PDA における Java 実行の高速化の一方式

有馬 啓† 並木 美太郎†

Java 言語は Web ブラウザの組込み言語として普及してきたが，近年は PDA や組込みシステムなどのソフトウェア開発用の言語としても注目を浴びている．しかし，Java プログラムはインタプリタ形式で実行されるため実行速度が遅い，という問題がある．パーソナルコンピュータ（以下，PC）では，Java を高速に実行するために JIT や HotSpot の技術が用いられるが，PDA では，翻訳処理やプロファイリング処理が大きな負荷となってしまう．また Java クラスファイルをネイティブコードに翻訳する方法では，プラットフォーム独立やアプリケーションサイズが小さいなどの Java の特徴を失ってしまう．本稿では，PDA 上で Java を高速に実行する方式を提案する．提案する方式は，Java クラスファイルをすべて翻訳するのではなく，頻繁に利用されるメソッドだけをネイティブコードに翻訳する．また PDA の負荷を軽減するために，ホスト PC と連携してプロファイリングや翻訳作業を行う．具体的には PDA 側ではプロファイリングに必要なデータを取得し，ホスト PC 側で最終的なプロファイリングとネイティブコードへの翻訳を行う．本方式に基づいて，PalmOS 上で動作する KVM を対象としたメソッドカウンタとプラファイラデータベースを実現し，方式の有効性を確認した．その結果，呼出し回数全体で 5%以上の呼出し回数があるメソッドを機械的に選択するだけで，1.5 倍から 3.5 倍の速度向上を，メモリ使用量 10%から 20%増加のオーバヘッドで達成できることが分かった．

An Approach to Accelerating the Execution of Java for PDA

Akira Arima† and Mitaro Namiki†

It is expected that software of PDA is developed in Java programming language. However, instructions of JavaVM are executed with low performance. JIT or HotSpot technologies on the desktop computers can solve this problem, but these technologies are not enough for PDA. Because PDA has few memory capacity and low CPU performance. Moreover, the approach that is to compile all of Java class file loses the advantage of Java such as platform-independent. This paper describes a scheme of collaborative compilation for Java byte code on PDA and host PC. This approach is to translate some methods invoked many times to native codes on host PC with profile data logged at PDA. This language processor profiles counts of method call, and compiles effective methods to reduce overhead of interpreter. In this paper, KVM based on this approach for PalmOS is discussed.

1. はじめに

Java 言語は，インターネットの発展にともなって普及してきたが，近年は PDA のソフトウェア開発言語としても注目を浴びている．PDA や組込みソフトウェアの分野では生産性よりもコストパフォーマンスを重視しているため，アセンブラや言語 C などでハードウェア性能を最大限に引き出すようなソフトウェア開発が行われている．同時に，近年，PDA など組込み機器に対する高機能化の要求が高まっており，ソフトウェアも複雑化していることから，生産性も重要視されるようになってきた．しかし，従来のアセンブラや言語 C でのソフトウェア開発はプラットフォームに依存してしまうので，開発効率が悪い．もし PDA 上で Java を実行することができれば，あらゆるプラットフォームで実行できるアプリケーションが提供され，開発効率が向上する．さらにプラットフォームに依存しないという特徴は，メールや Web 上でのプログラムが配信可能になるなどの利点を生じる．

PDA で Java を実行しようとした場合，ハードウェアの性能が問題となってくる．Java は通常インタプリタ形式でプログラムを実行するため，実行速度が遅い．パーソナルコンピュータ（以下，PC）では Java を高速に実行するために，いくつかの高速化方式が採用されているが，PDA に PC のような方式をとり入

† 東京農工大学大学院工学研究科
Graduate School of Technology, Tokyo University of Agriculture and Technology

れると速度低下を招いてしまう可能性がある．たとえば PDA で Just In Time Compiler[1]（以下，JIT）技術を取り入れた Java 仮想機械[2]（以下，JavaVM）を実装すると，PDA のメモリ容量が小さいので翻訳時にメモリが足りなくなり，頻繁にメモリの確保や開放が繰り返されフラグメンテーションなどが起こる可能性がある．プロファイリングを行う HotSpot[3]を採用した場合，JIT と比べると実行時のメモリ消費量は減るが，高速化のためにプロファイリングの精度を上げると，PDA のような性能の低い CPU では負荷が増加し，性能の向上は望めない．

このようにハードウェア性能の制約上，従来 PC で用いられてきた方式では高速化が望めない．そこで GCJ[4]のように Java クラスファイルをすべてネイティブコードに翻訳して実行する方法が考えられる．しかし Java クラスファイルのすべてをネイティブコードに翻訳する方式は，Java のプラットフォーム独立という特徴をなくしてしまう．さらに，アプリケーションのコードサイズが大きくなってしまう，といった問題がある．

PDA 上などの組込み機器で Java の特徴を維持したまま高速化を行うためには，Java を実行するときにボトルネックとなる部分だけをネイティブコードに翻訳し，他の大部分は Java クラスファイルのまま実行する方式[5]が考えられる．この方式では組込み機器向けの方式として，Java のボトルネックとなるメソッドを事前にネイティブコードに翻訳する．組込み機器側ではボトルネックとなる部分はネイティブコードで実行し，その他の部分は Java クラスファイルのまま実行される．実行時にボトルネックとなる部分だけをネイティブコードで実行することで，Java の特徴を維持し，高速に実行できる．さらに，メモリを大量に消費することなく実行することができる．しかし，どのメソッドを翻訳対象とするかは，静的情報からは判別が困難である．

一般に，PDA はユーザによって使用方法が異なり，インストールされたアプリケーションは，頻繁に使われるアプリケーションもあれば，ほとんど使われないものもある．使用頻度が異なるアプリケーションを同じように高速化の対象とすると，ほとんど使われないアプリケーションに対しても同じようにメモリが消費され効率が悪くなる．

そこで本稿では，PDA において，ネイティブコードと Java クラスファイルを混在させる方法で，PDA の負荷を軽減する Java の高速化方式について提案する．具体的には，メソッドの使用頻度に応じて，メソッドを部分的にネイティブコードに翻訳する．HotSpot でもプロファイリングして同様の方法で高速化を行うが，HotSpot のように JavaVM 側でプロファイリングから翻訳処理のすべてを行うのではなく，PDA 上の JavaVM では，メソッドの呼出し頻度を予測するために必要なメソッドの呼出し回数だけを計測する．メソッドの呼出し回数を計測するだけなので，データ収集は軽量に実行できる．計測したデータから，ホスト PC で詳細なプロファイリング解析を行い，ネイティブコードへの翻訳処理を行う．このような方式で翻訳処理を行うことによって，PDA 側の負荷を軽減し，性能の低い CPU でも Java を高速に実行することができる．さらに PDA はユーザの利用形態を強く反映するので，メソッドの参照特性はユーザによって異なることが推測される．したがってユーザごとの要求に応えるために頻繁に使用されるメソッドを部分的にネイティブコードに翻訳できる可能性がある．

表 1 高速化方式の比較
Table 1 Comparison of accelerating the execution of Java.

	JIT	HotSpot	Native Code	本方式
メモリ使用量	×	△	○	○
オーバヘッド	△ 翻訳	× 翻訳 profile	○	○
ファイルサイズ	◎	◎	×	○
Java の特徴	○	○	×	△ 翻訳

表 1 は，PDA 上での Java 実行の高速化方式を比較したものである．JIT や HotSpot ではファイルサイズが小さく，Java の特徴であるプラットフォーム独立や動的呼出しなどが維持できるが，PDA にこれらの方式を用いると実行時のメモリ使用量や負荷が大きく，高速に Java を実行することが期待できない．HotSpot では，JIT よりも実行時のメモリ消費量は減るが，HotSpot 技術を用いた JDK1.3.0 では，Pentium166 MHz 以上の CPU を要求し，これは HotSpot 技術を使用していない JDK1.2.2 が 486/DX 以上の CPU であれば実行できたことを考えると，プロファイリングにかかる負荷が大きいことが推測される．本稿で用いた PDA は WorkPAD3.0J であるが，M68000 互換の 16 MHz の CPU で，現在多くのデスクトップ PC として使用されているマシンと比較すると，10 倍以上の性能差がある．PC 上でコンパイル処理が数百 ms かかったとすると，PDA 上では数秒にもなり，対話性能が要求される PDA では深刻な問題となる．さらに，JIT コンパイラのコードは数百 KB にも及ぶ．PC では 100 MB を超える主記憶を有し，仮想記憶を用いるので問題はないが，PDA では主記憶の数割を

占有してしまう．実記憶系の PDA や組込み機器では，JIT のコードが占めるメモリサイズは，ますます深刻な問題となる．また，すべてネイティブコードに翻訳した場合には，メモリ使用量を低く抑えることができ高速に実行できるが，アプリケーションサイズが大きくなり，プラットフォーム独立などの Java の特徴を失ってしまう．

本稿では，プロファイリングと翻訳処理をホスト PC と PDA で分散することで，低性能の PDA の負荷を減らしつつ，高速化を達成する方式を提案する．本方式に基づいて KVM 上にメソッドカウンタとプロファイリングデータベースを実装し，その有効性を確認することができた．

2. 目的と特徴

本方式では，PDA 上で Java を実行する際に，プロファイル結果を基にボトルネックとなる部分をネイティブコードに翻訳し，効率的な高速化を行うことを目的する．さらに PDA の性能を考慮して，ホスト PC と連携して，PDA の負荷を軽減することも目的とする．本方式の特徴は次のとおりである．

(1)　PDA とホスト PC が連携してプロファイリングを実行

PDA はユーザによって使い方が多種多様であり，Java アプリケーションで使用されるメソッドの呼出し頻度が使い方によって異なる．このことから頻繁に使用されるメソッドをプロファイリングによって選択し，高速化することで，Java アプリケーションの実行速度が向上されることが期待できる．PDA 上でプロファイリングに必要なデータを取得し，同期時にホスト PC へデータが転送される．取得したデータからホスト PC 側で最終的なプロファイリングを行う．

(2)　メソッドを部分的にネイティブコードに翻訳

プロファイリングによって使用頻度の高いメソッドをネイティブコードに翻訳する．クラスファイル全体を翻訳すると PDA のメモリを大量に消費してしまい，Java の特徴も失うことから，実行時にボトルネックとなる部分だけをネイティブコードに翻訳することで，Java の特徴を保持したまま実行できる．

(3)　ホスト PC 側で翻訳処理

PDA では PC と比較すると実装されている実メモリの容量が少ないので，JIT のようにメモリが大量に必要となる技術は適さない．本方式では，ホスト PC 上でネイティブコードへの翻訳処理を行う．ホスト PC 上で翻訳されたネイティブコードは，PDA とホスト PC との同期時に転送される．

(4)　ネイティブメソッド呼出し

ネイティブコードに翻訳されたメソッドは，JavaVM の中でネイティブメソッドとして呼び出すことで，既存の JavaVM に特別な改良を加えることなく実行できる．

3. 方 式 設 計

3.1　本方式が適用される環境

本方式は，PDA が頻繁にホスト PC と同期をとることを前提としている．PDA は，アプリケーションをインストールする場合，一度ホスト PC にデータをダウンロードして，ホスト PC と同期をとった際にアプリケーションがインストールされる．したがって，ホスト PC 側には PDA にインストールしたアプリケーションやデータが保存される．また PDA 上でよく使用されるメールソフトやスケジュール管理ソフトなどは，ホスト PC と連携して使用され，同期の際にホスト PC 側にあるデータと PDA 側にある送受信したメールのデータやスケジュール表などのデータと同期する．本方式では，このような環境を想定している．

3.2　方式の概要

本方式では，PDA 側の負荷を軽減するために，ホスト PC 側で高速化の解析に必要なプロファイリングやネイティブコードへの翻訳，クラスファイルの変更を行う．高速化に必要な大部分の処理をホスト PC 側で実行することで，PDA 側の処理はプロファイリングに必要なメソッド呼出しの回数をカウントする機能とカウント結果を一定量保持しておくためのプロファイリングデータベースが必要となるだけである．

図 1 は，本方式における処理の流れを示したものである．まず，Java アプリケーションが PDA にインストールされる．PDA 上の JavaVM は，Java が実行されるたびにメソッド呼出し回数を計測する．さらに計測結果をプロファイリングに使用するためにアプリ

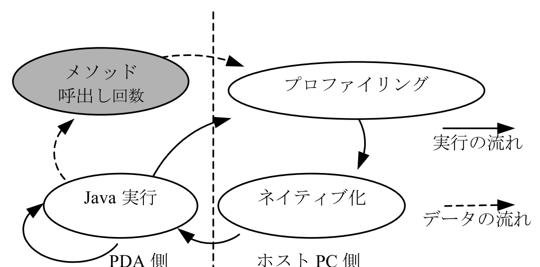

図 1　本システムの処理方式
Fig. 1　The system of processing.

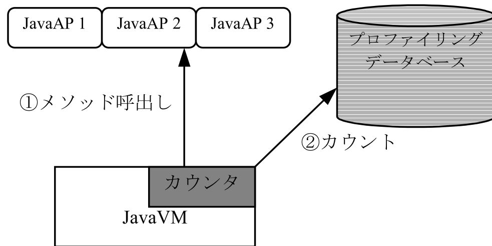

図 2 PDA 側の処理
Fig. 2 Process of PDA side.

ケーション終了後もプロファイルしたデータを保持する．その後ホスト PC と同期をとった際に，このデータをホスト PC に渡す．このデータからホスト PC 側でプロファイリングとネイティブコードへの翻訳が行われ，変更後の Java アプリケーションを PDA 側に転送する．

3.3 PDA 側の処理

PDA 側では JavaVM とプロファイリングに必要なメソッド呼出しの回数を計測する機能だけを提供する．ユーザがホスト PC との同期をとった際に，プロファイリングデータをホスト PC へ転送する．

また，ホスト PC でネイティブコードに翻訳されたメソッドは，JavaVM のネイティブメソッド呼出しを利用して呼び出される．したがって，ネイティブメソッドの呼出しには，JavaVM とのリンケージが必要になる．Java2 Standard Edition[6]（以下，Java2SE）では，Java Native Interface[7]（以下，JNI）を利用することによってネイティブメソッドとのリンケージが可能になる．JNI のようにネイティブメソッドとのインタフェースを有する JavaVM では，既存の方式をそのまま利用することができる．しかし KVM[8]のような PDA 用の JavaVM には，あらかじめ JavaVM に組み込まれたネイティブメソッドだけが使用可能であり，ユーザが新たにネイティブメソッドを追加するためのインタフェースを持たないものもある．本稿ではこのような JavaVM である KVM に対して設計を行った．そのため今回は，試作として JavaVM そのものにネイティブメソッドのコードを埋め込む方式を採用した．詳細については 4 章で述べる．

PDA 側の処理を**図 2** に示す．PDA には JavaVM 内にメソッドカウンタと，プロファイリング処理に必要なメソッド呼出し回数を記録するためのプロファイリングデータベースがある．プロファイリングデータは同期時にホスト PC 側に転送する．

3.4 ホスト PC 側の処理

図 3 は，ホスト PC 側の処理の流れを示したものである．本方式では，ホスト PC 上に，インストールした Java アプリケーションをすべて保持していることを想定している．ホスト PC 側では高速化の処理に必要な処理の大部分を行う．PDA 側からプロファイリングデータを取得し，プロファイリングデータを基にネイティブコードに翻訳するメソッドを決定する．そしてネイティブコードに翻訳後，PDA 上の Java アプリケーションを置き換える．これらの処理の詳細は次のとおりである．

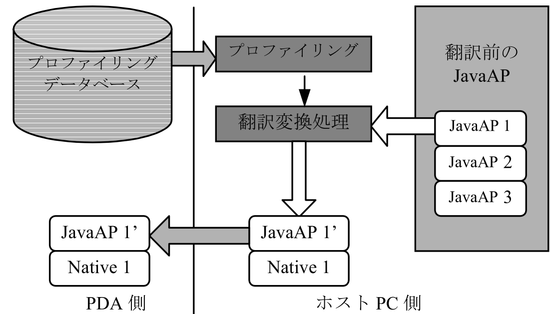

図 3 ホスト PC 側の処理
Fig. 3 Process of host PC side.

(1) プロファイリング
PDA 側で取得したプロファイリングデータから，頻繁に呼び出されているメソッドを選出し，ネイティブコードへの翻訳対象とする．

(2) ネイティブコードへの翻訳
翻訳対象のメソッドを含むクラスファイルはあらかじめホスト PC 側に保持されている．翻訳対象のメソッドが含まれるクラスファイルからメソッドを抽出しネイティブコードへ翻訳する．

(3) クラスファイルの書き換え
ネイティブコードに翻訳されたメソッドはネイティブメソッドとして呼び出されるため，オリジナルのクラスファイルを書き換える必要がある．まず，Java クラスファイル内にあるメソッドの属性を示すフラグをバイトコードからネイティブメソッドに変更し，翻訳するメソッドのバイトコードを Java クラスファイルから取り除く．またネイティブコードへの翻訳対象は使用頻度に応じて変わるので，オリジナルのクラスファイルを保持し，ホスト PC 側にはクラスファイルを元に戻す機能を提供する．この場合にはホスト PC にあるインストール履歴から Java クラスファイルが復元される．

(4) PDA 上の Java アプリケーションの置き換え
(1) から (3) の処理が終了すると，変更されたアプリケーションとネイティブメソッドを PDA 側に転

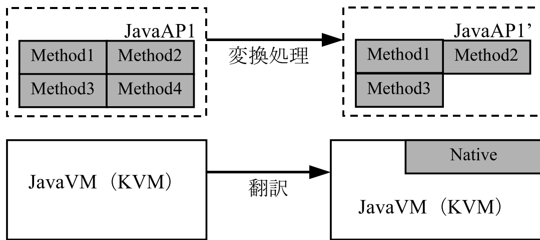

図 4 KVM での実装方式
Fig. 4 Custom design for KVM.

```
typedef UInt unsigned short int;

struct ProfileAppInfo{              // profiling APPINFO
        UInt infosize;              // APPINFO のサイズ
        UInt num;                   // 履歴取得の回数
        UInt offset;                // 現在の履歴数
};
```

図 5 アプリケーション情報ブロック
Fig. 5 Application information block.

送し，Java アプリケーションを置き換える．

4. PDA 側の設計

4.1 PDA 側の全体設計

本稿では，ターゲット PDA として PalmOS 3.1 [9),10)]が動作する WorkPad を使用し，JavaVM は Java2 Micro Edition [11)]の KVM を使用する．また開発環境は CodeWarrior Release 6 を用いる．

PDA 側には，プロファイリングのためのメソッドカウンタと，ホスト PC にプロファイリングデータを渡すデータベースがある．KVM はユーザが作成したネイティブメソッドを呼び出すインタフェースが存在しないので，ネイティブメソッドを KVM の中に直接組み込む設計とした．

JavaVM では，メソッドは `invokestatic` など数種類の命令が必ず呼び出されるため，カウントするタイミングを限定することができる．またどのような JavaVM でも，クラスやメソッドをテーブルで管理していることから，今回の設計は汎用性があると考えている．そのため PalmOS 以外の WindowsCE などの PDA でも，ネイティブメソッド用のインタフェースがあれば本方式を取り入れた JavaVM を実装することができ，同様の効果が期待される．

4.2 ネイティブメソッド呼出し

KVM ではネイティブメソッドを呼び出すことはできるが，JNI のようなユーザが作成したネイティブメソッドを実行する機構がないため，KVM にあらかじめ組み込まれたネイティブメソッドだけしか呼び出すことができない．そこで今回の実装では，**図 4** のように翻訳対象のメソッドのバイトコードを一度言語 C のソースコードに変換し，KVM のソースコードに追加する．その後 KVM 全体をコンパイルし直すことで，KVM が提供するネイティブメソッドとして実現した．

4.3 プロファイルデータベースについて

プロファイリングデータベースは，アプリケーションの実行ごとにメソッドの呼出し回数の計測結果を保持する．PalmOS のデータベースはストレージメモリと呼ばれる保護されたメモリ上に存在する．したがってプロファイリングデータベースは PalmOS のデータベースに特化した形で設計を行った．PalmOS のデータベースは，個々のデータベースに関する情報が格納される領域と，情報が格納されたレコードと呼ばれる領域がある．メソッドの呼出し回数をカウントした結果をレコードの領域に保存する．個々のデータベースに関する情報が格納される領域には，アプリケーション情報ブロックというデータベース自体の情報を格納できる領域にすべてのレコードに共通する情報が格納される．アプリケーション情報ブロックにはアプリケーションの実行回数と，ユーザが設定する履歴の取得数があり，**図 5** のような構造になっている．

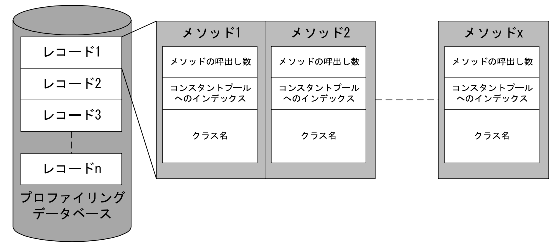

図 6 プロファイリングデータベースの構造
Fig. 6 Structures of profiling database.

プロファイリングデータベースの各レコードを**図 6** に示す．アプリケーション実行ごとにその実行で使用されたすべてのメソッドのプロファイルデータを異なるレコードに書き込む．レコードに保存される情報はメソッドの呼出し回数とメソッドを指し示す `cp_info`，クラス名である．これらの情報を使用して，ホスト PC 側でプロファイリングを行う．

ユーザが指定した履歴回数分だけレコードを確保し，プロファイリングデータを書き込む．プロファイリングは，最新の利用形態を反映するためにアプリケーションが実行されるたびに行われ，古いものから削除される．アプリケーション実行回数が履歴取得回数を超えた場合には，先頭のレコードから上書きされる．

4.4 メソッドカウンタについて

KVMでは，invokestaticなどのメソッド呼出し命令が実行されたときは，KVM中のメソッド呼出しを処理する関数が必ず実行される．したがってこの関数内でメソッドをカウントすることによって，メソッド呼出し回数をすべてカウントすることができる．

メソッド呼出しを処理する関数はメソッドテーブルを参照する．メソッドテーブルには，メソッド呼出し時に必要な情報が格納されている．そこで，このメソッドテーブルにプロファイリングデータベース上のメソッドを示すインデックスを追加した．メソッド呼出し時にはこのインデックスを参照してメソッドごとに対応した場所への書き込みを行い，呼出し回数を計測する．またメソッドテーブル作成はクラスロード時に行う．

5. ホストPC側の設計について

5.1 ホストPC側の全体設計

ホストPCは，Windows98が動作するAT互換機を使用する．ホストPCではJavaのメソッド部分をPalmOS用のネイティブコードに翻訳するため，CodeWarriorを使用する．プロファイリングやクラスファイルの書き換えなどのホストPC上で動作するプログラムは言語Cで記述し，コンパイラはgccを使用する．また，Javaバイトコードから言語Cを生成するために，Toba[12]を一部改良して使用する．プロファイリング処理には，PDAから受け取ったプロファイリングデータを使用する．ネイティブコードへの翻訳には，主にクラスファイルのコンスタントプールとメソッドの実行コードがあるcode属性部分を使用する．コンスタントプールはメソッドの型や引数を決定するために必要であり，code属性はネイティブコードに翻訳する部分になる．KVMではJNIのようなインタフェースがない．そこで今回は，**図7**のように対象となるメソッドを一度言語Cに変換し，KVMのソースコードに言語Cに変換されたメソッドのソースコードを追加することによってネイティブメソッドとして扱う設計とした．追加した後に，追加されたKVM全体を再コンパイルする．また，型や引数もKVMでJavaスタック操作命令を使用することによってネイティブメソッドからでもJavaスタックを操作できる．そのためこれらの命令を使用してJavaバイトコードとのリンケージをとることができる．

5.2 プロファイリング処理部

プロファイリング処理部では，PDA側のプロファイリングデータベースからプロファイリングデータを取得する．このデータを基にメソッド呼出しの総数と個々のメソッドごとの呼出し回数を調べる．その結果から頻繁に呼び出されるメソッドが翻訳対象として決定される．今回はメソッドの呼出し回数が，メソッドの全呼出し回数の5%以上呼出しがあるメソッドを翻訳対象とした．

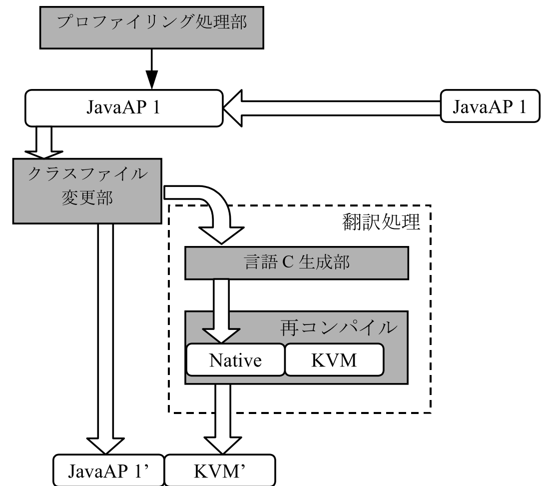

図7 ホストPC側の翻訳処理
Fig. 7 Compilation of the host PC side.

翻訳対象となったメソッドは，プロファイリングデータからコンスタントプールIDとクラス名が分かるので，Javaアプリケーションのインストール履歴から，対象のメソッドが含まれるJavaクラスファイルとメソッド名がクラスファイル変更処理に渡される．

5.3 クラスファイル変更部

クラスファイル変更部では，翻訳対象となるメソッドを含んだクラスファイルをプロファイリング処理部から受け取る．その後，翻訳対象となるメソッドの属性を，JavaVMがネイティブメソッドとして呼び出せるように，ネイティブコードの属性に変更する．クラスファイル内にはメソッドの修飾子を示すaccess_flagsがあり，この値の8ビット目のフラグをセットする．

翻訳対象のメソッドのバイトコードが記述されているcode属性を変更し，同時に不要となるメソッドのバイトコードを取り除く．この取り除かれたバイトコードから言語Cが生成されるので，抜き出されたコードは言語C生成部に渡される．さらにcode属性には，JavaスタックやJavaVM内でのローカル変数の個数を示す値が格納されているが，ネイティブメソッドでは不要になるために，これらの属性値の変更を行う．

5.4 言語C生成部

翻訳処理には，クラスファイル変更部から渡された対象メソッドのクラス名，メソッド名，ディスクリプタ，メソッドの実行コードを基にネイティブコードを生成す

表 2 Java から言語への変換例
Table 2 Sample of a translation from Java to C.

Java	言語 C

Static void Hoge(int x, int y){
  /* 実行部分 */
}
	/* インタフェース関数 */
	Static void Foo_Hoge(){
	  int y = popStack();
	  int x = popStack();
	  Foo_Hoge_Exe(x, y)
	}
	
	/* 実行関数 */
	static void Foo_Hoge_Exe(int x, int y){
	  /* Toba が生成した実行コード */
	}

る．ただし，今回実装を行った KVM ではネイティブメソッド用のインタフェースがないので，言語 C 生成部では Toba を利用してクラスファイルから言語 C を生成し，KVM のソースコードに埋め込める形式にする．

Toba は Java クラスファイルをすべて言語 C のソースコードに変換するが，本方式では翻訳対象となったメソッドだけを言語 C にすればよい．したがって，翻訳対象メソッドに対応する関数を KVM のソースコードに組み込む．ただし Toba が生成したソースコードは，Java との引数やスタックの扱い方が異なる．そこで Java メソッドのインタフェース用の関数を生成し，この関数内で，KVM 内のスタック操作命令を使用し，Java メソッドとネイティブメソッドとの引数や返り値の受渡しを行う．具体的には**表 2** のようになる．

たとえば，Java のクラス名が `Hoge` で，メソッド名が `void Foo(int x, int y)` から生成される関数から，インタフェース関数が `void Hoge_Foo(void)`，実行関数が `void Hoge_Foo_Exe(int x, int y)` という関数が生成される．

5.5 KVM の翻訳と転送

翻訳対象のメソッドは言語 C への変換後に，KVM のソースコードに組み込まれ再コンパイルされる．本方式では，追加されたネイティブメソッドとクラスファイルだけを PDA に転送するが，KVM にはネイティブメソッド用のインタフェースが存在しないので，KVM 全体を再コンパイルするような実装を行った．したがって PDA への転送についても，ネイティブメソッドが追加された KVM と Java アプリケーションを PDA に再インストールするという方式になり，転送されるデータが大きいために同期に時間がかかる．実測結果を，次章に示す．PDA や組込み機器に適した，標準的なネイティブメソッドの扱いが望まれる．

6. 実装と考察

本方式を実装した環境を**表 3** に示す．本方式の設計中，メソッドカウンタおよびプロファイリングデータベースを KVM 上に実装した．ホスト PC 上にクラスファイル変更部，KVM へのネイティブ化された言語 C コードを挿入する環境を構築した．現在，抽出されたバイトコードで表現されたメソッドを Toba により言語 C に翻訳する部分だけは未完成なことから，言語 C への変換は手動で行った．本方式による KVM は 272 KB で，オリジナルの KVM より約 2 KB 増で実装することができた．

表 3 実装環境
Table 3 Implement environment.

	PDA	ホスト PC
機種	WorkPad 30J	AT 互換機
OS	PalmOS 3.1	MS Windows 98
CPU	Motorola DragonBall-EZ 16MHz	Intel Celeron 333MHz
RAM	4MB	128MB

表 4 実行時間の比較
Table 4 Comparison of execution time.

	App1	App2	App3
翻訳処理前	11.55	90.03	11.16
全メソッド呼出し回数の 10%以上呼出しのあるメソッドを対象	3.21	90.03	11.16
全メソッド呼出しの 5%以上呼出しのあるメソッドを対象	3.21	59.57	11.16
全メソッド呼出しの 3%以上呼出しのあるメソッドを対象	3.21	47.97	11.16

単位：秒

本実装で方式の有効性を確認するために，実際に PDA 上でアプリケーションプログラムを実行し，実行時間，メモリの使用量などを評価した．評価では，次の 3 つのプログラムをサンプルプログラムとして用意した．まず，簡単なベクトルの総和計算を繰返し計算の基本性能向上の基本指標と考え，APP1 として採用した．また，3D オブジェクトの座標計算を PDA の GUI 表示の一例と考え，APP2 として採用した．これは，PDA はクライアントサイドで UI の処理が多く，表示の基本演算は頻繁に実行されると考えたためである．最後の APP3 はダイアログボックスからの要素選択という，PDA で重要となるユーザからの対話が必要なプログラムである．このサンプルは，ユーザの入力待ちが入り，正確な値の計算が困難な点もあるが，対話性への影響を見るためにサンプルとして採用した．この 3 つのサンプルは，いずれも基本性能をはかるために最低限の項目と判断した．

本システムはユーザの使用頻度に応じて，翻訳対象のメソッドが変更されるので，測定では APP1 を日に 3 回，APP2 を日に 1 回，APP3 を週 2 回使用したと仮定し，4 週間分のデータ（120 回分の実行）からプロファイリングを行った．プロファイリングデータの大きさは 24.6 KB であった．

まず，本方式で部分的にネイティブコードに変換したときの実行時間の性能向上を**表 4** に示す．ここでの実行時間は，JavaVM の初期化から終了までのす

表 5 アプリケーションコードサイズの比較
Table 5 Comparison of application size.

	App1		App2		App3	
	Java Classfile	Native Code	Java Classfile	Native Code	Java Classfile	Native Code
翻訳処理前	2.02	-	8.42	-	2.67	-
全メソッド呼出し回数の10%以上呼出しのあるメソッドを対象	1.95	0.20	8.42	-	2.67	-
全メソッド呼出しの5%以上呼出しのあるメソッドを対象	1.95	0.20	8.26	1.50	2.67	-
全メソッド呼出しの3%以上呼出しのあるメソッドを対象	1.95	0.20	7.97	4.50	2.67	-
すべてのメソッドを翻訳(予測値)	-	8.08	-	33.7	-	10.7

単位：Kbyte

べての実行時間を含んでいる．翻訳処理前の実行時間は，APP1 が 11.55 秒，APP2 が 90.03 秒，APP3 が 11.16 秒であった．

ネイティブコードへの翻訳対象を全メソッド呼出し回数の 10%を超える呼出しがあったメソッドとした場合，APP1 のメソッド 1 つが翻訳対象となった．このメソッドをネイティブコードに翻訳した場合，APP1 の実行時間は 3.21 秒になり約 3.5 倍高速になったが，APP2 および APP3 には効果がなかった．

翻訳対象を全メソッド呼出し回数の 5%以上呼出しがあるメソッドとした場合，APP1 では 10%のときと変化なかったが，APP2 では翻訳対象となるメソッドが 3 つになった．これら 3 つのメソッドを翻訳対象とした場合，APP2 の実行時間は 59.57 秒で約 1.5 倍高速に実行できた．

翻訳対象を全メソッド呼出し回数の 3%以上呼出しがあるメソッドとした場合，APP2 でさらに 3 つのメソッドが翻訳対象となった．これらのメソッドを翻訳した結果，APP2 の実行時間は 47.97 秒で，翻訳前に比べて約 1.9 倍高速になった．測定の結果から，速度向上比は APP1 で約 3.5 倍，APP2 で 1.5 倍から 1.9 倍程度の実行性能の改善が見られる．

APP1 では，10%の場合でも 3%でも性能に大差はない．これは APP1 では総和計算をするメソッド部分だけが頻繁に呼び出されており，ここでほとんどの処理が費やされているため，3%の場合でも性能向上に寄与しない．APP2 では，10%の場合には翻訳対象となるメソッドがなかったが，5%の場合には 1.5 倍，3%の場合には 1.9 倍となった．5%の場合には，三角比を求めるメソッドが翻訳対象となり，3%の場合には，座標を求めるメソッドが翻訳対象となった．APP3 は実行速度の改善が見られないが，これは APP3 で呼び出されるメソッドの使用頻度が少ないためである．

次にメモリサイズについて述べる．ネイティブコードへの翻訳対象の比率を変えたときの Java のアプリケーションサイズとネイティブコードのサイズを実測したものを**表 5** に示す．表 4 と同様に翻訳処理前と，全メソッド呼出し回数の 10%，5%，3%以上呼出し回数があるメソッドを翻訳対象としたものを示したもので，Java アプリケーションと変換されたネイティブコードの大きさを比較した．さらに Java アプリケーションをすべてネイティブコードに変換した場合のコードサイズについても予測した．単純に計算すると，すべてをネイティブコードに翻訳すると Java アプリケーションよりも 4 倍程度のコードサイズの増加を招く．本方式では部分的に翻訳するので，最悪でも 4 倍のコードサイズ増になる．本実装では，CISC アーキテクチャの M68000 を対象にしたが，RISC アーキテクチャを採用している PDA や組込みシステムではさらにコードサイズが増加することが予想される．

表 6 プロファイリング後のコードサイズと実行速度の比
Table 6 Ratio of code size and execution time.

	App1	App2	App3
アプリケーションサイズ	1.1 倍	1.2 倍	効果なし
実行速度	3.5 倍	1.5 倍	効果なし

表 5 を見ると，APP1 では 10%，5%，3%いずれの場合でも全体で 10%のコードサイズ増にとどまっている．APP1 では，総和を計算するメソッドだけが頻繁に呼び出されており，他のメソッドが翻訳対象とならなかったためである．APP2 では，10%では変化がないが，5%で 2 割増，3%で 5 割近いコードサイズの増加となる．

表 4 および表 5 の測定結果から，翻訳対象となるメソッドは全メソッド呼出し回数の 5%を超える呼出しがあったメソッドを対象とすることが速度向上とメモリ使用量のトレードオフで適切な閾値と考えた．全呼出し回数の 5%を超えるメソッドを翻訳対象として選んだときの実行コードのサイズと実行速度の向上比を**表 6** に示す．APP1 では 1 割のコードサイズ増で，1.5 倍の速度向上を達成できている．また，APP2 でも 2 割のコードサイズ増で，1.5 倍の実行速度向上が

表 7 プロファイリングのオーバヘッド
Table 7 Load of profiling on KVM.

	App1	App2	App3
メソッドカウンタなし	9.71	57.50	11.97
メソッドカウンタあり	11.51	90.03	11.16

単位：秒

可能になることが分かった．

次にメソッドの呼出し回数計測のオーバヘッドについて述べる．**表 7** に KVM の実行時間を示す．“メソッドカウンタなし”がオリジナルの KVM 上での実行時間，“メソッドカウンタあり”が本方式を採用した KVM 上での実行時間である．APP1 で 1.2 倍の実行時間増，APP2 で 1.5 倍の実行時間増となっている．これは実装が，JavaVM でメソッド呼出しがあるごとに直接プロファイリングデータベースに書き込みを行っているためである．この処理を JavaVM 終了時にプロファイリングデータベースに書き込みを行うようにすれば，メソッドカウンタのオーバヘッドを軽減できると考えている．

プロファイリングには，メソッドの呼出し回数ではなくメソッドやブロックの実行時間を計測した方が精度の良いプロファイリングができるのは周知の事実であるが，軽量な本方式でも数 10%のオーバヘッド増となるので，メソッドやブロックの実行時間を計測したときは，オーバヘッドが増加してしまう．本方式は，PDA 上でのオーバヘッドが小さいことが特徴となっており，今後ホスト PC 側のプロファイリング処理部を改良すれば，より良い精度でプロファイルできると考えている．

最後に，ホストとの同期，ホストでの翻訳処理のオーバヘッドについて述べる．今回の実装で用いた PDA では，ホストと PDA は RS232C で接続されている．RS232C は，19.2 kbps〜115 kbps の転送速度で使用されるが，本実装で計測したところ，修正された KVM 全体と AP を転送するのに，約 2 分を要し，実用上問題のない結果である．さらにホスト側で行われる処理は，プロファイルデータベース解析とクラスファイルの変更およびネイティブコードへの翻訳がある．翻訳処理部分が未完成であるために正確な数値は示せないが，KVM の再コンパイルに 35 秒程度かかり，Toba の翻訳時間は十数秒程度と予想されるので，ホスト PC 側での処理には約 1 分以内には処理が終了すると予想している．したがってホスト PC での同期処理には，全体で約 3 分程度で行えると考えている．今回の実装では，KVM を再コンパイルする方法で実装を行ったが，ネイティブメソッドだけを追加できる JavaVM を使用すれば，さらに転送時間は短くなると考えられる．

これらの結果から，本方式ではプロファイルを行うために 20%から 50%ほどのオーバヘッドがかかるが，翻訳処理後には 1 割から 2 割のサイズ増で，1.5 倍から 3.5 倍の実行速度改善が行え，本方式が有効であることが分かった．同時に，プロファイリングの精度やメソッドのカウント方式の改善などの課題があることが分かった．

7. おわりに

本稿では PDA 側の負荷を軽減でき，Java の特徴を失わない高速化の方式を提案し有効性を示した．本方式は既存の JavaVM を大幅に変更することなく実装ができ，本方式を用いた JavaVM は，次のような利点がある．

(1) 低性能な PDA のハードウェアに対応
実行時にネイティブコードへ翻訳したりプロファイリングを行って翻訳する方式ではなく，ホスト PC 側で翻訳を行う．したがって PDA 側でのメモリ使用容量が少なく，オーバヘッドも少ない高速化が行える．

(2) ユーザの使用状況を反映
PDA はユーザごとに使用方法が異なり，様々なアプリケーションが使用される．そのため本システムでは Java メソッドの使用頻度に応じて翻訳対象のメソッドを決定することで，効果的な高速化が行える．

(3) Java の特徴を保持
すべての Java アプリケーションをネイティブコードに翻訳するという方法とは異なり，本システムでは Java の特徴である動的呼出しや実行コードが小さいという特徴を保持した高速化が行える．

今後の課題として，プロファイリングの精度の向上とメソッドカウンタのオーバヘッドの削減がある．さらにネイティブメソッド用のインタフェースが用意されていない KVM に対して設計を行ったため，ネイティブメソッドを KVM に組み込むという実装に依存する処理が生じた．JNI は現在標準的なネイティブメソッド用のインタフェースであるが，PDA のようなハードウェアの制限があるものには適さない．したがって PDA などハードウェア性能が低い機器用のネイティブメソッドインタフェースについても提案し，本方式を他の JavaVM にも適用できるようにしたい．

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, F.: The Java Virtual Machine Specification, 2nd Edition. http://java.sun.com/docs/books/vmspec/2nd–edition/html/VMSpecTOC.doc.html, Sun Microsystems, Inc.
3) The JAVA HOTSPOT Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, Sun Microsystems, Inc.
4) GCJ. http://sources.redhat.com/java/, Red Hat, Inc.
5) 中村寿彦，白井和敏，中本幸一：Java メソッドとの混在可能なバイトコードコンパイラの実装と評価，情報処理学会研究会報告，No.98-OS-81, pp.149–154 (1998).
6) JAVA2 PLATFORM STANDARD EDITION Product Family. http://java.sun.com/j2se/, Sun Microsystems, Inc.
7) Java Native Interface 仕様. http://java.sun.com/products/jdk/1.2/ja/docs/ja/guide/jni/spec/jniTOC.doc.html, Sun Microsystems, Inc.
8) CLDC and the K Virtual Machine (KVM). http://java.sun.com/products/cldc/, Sun Microsystems, Inc.
9) Rhodes, N. and McKeehan, J.: Palm プログラミング，オライリー・ジャパン (1999).
10) 3com: PalmOS バイブル，日経 BP(1999).
11) Java2 Platform Micro Edition. http://java.sun.com/j2me/index.html, Sun Microsystems, Inc.
12) Toba. http://www.cs.arizona.edu/sumatra/toba/index.html. The University of Arizona.

(平成 12 年 12 月 18 日受付)
(平成 13 年 4 月 6 日採録)

有馬　　啓（学生会員）
1976 年生．1999 年東京農工大学工学部電子情報工学科卒業．同年東京農工大学大学院工学研究科電子情報工学専攻修士課程入学，2001 年 3 月同課程修了．Java 言語の処理系を通じ言語処理系やシステムソフトウェアに興味を持つ．2001 年 4 月より（株）PFU 勤務．

並木美太郎（正会員）
1984 年東京農工大学工学部数理情報工学科卒業．1986 年東京農工大学大学院工学研究科情報工学専攻修了．同年日立製作所基礎研究所入社，1988 年東京農工大学工学部数理情報工学科助手を経て，1993 年より同大学電子情報工学科助教授，1998 年同大学情報コミュニケーション工学科助教授．博士（工学）．OS やコンパイラ等のシステムソフトウェア，日本語情報処理等の研究開発に従事し，近年は PDA や組込み用システムソフトウェアに興味を持つ．ACM，IEEE 各会員．