimization of d8

org.dex

```
        ...
0x0000: nop
0x0001: nop
0x0002: sget v1, I test.x1 // field@1
0x0004: const/4 v0, #+1
0x0005: div-int/2addr v0, v1
0x0006: const/4 v1, #+0
0x0007: mul-int v1, v1, v0
0x0009: sget-object v0, Ljava/io/PrintStream;
          java.lang.System.out // field@0
0x000b: invoke-virtual {v0, v1}, void
          java.io.PrintStream.println(int) // method@0
0x000e: return-void
        ...
```

opt.dex

```
        ...
0x0000: nop
0x0001: nop
0x0002: sget v1, I test.x1 // field@1
0x0004: const/4 v0, #+1
0x0005: div-int/2addr v0, v1

0x0006: mul-int/lit8 v0, v0, #+0
0x0008: sget-object v1, Ljava/io/PrintStream;
          java.lang.System.out // field@0
0x000a: invoke-virtual {v1, v0}, void
          java.io.PrintStream.println(int) // method@0
0x000d: return-void
        ...
```

Fig. 13. DEX codes for org.java and opt.java

B. Equivalent program method

The d8 compiler was tested by 3,000 pair of programs, targeting the x86_64 ISA. The result is summarized in Table III. By this test, 558 programs identified optimization shortcomings in d8.

Fig. 12 shows one of the (minimized) test programs. "org.java" and "opt.java" are unoptimized and optimized Java programs, respectively, and "org.s" and "opt.s" are resulting native codes. Lines 4–8 of "org.s" which includes a divicaion (idiv) instruciton should be optimized away.

For reference, Fig. 13 is a part of the bytecode (DEX code), which is generated before being converted into native code by the AOT compiler in ART. At this stage, both programs generate the div instruction, indicating that the deficiency in optimization is not in the DEX compiler but rather in the AOT compiler of ART.

V. Conclusion

This paper has proposed performance optimization test methods for Android compilers based on native code comparison. Through experiments, the both differential testing and equivalence testing successfully detect optimization deficiencies in the d8 compiler and the AOT compiler in ART.

Our current testing system is limited in its scope, primarily addressing a small portion of compiler optimization, specifically focusing on arithmetic optimization. Extending our testing methodology to encompass other aspects of compiler optimization is a direction for future research.

Acknowledgments

Authors would like to express their appreciation to the members of Ishiura Lab. of Kwansei Gakuin Univ. for their cooperation.

References

[1] Android Open Source Project (Android core technologies), https://source.android.com/devices/tech/dalvik/ (accessed 2021-12-17).

[2] S. Kyle, H Leather, B. Franke, D. Butcher, S. Monteith: "Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing," ACM SIGPLAN Notices, vol. 50, Issue 7, pp. 121132 (July 2015).

[3] H. Ikeo, R. Shimizu, and N. Ishiura: "Random Testing of Android Virtual Machine by Valid DEX File Generation" (in Japanese), in *Technical Report of IEICE*, VLD2017-88, (Feb2018).

[4] R. Shimizu and N. Ishiura: "Reinforcing Generation of Instruction Sequences in Random Testing of Android Virtual Machine" (in Japanese), in *Technical Report of IEICE*, VLD2018-124, (Mar. 2019).

[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] G. Barany: "Finding Missed Compiler Optimizations by Differential Testing," in *Proc. International Conference on Compiler Construction (CC 2018)*, pp. 92–91 (Feb. 2018).

[7] 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).

[8] 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).

[9] 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. 676679 (Oct.2016).