M-020

携帯端末 Java の高速化手法の検討と評価
Accelerating Execution of Java for Hand-held Device

高橋　克英[†]　　清原　良三[†]
Katsuhide TAKAHASHI　　Ryozo KIYOHARA

1. はじめに

近年、携帯端末に搭載された Java 実行環境には、ゲームや株価表示等の様々アプリケーションが提供されている。

携帯端末上の Java 仮想マシン(以下、Java VM)[1]は、プログラム内のバイトコードをインタプリタ処理にて実行することが一般的であり、実行速度が遅い。株価表示、案内等の情報提供を行うアプリケーションでは、実行速度は問題とならないが、シューティング・ゲームでは重要である。

パーソナル・コンピュータに搭載された Java 実行環境には、Java バイトコードをネイティブコードに変換するネイティブコンパイラを用いる JIT[2]や Hotspot[3]が採用されている。しかし、多くのメモリを使用するため、メモリ資源の限られた携帯端末にそのまま適用することはできない。[4]

筆者等は、メモリ資源の限られた携帯端末上で、シューティング・ゲームを高速化する、アプリケーションの特性と構造に着目した動的コンパイル方式の検討[5]を行った。本論文では、実装と評価を述べる。

2. 特性と構造に着目した動的コンパイル方式

本方式では、アプリケーションの特性と構造を利用することにより、プロファイリング処理に用いるメモリを削減し、ユーザ操作に対する応答時間を保証する。

図 1 に示すように、クラスファイルのロード時に、静的な解析により頻繁に実行されるメソッドを判定し、ネイティブコードに変換する。

また、プロファイル対象のメソッドを絞り込むために、事前に取得した使用頻度に基づいて選択を行う。

さらに、アプリケーションの実行中には、頻繁に実行されているメソッドを判定し、ネイティブコードに変換する。

2.1 静的な解析による判定

1）フレームワークによる判定

携帯端末に搭載される Java 実行環境の仕様である MIDP[6]やｉアプリ API[7]には、ユーザインタフェースを実現するためのフレームワークが定義されており、アプリケーションに、キー、描画に対する処理を行うメソッドを実装する。

クラスファイルのロード時に、これらのフレームワークを実装するクラス及びそのメソッドを判定し、ネイティブコードに変換する。

2）呼出メソッドによる判定

携帯端末の Java アプリケーションのクラス構造は単純であり、クラスの継承関係は用いない傾向にある。また、頻繁に利用するメソッドは、クラス内で一意に特定できる private メソッドとして実現されていることが多い。キー、描画イベントを処理するメソッドから呼び出されている private メソッドを判定し、ネイティブコードに変換する。

3）コードキャッシュからの追出抑制

静的な解析により判定されたメソッドは、使用頻度が高いメソッドであると考えられる。ネイティブコードを格納するコードキャッシュから追い出されないようにすることで、プロファイリング処理に用いるメモリを削減することができる。

[†]三菱電機(株)　情報技術総合研究所
Mitsubishi Electric Corporation Information Technology R&D Center

2.2 プロファイル対象の設定

1）クラスライブラリ・メソッドの選定

位置計算、衝突判定、描画処理は、シューティング・ゲームに共通な処理であり、多くの処理時間を占める。幾つかのシューティング・ゲームについて、使用頻度の高いクラスライブラリ・メソッドを事前に調査し、プロファイル対象として設定する。

2）アプリケーション・メソッドの選定

静的な解析にて判定されたメソッド以外のアプリケーションのクラスのメソッドは、動的な解析による判定の対象となるようにプロファイル対象として設定する。

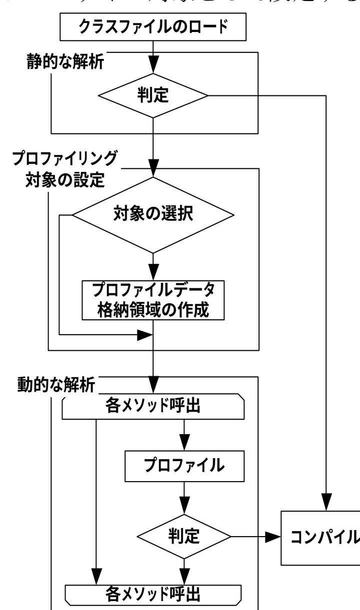

図 1 解析とプロファイル対象の設定

2.3 動的な解析による判定

1）サイズ小による判定

ユーザ操作に対する応答時間を保証するためには、操作者に認識される程の長い時間のコンパイル処理を行わないことが必要である。

コンパイル時間は、コンパイル対象であるメソッドに含まれるバイトコードのサイズに比例する。一定時間内に、ネイティブコードに変換できるように、一定のサイズ以下のメソッドを判定する。

2）呼出回数による判定

メソッドの累積実行時間を計測することで、コンパイルを行うかどうかの判定の精度を上げることができるが、前回の呼び出し時刻及び累積実行時間を保持するメモリが必要となり、また、メソッドの呼出毎に、経過時間の取得が必要である。そのため、メソッドの呼出回数を計測し、一定回数毎に経過時間を求める。

3. 実装

3.1 静的な解析による判定

アプリケーションのクラスのロードは、クラスファイルの解析及び検証処理を行うため、ある程度の処理時間が必要である。携帯端末の Java アプリケーションは、操作中に

クラスファイルのロードが発生しないように、起動直後に、アプリケーションの全クラスファイルのロードを完了させる傾向にある。
　そのため、アプリケーションの実行前に、アプリケーション内の全クラスファイルの静的な解析によるメソッドの判定と判定されたメソッドに含まれるバイトコードのネイティブコードへの変換処理は完了する。表 1 に静的な解析による判定されるメソッドとコードキャッシュからの追い出し抑制の設定を記載する。

表 1　静的な解析による判定

処理名	メソッド	コンパイル	追出抑制
キー操作	本体	○	○
	呼出(private)	＼	＼
描画処理	本体	○	○
	呼出(private)	○	○

3.2 動的な解析による判定

1）クラスライブラリ・メソッドの選定
　クラスライブラリ・メソッドの選定を行うために用いた呼出頻度データは、幾つかのシューティング・ゲームの各メソッド累積実行時間を取得し、解析を行うことで求めた。表 2 に示す様に、クラスライブラリ・メソッドの実行時間上位 49 個により、クラスライブラリ・メソッドによる実行時間の 50%以上を占めている。

表 2　機能毎の実行時間の割合

機能	メソッド個数	実行時間(%)
描画処理	24	27
イベント処理	12	10
文字列操作	10	10
数値処理	3	7
計	49	54

2）サイズ小による判定
　操作に対する応答時間に影響を与えるコンパイル時間を、1 フレームレートを下げる時間(12 フレームレート:83msec)として定義することで、1 バイトあたりのコンパイル時間(25μsec)の測定結果からコンパイル可能なサイズ小のメソッドを 3.3K バイト以下のサイズのメソッドと設定した。
3）呼出回数による判定
　メソッドが一定回数 C(c)呼び出された場合の、経過時間 T(c)に対してコンパイルを行うか判定を行う。1 秒間に 1 回以上呼ばれるメソッドを頻繁な呼出であるとして、C(c)：10 回　T(c)：10 秒とした。

4. 評価

4.1 プロファイリングに用いるメモリ量の削減

　本方式の検討[5]にて述べているように、JIT 方式では、ネイティブコードを常に実行するため、コードキャッシュのサイズを大きくする必要がある。コードキャッシュを削減するためには、プロファイリング処理を実装し、ボトルネックとなる部分のみをネイティブコードに変換する動的コンパイラ方式は有効である。
　また、本方式では、静的なデータを基にプロファイル対象を絞り込んだことで、プロファイル対象として選定されたクラスライブラリ・メソッド数は、クラスライブラリの全メソッドの約 2.5%に過ぎない。メソッド毎に必要な呼出回数や時刻を保持するプロファイル領域を大幅に削減することができた。

4.2 ユーザ操作に対する応答時間の保証

　JIT 方式を実装した Java 実行環境では、シューティング・ゲームの操作中に、一瞬画面が停止する現象が発生した。本方式を実装した Java 実行環境においては、インタプリタでの実行と比較してキー反応速度、弾丸の発射速度が高速化されている。また、マージャン・ゲームでは、対戦相手が思考する時間が短縮されている。
　本方式を用いることで、応答時間に影響を与えることなく、Java アプリケーションを高速化することができた。

4.3 速度性能への影響

　携帯電話に搭載された Java 実行環境で一般的に利用されるベンチマーク[8]を用いて評価を行った。表 3 は、JIT 方式と本方式のインタプリタ実行に対する性能向上度を示している。全般的に JIT 方式に比べて性能向上度が下回るが、最大 0.2 倍の差しかない。
1）コンパイル判定までのインタプリタ実行時間
　本方式では、処理メソッドがコンパイルされるまでのインタプリタにおける実行時間が含まれる。そのため、特定の処理のみが実行されるベンチマークプログラムでは、僅かながら性能差が検出される。
1）プロファリング処理のオーバヘッド
　「メソッド呼出」では、メソッドの呼出が頻繁に行なわれるために、他の測定値に比べてプロファイリング処理のオーバヘッドの影響を受けていると考えられる。
2)プロファイル対象外のメソッド
　「記憶領域 I/O」では、ストリーム処理のクラスライブラリのメソッドがプロファイル対象となっていないため、インタプリタ実行における実行時間が含まれている。

表 3　インタプリタに対する性能比

測定項目	JIT 方式	本方式
基本演算	4.95 倍	4.92 倍
ループ	4.95 倍	4.91 倍
メソッド呼出	2.44 倍	2.20 倍
記憶領域 I/O	1.35 倍	1.18 倍
パネル描画	1.23 倍	1.21 倍
2D 描画	1.30 倍	1.28 倍
画像描画	1.31 倍	1.25 倍

5. おわりに

　アプリケーションの特性と構造を利用する動的コンパイル方式を用いることで、ユーザ操作に対する応答時間に影響を与えることなく、ゲームを高速化し、性能向上が得られることを確認した。
　今後、プロファイリング処理のオーバヘッドの低減やプロファイル対象のメソッド数を増加させた場合の影響について検討を進める予定である。

[1]Lindholm, The Java Virtual Machine Specification, 2nd Edition, SunMicrosystems, Inc.
[2]The JIT Compiler Interface Specification, http://Java.sun.com/docs/jit_interface.html
[3]http://Java.sun.com/products/hotspot/whitepaper.html, Sun Microsystems, Inc.
[4]川本他，家電向け JIT コンパイラの構成方法とその評価，情報処理学会論文誌,Vol43. SIG8(PRO15), pp.37-48, 2002
[5]高橋他,携帯端末向け Java の高速化手法の検討,情報処理学会研究報告,Vol. 2002,MBL-22-11,Oct.2002
[6] The Mobile Information Device Profile, http://Java.sun.com /products/midp/, Sun Microsystems, Inc.
[7]http://www.nttdocomo.co.jp/p_s/imode/java
[8]http://www.seckey.net/iappli/KVMMark.html