モバイルコンピューティングと
ワイヤレス通信 22−11
（２００２．１０．１８）

携帯端末向け Java の高速化手法の検討

高橋　克英*、清原　良三*、坂本　守**

携帯電話等の携帯端末に Java が普及してきている中で、シューティング・ゲーム等の Java プログラムの高速化が期待されている。Java バイトコードをネイティブコードにコンパイルする Just-In-Time(JIT)と HotSpot の高速化方式が PC 上で利用されている。しかし、限られたメモリ資源の携帯端末では、二つの問題があるため、それらの方式を適用することはできない。1)コンパイル及びプロファイリングのためのメモリが足りない。2)アプリケーションの処理に関係なくコンパイル処理が発生し、ユーザ操作時の応答時間に悪影響を与える。本報告では、ユーザに対する応答時間を保ちながら、Java バイトコードをコンパイルする高速化方式を提案する。本方式は、クラスファイルのロード時に、アプリケーションを解析して頻繁に使用されると判定したメソッドをコンパイルし、実行時に、短時間にコンパイルできる小さな、頻繁に使用されるメソッドをコンパイルする。

Accelerating Execution of Java for Hand-held Device

Katsuhide TAKAHASHI*, Ryozo Kiyohara*, Mamoru SAKAMOTO**

The recent spread of java-enabled handheld devices (e.g. cellular phones) has led to increasing interest in improving execution speed of java programs (particularly shooting games). Just-in-time (JIT) compilation and HotSpot acceleration technology, widely used solutions on PCs, compile java byte codes into native codes at runtime. These existing solutions, however, are not suited to memory-limited devices, due to lack of memory space needed for compilation and profiling. Furthermore, because of unexpected runtime invocation of the compiler, these methods can deteriorate real time response in interactive applications such as shooting games. In this paper we propose a new acceleration technique to compile java byte-codes, while assuring quick response to user interaction. In this technique, when a class file is loaded, application programs are analyzed and then determinable frequently-executed methods are compiled, at runtime, only frequently- executed small methods are compiled in short time.

1. はじめに

インターネットの発展に伴って Java 言語が普及してきている。近年は、携帯電話を代表とする携帯端末に搭載されており、注目を集めている。携帯端末に Java を搭載することで、異なるプラットフォーム上で実行できるアプリケーションを開発することが可能となり、また、Java の検証機能がプログラムの安全な実行を保証することで、Web サーバからのプログラム配信が可能となった。携帯端末に搭載された Java には、ゲームや株価表示、案内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)[2]は、プログラム内のバイトコードをインタプリタで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、実行速度は問題にならないが、高速な描画処理を主にしたゲームでは重要な要素である。

*三菱電機(株)　情報技術総合研究所
Mitsubishi Electric Corporation
Information Technology R&D Center
**三菱電機(株)　システムＬＳＩ事業化推進センター
Mitsubishi Electric Corporation
System LSI Development Center

パーソナルコンピュータ(以下、PC)では、Java バイトコードをネイティブコードにコンパイルするネイティブコンパイラを用いた高速化方式 Just-In-Time Compiler[1] (以下、JIT)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。[6]

コンパイルしたネイティブコードの格納領域(以下、コードキャッシュ)が不要であり、メモリコストに影響を与えない Java アクセラレータを搭載するということも考えられるが、Java アクセラレータ自身のコストが影響する。

本報告では、メモリ量が少ない携帯端末上で、描画処理を主にした高速シューティング・ゲームを、ネイティブコンパイラを用いて高速化する方式を提案する。

2. コンパイル方式の検討

2.1. 通信量に対する影響

携帯端末のためのコンパイル方式として、PDA における高速化[4]のように、Java クラスファイルの一部のクラスファイルを、携帯端末にダウンロードする前にネイティブコードにコンパイルすることが考えられる。しかし、Java のプラットフォーム独立という特徴がなくなる。

また、一般に、ネイティブコードはバイトコードの数倍に増大するため、アプリケーションのダウンロードファイルのサイズが大きくなる。単一の携帯端末に対する高速化のために、ダウンロードファイルが大きくなることは、通信料を負担する携帯端末のユーザには受け入れられない。

さらに、ダウンロードファイルのサイズには制限がかけられていることが一般的であり、格納できるプログラムや画像データ量を圧迫することになる。

通信量に対する影響を考えた場合、携帯端末上でネイティブコードにコンパイルすることが必要である。

2.2. メモリ量に対する影響

ネイティブコンパイラを用いて Java アプリケーションを高速化する場合、多くのメモリ量を必要とする。

常にネイティブコードを実行する JIT を用いた Java VM では、コードキャッシュのメモリ容量が少ない場合には、コンパイルコードの追出とコンパイル処理が頻繁に行われるためにオーバヘッド時間が長くなる。

プログラムの挙動に関するデータの取得(以下、プロファイリング)を行う Hotspot を用いた Java VM では、最も頻繁に実行される部分のみをコンパイルするため、コードキャッシュに使用するメモリ量は少なくて済む。しかし、高速化のためにプロファイリングの精度を上げるために、プロファイリングに必要なメモリ量が増加する。

コードキャッシュに格納するコンパイルコードを少なくするために、メソッド内のバイトコードの一部をコンパイルする JIT の改良[7] 方式に関する研究が進められている。頻繁に実行される部分を抽出し、この方式を適用することで、更なるコンパイルコードの削減が期待できる。

メモリ量に対する影響を考えた場合、少ないメモリ量で動作するプロファイリング機能を実現する必要がある。

2.3. ユーザ応答時間に対する影響

ネイティブコンパイラを用いて Java アプリケーションを高速化する場合、アプリケーションの動作も考慮する必要がある。

高速シューティング・ゲームでは、ユーザの代理物であるキャラクタを、前後左右の移動キー及びジャンプやミサイル発射等のアクション・キーを押下することで操作し、敵や障害物、弾丸等の移動物体の位置と衝突を計算し、描画を行う。

図 1ゲーム操作画面

キー操作に対する処理、位置と衝突の計算処理や描画処理は、多くの実行時間を使用する。ゲーム操作中に、これらの処理のコンパ

イルを行った場合には、コンパイル処理時間がオーバヘッドとなり、ユーザ操作に対する応答を遅延させて、画面が一瞬停止する現象(以下、瞬停)が発生する。

ユーザ応答時間に対する影響を考えた場合、ネイティブコンパイルを用いて各処理の実行速度を上げると共に、ユーザ操作に対する応答時間を保証する必要がある。

3. 携帯端末のためのコンパイル方式の提案

本方式は、図 2に示すように、各クラスファイルについて、静的な解析によるコンパイルと動的な解析によるコンパイルによりネイティブコードを生成、実行する。その他の部分はインタプリタによりバイトコードを実行する。

1) 静的な解析による判定

各メソッドを実行する前に、静的な解析で判定できる頻繁に実行される部分をコンパイルする。

2) プロファイリング対象の選択

頻繁に実行されると期待できる部分のみをプロファイリングの対象とする。

3) 動的な解析による判定

プロファイリングデータに基づく動的な解析で判定する頻繁に実行される部分をコンパイルする。

3.1. 静的な解析による判定

1) サイズ大による判定

Java アプリケーション内で頻繁に呼び出されるコードは、性能向上を計るために、一つのメソッドに多くの処理コードを含めて、メソッド呼出のオーバヘッドを少なくしていると考えられる。

サイズの大きいメソッドは、頻繁に実行されると判断し、クラスロード時に一定のサイズ以上のメソッドをコンパイルする。

また、コンストラクタもサイズが大きくなると考えられるが、頻繁に実行されないメソッドであり、コンパイルの対象外とする。

2) フレームワークによる判定

携帯端末に搭載される Java は、MIDP[5]やiアプリ等が定義する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を提供する。アプリケーションは、フレームワークが提供するキー、描画イベントを処理するメソッドをオーバライドすることでキー操作、描画処理を実現する。そのため、キー、描画に対する処理を行うメソッドを、クラスロード時に特定することが可能である。

瞬停を起さずに、処理性能を向上させるために、クラスロード時に、キー、描画イベントを処理するメソッドをコンパイルする。

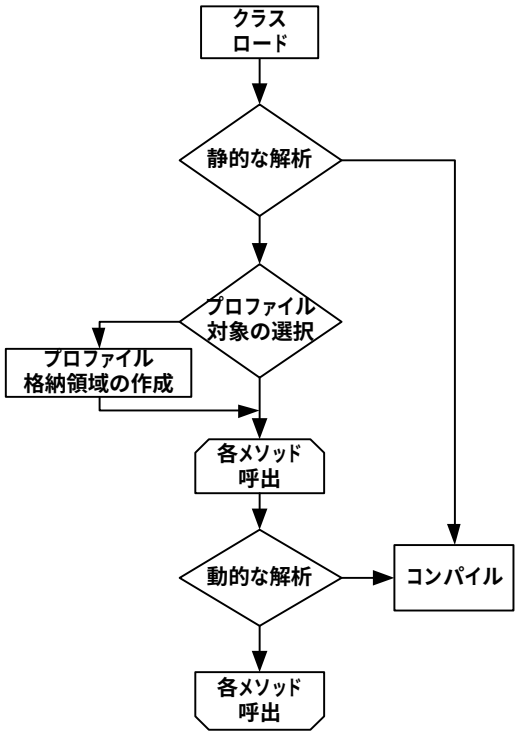

図 2 クラスファイルのコンパイル

3) 呼出メソッドによる判定

アプリケーションのクラス構造は単純であり、動的なインスタンスの生成が伴うような、実行時しかメソッドを特定できない構造を用いておらず、各インスタンスの private メソッドを呼び出していると考えられる。

瞬停を起さずに、処理性能を向上させるために、クラスロード時に、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 メソッドをコンパイルする。

4) コードキャッシュからの追い出し抑制

ゲーム操作中は、瞬停となるコンパイル処理を行わないことが必要である。

静的な解析による判定にてコンパイルを行ったメソッドは、コードキャッシュから追い出されないようにする。

3.2. プロファイリング対象の選択

1) メソッド単位のプロファイリング

コンパイル処理単位でプロファイリングを行うことも考えられるが、プロファイリングに伴うオーバヘッドとメモリ量を低減するために、メソッド単位でプロファイリングを行う。

2) クラスライブラリ・メソッドの選定

物体の移動、衝突判定と描画は、高速シューティング・ゲームに共通な処理であり、多くの実行時間を使用する。そのため、これらの処理の中で呼び出されるクラスライブラリのメソッドは、高速シューティング・ゲームに、共通に呼び出される使用頻度の高いメソッドと考えられる。

事前に、幾つかの高速シューティング・ゲームについて、使用頻度の高いクラスライブラリ・メソッドを調査し、選定を行う。選定されたクラスライブラリのメソッドのみを、プロファイリングの対象とする。

3.3. 動的な解析による判定

1) サイズ小による判定

瞬停を起さないためには、ユーザのゲーム操作中に、ユーザに認識される程、長時間のコンパイル処理を発生させないことが必要である。一般的にコードの最適化を行うコンパイラはコンパイル処理時間が長い。そのため、コードの最適化を行わないコンパイラを用いる。

最適化を行わないコンパイラは、バイトコードとネイティブコードを対応付けてコンパイル処理を行う。対応付けを行うコンパイラのコンパイル時間は、コンパイル対象である各メソッドに含まれるバイトコードのサイズに比例する。

瞬停を発生させないコンパイル時間となるように、一定のサイズ以下のメソッドをコンパイルする。

2) 呼出回数による判定

プロファイリングの精度を上げるために、各メソッドの累積実行時間を計測することが考えられる。しかし、累積実行時間を取得するためには、前回呼び出された時刻及び累積実行時間を保持する必要があり、多くのメモリを使用する。そのため、呼出回数を計測するプロファイリングを行うこととする。

高速シューティング・ゲームでは、描画イベントを処理する paint メソッドが、頻繁に呼び出されており、コンパイル判定に用いる呼出回数の基準値として利用することができる。paint メソッドの呼出回数又はその比例値よりも、呼出回数の多いメソッドをコンパイルする。

4. コンパイル方式の評価

高速シューティング・ゲームの動作データを用いて、

1)クラスライブラリ・メソッドの選定

2)コンパイル判定

の評価を行った。

4.1. クラスライブラリ・メソッドの選定

4.1.1. メソッドの実行時間

クラスライブラリ・メソッドの選定が有効であることを確認するために、Java VM のインタプリタに情報取得のための機能を実装し、7 個の高速シューティング・ゲームの各メソッドの実行時間を取得した。

図 3は、呼び出されたクラスライブラリのメソッド(ネイティブメソッドを除く)とアプリケーションのメソッドのパッケージ毎の実行時間の割合を示している。3/4 程度の実行時間をクラスライブラリのメソッドが占めていることが分かる。性能向上に関して、クラスライブラリのメソッドをコンパイル対象とすることは有効である。

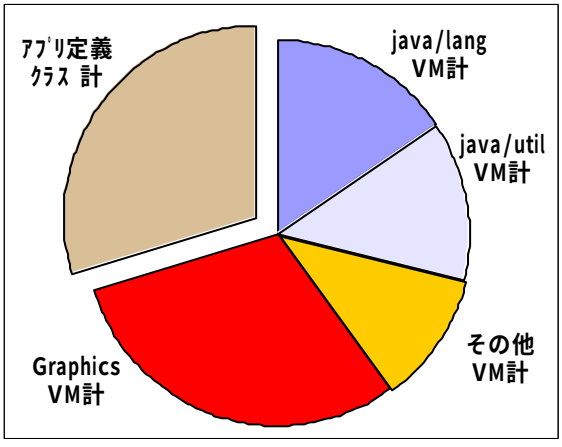

図 3　高速シューティングのメソッド実行時間分布

4.1.2. 使用頻度の高いメソッドの処理

表 1に示すように、アプリケーションが使用しているクラスライブラリのメソッドの実行時間上位 49 個により、クラスライブラリのメソッドによる実行時間の 50%を占めていた。

表 1　機能毎の実行時間の割合

機能	個数	実行時間(%)
描画処理	24	26.90%
イベント処理	12	9.66%
文字列操作	10	9.60%
数値処理	3	7.08%
計	49	53.24%

1) 描画処理

キャラクタ等の移動物体を描画するために、画像イメージ、線や多角形の描画処理が行われている。また、ディスプレイや描画範囲のサイズを取得するメソッドも頻繁に呼び出されている。アプリケーションが携帯端末との非依存性を保つために、常に値を取得して利用するためと考えられる。

2) イベント処理

キー、描画イベントの取得及びイベント・キューの操作処理を行っている。ユーザインタフェースを実現するために、イベント配送、描画処理等を行う複数の Java スレッドが実行されており、1 割程度の時間がイベント処理に使用されている。

3) 文字列操作

整数から文字列に変換する処理が頻繁に呼び出されている。点数や経過時間の表示に使用していると考えられる。

4) 数値処理

絶対値とランダム値の取得が行われている。移動物体の位置計算に用いていると考えられる。

4.1.3. 選定による効果

クラスライブラリ全体のメソッド数は 2003 個であり、50%の実行時間を占める 49 個のメソッドは、クラスライブラリ全体の 2.5%に過ぎない。また、上位 49 個のメソッドは、高速シューティング・ゲームに特有な処理に用いられている。

それらのクラスライブラリのメソッドのみをプロファイリング対象として選択することで、プロファイリングによる処理及びメモリ使用量を大幅に低減することができる。

4.2. コンパイル判定

静的な解析による判定、動的な解析によるコンパイル判定方法を評価するために、サイズ大による判定に対する特性が異なる 3 つの高速シューティング・ゲームについて、コンパイル判定シミュレーションを行った。

ゲーム A：サイズ大による判定よりもかなり大きなメソッドが存在する。

ゲーム B：サイズ大による判定程度の大きさのメソッドが存在する。

ゲーム C：サイズ大による判定によりコンパイル対象となるメソッドが存在しない。

シミュレーションでは、情報取得を行う Java VM のインタプリタを用いて取得したアプリケーション内クラスのメソッドのサイズ、呼出回数、実行時間の割合を用いる。

4.2.1. シミュレーション条件の設定

1) サイズ大による判定

シミュレーションに用いた Java アプリケーションのダウンロードファイルは、10K バイトである。ダウンロードファイルは jar 形式であり、圧縮されている。また、ゲームに使用する画像データも含まれている。

圧縮と含まれる画像データが相殺されると仮定し、アプリケーションに含まれる全バイトコードを 10K バイトとする。その 2 割である 2K バイト以上のサイズのメソッドをコンパイルする。

2) フレームワークと呼出メソッドによる判定

描画イベントを処理メソッド paint()とキーイベントを処理するメソッド keyEvent()及びそのメソッドから呼び出される private メソッドをコンパイルする。

3) サイズ小による判定

瞬停を発生させる時間を、1 フレームレート分の時間とする。動画表示と同様な 24 フレームレートが実現されると仮定し、1sec/24 = 42msec よりも少ない値である 10msec として設定する。

コンパイル処理性能を 50μ秒/バイトと仮定し、10msec で行えるコンパイル可能なバイトコードサイズは、200 バイトである。200 バイト以下のメソッドをコンパイルする。

4) **呼出回数による判定**

paintメソッドよりも呼出回数が多いものをコンパイルする。24 フレームレートであれば、1 秒間に 24 回程度の頻度で呼ばれるメソッドをコンパイルすることになる。

4.2.2. 総合的な評価

表 2、3、4 は、各判定方法のコンパイル対象メソッドのサイズと実行時間の占める割合を示している。

・ゲーム A

サイズの大きな一つのメソッドと描画イベントを処理するメソッドである paint が 90%の処理時間を占めており、また、全体に占める値は 66%に抑えている。静的な解析による判定が有効である。

・ゲーム B

本方式によりコンパイル対象となる実行時間は全体の 57%であり、サイズは全体の 32%に抑えている。

動的な解析によりコンパイル対象となるサイズは全体の 5%、実行時間は全体の 40%である。動的な解析によるコンパイル判定が有効に機能していることが分かる。

・ゲーム C

本方式によりコンパイル対象となる実行時間は全体の 61%であり、サイズは全体の 41%に抑えている。

動的な解析によりコンパイル対象となったサイズは全体の 5%であるが、実行時間は全体の 40%である。動的な解析によるコンパイル判定が有効に機能していることが分かる。

4.2.3. 各判定方法の評価

表 5、6、7 は、ゲーム A、B、C のそれぞれのシミュレーションの結果である。網掛した列は、コンパイラ対象となったメソッドを示している。

4.2.3.1 サイズ大による判定

1) **判定結果**

・ゲーム A

コンパイル対象となった大きなメソッド a()が存在し、移動物体等の計算処理の全般を行っていると考えられる。全体の実行時間の 54%を占めている。

・ゲーム B

アプリケーション内メソッドのバイトコードの 18%を占めるメソッド loop が存在する。呼出回数は多いが、実行時間では、8%を占めているに過ぎない。ゲーム操作に関連しないアプリケーションの開始、終了処理、データ保存等の処理コードも含まれているものと考えられる。

・ゲーム C

コンストラクタ以外に、サイズが 2K バイト以上のメソッドは存在しない。

2) **評価**

ゲーム B には、有効性がない。メソッド数が少なく、非常に大きなメソッドが存在するアプリケーションに対してのみ有効であると考えられる。適用は、効果のあるアプリケーションに限定する必要がある。

4.2.3.2 フレームワークによる判定

1) **判定結果**

・ゲーム A

paint メソッドは、サイズの大きなメソッドとして構成されており、全体の実行時間の 37%を占めている。keyEvent メソッドは、全体の実行時間の 0.74%しか占めていない。しかし、サイズは動的な解析によるコンパイル対象外の大きさである。

・ゲーム B

paint メソッドは、サイズは 11%、実行時間は 9%程度を占めている。keyEvent メソッドは、実行時間では 1%を占めているに過ぎない。サイズは動的な解析によるコンパイルの対象とはならない大きさである。

・ゲーム C

paint メソッドは、サイズは 7%、実行時間は 7%を占めている。keyEvent メソッドでは、0.04%を占めているに過ぎない。サイズは、動的な解析によるコンパイルの対象外の大きさである。

2) **評価**

ゲーム B、C の描画イベントを処理するメソッドの全体に占める実行時間の割合とサイズの割合は、同程度であり、性能向上に対するコードキャッシュの使用効率は高くない。しかし、瞬停を起さずに、実行時間の 7%以上を占める処理の高速化を行うことがで

きており、描画イベントを処理するメソッドをコンパイルすることは有効である

キーイベント処理するメソッドでは、実行時間の割合が少ないために、アプリケーション全体から見た性能向上の観点からは意味がない。しかし、キー応答時間を向上させるために、有効であるとも考えられる。

4.2.3.3 呼出メソッドによる判定

1) **判定結果**

・ゲーム A

paint 及び keyEvent メソッドから呼び出されている private メソッドはない。

・ゲーム B

paint メソッドは、private メソッドを呼び出さない。KeyEvent メソッドから呼び出されている 2 個の private メソッドは、呼出回数が少なく、ゲーム操作に関する処理ではないと考えられる。

・ゲーム C

paint メソッドは、5 個の private メソッドを呼び出している。メソッド 5 個の内の 1 個は、全体 9%の実行時間を占めており、呼出メソッドによる判定の効果を示している。しかし、その他の 5 個メソッドは、各 1%以下の実行時間を占めているに過ぎない。

keyEvent メソッドは、private メソッドを呼び出さない。

2) **評価**

ゲーム操作と関係しない処理や実行時間の少ないメソッドもコンパイル対象として判定しており、有効に機能していない。

コードキャッシュの効率的な利用を考えた場合、呼出メソッドの判定でコンパイルしたネイティブコードは、コードキャッシュに貼り付けず、呼出頻度の少ないメソッドを追い出し、動的な解析にコードキャッシュの領域を明け渡すことが必要である。

4.2.3.4 サイズ小、呼出回数による判定

・ゲーム A

アプリケーションはサイズの大きなメソッドから構成されており、コンパイル可能と判定できるメソッドはなく、paint メソッドよりも多く呼ばれたメソッドは、静的解析によるコンパイル対象である a()だけである。

・ゲーム B

paint メソッドの呼出回数以上に呼び出されたメソッド 8 個の内の 5 個がコンパイル対象として判定されている。

・ゲーム C

動的な解析によるコンパイル判定では、paint メソッド以上の呼出が行われた 8 個内の 4 個のメソッドがコンパイル対象として判定されている。

2) **評価**

コンパイルが行えるサイズの小さいメソッドでも、実行時間の割合の高いメソッドが存在し、性能向上に効果がある。

表 2 ゲーム A コンパイル対象の実行時間とサイズ

コンパイル判定方法			サイズ割合(%)	実行時間割合(%)
静的解析	サイズ大		44	54
	フレームワーク	paint	17	37
		keyEvent	5	1
	呼出メソッド	paint	0	0
		keyEvent	0	0
	計		66	91
動的解析	サイズ小,呼出回数		0	0
合計			66	91

表 3 ゲーム C コンパイル対象の実行時間とサイズ

コンパイル判定方法			サイズ割合(%)	実行時間割合(%)
静的解析	サイズ大		0	0
	フレームワーク	paint	7	7
		keyEvent	5	0
	呼出メソッド	paint	15	10
		keyEvent	0	0
	計		27	17
動的解析	サイズ小,呼出回数		5	40
合計			32	57

表 4 ゲーム B コンパイル対象の実行時間とサイズ

コンパイル判定方法			サイズ割合(%)	実行時間割合(%)
静的解析	サイズ大		18	8
	フレームワーク	paint	10	9
		keyEvent	7	1
	呼出メソッド	paint	0	0
		keyEvent	1	3
	計		36	21
動的解析	サイズ小,呼出回数		5	40
合計			41	61

5. おわりに

本報告では、携帯端末の Java で動作する高速シューティング・ゲームに対して、アプリケーションの特性と構造を利用して、プロファイリングのメモリ量を低減し、瞬停が発生しないコンパイル方式を提案し、判定シミュレーションによる評価を行った。

今後、下記の課題を解決し、コンパイラを搭載して評価を行う予定である。

1)サイズ大による判定の精度の向上

2)キーイベント処理するメソッドのコンパイルの必要性の検証

3)呼出メソッドによる判定にて引き起こされるメモリ使用効率の低下の防止

参考文献

[1].The JIT Compiler Interface Specification , http://java.sun.com/docs/jit_interface.html,Sun Microsystems, Inc.

[2]Lindholm, T. and Yellin, The Java Virtual Machine Specification, 2nd Edition,SunMicrosystems, Inc.

[3]The JAVA HOTSPOT Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, Sun Microsystems, Inc.

[4]有馬 啓,並木美太郎, PDA における Java 実行の高速化の一方式, 情報処理学会論文誌,Vol 42. No 6,pp.1535-1544(2001)

[5] The Mobile Information Device Profile, http://java.sun.com / products/midp/, Sun Microsystems, Inc.

[6]川本 琢二,春名 修介,金丸 智一, 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,Vol 43. No SIG 8 (PRO15),pp.37 -48 (Sep 2002)

[7]Manjuath, G. and Krishnan, V.:A samall Hybrid JIT for Embedded Systems, ACM SIGPLAN Notices, Vol.35, No.4, pp.44-49(2000)

表 5 ゲーム A 各メソッドの実行時間とサイズ

メソッド	サイズ (byte)	呼出回数	実行時間割合(%)
G$a::a()V	3818	1647	53.68
G$a::paint(G;)V	1486	345	36.81
G$a::b(I)V	988	12	3.80
G$a::a([BI)I	312	10	1.60
G$a::c(I)I	518	5	1.55
G$a::run()V	156	1	1.31
G$a::keyEvent(II)V	416	31	0.74
G::start()V	34	3	0.32
G$a::a(I)V	256	10	0.17
G$a::<init>(LG;)V	731	1	0.02
計	8715	2065	100

表 6 ゲーム B 各メソッドの実行時間とサイズ

メソッド	サイズ (byte)	呼出回数	実行時間割合(%)
M::moveI(II)V	1312	276	13.72
M::paintI(L/G;II)V	69	322	13.81
M::paintB(L/G;)V	147	79	12.15
M::hitChk()V	952	71	11.98
M::paint(L/G;)V	1257	69	9.23
M::loop()V	2116	452	7.69
M::hitChkSh(IIII)Z	1224	6	6.77
M::rayHitChk(I)V	190	93	6.39
M::run()V	423	1	2.76
M::paintR(L/G;I)V	138	104	5.50
M::paintS(L/G;)V	109	130	2.24
M::goTitle()V	36	4	2.63
M::keyEvent(II)V	844	50	0.96
M::drawBe(L/G;I)V	170	11	1.65
M::fireR(I)V	250	11	0.82
M::drawSh(L/G;II)V	922	7	0.62
M::moveIBase()V	128	7	0.51
M::drawS(L/G;II)V	166	6	0.18
M::drawIe(L/G;II)V	35	4	0.22
I::start()V	952	1	0.05
M::drawR(L/G;)V	52	4	0.09
M::writeGData()V	56	1	0.01
M::<init>()V	530	1	0.01
計	12078	1710	100

表 7 ゲーム C 各メソッドの実行時間とサイズ

メソッド	サイズ (byte)	呼出回数	実行時間割合(%)
S::sort(I)V	332	115	18.91
S::pmiss()V	76	319	17.75
S::obj_bin()V	607	320	15.91
S::pDraw(L/G;)V	127	666	11.29
S::d_block(L/G;)V	535	310	8.82
S::chg_key(LS;LS;)V	13	310	7.40
S::paint(L/G;)V	555	237	6.90
S::shot()V	198	305	3.77
S::run()V	182	1	3.45
S::cmiss(I)Z	276	97	1.64
S::in_key()V	334	304	1.21
S::cshot(II)Z	544	26	0.95
S::dhot(L/G;I)V	241	6	0.87
S::getRndInt(I)I	12	473	0.65
S::clear()V	43	1	0.16
S::MkState(L/G;)V	378	58	0.12
S::dScore(L/G;LS;I)V	17	56	0.09
S::keyEvent(II)V	366	21	0.04
S::mAction(L/M;II)V	56	10	0.04
S::createI(ILS;)L/I;	38	4	0.02
S::terminate()V	26	1	0.01
S::score_save()V	111	1	0.01
S::ld_chk()Z	32	3	0.00
S::<init>()V	2596	2	0.00
S::score_load()V	84	1	0.00
S::init()V	37	1	0.00
計	7816	3648	100