携帯端末向けのJava高速化手法とその評価

高橋克英[†] 清原良三[†]

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求められている．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た高速化方式である JIT 方式や HotSpot 手法が利用されている．しかし，限られたメモリ資源の携帯端末にネイティブコンパイラを適用する際には，2 つの問題がある．1) コンパイル処理およびプロファイリングのためのメモリが足りない．2) コンパイル処理時間がユーザ操作の応答時間に悪影響を与える．本論文では，ユーザ操作の応答時間を保ちながらネイティブコードに変換する方式を提案する．本方式では，クラスロード時にアプリケーションを解析し，頻繁に使用されると判定したメソッドをネイティブコードに変換する．さらに，本方式では，短時間に変換可能なメソッドをプロファイリング対象として設定し，頻繁に利用されるメソッドをネイティブコードに変換する．本方式を用いることで，コンパイラが利用できるメモリが少ない場合でも，ユーザ操作に対する応答時間を保ち，JIT 方式と同等の性能が提供できる．

Accelerating Technique of Java for Hand-held Devices and Its Evaluation

Katsuhide Takahashi[†] and Ryozo Kiyohara[†]

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, 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 bytecodes, 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. Our evaluation results show the compiler memory size is small, and that JIT compilation is almost same a performance.

1. はじめに

インターネットの発展にともなって，ポータビリティ，セキュリティを提供する Java 言語[1] が普及し，近年は，携帯電話を代表とする多くの携帯端末に Java 仮想マシン[2]（以下，Java VM）が搭載されている．携帯端末に Java VM が搭載されることで，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，プログラムの安全な実行とプログラム配信が可能となった．携帯端末には，ゲームや株価表示，案内等の様々なアプリケーションが提供されている．

† 三菱電機株式会社情報技術総合研究所
Information Technology R&D Center, Mitsubishi Electric Corporation

携帯端末向けの Java VM は，アーキテクチャ非依存コードであるバイトコードを実行するために，インタプリタを用いる実装[3),4)] が一般的であった．しかし，インタプリタは，バイトコードを逐次解釈して実行するために実行速度が遅いという問題がある．携帯電話の Java アプリケーションでは，実行速度を重視するゲームが多く配信されており，特に，**図 1** のような高速シューティング・ゲームでは実行速度の遅さは重要な問題である．

インタプリタの実行速度の問題は，携帯端末に限らず Java VM の一般的な課題である．実行速度を向上させるために，バイトコードを CPU が直接実行できる命令（以下，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た高速化方式を搭載する取り組み[5)〜7)] が行われてきた．しかし，ネイティブコ

ンパイラを用いた高速化方式では，変換したコードを保持するメモリが必要となるため，メモリ資源の限られた携帯端末に搭載する際にはそのままでは適用できず，使用するメモリ量を削減しなければならない．ネイティブコンパイラが使用するメモリ量を削減するためには，ネイティブコードに変換するバイトコードを頻繁に実行されるホットスポットに限定し，その他のバイトコードはインタプリタを用いて実行する，バイトコードを選択して変換する方式[8] が有効である．

ネイティブコンパイラを利用しない高速化方式として，バイトコードを直接実行する Java アクセラレータの搭載があげられる．Java アクセラレータは，ネイティブコードを格納するメモリ領域を必要としない．しかし，Java 専用ハードウェアである Java アクセラレータのコストが製品コストの増大につながる．メモリ資源は，Java VM の実行速度の向上以外の用途にも利用することが可能であり，製品コストの低減が必要な携帯端末では，メモリ資源への投資を選択し，ネイティブコンパイラを利用すると考えられる．

また，ネイティブコンパイラを用いた高速化方式を，実行速度を重視するゲームに適用する場合には，ネイティブコンパイラの変換処理時間を考慮する必要がある．特に，高速シューティング・ゲームでは，ユーザ操作に対する応答時間の制約が厳しく，ユーザ操作に対する応答を遅延させてはならない．

本論文では，メモリ量が少ない携帯端末上で，高速シューティング・ゲームを代表とするユーザ操作がともなう Java アプリケーションを，ネイティブコードに変換する時間を考慮しながらネイティブコンパイラを用いて高速化する方式を提案し，実装およびその評価について述べる．

以下に，本論文の章構成を示す．2 章では，携帯端末の特性を明らかにし，ネイティブコンパイラを用いた高速化方式に求められる機能を定義する．3 章では，関連研究について述べる．4 章では，提案方式の内容を説明する．5 章では，動作データの分析による判定シミュレーションと試作実装を用いた評価結果を記述する．6 章で本研究のまとめと今後の課題について述べる．

図 1 ゲーム操作画面
Fig. 1 Playing screen in a game.

2. 携帯端末の特性と機能

本章では，ネイティブコンパイラを適用する携帯端末の特性を記述し，ネイティブコンパイラを用いる高速化方式に求められる機能を記載する．

2.1 通信量に対する影響

ネイティブコンパイラを用いた高速化方式として，PDA における高速化[9] のように，Java クラスファイルの全体または一部を，携帯端末にダウンロードする前にネイティブコードに変換する方式がある．しかし，変換したネイティブコードは変換前のバイトコードの数倍に増加するため，Java アプリケーションをダウンロードする際の通信量が増大する．**表 1** に，Java クラスライブラリのパッケージに含まれるバイトコードをネイティブコードに変換した際の増大率を示す．表 1 は，バイトコードとネイティブコードの非圧縮時のサイズを示しており，変換したネイティブコードが約 7 倍から 8 倍程度に増加していることが分かる．

通信量の増大は，ユーザ，通信事業者，プログラム開発者のそれぞれに問題を発生させる．携帯端末のユーザは，課金される通信サービスを用いて Java アプリケーションをダウンロードして利用する．ダウンロード前にネイティブコードに変換した場合，ダウンロードファイルのサイズが大きくなり，ユーザの通信料の負担が増大する．通信サービスを提供する通信事業者は，通信量の増大の対策として通信網の整備という新たな投資が必要となる．

プログラム開発者は，通信事業者の通信網の設備投資を抑える処置により影響を受けることになる．通信事業者は，通信量を抑制するために，ダウンロードファイルのサイズに制限をかけることが一般的である．そのため，プログラム開発者は，クラス名やメソッド名等のシンボル情報を短縮するツール等を用いて，ダ

表 1 コンパイルコードのサイズ
Table 1 Size of the compiled code.

パッケージ	バイトコード (KB)	ネイティブコード (KB)	増加率
java.io	3.9	34.8	8.9 倍
java.lang	1.5	11.8	7.8 倍
java.util	3.0	26.6	8.5 倍

ウンロードファイルに含まれるロジック，データ容量を確保しなければならない状況にある．実際に，後述の判定シミュレーションに用いた 3 つのゲームにおいても，**表 8**，**表 9**，**表 10** に示すようにクラス名，メソッド名の短縮が行われている．ネイティブコードの変換による通信量の増大は，ダウンロードファイルに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り，プログラム開発者は，プログラム，画像データの削減に迫られる．Java アプリケーションを有料コンテンツとして提供するコンテンツプロバイダにとっては，魅力あるコンテンツを提供できなくなる恐れもある．

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変換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た高速化方式では，変換したコードを保持するメモリ（以下，コードキャッシュ）を使用する．つねにネイティブコードを実行する JIT（Just-In-Time）方式の Java VM では，コードキャッシュのメモリ容量が不足した場合に，ネイティブコードの追い出しとバイトコードの変換処理が頻繁に発生し，ネイティブコードに変換する処理時間が増加して性能向上が得られなくなる．

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）を行うことで，頻繁に実行されるホットスポットを検出，ネイティブコードに変換して実行し，その他のバイトコードはインタプリタを用いて実行する HotSpot 手法では，ネイティブコードを格納するメモリ量が少ない場合でも，ホットスポットを選択してネイティブコードに変換するため，変換処理時間が増加せず，効率的に性能を向上させることができる．

しかし，携帯端末のようなメモリ資源が限られた環境では，変換したコードを保持するメモリ容量が非常に限られていることが考えられ，ホットスポットを検出するプロファイリングの精度を上げることが必要である．

プロファイリングの精度を上げるためには，プロファイリングに必要なメモリ量，処理時間が増大する．プロファイリングの方法には，1) 計測コード方式，2) サンプリング方式の 2 つの方式[10] がある．計測コード方式では，ホットスポットを検出するために，メソッドの呼び出しコードにカウンタ値を増加させる等の計測コードを追加する．サンプリング方式では，実行中のスレッドのスタックからメソッド等の情報を取得するサンプリングコードを周期的に実行する．

計測コード方式は，メソッドの呼び出し時に計測用コードを実行させるために，サンプリング方式に比べてオーバヘッドが大きい．また，プログラム内のメソッド数が多いとプロファイリングに用いるメモリ容量が増大する．サンプリング方式は，周期的にプロファイリング処理を実行するため，各メソッドの呼び出しごとにプロファイリング処理を実行する計測コード方式に比べてオーバヘッドが少ない．しかし，サンプリング方式は，一定周期ごとにプロファイルが行われるため，ホットスポットの検出の精度が下がる．

メモリ量に対する影響を考えた場合，ホットスポットの検出精度が高く，少ないメモリ量で動作するオーバヘッドの少ないプロファイリング機能を実現することが必要である．

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・ゲーム等のアプリケーションは，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およ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を計算し，各移動物体の衝突を検出し，移動物体の描画処理を行う．

そのため，キー操作に対する処理，位置計算と衝突検出の処理や描画処理は，多くの実行時間を使用し，ホットスポットとして検出される．ゲーム操作中に，これらの処理を実装したバイトコードを変換する処理を行った場合には，ネイティブコードに変換する処理時間が，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問題（以下，瞬停）が発生する．

また，プロファイリング方法としてサンプルリング方式を用いた場合，ユーザ操作に対する応答時間が一定時間とはならない．サンプリング方式では，周期的に各スレッドのメソッド等を調査するために，ユーザ操作を処理するコードとは無関係にサンプリングコードが実行されるからである．

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ラを用いて各処理の実行速度を上げるとともに，プロファイリング，ネイティブコードに変換する処理の影響を削減し，ユーザ操作に対する影響を最小限にとどめる機能を実現することが必要である．

3. 関連研究

携帯端末向けのネイティブコンパイラを用いた高速化方式の研究として，ダウンロードする前にネイティブコードに変換する方式[9] がある．しかし，携帯端末の特性から通信量が増加することは容認できず，携

帯端末上でネイティブコードに変換することが必要である．

パーソナルコンピュータに代表されるコンピュータ分野のネイティブコンパイラを用いた高速化方式の研究[5)~7)]では，高い性能を得ることに主眼を置いている．そのため，メモリ資源の制約が厳しい携帯端末に対して，そのまま利用することはできない．

性能向上を犠牲にしてメモリ消費量の低減，変換処理の時間短縮を優先するネイティブコードの変換方式の研究[11)~13)]が行われている．これらの研究では，頻繁に実行されるホットスポットを判定してネイティブコードに変換する高速化方式については議論されていない．

検出したホットスポットをネイティブコードに変換し，コードキャッシュのメモリ量を低減する方式が，携帯端末向けの Java VM[8)] に採用されている．しかし，ユーザ操作に対する応答時間への影響を最小限に抑えるための検討が行われていない．高速シューティング・ゲームのようなアプリケーションでは，ネイティブコンパイラの変換処理が動作した場合の瞬停の発生は致命的な問題であり，検討が必要である．

また，プロファイリングに必要なメモリ量，オーバヘッドを低減するサンプリング方式を採用した研究[10),14)]が行われている．しかし，サンプリング方式では，プロファイリングが一定した処理時間とならず，ユーザ操作に対する応答時間が一定とならない．

本論文では，ユーザ操作がともなうアプリケーション，特に，高速シューティング・ゲームを意識して，実行中のプロファイリング処理時間，ネイティブコードに変換する処理時間の制約が厳しい携帯端末におけるネイティブコンパイラを用いた高速化方式を提案する．

4. 提案方式

メモリ資源の少ない携帯端末では，多くのネイティブコードを保持することはできない．そのため，頻繁に実行されるホットスポットをネイティブコードに変換する方式を採用する．本論文で提案するネイティブコンパイラを用いる高速化方式は，**図 2** に示すように，静的な解析による変換処理と動的な解析による変換処理を行うことで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実装を用いてバイトコードを実行する．

4.1 静的な解析

ユーザ操作がともなう Java アプリケーションでは，瞬停が発生する恐れがあるために，アプリケーション実行中のネイティブコードに変換する処理時間に対して厳しい制約を設けなければならない．そのため，アプリケーションの構造を静的に解析してホットスポットを判定し，アプリケーションの起動前のクラスロード時にネイティブコードに変換する手法を提案する．

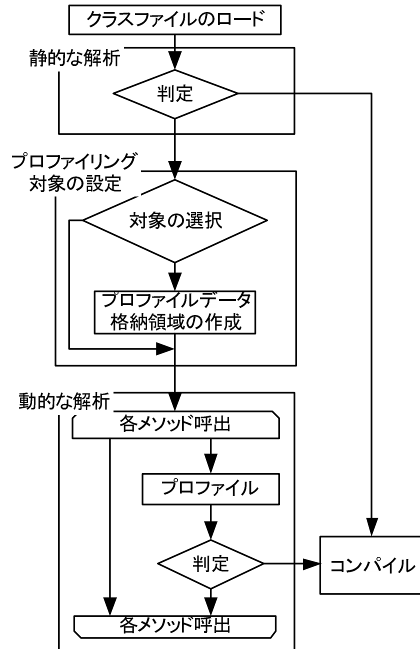

図 2 解析とプロファイリング
Fig. 2 Analyzing and profiling.

以下に，静的な解析に用いる判定方法を記載し，**表 2** に，ネイティブコードに変換するメソッドを示す．

4.1.1 メソッドサイズによる判定

頻繁に実行される部分として開発者が判断したコードは，最適化が行われる．頻繁に呼び出されるメソッドは，呼び出し時のオーバヘッドを削減するために，インライン展開が行われ，1 つのメソッドが肥大化する傾向がある．

サイズの大きいメソッドは，頻繁に実行される部分として判断し，クラスロード時に一定のサイズ以上のメソッドをネイティブコードに変換する．データ初期化を行うコンストラクタもサイズが大きくなる傾向にあるため，この条件に合致する．しかし，インスタンス生成にともなうデータの初期化は頻繁に実行されないため，この処理の例外としてネイティブコードに変換しない．

4.1.2 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[15)] や i アプリ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．Java アプリケーションは，フレームワークに含まれるキー，描画イベントを

表 2　静的な解析によるネイティブコード変換
Table 2　Compile methods in static analyzing.

実装箇所	メソッド形態	メソッドサイズ	フレームワーク（キーイベント，描画）	呼び出しメソッド	アクセス修飾子	ネイティブコードに変換
クラスライブラリ						×
アプリケーション	コンストラクタ					×
	メンバ	大きなメソッド（一定サイズ以上）				○
		それ以外	イベント処理メソッド			○
			それ以外	イベント処理メソッドから呼び出される	private	○
					それ以外	×
				それ以外		×

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．

キー，描画に対する処理は頻繁に実行される部分であり，クラスロード時に，アプリケーション内のキー，描画イベントを処理するメソッドをネイティブコードに変換する．

4.1.3　呼び出しメソッドによる判定

携帯端末に搭載される Java VM は，CLDC 仕様[16),17)] に準拠しており，ユーザ定義のクラスロード機能を提供していないため，携帯端末上に組み込まれたシステムクラス以外のメソッドは，すべてダウンロードファイルのユーザ定義クラスに含まれる．また，Java アプリケーションのサイズ制限が厳しいことから，ユーザ定義クラスでは，クラス継承やインタフェース定義を利用するような実行時にしか特定できない動的なメソッド呼び出しを用いておらず，各インスタンス内のメソッドを呼び出していると考えられる．また，プログラム開発者は，メソッド呼び出しの効率化を期待して頻繁に呼び出されるメソッドを private とする傾向がある．

以上のことから，頻繁に実行される部分から呼び出される private メソッドは頻繁に呼び出されるメソッドであると考えられる．本方式では，クラスロード時に，キー，描画イベント処理メソッドから呼び出される private メソッドをネイティブコードに変換する．

4.1.4　コードキャッシュの追い出し制御

本方式では，静的な解析で変換したコードをコードキャッシュから追い出さないように制御する．コードキャッシュの追い出し制御を行うことで，プロファイリングを用いた動的な解析による判定を回避し，アプリケーション動作中のネイティブコンパイラの実行を抑制することができる．

4.2　動的な解析

周期的にプロファイリングを行うサンプリング方式は，オーバヘッドが少ないという利点がある．しかし，

表 3　動的な解析に用いるプロファイリング対象
Table 3　Profiling methods in dynamic analyzing.

実装箇所	メソッドの事前選定	メソッドサイズ	プロファイリング対象
クラスライブラリ	選定内	小さなメソッド（一定サイズ以下）	○
		それ以外	×
	選定外	-	×
アプリケーション	-	小さなメソッド（一定サイズ以下）	○
		それ以外	×

サンプリング方式では，ホットスポットの検出精度が悪く，プロファイリングが一定した処理時間とならないという問題点がある．本方式では，ユーザ操作に対する応答時間の影響を考慮して，オーバヘッドが一定している計測コード方式を採用する．

また，プロファイリングに必要なメモリ量を低減するために，頻繁に実行される可能性が高い部分のみをプロファイリングの対象として選択する．

以下に，動的な解析に用いるプロファイリング対象の選定方法，プロファイリングの取得データについて記載し，**表 3** に，動的な解析に用いるプロファイリング対象のメソッドを示す．

4.2.1　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．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，いく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，プロファイリングの対象メソッドの選定を行う．

4.2.2　メソッドサイズによる選定

瞬停を発生させないためには，ゲーム操作中に，ユー

ザに認識される変換処理を行わないことが必要である．ネイティブコンパイラの変換処理時間を推定するために，最適化処理をともなわずバイトコードをネイティブコードに対応付けて変換するのみの単純なネイティブコンパイラを採用する．本方式では，瞬停を発生させない変換処理時間となるように，一定のサイズ以下のメソッドをプロファイリング対象として選択する．

4.2.3 呼び出し回数による判定

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，メソッドが前回呼び出された時刻および累積実行時間を保持する必要があり，多くのメモリ領域を使用する．そのため，一定時間内の呼び出し回数（メソッドの呼び出し頻度）を計測するプロファイリングを行う．

5. 評　　価

5.1 クラスライブラリ・メソッドの選定の有効性

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[3] のインタプリタにメソッドの実行時間を取得する機能を実装し，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，アプリケーションのメソッドとクラスライブラリのメソッドのパッケージごとの実行時間の割合を示している．3/4 程度の実行時間がクラスライブラリのメソッドで占められていることが分かる．クラスライブラリのメソッドをネイティブコードに変換するプロファイリング対象として設定することは，性能向上に有効である．

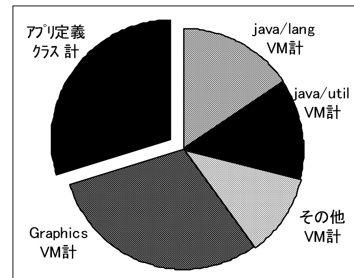

図 3 高速シューティングのメソッド実行時間分布
Fig. 3 Execution time of methods in shooting game.

表 4 機能ごとの実行時間の割合
Table 4 Execution time of functions.

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

5.1.1 使用頻度の高いメソッドの処理

表 4 に示すように，実行時間が長い上位 49 個のメソッドで，クラスライブラリに含まれるメソッドの実行時間の 50%を占めていた．以下に，該当するメソッドを処理内容を示す．

（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への非依存性を保つために，つね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およびイベント・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
整数を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用いていると考えられる．

（4） 数値処理
絶対値と乱数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．

5.1.2 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，53%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%にすぎない．また，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およびメモリ使用量を大幅に低減させることができる．

5.2 静的，動的な解析による判定の有効性

静的な解析による判定，動的な解析による判定の有効性を確認するために，3 つの高速シューティング・ゲームを用いて判定シミュレーションを行った．用いたゲーム A，B，C は，図 1 にようにキー操作にともない，敵，キャラクタ，弾丸等の移動体の位置を計算し，相互に衝突検知を行って移動物体の描画を行う．

ゲーム A 50 程度の敵が整列，移動する処理とともに，数十個の弾丸による攻撃が行われる．キャラクタは弾丸を避けながら敵を打ち落とす．移動体は単純な動作であるが移動体の数は多い．

ゲーム B 50 程度の敵が整列，移動する処理とと

もに，数個の弾丸，体当りによる攻撃が行われる．キャラクタは弾丸，体当りを避けながら敵を打ち落とす．敵の体当りは複雑な動作として実現されている．

ゲーム C　背景を描画することでキャラクタ自身を高速に前進させながら，正面から現れる 2，3 の敵を攻撃する．描画する移動体は少ないが，移動速度を表現するために描画速度を必要としている．

シミュレーションでは，アプリケーション内クラスのメソッドのサイズ，呼び出し回数，実行時間の全体に対するメソッドの実行時間の割合を用いた．以下に，静的な解析，動的な解析による判定条件を記載し，効果について述べる．

5.2.1　静的な解析による判定条件

（1）　メソッドサイズによる判定

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，10 K 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，音声データも含まれている．表 8，表 9，表 10 に含まれる全バイトコードは，それぞれ約 8 K バイト，約 12 K バイト，約 8 K バイトである．ばらつきがあるが平均はほぼ 10 K である．

圧縮と含まれる画像データ，音声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 10 K バイトとする．その 2 割である 2 K バイト以上のサイズのメソッドをネイティブコードに変換する．

（2）　フレームワークと呼び出しメソッドによる判定

描画イベントを処理するメソッド paint() とキーイベントを処理するメソッド keyEvent() およびそのメソッドから呼び出される private メソッドをネイティブコードに変換する．

5.2.2　動的な解析による判定条件

（1）　メソッドサイズによる選定

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，$1\,\mathrm{sec}/24 = 42\,\mathrm{msec}$ よりも少ない値である 10 msec として設定する．変換処理時間の性能を $50\,\mu$ 秒/バイトと仮定し，10 msec 以内でネイティブコードに変換処理可能なバイトコードのサイズは 200 バイトである．200 バイト以下のメソッドをプロファイリング対象として設定する．

（2）　呼び出し回数による判定

描画イベントを処理する paint メソッドは頻繁に呼び出されるため，プロファイリング処理の代用として，呼び出し回数の基準値とすることができる．判定シミュレーションでは，paint メソッドよりも呼び出し回数が多いメソッドをネイティブコードに変換する．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ネイティブコードに変換することになる．

5.2.3　静的な解析による判定方法の効果

ゲーム A，B，C のそれぞれのシミュレーション結果を表 8，表 9，表 10 に示す．変換対象となったメソッドを判定方法ごとに ◯ を記載している．フレームワーク列の ⊕ を記載したメソッドは，paint メソッド，keyEvent メソッドから呼び出されているメソッドである．

（1）　メソッドサイズによる判定

ゲーム A　静的な解析により変換対象となるサイズの大きなメソッド G$a::a()V が存在し，全体の実行時間の 54%を占めている．

ゲーム B　バイトコード全体の 18%を占めるメソッド M::loop()V が存在する．呼び出し回数は多いが全体の実行時間の 8%を占めているにすぎない．

ゲーム C　コンストラクタ以外は，サイズが 2 K バイト以上のメソッドは存在しない．

メソッドサイズによる判定はゲーム B には有効性がない．メソッド数が少なく，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．

（2）　フレームワークによる判定

ゲーム A　paint メソッドは全体の実行時間の 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．

ゲーム B　paint メソッドは全体の実行時間の 9%程度を占めている．keyEvent メソッドは全体の実行時間の 1%しか占めていない．

ゲーム C　paint メソッドは全体の実行時間の 7%を占めている．keyEvent メソッドは全体の実行時間の 0.04%しか占めていない．

ゲーム B，C では，描画イベントを処理する paint メソッドの全体に占める実行時間の割合はそれほど高くない．しかし，ゲーム B，C の paint メソッドは，動的な解析による判定でネイティブコードに変換できるサイズよりも大きく，動的な解析ではネイティブコードに変換されない．静的な解析で描画メソッドをネイティブコードに変換することで，ユーザの応答時間に影響を与えずに実行時間の 7%以上を占める処理を高速化できる．

キーイベントを処理するkeyEventメソッドは，アプリケーション全体の実行時間と比べると実行時間が短く，本シミュレーション結果から静的な判定による変換の効果は示せていない．

(3) 呼び出しメソッドによる判定

ゲームA paintおよび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，呼び出し回数が少なく，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 keyEventメソッドは，クラス内のメソッドを呼び出さない．paintメソッドは，クラス内の5個のメソッドを呼び出している．メソッド5個のうちの1個は，全体の実行時間の9%の実行時間を占めており，呼び出しメソッドによる判定に効果があることを示している．その他の4個メソッドは，1%以下の実行時間を占めているのみである．

keyEventメソッドからの呼び出しメソッドには，ゲーム操作中の処理とは関係しない処理が含まれており，呼び出しメソッドによる判定は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，呼び出しメソッドによる判定で変換したメソッドのネイティブコードの実行頻度が少ない場合，コードキャッシュから追い出し，動的な解析に対してコードキャッシュの領域を明け渡すことが必要である．

5.2.4 動的な解析による判定方法の効果

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，ネイティブコードに変換可能な小さなメソッドはない．paintメソッドよりも多く呼ばれたメソッドは，静的解析による判定において変換対象としたa()だけである．

ゲームB paintメソッドの呼び出し回数以上に呼び出されたメソッド8個のうちの5個がネイティブコードに変換可能と判定される．

ゲームC paintメソッド以上の呼び出しが行われた8個のうちの4個がネイティブコードに変換可能と判定されている．

ネイティブコードに変換が可能なサイズの小さなメソッドでも，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．

5.2.5 総合的な効果

各判定方法の変換対象として設定したメソッドのサイズと実行時間に占める割合を**表5**に示す．

表5 ゲームA，B，C：選択されたメソッドの実行時間とサイズ
Table 5 Game A, B, C: size and execution times of the selected methods.

コンパイル判定方法			ゲームA		ゲームB		ゲームC	
			サイズ %	実行時間 %	サイズ %	実行時間 %	サイズ %	実行時間 %
静的解析	サイズ大		44	54	18	8	0	0
	フレームワーク	paint	17	37	10	9	7	7
		key Event	5	1	7	1	5	0
	呼び出しメソッド	paint	0	0	0	0	15	10
		key Event	0	0	1	3	0	0
	計		66	92	36	21	27	17
動的解析	サイズ小，呼び出し回数		0	0	5	40	5	40
合計			66	92	41	61	32	57

ゲームA 静的な解析によりサイズが大きいと判定したメソッドと描画イベントを処理するメソッドpaintの実行時間を合わせた時間は，全体の実行時間の92%を占めている．静的な解析による判定が有効であることが分かる．

ネイティブコードに変換した場合の高速化率を3倍と仮定すると，実行時間の92%が1/3となるため，同一の処理を40%程度の時間で実行できる．

ゲームB 動的な解析による判定において変換対象となるメソッドに含まれるバイトコードのサイズは，全体のバイトコードの5%である．しかし，それらのメソッドの実行時間を合わせた時間は，全体の実行時間の40%になる．動的な解析による判定が有効に機能し，実行時間が長いメソッドをネイティブコードに変換できることが分かる．静的な解析により変換されるメソッドの実行時間と合わせた時間は，全体の61%である．

ネイティブコードに変換した場合の高速化率を3倍と仮定すると，実行時間の61%が1/3となるため，同一の処理を60%程度の時間で実行できる．

ゲームC 動的な解析による判定において変換対象となるメソッドに含まれるバイトコードのサイズは，全体のバイトコードの5%である．しかし，それらのメソッドの実行時間をあわせた時間は，全体の実行時間の40%になる．動的な解析による判定が有効に機能し，実行時間が長いメソッドをネイティブコードに変換できることが分かる．静的な解析により判定されたメソッドの実行時間と合わせた時間は，全体の57%である．

ネイティブコードに変換した場合の高速化率を3倍と仮定すると，実行時間の57%が1/3となるため，同一の処理を60%程度の時間で実行できる．

表 6 フレームワークによる判定
Table 6 The determinaton in framework analyzing.

処理名	メソッド	判定	追出抑制
キー操作	本体	変換する	する
	呼び出しメソッド	変換しない	しない
描画処理	本体	変換する	する
	呼び出しメソッド	変換する	する

表 7 インタプリタに対する性能比
Table 7 Ratio of accelerating from interpreter.

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
メソッド呼び出し	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

5.3 試作実装を用いた評価

ネイティブコードをつねに実行する JIT 方式と提案方式を M32R 40 Mhz を搭載した携帯端末上に実装した試作環境を用いて，ユーザ操作の応答時間に対する影響と速度性能の向上を評価した．試作環境は，ネイティブコンパイラを搭載した Java VM [3] に，コードキャッシュとして 32 KB を設定している．

5.3.1 静的な解析による判定条件

判定シミュレーションの結果から一部の判定条件を修正し，試作実装を行った．

（1） メソッドサイズによる判定
判定シミュレーションにて，アプリケーションに含まれる大半のバイトコードが含まれた場合に有効性が示せた．全バイトコードがおおよそ 10 K バイトであるため，4 K バイト以上のメソッドをネイティブコードに変換するように実装した．

（2） フレームワークと呼び出しメソッドによる判定
判定シミュレーション結果から，キーイベント処理メソッドからの呼び出しメソッドの判定に有効性が認められなかった．そのため，キーイベント処理メソッドの呼び出しメソッドは静的な解析による判定の対象から除外して実装した．**表 6** に，静的な解析でネイティブコードに変換するメソッドと追い出し抑制の設定を示す．

5.3.2 動的な解析による判定条件

（1） クラスライブラリ・メソッドの選定
5.1 節において選定した実行時間の長い上位 49 個のクラスライブラリ・メソッドをプロファイリング対象とするように実装した．

（2） メソッドサイズによる選定
判定シミュレーションでは，瞬停を発生させる時間を動画表示と同様の 24 フレームレートとして設定して分析を行った．実際の高速シューティング・ゲームでは，12 フレームレートを基準として利用している．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：83 msec）として定義した．評価に利用したネイティブコンパイラの 1 バイトあたりの変換時間（25 μsec）からコンパイル可能な小さなメソッドを 3.3 K バイト以下のメソッドと設定し，プロファイリング対象となるように実装した．

（3） 呼び出し回数による判定
判定シミュレーションでは，paint メソッドよりも呼び出し回数が多いメソッドを高頻度であると設定して分析を行った．試作実装では，メソッドが一定回数呼び出された場合の経過時間を測定し，1 秒間に 10 回以上呼び出されるメソッドを頻繁な呼び出しであると判定するように実装した．

5.3.3 ユーザ操作に対する応答時間

JIT 方式を実装した Java VM では，ゲーム A の操作中に一瞬画面が停止する瞬停が発生した．JIT 方式を用いた場合，ネイティブコードに変換する処理とコードキャッシュからの追い出しが頻繁に発生していた．

本方式を実装した Java VM では瞬停は発生せず，応答時間に対する影響を排除することができている．

5.3.4 速度性能の向上

携帯電話に搭載された Java において利用されるベンチマークを用いて評価を行った．**表 7** に，本方式のインタプリタ実行に対する性能向上率を示す．

本方式を用いることで，各測定項目においてインタプリタ実行と比較して速度性能が向上していることが分かる．特に，測定項目「基本演算，ループ」では，5 倍近い速度性能の向上が得られている．

5.3.5 提案方式の影響

表 7 には，ベンチマークを用いて取得した JIT 方式によるインタプリ実行に対する性能向上率も合わせて示している．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，コードキャッシュの不足はおきず，キャッシュからの追い出しが行われ難いために，つねにネイティブコードを実行する JIT 方式による測定結果が最高性能を示す．

そのため，本方式を用いた場合には，JIT 方式に比べて性能向上率が全般的に下回る．しかし，その差は最大で 0.2 倍しかなく，提案方式は速度性能に対して大きな影響を与えていない．以下に，JIT 方式と比べて性能向上率が低い理由を述べる．

表 8 ゲーム A：各メソッドの実行時間とサイズ
Table 8 GameA: Size and execution time of methods.

メソッド	サイズ byte	呼び出し回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
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 9 ゲーム B：各メソッドの実行時間とサイズ
Table 9 GameB: Size and execution time of methods.

メソッド	サイズ byte	呼び出し回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
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

（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，ホットスポットを検出するまでのインタプリタにおける実行時間が含まれるため，JIT 方式と比べて性能低下が検出される．しかし，測定項目「基本演算，ループ」の性能差 0.02 倍が示すようにわずかな差でしかない．

（2） プロファリング処理のオーバヘッド
測定項目「メソッド呼び出し」では，メソッド呼び出しが頻繁に行われるために，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．

（3） プロファイル対象外のメソッド
測定項目「記憶領域 I/O」が頻繁に利用するストリーム処理のクラスライブラリ・メソッドをプロファイル対象と設定しておらず，インタプリタを用いた実行が行われている．

表 10 ゲーム C：各メソッドの実行時間とサイズ
Table 10 GameC: Size and execution time of methods.

メソッド	サイズ byte	呼び出し回数	静的解析		動的解析	実行時間 %
			サイズ大	フレームワーク	サイズ小	
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

6. おわりに

本論文では，メモリ量が少ない携帯端末上で，高速シューティング・ゲームを代表とするユーザ操作をともなう Java アプリケーションを，ネイティブコンパイラを用いて高速化する方式の提案を行った．

本方式は，携帯端末の Java アプリケーションの特性であるダウンロード時の通信量，実行時のメモリ量の制約が厳しく，かつ，ユーザ操作に対する応答時間に制約がある高速シューティング・ゲームを代表とする Java アプリケーションに適用することを目的としている．

本方式では，頻繁に実行されるホットスポットを検出するために，Java アプリケーションの起動前にメソッドサイズ，フレームワーク，呼び出しメソッドの

静的な解析を行い，アプリケーションの実行中にサイズおよび処理内容に基づいて選定したメソッドのみをプロファイリング対象とする動的な解析を行うことにより，プロファイリングに必要なメモリ量を低減し，ユーザ操作の応答時間を遅延させないネイティブコンパイラを用いた高速化を実現する．

評価では，インタプリタ実装を用いて取得したメソッドの実行時間と呼び出し回数を用いた判定シミュレーションによりクラスライブラリ・メソッドの選定および静的，動的な判定方法の有効性の確認を行った．また，試作実装を用いた評価では，ユーザ操作に対する応答時間の影響を排除したうえで，JIT 方式と同様に最大で約 5 倍の速度性能の向上を実現し，本方式の速度性能に対する影響が少ないことを確認した．

本方式の判定シミュレーション，試作実装による評価は，10 K バイト程度の Java アプリケーションを実行するメモリ量，CPU 性能を持つ携帯端末を用いて行った．現在，100 K バイト以上の Java アプリケーションが利用できる携帯電話が登場しており，評価に用いた環境と比較すると利用できるメモリ量が増加し，高速な CPU が利用できる．本方式は，メソッドサイズ，フレームワーク，呼び出しメソッドおよび処理内容に基づいた静的な解析，動的な解析を用いることに特徴がある．そのため，より高速な CPU が利用可能な環境においては，ネイティブコードに変換する時間が短縮されるため，ユーザ操作に対する応答時間に影響を与えないで変換できるメソッドのサイズ制限を緩和することができる．また，より多くのメモリが利用可能な環境においては，プロファイリングに利用可能なメモリ量が増大し，プロファイリング対象として設定するメソッドの処理内容を拡大することができる．本方式では，メモリ量，CPU 性能の制約が緩和された場合にも，ユーザ操作に対する応答時間の影響を排除することでき，また，利用するメモリ量を調整することができる．

今後，試作実装による速度性能の評価において測定されたプロファイリング処理のオーバヘッドの調査とその低減を行い，サイズの大きな Java アプリケーションを実行できる，より高速な CPU，より多くのメモリが利用可能な携帯端末において，プロファイリング対象とするメソッドの選定方法を検討し，プロファイリング対象が増加した場合の影響を調査することで，本方式の完成度を高めていきたいと考えている．

謝辞 本研究を進めるにあたり，組み込み機器向けのネイティブコンパイラに関してご指導をいただいた株式会社ルネサステクノロジ（元三菱電機株式会社 LSI 事業化推進センタ）坂本守氏に感謝いたします．

参 考 文 献

1) Gosling, J., Joy, B., et al.: *Java Language Specification, 2nd Edition*, Addison Wesley (2000).
2) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification, 2nd Edition*, Sun Microsystems Inc. (1999).
3) Java 2 Platform Micro Edition Technology for creating mobile devices. White Paper. http://java.sun.com/products/cldc/wp/
4) Kaffe—An Opensource Java Virtual Machine. http://www.kaffe.org
5) Suganuma, T., Ogasawara, T., et al.: Overview of the IBM Java Just-in-Time compiler, *IBM Systems J.*, Vol.39, No.1, pp.175–193 (2000).
6) Alpern, B., Attansio, C.R., et al.: The Jalape˜no Virtual Machine, *IBM Systems J.*, Vol39, 1, pp.211–221 (2000).
7) The Java HotSpot Virtual Machine. http://java.sun.com/javase/technologies/hotspot/
8) The cldc hotspot (tm) implementation virtual machine, White Paper. http://java.san.com/products/cldc/
9) 有馬 啓ほか：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).
10) Whaley, J.: A Portable Sampling-Based Profiler for Java Virtual Machines, *ACM 2000 Java Grande Conference*, pp.78–87 (2000).
11) 首藤一幸ほか：Java Just-in-Time コンパイラのためのコスト効率の良いコンパイル手法，電子情報通信学会論文誌，Vol.J86-DI, No.4, pp.217–231 (2003).
12) 川本琢二ほか：家電向け JavaJIT コンパイラの構成法とその評価，情報処理学会論文誌，Vol.43, No.SIG08(PRO15), pp.37–48 (2002).
13) Shaylor, N.: A just-in-time compiler for memory-constrained lowpower devices, *Proc. Usenix Java Virtual Machine Research and Technology Symp.*, pp.119–126 (2002).
14) 舩田雅史ほか：携帯機器を対象とした Java 動的コンパイラにおけるプロファイリングシステム，情報処理学会研究報告，2003-MBL-028, pp.55–62 (2004).
15) The Mobile Information Device Profile, Sun Microsystems, Inc. http://java.sun.com/products/midp/
16) JCP: JSR-000030 J2ME Connected, Limited Device Configuration. http://www.jcp.org/
17) JCP: JSR-000139 Connected Limited Device

Configuration 1.1. http://www.jcp.org/

(平成 18 年 5 月 12 日受付)
(平成 18 年 11 月 2 日採録)

高橋 克英

昭和 42 年生．平成 3 年高知大学数学科卒業．同年三菱電機（株）入社．JavaVM 実装に関する研究開発に従事．平成 18 年より（株）ルネサステクノロジに出向．携帯電話ソフトウェアの開発に従事．

清原 良三（正会員）

昭和 35 年生．昭和 60 年大阪大学大学院工学研究科応用物理学専攻前期課程修了．同年三菱電機（株）入社．昭和 63 年より（財）新世代コンピュータ技術開発機構に出向．平成 4 年三菱電機（株）に復職．携帯電話のソフトウェア更新，JavaVM 実装に関する研究開発に従事．ACM 会員．