Vol.2020-OS-149 No.4
2020/5/28

JIT とコンパイラの動作を考慮した Kotlin 記述プログラムの性能向上

園山敦也[1] 神山剛[2] 福田晃[2] 小口正人[3] 山口実靖[1]

概要：2016 年に新しいプログラミング言語である Kotlin がリリースされ，注目を集めている．Kotlin は Java との高い相互運用性を持つ言語であり今後は Java が利用されている環境などを中心に普及していくと考えられ，非常に重要な言語の一つとなっている．本研究では，Kotlin 記述プログラムと Java 記述プログラムの性能を比較し，JIT コンパイラとコンパイラの振る舞いを考慮した性能差が生じる理由を考察し，本考察に基づく性能向上手法を提案する．そして，本提案手法の性能評価を行い，本手法の有効性を示す．具体的には，両言語で記述した同等の意味のプログラムの性能を比較し，性能差が存在することを示す．そして，性能差は各言語のコンパイラが生成するバイトコードの差異と，その差異により生じる JIT コンパイラの最適化の振る舞いの違いに起因することを明らかにする．

キーワード：Java，Kotlin，JVM，JIT コンパイラ，Java bytecode

1. はじめに

静的型付けのプログラミング言語 Kotlin が，Breslav らにより 2011 年に開発され，2016 年に公式にリリースされた[1]. Kotlin は Java との高い相互運用性を持つなどの特徴を有しており，Android OS 用アプリケーション開発の公式の第一言語に指定されるなど，近年非常に注目を集めている．Kotlin は Java と高い相互運用性を持ち，Kotlin で記述されたプログラムは Kotlin コンパイラで Java バイトコードにコンパイルされ, Java 実行環境である JVM(Java Virtual Machine)にて実行される．Java 言語で開発された既存の Java バイトコードのソフトウェアを Kotlin プログラムで用いることや，Kotlin で開発し Kotlin コンパイラにより生成された Java バイトコードを Java プログラムで用いることが可能となる．すなわち，Kotlin で開発されたプログラムは既存の Java のソフトウェア群(JVM やその中に実装されている JIT コンパイラなど)を用いて処理や実行がされていくこととなり，これらの既存ソフトウェア実装環境にて高い性能を示すことが重要となる．Java 言語は 1995 年にリリースされ，その後の約四半世紀にわたり開発や実装の改善が行われてきており, Java コンパイラが出力した Java バイトコードを JVM や JIT(Just-in-time)コンパイラが効率的に処理できるよう改善がなされてきた．一方，Kotlin は登場および広い普及からの年月が比較的短く，これら実装群が必ずしも効果的に動作するように十分な最適化がなされていない可能性も予想できる．

本稿では，Kotlin で記述したプログラムを既存の Java のソフトウェアの実装群で処理や実行することの重要性に着目し, 著名な Java 実行環境における Kotlin 記述のプログラムの性能について考察をする．具体的には，Java および Kotlin で記述した意味が同等のマイクロベンチマークソフトウェアを著名な Java 実行環境(Windows OS における Oracle JVM)で実行しそれらの性能を比較し，意味が同等であっても Kotlin で記述したプログラムの性能が劣ることを示す．次に，Java コンパイラ(同様に Windows OS 用 Oracle Java 開発キッド)が出力した Java バイトコードと, Kotlin コンパイラ(公式の Windows 版開発キッド(version.1.3.31)[3]が出力したバイトコードを比較し，両者にわずかな差異が存在することを示す. そして，両バイトコードを上記 JVM で実行したときの JIT コンパイラの振る舞いを示し，前述の両バイトコードのわずかな差異が JIT コンパイラの振る舞いに影響を与え，結果的に性能に差が生じることを示す. さらに，Kotlin コンパイラから出力されたバイトコードを改変し Java コンパイラが出力したバイトコードと類似のものとすることにより Kotlin 記述プログラムの性能を向上させる手法を提案し，性能評価によりその効果を示す.

2. Kotlin

Kotlin は Jetbrains によって開発された静的型付けのプログラミング言語であり，オブジェクト指向と関数型プログラミングのスタイルをサポートしている.

Kotlin は JVM(Java Virtual Machine)環境をターゲットにして使用することができ，その場合は Kotlin で記述されたソースコードが Java バイトコードにコンパイルされ, そのバイトコードが JVM によって解釈，実行される．多くの場合 JVM 実装は JIT(Just-In-Time)コンパイラを搭載しており，バイドコード実行中には JIT コンパイラによるバイトコードのネイティブコードへの置き換え(コンパイル)が行われ，これにより高速化が達成されている.

3. 関連研究

Java 記述プログラムと Kotlin 記述プログラムの性能に関する考察としては，Schwermer による Kotlin プログラムと Java プログラムの性能評価[2]がある．本研究では，メモリ消費量や GC(Garbage Collection)の動作などに関して詳細な評価が行われ，Java 言語プログラムの高速性などが示されている．ただし，当該研究は ART(Android Runtime)上での性能のみを評価しており，本稿で注目した著名な JVM 実装については考察をしてない．また，コンパイラが出力するバイトコードへの評価は統計的な分析に

留まっており，JIT コンパイラの動作を考慮したバイトコード出力やその改変による性能高以上などに関する考察はなされていない．

我々は過去の研究[4][5][6][7]にて，Java 記述ソースコードの性能と Kotlin 記述ソースコードの性能についての評価を行っているが，評価は非常に簡易なものにとどまっている．すなわち，Java バイトコードの詳細な比較，ネイティブコードの比較，ループアンローリングなどの JIT コンパイラの最適化動作を考慮した考察などは行われていない．

4. 性能評価

本章にて，Kotlin 記述プログラムと Java 記述プログラムの性能比較を行う．具体的には，プログラムの基礎的な機能の一つである for 文(繰り返し文)の処理性能について比較を行う．

4.1 評価方法

中身がなく for 文による繰り返しのための処理のみを行うマイクロベンチマークと，所定の浮動小数点計算を for 文により繰り返すマイクロベンチマークを Java と Kotlin で作成し，これらベンチマークを JVM 上で実行してその実行時間を計測した．Java で記述したマイクロベンチマークのソースコードの繰り返しの例を図 1 に，Kotlin で記述における繰り返し部の例を図 2 に示す．
図 1，図 2 に示すソースコードは for 文内の浮動小数点計算の行数が 2 行，ループの繰り返し回数が 1 億回の場合の例である．中身がなく for 文による繰り返しのための処理のみを行うマイクロベンチマークにおいては，for 文内の浮動小数点計算の行数が 0 行となる．基礎性能調査では，for 文の内部計算行数とループ回数を変化させ，計算行数とループ回数と処理時間の関係を調査した．実験環境は表 1 の通りである．

表 1．本実験の測定環境

OS	Windows 10 Home Edition ver.1903 (build 18362.778)
CPU	Intel corei5-7200U
RAM	8GB
SSD	256GB
JVM	Oracle Hotspot VM version 1.8.0_231
Java Compiler	version 1.8.0_231
Kotlin Compiler	version 1.3.31

```
public class Bench_For_Java_winVM_sonoyama_simple2_contents {
   public static void main(String args[]){
      double a=30000.0;
      long timestart = System.nanoTime();
      for(int i=1;i<=100000000;i++){
         a*=1.0001;
         a*=0.9999;
      }
      long timeend = System.nanoTime();
      double timecount = (timeend - timestart)/1000000.0;
      System.out.println(a+";"+timecount);
   }
}
```

図 1．Java 記述 for 文マイクロベンチマーク

```
fun main(){
   var a:Double = 30000.0
   val timestart = System.nanoTime()
   for(i:Int in 1..100000000){
      a*=1.0001
      a*=0.9999
   }
   val timeend = System.nanoTime()
   val timecount:Double = (timeend - timestart)/1000000.0
   println("$a;$timecount")
```

図 2．Kotlin 記述 for 文マイクロベンチマーク

4.2 実行時間

4.2.1 for 文の繰り返し処理のみを行うベンチマーク

ループ回数 1 億回，計算行数を 0 行(空回り)における測定結果を図 3，図 4 に示す．図 3 は JIT コンパイラ有効時の，図 4 は JIT コンパイラ無効時の結果である．縦軸は 1 億回のループの実行に要する時間であり，100 回の試行(1 回の試行は 1 億回のループ)の平均である．

JIT コンパイラ有効時の図 2 に着目すると，Java 記述プログラムの実行時間が Kotlin 記述のものより 95%以上短く，極めて大きな実行時間の差があることが分かる．JIT コンパイラ無効時の図 3 に着目すると，JIT コンパイラ有効時の結果とは大きく異なり Java 記述と Kotlin 記述で大きな実行時間の差はなく，Kotlin 記述の方が 10%以上高速であることが分かる．JIT コンパイラ無効時の結果である図 3 にて大幅な速度差が確認されなかったことより，図 2 の大幅な速度差は JIT コンパイラの動作に起因するものであると予想できる．

4.2.2 浮動小数点計算を繰り返す for 文ベンチマーク

JIT コンパイラが有効でループ回数が 1 億回における，計算行数(2 行から 16 行)と平均実行時間の関係を図 5 に示す．また，計算行数 2 行における，ループ回数(5000 万回から 2 億回)と平均実行時間の関係を図 6 に示す．図 5，図 6 に示す両実験で Java 記述プログラムの方が高速であり，JIT コンパイラ有効時は空回りだけでなく中に処理を記述した for 文においても Java が高速であることが確認された．

計算行数を変化させた図 5 に着目すると，Java 記述プロ

グラムと Kotlin 記述プログラムの実行時間の差は，for 文内の計算行数に関わらず一定であり，約 128[ms]であることがわかる．次にループ回数を変化させた図 6 に着目すると，ループ回数を変化させると，図 5 の結果とは異なり，ループ回数を増やすにつれ Java 記述プログラムと Kotlin 記述プログラムの実行時間の差がほぼ比例して大きくなっていくことがわかる．Java 記述と Kotlin 記述の間の実行時間が，for 文内の計算の数に依存せず，ループ回数に比例して変化することから，確認された for 文マイクロベンチマークの性能差は for 文処理を実行する方法(Java バイトコード上の実装)や JIT コンパイラによるその最適化の違いから生じていると予想できる．

中身がある for 文ベンチマークについて JIT コンパイラを無効化して実行時間を測定したところ，図 5，図 6 に示すような Java の高速性は確認されなかった．よって，for 文における Java の高速性は JIT コンパイラの最適化の優位性によるものだと予想できる．

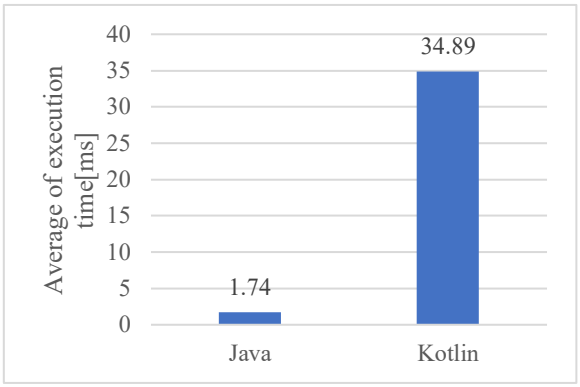

図 3．JIT コンパイラ有効時の for 文の繰り返し処理のみを行うベンチマーク

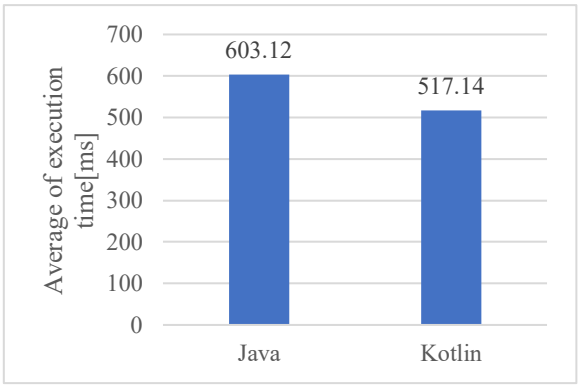

図 4．JIT コンパイラ無効時の for 文の繰り返し処理のみを行うベンチマーク

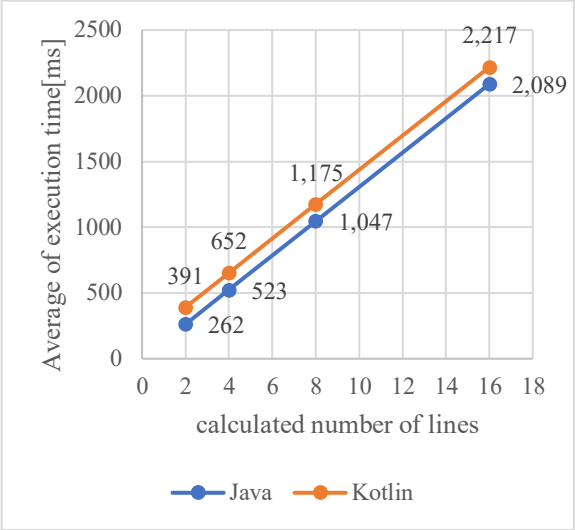

図 5．計算行数と実行時間の関係 (浮動小数点計算を繰り返す for 文ベンチマーク，JIT コンパイラ有効)

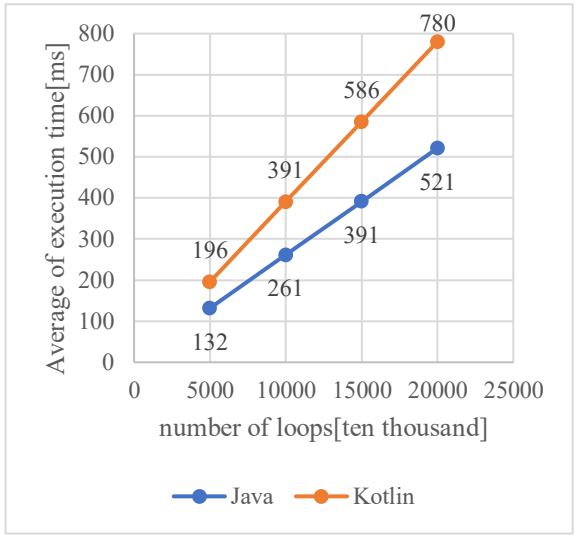

図 6．ループ回数と実行時間の関係(浮動小数点計算を繰り返す for 文ベンチマーク，JIT コンパイラ有効)

4.3 ニーモニックの比較

Java と Kotlin で記述された for 文マイクロベンチマークをそれぞれのコンパイラでコンパイルしたときに得られたバイトコードの for 文処理部をニーモニックで表したものを図 7，図 8 に示す．図 7 は Java 記述，図 8 は Kotlin 記述の中身無し for 文のものである．

図 7，図 8 から，ソースコードがほぼ同一である中身が空の for 文であっても，Java 記述ソースコードをコンパイルして得られたバイトコードと，Kotlin 記述ソースコードをコンパイルして得られたバイトコードには差異が存在することが分かる．

中身がある for 文のマイクロベンチマークについても同様に，両言語記述のソースコードから生成されたバイトコードを比較したところ，上記と同様の結果が得られた．すなわち，中身がある Java 言語記述の for 文と中身がない

Java 言語記述の for 文のバイトコードは，for 文内に浮動小数点計算用の命令があるか否かのみの違いがあり，for 文の繰り返しの処理部(ループカウンタや終了条件の管理など)には差異は確認されなかった．

図 7 と図 8 の両ニーモニックを比較すると，図 7 に示す Java 記述ソースコードから生成されたバイトコード(およびそのニーモニック)ではループ回数である 1 億を毎回スタックへ積み比較を行っており，一方で図 8 に示す Kotlin 記述ソースコードから生成されたバイトコードでは 1 億をあらかじめ局所変数に格納して呼び出していることが分かる．先述の通り，図 3 で確認された Java と Kotlin の大幅な性能差は JIT コンパイラの最適化の違いによって生じていると考えられるが，最適化の違いはこれらのバイトコードの差異により生じていると考えられる．

```
L4:   iconst_1
L5:   istore_3

      .stack append Long Integer
L6:   iload_3
L7:   ldc 100000000
L9:   if_icmpgt L18
L12:  iinc 3 1
L15:  goto L6
```

図 7．Java 記述空回り for 文マイクロベンチマークのバイトコード(for 文該当部)

```
L4:   iconst_1
L5:   istore_2
L6:   ldc 100000000
L8:   istore_3

      .stack append Long Integer Integer
L9:   iload_2
L10:  iload_3
L11:  if_icmpgt L20
L14:  iinc 2 1
L17:  goto L9
```

図 8．Kotlin 記述空回り for 文マイクロベンチマークのバイトコード(for 文該当部)

4.4 JIT コンパイル後の機械語(ネイティブコード)の比較

JIT コンパイラが実行時に Java バイトコードをコンパイルし出力した機械語(CPU のネイティブコード)を記録し，比較を行った．

Java 記述のソースコードから生成されたバイトコードの JIT コンパイル結果では，単純に(すなわち最適化による書き換えを行うことなく Java バイトコードを機械的に置換し)ループ処理を行っているわけではないことが確認された．中身のある for 文では，一部がアンロールされ，ループカウンタは 1 ずつではなくより大きな値ずつ(展開した量に応じて)増加しながら実行されていることが確認された．空回りの for 文では，ループのアンロールではなく，for 文部を繰り返さずに(すなわち実行せずに)抜けるネイティブコードとなっていた．すなわち，Java コンパイラが出力したバイトコードに関しては，JIT コンパイラの最適化により当該部は実行の必要ないコードだと判断され実行されないコードが出力されたと予想できる．これに対して，Kotlin 記述のソースコードから生成されたバイトコードでは，中身のある for 文，空回りの for 文の両者ともに JIT コンパイラによるループのアンロールがされず，単純に(最適化による書き換えを行うことなく)与えられた回数のループを処理していることが確認された．

以上より，本 Java 実行環境(Oracle JVM)の JIT コンパイラは，Java 記述のソースコードから生成されたバイトコードに対してはループを検出し，アンロールなどの最適化を行い，実行時間の短縮が実現されているが，Kotlin 記述のソースコードから生成されたバイトコードに対してはこれらを行なわず，Java 言語記述のプログラムより低い性能となっていることが分かる．

5. 提案手法

本章にて，Kotlin バイトコードを Java バイトコードと同等になるように改変し Kotlin ソースコードをコンパイルして作成したアプリケーションの性能を向上させる方法を提案する．

Kotlin 記述のソースコードからコンパイルされた Java バイトコードの for 文処理部を編集し，Java コンパイラが出力したバイトコードの for 文処理部と同等のものとする．これにより，既存の Java 実行環境実装の JIT コンパイラなど最適化実装の最適化が施されやすくなり，結果として現在の実行環境における実行性能が向上することと期待される．

本修正は，既存の実行環境実装の最適化が施されやすくするためのものである．すなわち，効率的な Java バイトコードに書き換えるものではない．具体的には図 7，図 8 で見られた様に，Kotlin コンパイラが出力した Java バイトコードにてループ回数を指定する値(図では 1 億)を局所変数に格納してループ内で呼び出す命令記述を，Java コンパイラが出力した Java バイトコードの様に局所変数に格納せずにループ内で即値として 1 億を記述するように修正する．

6. 性能評価

本章にて，5 章で提案した Kotlin 由来の Java バイトコード(Kotlin ソースコードを Kotlin コンパイラによりコンパイルして得られた Java バイトコード)を Java 由来のバイトコードと同等に改変する手法の性能の評価を行う．

6.1 実行時間

6.1.1 for 文の繰り返し処理のみを行うベンチマ

ーク

4 章 2 節 1 項と同様に，ループ回数が 1 億，計算行数が 0 という設定にて，Kotlin 改変バイトコードと無改変の Java バイトコードを用いて実験を行った．実行時間を図 9 と図 10 に示す．図 9 は JIT コンパイラ有効時，図 10 は JIT コンパイラ無効時の結果である．改変した Kotlin バイトコードを用いた結果は Kotlin_edited と表記されている．

図 9，図 10 から，JIT コンパイラの有効時，無効時の両方で Kotlin 改変バイトコードの実行時間が Java と同等になっていることがわかる．図 3 に見られた大きな速度差はなくなり，改変により Kotlin のバイトコードも Java のバイトコードと同様に JIT コンパイラにより大きく最適化されたことがわかる．図 10 に示す JIT コンパイラ無効時の結果では，図 4 に示す無改変の Kotlin バイトコードで得られた実行時間よりも遅くなっていることが分かる．図 9 同様に，Java コンパイラ出力のバイトコードと同等のバイトコードに修正しているため，性能が Java 記述のバイトコードと同等になったためである．この性能の劣化については 7 章にて考察を行う．

6.1.2 浮動小数点計算を繰り返す for 文ベンチマーク

4 章 2 節 2 項と同様に，計算行数を変化させた中身有り for 文の実験とループ回数を変化させた中身有り for の実験を，Kotlin 改変バイトコードを用いて行った．計算行数を変化させた実験の平均実行時間を図 11 に，ループ回数を変化させた実験の平均実行時間を図 12 に示す．図 11，図 12 共に，Java 由来のバイトコードの実行時間と Kotlin 由来のバイトコードを改変したものの実行時間がほぼ等しくなっている(図においてはほぼ重なっている)ことが分かる．

Kotlin_edited を用いて両実験を JIT コンパイラ無効状態において行ったところ，実行時間が Java 由来のものとほぼ等しくなることが確認された．前節同様にこれは性能の悪化となっている．同様に，7 章に考察を記す．

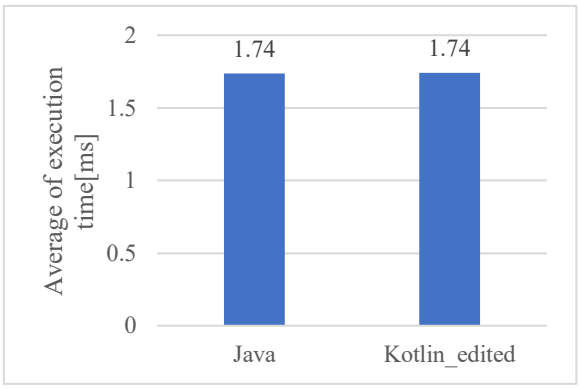

図 9.　Kotlin 改変バイトコードを用いた
JIT コンパイラ有効時の for 文空回し平均実行時間

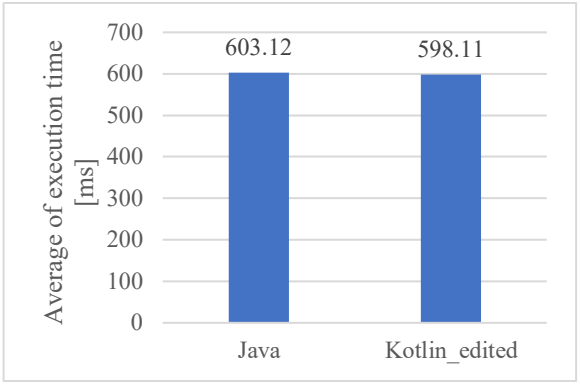

図 10.　Kotlin 改変バイトコードを用いた
JIT コンパイラ無効時の for 文空回し平均実行時間

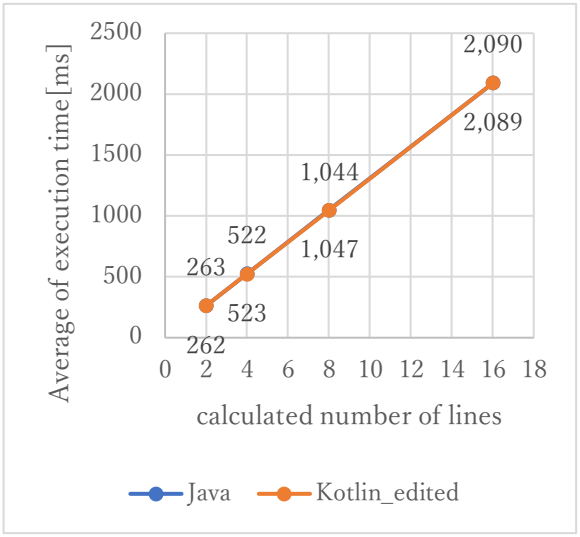

図 11．Kotlin 改変バイトコードを用いた
計算行数を変化させた中身あり for ベンチマーク
(JIT コンパイラ有効)

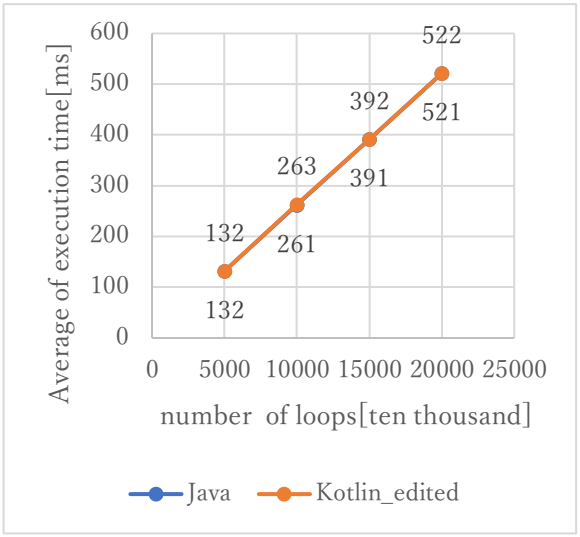

図 12．Kotlin 改変バイトコードを用いた
ループ回数を変化させた中身あり for ベンチマーク
(JIT コンパイラ有効)

6.2 JIT コンパイル後の機械語の比較

Kotlin 改変バイトコードを実行した際に JIT コンパイラが出力したネイティブコードを記録し，比較を行う．Kotlin の無改変のバイトコードを実行した際のネイティブコードは，4 章 4 節で述べた様にループを単純に繰り返すものであったが，Kotlin 改変バイトコードを実行した際に得られたネイティブコードでは，4 章 4 節の Java 由来のバイトコードで見られたようなループのアンロールとループカウンタ増加値の変化が見られた．すなわち，Kotlin 由来のバイトコードも，その一部を改変することにより JIT コンパイラにより Java 由来のバイトコードと同様にループの検出や最適化が行われることが確認された．

7. 考察

基礎性能調査にて, JIT コンパイル有効の環境では Kotlin 由来のバイトコードは Java 由来のものに対して性能が劣ることが確認された．そして提案手法を適用した Kotlin 改変バイトコードは JIT コンパイラによって Java と同等に最適化を施され速度差が解消されることが確認された．これにより JIT コンパイラの最適化の振る舞いは入力されるバイトコードに依存し，大きな性能差を生むことが明らかになった．ただし，Kotlin 由来のバイトコードへ全く性能向上処理が施されなかったということではない. 図 3, 図 4 に示されている通り，JIT コンパイラ有効時と無効時では大きな実行時間の違いがある．このことから，Kotlin 由来のバイトコードも JIT コンパイラにより大きな性能向上がなされていることが分かる．ただし，4 章 4 節で述べたように JIT コンパイラによるループのアンロールは行われていない．つまり，JIT コンパイラが Kotlin 由来のバイトコードを解釈した時に, for 文のループをアンロールすべきループとして検出できていない可能性が考えられる．

図 10 などの JIT コンパイラ無効の例においては Kotlin バイトコードを Java バイトコードと同様のものに修正し，性能が劣化することが確認された．このことから，Java コンパイラが出力するバイトコードは，バイトコード群が最適化されており短時間の実行が可能なものであるわけではなく，JIT コンパイラなどの既存実装が効果的に動作して高い性能が得られるものであることがわかる．また，図 9 の大幅な性能向上が既存の JIT コンパイラの動作を制御する(すなわち最適化がかかりやすくする)ことにより得られていることから, 既存の Java ソフトウェア実装群の上で効率的に動作する様にバイトコードなどを実装に依存させて最適化していくことが性能向上を行う上で非常に重要であることが分かる．

また，JIT コンパイラ無効時の性能において Java 由来のバイトコードは Kotlin 由来のバイトコードよりも劣る結果が得られ，Java コンパイラが出力したバイトコードには容易な最適化の余地が残されていると考えることもできるが，多くの場合は JIT コンパイラが有効の状態で実行されるためこの性能の低さが実際にユーザエクスペリエンスの低下につながる事例はまれであると予想され，同コンパイラは既存の実行環境実装に対して十分に最適化がなされていると考えることもできる．

8. おわりに

本稿では，Kotlin と Java で記述されたマイクロベンチマークプログラムの性能を比較し，内容がほぼ同一であっても性能に大きな差が生じることを示した．そして，両言語のソースコードからそれぞれのコンパイラが生成する Java バイトコードを比較し，バイトコードに差があることを示した．また，バイトコードの違いにより JIT コンパイラの最適化に差が生まれていることを，JIT コンパイル後ネイティブコードの比較により示した．最後に Kotlin 由来のバイトコードを Java 由来のものと同等に改変することにより Kotlin 由来のプログラムの性能を向上する方法を提案し，性能評価によりその有効性を示した．

今後は，JIT コンパイラのループの最適化を考慮した性能向上についての考察，Kotlin 由来のバイトコードに対しても Java 由来のものと同様にループのアンロールなどの最適化を行える JIT コンパイラの開発などに取り組む予定である．

謝辞 本研究は JSPS 科研費 17K00109，18K11277 の助成を受けたものである。本研究は、JST、CREST JPMJCR1503 の支援を受けたものである。

参考文献

[1] Andrey Breslav, Kotlin 1.0 Released: Pragmatic Language for JVM and Android. https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/ <accessed May 1, 2020>.

[2] Schwermer, Patrik, “Performance Evaluation of Kotlin and Java on Android Runtime,” TRITA-EECS-EX; 2018:417, 2018.

[3] Kotlin 1.3.72.Github.https://github.com/JetBrains/kotlin/releases/download/v1.3.72/kotlin-compiler-1.3.72.zip.(参照 2020-05-07)

[4] 園山敦也, 柴田涼一, 佐藤悠祐, 神山剛, 福田晃, 小口正人, 山口実靖, ``JIT とコンパイラの動作を考慮した Kotlin 記述プログラムの性能向上に関する一考察", 第 31 回コンピュータシステム・シンポジウム（ComSys2019）, poster-13, 2019.

[5]Ryoichi Shibata, Akira Fukuda, Yusuke Sato, Takeshi Kamiyama, Masato Oguchi, Saneyasu Yamaguchi, ``A Study on Compiler Dependent Performance Improvement," HPC Asia 2020: International Conference on High Performance Computing in Asia-Pacific Region, 2020.

[6]Atsuya Sonoyama, Masato Oguchi, Takeshi Kamiyama, Akira Fukuda, Saneyasu Yamaguchi, ``Performance Improvement of Kotlin Program in Consideration of JIT Compiler Optimization," 2020 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-Taiwan) - Advanced Computing Systems and Applications, 2020.

[7]Ryoichi Shibata, Atsuya Sonoyama, Masato Oguchi, Takeshi Kamiyama, Akira Fukuda, Saneyasu Yamaguchi, ``Java Android Application Performance Improvement by Kotlin DEX Bytecode Analysis without JIT Compiler," 2020 IEEE International

Conference on Consumer Electronics - Taiwan (ICCE-Taiwan) -
Advanced Computing Systems and Applications, 2020.