一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2021-74,CPSY2021-43,RECONF2021-82(2022-01)

ネイティブコード比較に基づく Android DEX コンパイラの最適化性能テスト

吉田　直生[†]　石浦菜岐佐[†]

† 関西学院大学　〒669–1337 兵庫県三田市学園 2–1

あらまし　本稿では, ネイティブコードの比較に基づく Android DEX コンパイラの最適化性能のランダムテスト手法を提案する. Android の DEX コンパイラ dx と d8 がクラスファイルから生成するコードを比較することにより最新版である d8 の最適化不足の検出を試みる. コードの比較は, DEX ファイルのバイトコードではなく, Android の実行系 ART のバックエンドコンパイラがバイトコードから生成するネイティブコードで行う. テストの入力となるクラスファイルは, Java 生成用に修正した Orange4 が生成するランダムな Java プログラムをコンパイルして得る. 提案手法に基づくテストシステムを実装して実験を行った結果, d8 から生成される x86_64 用ネイティブコードにおいて, 最適化不足を検出することができた.

キーワード　Android, DEX コンパイラ, dx, d8, ランダムテスト, 最適化性能テスト

Testing of Optimization Performance of Android DEX Compilers Based on Native Code Comparison

Naoki YOSHIDA[†] and Nagisa ISHIURA[†]

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This paper presents a method for testing optimizing performance of Android DEX compilers based on comparison of resulting native codes. By comparing object codes generated from class files by DEX compilers dx and d8, detection of missed optimization by the newer d8 is attemted. The code comparison is made on native codes which are produced by the back-end compiler of the ART (Android Runtime) from the byte codes generated by dx and d8. The test class files to dx and d8 are obtained by compiling random Java programs generated by Orange4. A test system based on the proposed method has successfully detected missed optimization in x86_64 native codes originated from d8.

Key words　Android, DEX compiler, dx, d8, random tesing, performance test

1. はじめに

Android [注1] はモバイルデバイス向けのオープンソースオペレーティングシステムであり, 近年活用される局面が増えてきている． 特に Android はスマートフォンを始めとする多くの携帯型デバイスで使用されるため, 処理性能の向上は非常に重要な課題となっている. これに応えるべく, Android のコンパイラや実行系には日々改良が行われているが, これに伴いその信頼性を確保するためのテストも重要な課題となる.

Android 実行系の仮想マシンとして長期間採用されていた dalvik よりも性能を向上させるべく新たに採用されたのが ART [1] である. ART は AOT (Ahead-Of-Time) コンパイルに基づいており, コードの一部を事前にコンパイルしたネイティブコードで実行するため, dalvik と比較してより良い実行時性能を発揮する.

Android の実行環境の仮想マシンに入力するバイトコードをクラスファイルから生成する DEX コンパイラとしては dx が長期間採用されてきたが, 最近新たに d8 が採用された. d8 は dx と比較してより良い実行時性能を発揮するとされている.

Android 実行系のランダムテストに関しては 文献 [2] [3] [4] がある. 文献 [2] では, ランダムに生成した Java プログラムから DEX ファイルを生成することにより, DEX コンパイラと ART の実行系に誤りがないかのテストを行っている. 文献 [3] [4] は, ランダムなバイトコードを生成することにより ART のテストを行っている. しかしこれらはコンパイラや実行系の誤り検出を目的としたものであり, 性能を対象としたテストではない.

これに対し本稿では, DEX コンパイラを対象とした最適化性能のテスト手法を提案する. 本手法では差分法 [5] を用いたネ

(注1)：https://source.android.com (accessed 2021-12-17)

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2022 by IEICE

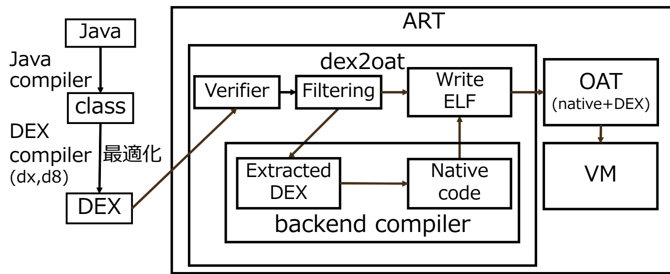


図 1: 処理系の概要 [4]

イティブコード比較により最適化不足を検出する．テストの入力となるクラスファイルは, 文献 [6] の手法でランダム生成した Java プログラムをコンパイルして得る．提案手法に基づくテストシステムを実装し実験を行った結果, dx と比較して d8 から生成される x86_64 用ネイティブコードにおいて, 最適化不足を検出することができた．

以下, 本稿では 2 章で ART の構造と DEX コンパイラ, コンパイラの最適化性能のテストついて述べ, 3 章では本手法に基づくシステムについて述べる．4 章で実装及び実験について述べた後, 5 章でまとめと今後の課題について述べる．

2. Android の処理系とコンパイラ最適化のランダムテスト

2.1 DEX コンパイラと Android 仮想マシン ART

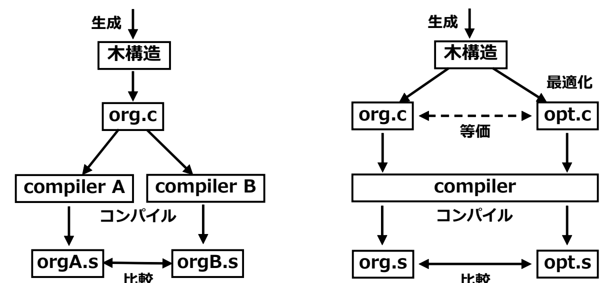
Android の処理系の概要を図 1 に示す．まず, プログラムを Java または Kotlin コンパイラでコンパイルし, クラスファイルを生成する．DEX コンパイラ (dx, d8) はクラスファイルから Android の実行環境である ART [1] の仮想マシン (VM) のバイトコードを含む DEX ファイルを生成する．仮想マシンはバイトコードをインタープリタで実行するが, 実行効率向上のためにコードの一部をバックエンドコンパイラでネイティブコードに変換してプロセッサで直接実行する．このコンパイルは JIT ではなく事前 (アプリケーションのインストール時) に行われる．

DEX コンパイラ dx, d8 はともにバイトコードのレベルで最適化を行う．このうち d8 が最新のコンパイラであり, dx と比較して同等あるいはそれ以上の実行時性能を発揮するとともに, dx よりコンパイル時間が短く, 生成される DEX ファイルのサイズも小さいとされている．

ART は 受け取った DEX ファイルを, dex2oat で ELF 形式の実行可能ファイル OAT に変換し, それを仮想マシンへの入力とする．dex2oat はベリファイアで DEX ファイルの正当性検査を行った後, フィルタリングでバイトコードからネイティブコードにコンパイルする箇所を抽出する．即ち, dex2oat は全てのバイトコードではなく, メソッドの規模 (命令数あるいは使用レジスタ番号の最大値等) が一定の条件を満たすものだけをネイティブコードにコンパイルする．バックエンドコンパイラは, バイトコードの対象箇所を中間表現 (IR) に変換して最適化を行い, IR からネイティブコードを生成する．最後に dex2oat は元の DEX ファイルの内容にネイティブコードを追加して OAT ファイルを生成する．

2.2 コンパイラの最適化のランダムテスト

コンパイラの最適化性能のテストの手法としては, 差分法 [5] [7] と等価プログラム法 [8] がある．差分法では, 図 2 (a) のようにテストプログラム org.c を異なるコンパイラ (あるいは同じコンパイラの異なるバージョン) compilerA, compilerB でコンパイルし, 生成したアセンブリコード orgA.s, orgB.s を比較することによっていずれかのコンパイラの最適化不足を検出する．これに対し等価プログラム法では, 図 2 (b) のようにテストプログラム org.c と, それに期待される最適化をソースコードレベルで行ったプログラム opt.c から生成されるアセンブリコード org.s と opt.s を比較することによって, 意図通りの最適化が行われているかをテストする．



(a) 差分法 (b) 等価プログラム法

図 2: コンパイラの最適化性能のランダムテスト手法 [5] [7] [8]

テストへの入力となるプログラムは, 手動で作成したり, ベンチマークプログラムを用いることもあるが, 徹底的なテストを行うには, 大量のランダムなプログラムを自動生成して用いる．

コンパイラのテストのためにランダムなプログラムを生成する手法においては, ゼロ除算やオーバフロー等の未定義動作や無限ループを含むプログラムの生成をいかに回避するかが重要な課題になる．C プログラムのランダムテスト生成を行う Csmith [9] では, 式や文の構文に制限を設けることによって未定義動作を回避しつつ, 関数呼び出し, 配列や構造体, ポインタ等を含む C 言語の幅広い構文要素を含むプログラムを生成している．これに対し Orange4 [10] は, 生成できる構文は制限されるが, プログラム中の全ての変数と式の値を保持したデータ構造を用いて未定義動作や無限ループを全く引き起こさないプログラムを生成する．

Orange4 が生成するテストプログラムの例を図 3 に示す．6–26 行目が変数宣言, 28–35 行目が算術式や制御文を含む文, 37–42 行目が計算結果を照合する文である．

3. ネイティブコード比較に基づく Android DEX コンパイラの最適化性能のランダムテスト

3.1 概　　要

本稿では, Android DEX コンパイラの最適化性能のランダムテスト手法を提案する．テスト対象は dx, d8 である．d8 が最新のコンパイラであることに鑑み, d8 の生成するコードが dx よりも劣っていないか, 即ち d8 の生成するコードの最適化不足がないかをテストする．

DEX コンパイラの最適化性能は, dex2oat 中で生成されるネイティブコードの品質に基づいて判断する．DEX コンパイラが生成するバイトコードの外見に関わらず, 最終的な実行性能はネイティブコードによって決まると考えるからである．

最適化不足の検出には差分法を用い, DEX コンパイラの入力となるクラスファイルはランダム生成した Java プログラムから得る．本手法の流れを図 4 に示す．Java プログラムをランダム生成し, それをコンパイルして得られるクラスファイルを dx と d8 でコンパイルして 2 つの DEX ファイルを生成する．

```
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: }
```

図 3: Orange4 が生成するプログラムの例 [10]

図 4: 提案手法

それぞれから ART 中で生成されるネイティブコード[1]を抽出し，その比較を行うことによって 2 つのコンパイラの性能差を検出する．

Java プログラムのランダム生成は，Orange4 を Java 生成用に修正したものを用いる．この際 dex2oat でネイティブコード生成される条件を満たす Java プログラムだけを生成するようにする．

3.2　ネイティブコードの比較に基づく性能テスト

比較対象のネイティブコードは，DEX ファイルを ART に入力した際に内部で生成される OAT ファイルから抽出する．OAT ファイルをダンプすると，バイトコードやネイティブコードをはじめとした様々な情報が出力されるが，その中からターゲットプロセッサの命令とオペランドを含む行のみを抽出することによってネイティブコードを取得する．抽出するネイティブコードの例を図 5 に示す．

ネイティブコードの比較は文献 [5] の方法 (図 6) により行う．まずコードの対を先頭から走査し，2 つのコードで k 命令以上

```
 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:         83780800    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
```

図 5: 本手法で生成するネイティブコードの例

図 6: 重み和による比較 [5]

(k は 7 程度) 連続して命令とオペランドが一致している部分を一致部分，そうでない部分を不一致部分とする．不一致部分を一つ取り出し，その中で同じ命令の対を削除する．残った命令の重みの和 (実行サイクル数が大きいと考えられる乗除算，および分岐命令に大きな重みを与える) を比較し，その比率が閾値を超えた場合にコードに優位な性能差があると判定する．

最新版である d8 の方が性能が良いことが期待されるので，dx のコードの方が優れていれば，d8 の最適化性能に劣化があったと判定できる．

3.3　Java プログラムの生成

Orange4 は C コンパイラ用のランダムテストシステムであるため，下記 4 点の変更を加え，Java のテストプログラムを生成できるようにする [6]．

（1）　変数の型

生成する変数の基本型を Java の言語仕様に合わせて byte 型，short 型，int 型，long 型に変更し，unsigned は生成しないようにする．

（2）　変数の修飾子

修飾子も言語仕様に合わせて volatile, final, static, private, protected, public, transient を生成する．final 修飾された変数

には再代入できないため, 参照用の変数にのみ付けるようにする．static は クラス変数全てに付け, ローカル変数には付けないようにする．

（3） 生成する構文

生成する構文は if 文, for 文, switch 文である．Orange4 では変数と式の値をプログラム中で一意に保つため, ループは繰り返し回数が 1 回以下となるようにコードを生成するが, Java の while 文ではその工夫を施したコードに到達不能コードがあると判定されてエラーになるため, while 文は生成しない．

（4） boolean 型への対応

比較演算と論理演算の結果は, C では整数型であるが Java では boolean 型になる．これは, 抽象構文木からプログラムを生成する際に必要な型の変換を追加することにより対応する．

- 比較演算や論理演算が整数型コンテキスト式中に現れる場合には, 3 項演算子を用いて結果を整数型に変換する．例えば, C 用に

```
t1 = (x1 < x2) + x3;
```

という式を生成していた場合, Java 用では

```
t1 = ((x1 < x2) ?  1 :  0) + x3;
```

に修正して出力する．

- 比較演算や論理演算のオペランドが整数型であった場合は, 0 との不等号比較を挿入して boolean 型に変換する．例えば, C 用に

```
t1 = (x1 + x2) && ( x3 < x4 );
```

という式を生成していた場合, Java 用では

```
t1 = ((x1 + x2) != 0) && ( x3 < x4 );
```

に修正して出力する．

- if 文等の条件式の式が整数型であった場合, boolean 型に変換する．例えば, C 用に

```
if ( x1 * x2 ) {...};
```

という文を生成していた場合, Java 用では

```
if ( (x1 * x2) != 0 ) {...};
```

に修正して出力する．

これ以外は Orange4 のプログラム生成の枠組みをそのまま用いる．生成するデータ型はスカラおよび配列であり, 生成する演算子は算術演算子 (+, -, *, /, %), シフト演算子 (<<, >>), ビット演算子 (&, |, ^), 比較演算子 (<, >, <=, >=, ==, !=), および論理演算子 (&&, ||) である．

提案手法は正しいコードを生成しているかどうかのテストを行うものではないので, 計算結果と期待値の照合は行わない．ただし, 計算結果を使っていないと最適化により意図しないコード削除が行われるので, メソッドが結果のハッシュ値を返すようにする．

また, ART がネイティブコードを生成するためのメソッドの規模の制約を満たすため, クラス中のメソッド数は 1 つ, 1 つのプログラムあたりの演算子の数は最大で 25 個にする．

本手法で生成するテストプログラムの例を図 7 に示す．2–29 行目までが変数宣言であり, 30–32 行目が算術式や制御文を含む文であり, 33–35 行目がハッシュの計算である．

4. 実 験 結 果

提案手法に基づくテストシステムを Perl 5 で実装した．本システムは Ubuntu Linux 等で動作する．テスト生成部が Ubuntu 上で生成した DEX ファイルは, adb (Android Debug Bridge) ツールを用いてそれぞれのターゲットに転送して実行した．テスト対象の Android の実行には本システムと同じ Ubuntu 上で動作する Android エミュレータを使用した．

```
 1:class test {
 2:static volatile long []x12 =  { 809778608162550391L } ;
 3:static long [][]t2 =
      { {-4261992674539738 9L,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:}
```

図 7: 本手法で生成する Java プログラムの例

ターゲットプロセッサとしては, x86_64 と ARM を対象にテストを行った．実験の結果を表 1 に示す．テスト数は 3,000 で, テストに要した時間は x86_64 が約 3 時間, ARM が約 6 時間である．“target” はテスト対象のコンパイラを, “reference” は比較対象のコンパイラを表し, 表中の数は target が reference に比べて最適化不足と判定したプログラムの数である．例えば表 1 (a) では, dx が d8 に比べて最適化不足であることを検出したプログラムが 62 件, 逆に d8 の最適化性能不足を検出したプログラムが 7 件 であることを示している．

(a) (b) いずれにおいても dx をターゲットにした方の最適化不足の件数が多くなっていることから, d8 の性能が向上していると見ることができる．ARM では d8 の最適化不足を検出することはできなかったが, x86_64 の 7 つのプログラムで d8 の最適化不足を検出した．7 つのプログラムは 2 種類に分類できた．

x86_64 で最適化不足を検出した 1 種類目のプログラムを最小化したものを図 8 に示す．ネイティブコードの 4–8 行目以外は一致しており, d8 では 4–8 行目に除算 (idiv) を含むコードが余分に生成されている．idiv 命令の結果が代入された edx, eax は以降で使われることなく, eax には 11 行目で別の値が代入されている (10 行目のジャンプ先でも使われていない) ため, d8 では 4–8 行目の不要コード削除が行われていないことが分かる．

x86_64 で差分を検出した 2 種類目のプログラムを最小化したものを図 9 に示す．d8 では 10–22 行目に複数の乗算 (imul

表 1: テスト結果

(a) Target: x86_64

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

テストプログラム数: 3,000

(b) Target: ARM

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

テストプログラム数: 3,000

test.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:  }
```

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
      ...
```

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
      ...
```

図 8: d8 の最適化不足を検出したテストプログラム

test.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:  }
```

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
      ...
```

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
      ...
```

図 9: d8 の最適化不足を検出したテストプログラム

と mul) 命令が生成されており, ここでも不要コード削除が行われていないことが分かる.

5. むすび

本稿では, ネイティブコード比較に基づく Android DEX コンパイラの最適化性能テストを提案した. 実験の結果, dx と比較し d8 から生成される x86_64 用ネイティブコードにおいて, 最適化不足を検出することができた.

本稿の手法では, d8 の dx に対する最適化不足しか検出できない. これ以外の最適化不足を検出するために等価プログラム法によるテストを行うことが今後の課題として挙げられる.

謝　辞

本研究の遂行に際してご助言を頂いた元 関西学院大学 清水 遼太郎氏, および本研究に関してご討論頂いた関西学院大学石浦研究室の諸氏に感謝致します.

文　献

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

[2] 清水遼太朗, 池尾弘史, 石浦菜岐佐: “コンパイラのランダムテストシステム Orange3 の拡張による Java 処理系のテスト,” 信学ソ大, A-6-11 (Sept. 2016).

[3] 池尾弘史, 清水遼太郎, 石浦菜岐佐: “正当な DEX ファイルの生成による Android 仮想マシンのランダムテスト, 信学技報, VLD2017-95 (Feb. 2018).

[4] 清水遼太朗, 石浦菜岐佐: “Android 仮想マシンのランダムテストにおける命令列生成の強化,” 信学技報, 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] 吉田直生, 石浦菜岐佐: “等価変換に基づくランダムテストプログラム生成による Java 処理系のテスト,” 信学総大, A-6-6 (Mar. 2020).

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

[8] 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).

[9] 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).

[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).