

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

船田 雅史[†] 内田 純平[‡] 戸川 望^{†,‡,‡‡} 柳澤 政生[‡] 大附 辰夫[‡]

[†] 早稲田大学理工学部電子・情報通信学科

[‡] 早稲田大学理工学部コンピュータ・ネットワーク工学科

^{†‡} 北九州市立大学国際環境工学情報メディア工学科

^{‡‡} 早稲田大学理工学総合研究センター

〒 169-8555 東京都新宿区大久保 3-4-1

Tel: 03-3209-3211(5716), Fax: 03-3204-4875

E-mail: funada@yanagi.comm.waseda.ac.jp

あらまし 本稿では、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を提案する。本システムは Java 仮想マシンの実行中に、アプリケーションにおいて頻繁に実行されるメソッド(ホットメソッド)を検出する。ホットメソッドはコンパイラによってネイティブコードにコンパイルされ、ヒープ領域に格納される。ネイティブコードに使用するヒープ領域をプロファイラが決定することにより、ガーベッジコレクションの起動を抑制することが可能である。提案プロファイラは Java 仮想マシンの処理時間の 3% 程度のオーバーヘッドで動作可能である。ガーベッジコレクションを元の仮想マシンの 2 倍程度に抑え、プロファイリングによって得られたメソッドをコンパイルすることにより、平均で約 7 倍程度の高速化を実現した。

キーワード Java, 動的コンパイラ, プロファイラ, J2ME, ガーベッジコレクション

A Lightweight Profiling System of Java Dynamic Compilers for Hand-held Devices

Masashi FUNADA[†], Jumpei UCHIDA[‡], Nozomu TOGAWA^{†,‡,‡‡},
Masao YANAGISAWA[‡], and Tatsuo OHTSUKI[‡]

[†]Dept.of Electronics, Information and Communication Engineering, Waseda University

[‡]Dept.of Computer Science, Waseda University

^{†‡}Dept.of Information and Media Sciences, The University of Kitakyushu

^{‡‡}Advanced Research Institute for Science and Engineering, Waseda University

3-4-1 Okubo, Shinjuku, Tokyo 169-8555, Japan

Tel: +81-3-3209-3211(5716), Fax: +81-3-3204-4875

E-mail: funada@yanagi.comm.waseda.ac.jp

Abstract This paper proposes a lightweight profiling system of Java dynamic compiler for hand-held devices. The system detects the methods frequently invoked in application (hot method) during execution of a Java virtual machine. A hot method is compiled into a native code by the compiler, and is stored in a heap area. The profiler determines the heap area used for a native code, and it is possible to reduce a garbage collection. Our technique can profiles method informations with 3% overhead of the processing time of a Java virtual machine. By compiling the hot method, as a result, we achieve approximately 7 times speedup in average, by suppressing a garbage collections to approximately 2 times of the original virtual machine.

Key words Java, dynamic compiler, profiler, J2ME, Garbage Collection

1 はじめに

近年、情報通信技術および LSI 技術の発展により、携帯電話や PDA 等の携帯情報端末が普及するようになった。携帯電話では、Java プログラムを実行可能であり、ユーザーが作成したアプリケーションを携帯電話上で実行できる。

携帯機器に実装される Java 仮想マシンは、バイトコードを逐次解釈して実行する。そのため、C 言語等で記述されたものと比較して、Java アプリケーションの実行は低速である。実行速度を改善する手段として、プログラムにおいて頻繁に実行されるメソッド (ホットスポット) を選択的にコンパイルする、動的コンパイラが注目されている。

動的コンパイラの研究として、文献 [3, 4, 6] が挙げられる。これらの手法は、サーバ用途の Java 仮想マシンを対象としたものであるため、アプリケーションのプロファイリング、ホットスポットのコンパイルに多くのメモリを使用できる。消費メモリの点で、これらの手法を組み込み機器の Java 仮想マシンに対して、適用することは不可能である。

携帯機器を対象とした Java 仮想マシンの高速化手法は動的コンパイラの機能を制限することによって実現される。文献 [2] の手法は、組み込み機器はプロファイリングを行なって、ホットスポットを検出し、ホットスポット部を高速化した Java 仮想マシンを生成する手法である。プロファイリングによる時間的なオーバーヘッドより、ホットスポットのコンパイルは組み込み機器ではなく別のサーバが行う。この手法はサーバとの通信のオーバーヘッドが存在し、送信するのはアプリケーションに適した Java 仮想マシンであり、他のアプリケーションの高速化には対応することが不可能である。また、プロファイリングのためのメモリ不足の問題から、アプリケーションの種類によって、予めホットスポットを予測してコンパイルする手法 [7] も存在する。この手法は事前のプロファイリングによってホットスポットを推定するため、他アプリケーションに対応することが難しい。文献 [5] の手法はコンパイラを搭載して高速化を実現するものであるが、プロファイラによる空間的オーバーヘッドより、使用するメモリ量を従来の 2 倍必要とする。

本稿では、携帯機器においても少ない消費メモリで動作するプロファイラを提案する。これまで提案されてきた手法はプロファイラによる時間的、空間的オーバーヘッドにより、動的コンパイラの機能を制限し、アプリケーションに対応した高速化が困難であった。本システムは、時間的、空間的オーバーヘッドが少ないプロファイラによって、アプリケーションにおけるホットスポットを決定する。本システムは Java 仮想マシンのヒ-

ブ領域に、適切なネイティブコードの割当てを行うことにより、消費メモリ量の削減とアプリケーション実行速度の向上を両立させる。本プロファイリングシステムを組み込み機器を対象とした Java 仮想マシンの KVM に実装し、その有効性を評価する。

2 既存研究

プロファイリング方法は大きく分けて次の二つの方式が存在する [8]。

- 計測コード手法
- サンプリング手法

計測コード手法は、プロファイリングを行うための計測用コードを、プロファイル対象のプログラムに埋め込んでプロファイルする。プログラムの実行中に、計測用コードに含まれるカウンタ値を増やすことにより、プロファイル情報を得る。メソッドの起動回数のカウントとスタックトレースを計測コード手法にて行う場合、各メソッドの起動ごとに、プロファイリングのための計測用コードを実行する必要がある。そのため、計測用のコードが大きくなればなるほど、プロファイリングのためのオーバーヘッドが増大し、ホットスポット部に計測用コードが含まれる場合や、大規模なプログラムでも、オーバーヘッドが増大する。計測コード手法はプロファイリングのオーバーヘッドが大きいものの、サンプリング手法よりもホットスポットを検出しやすい。

サンプリング手法 [1] は、プロファイリングを一定周期で行う方法である。プログラム実行中に、プロファイリングを行うためのスレッドを別に動作させてプロファイル情報の収集を行う。プロファイリングを行うスレッドは、プロファイル情報を得るために、プログラムで実行されているすべてのスレッドを一定の周期ごとに停止させて、各スレッドが呼び出しているメソッド、スタックトレースの情報を得る。サンプリング手法は、周期的にプロファイリングを実行するので、前述の計測コード手法と比べて、アプリケーションの実行の妨げになりにくい。なぜならば、サンプリングの回数はメソッドの呼出し回数よりも少ないためである。しかし、サンプリング手法はすべてのメソッドを起動することに監視するのではなく、一定周期ごとにプログラムを停止させてプロファイルするため、計測コードによる手法よりもホットスポットは検出しにくい。

携帯機器を対象とした高速化手法である、文献 [2, 7] のいずれの手法も、計測コード手法を用いたプロファイリングを使用したものである。計測コード手法によるプロファイリングは、ホットスポットを検出しやすいが、時間的、空間的オーバー

ヘッドの問題から動的コンパイラの機能を制限する必要があった。また、携帯機器を対象とした動的コンパイラにおいてサンプリング手法を適用した場合、コンパイラとプロファイラを仮想マシン上に搭載可能であるが、ネイティブコードによるメモリ消費の問題から、携帯機器のメモリ量を増やす必要があった。

提案手法はサンプリング手法によるプロファイリングとヒープ領域のプロファイリングによって上記の問題を解決している。

3 携帯機器を対象とした Java 動的コンパイルシステム

提案手法が対象とする動的コンパイルシステムを図 1 に示す。動的コンパイルシステムは Java 仮想マシン内に含まれ (1) インタプリタ (2) プロファイラ (3) コンパイラによって構成される。

3.1 インタプリタ

インタプリタは、バイトコードの解釈・実行をする。バイトコードはアーキテクチャ非依存の、スタックマシンを対象とした命令である。バイトコードはネイティブコードと比較して、空間的なサイズが 1/10 程度である。

インタプリタはバイトコードの解釈・実行とともに、コンパイラによって生成されたネイティブコードを実行する。ネイティブコードはホットスポットと推定されたメソッドごとに生成される。インタプリタは、メソッドがネイティブコードを保持するかを確認する。メソッドがネイティブコードを保持している場合、バイトコードの解釈・実行をせずに、ネイティブコードを直接実行する。

3.2 プロファイラ

プロファイラはアプリケーション実行中にプロファイルする。プロファイル情報はホットスポットの決定のために、コンパイラによって使用される。プロファイラは、プロファイリングのために、インタプリタに対して一定周期で割り込みをかける。

プロファイラはプロファイル情報を作成して、アプリケーションのホットスポットを検出する必要がある。プロファイリングの情報として、メソッドの呼び出し回数等を収集する。プロファイルする情報については、4.2 節と 4.3 節にて述べる。プロファイリングの情報は Java 仮想マシン内のヒープ領域に保持される。アプリケーションの実行中に一定周期で、プロファイルすることにより、アプリケーションにおけるホットスポットを検出し、ホッ

トメソッドの情報はコンパイラに渡される。ホットメソッドだけをコンパイルすることにより、消費メモリ量の削減と実行時間の向上を実現する。

3.3 コンパイラ

コンパイラはプロファイラから得られた、ホットスポットの情報を元に、メソッドをコンパイルする。コンパイルによって生成されたネイティブコードは、Java 仮想マシン内のヒープ領域に格納される。コンパイル処理は Java 仮想マシンの開始時に行なわれ、ネイティブコードは Java 仮想マシンが終了するまで、ヒープ領域内に保持される。

使用メモリに制限がある組み込み機器において、既存の動的コンパイラで使用される最適化手法を、携帯機器を対象とした動的コンパイラに適用するのは、使用メモリと実行時間の点において不可能である。組み込み機器において最適化処理を行う場合、共通部分式の削除や定数伝播、レジスタ割り当てや命令スケジューリング等の最適化処理を、全て行って最適化するのは現実的ではない。対象機器にコンパイラを組み込むという前提において、最適化処理をすべて採用すると、使用メモリ量が大きくなり、コンパイルに必要な処理も大きくなってしまふ。

そこで、バイトコードの 1 命令をネイティブコードへと変換するアルゴリズムによってコンパイルする。このアルゴリズムはバイトコードからネイティブコードへのマッピングであるため、少ないメモリで高速にコンパイルが可能である。しかし、単純にバイトコード命令をネイティブコードへ変換するだけなので最適化はされない。そのため、コンパイラはネイティブコードへの変換に加えて、軽量の最適化処理を施す。

4 軽量プロファイラ

動的コンパイルシステムはホットスポットをコンパイルするために、プロファイラによってホットスポットを検出する必要がある。携帯機器のようなハードウェアに制約がある場合、動的コンパイルシステムを実現するには、プロファイラは時間的、空間的に軽量でなければならない。従って以下に示すようなプロファイラを提案する。

4.1 プロファイラの目的

プロファイラがアプリケーション実行中にプロファイリングを行なう目的は以下の二つになる。

- ホットスポットの検出
- 使用可能なヒープ領域の調査

Java仮想マシン

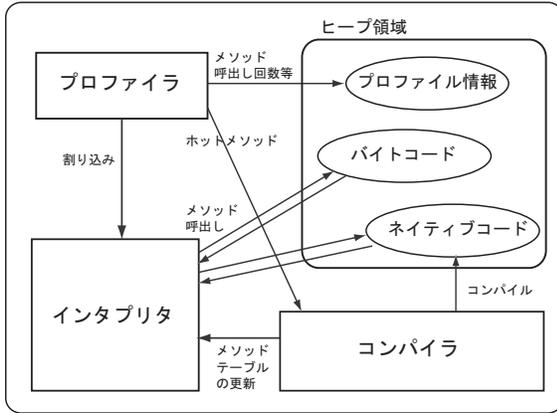


図 1: 携帯機器を対象とした動的コンパイルシステム。

ホットスポットの検出の目的は、アプリケーションのホットスポットを選択的にコンパイルするためである。ホットスポットとなるメソッドは、呼出し回数の多いメソッドである。プロファイラは頻繁に呼出されるメソッドを検出し、そのメソッドをホットスポットと推定する。

使用可能なヒープ領域を調査する目的は、コンパイルによって生成されるネイティブコードを格納する領域を見積るためである。ネイティブコードを格納するための領域は、Java 仮想マシン内のヒープ領域から、新たに確保される。ネイティブコード格納によって、Java 仮想マシンが使用できるヒープ領域が少なくなった場合、Java 仮想マシンはガーベジコレクションを起動する。ガーベジコレクションが発生すると、Java 仮想マシンは割り込みを受けつけることが出来なくなるため、ガーベジコレクションが頻繁に起動される事は好ましくない。このような状況では、ガーベジコレクションによるオーバーヘッドが増大し、結果として実行時間は遅くなる。また、ヒープ領域をネイティブコードの格納のために、必要以上に使用すると、インスタンス等に使用するメモリを十分に確保できずに、アプリケーションが実行できなくなる。携帯機器のような限られたメモリ使用量において、ネイティブコード割り当てによる、高速化を実現するために、ヒープ領域の情報をプロファイルする。

4.2 ホットスポット検出プロファイリング

コンパイルはメソッドを対象とする。ホットメソッドを決定するために、以下の情報が必要となる。

- 呼出し頻度の高いメソッド

STEP1. スレッドの切換え時にインタプリタを停止させる。

STEP2. カレントフレームにおけるメソッドの情報を取得し、取得したメソッドがプロファイル情報に存在する場合、STEP4 へ。

STEP3. プロファイル情報にメソッドを追加し、メソッドのコードサイズを取得する。

STEP4. メソッドの呼出し回数を増分する。

STEP5. カレントフレームの上位フレームにおけるメソッドの情報を取得する。取得したメソッドがプロファイル情報に存在する場合、STEP7 へ。

STEP6. プロファイル情報にメソッドを追加し、メソッドのコードサイズを取得する。

STEP7. メソッドの結合度を増分する。

STEP8. 次のスレッドに制御を渡し、バイトコードの解釈・実行を再開する。

図 2: プロファイリングアルゴリズム。

- 結合度の高いメソッド

呼出し頻度の高いメソッドは実行される回数の多いメソッドである。呼出し頻度の高いメソッドはホットスポットであり、コンパイルの対象となる。結合度の高いメソッドとはあるメソッドを呼出していたメソッドとの結合度を示す。呼出し元のメソッドとの結合度が強いメソッドは、呼出し元のメソッドに対してインライン展開される。

上記の情報をプロファイリングするために、提案手法はスレッドの切換え時にプロファイリングをする。提案プロファイリングアルゴリズムを図 2 に示す。

提案プロファイリングシステムは、スレッドを用いて一定周期のサンプリングによってプロファイルを行う。スレッドの切換えまでの時間は各スレッドで大差がないため、一定周期のプロファイリングになる。呼出し頻度の高いメソッドを検出するにあたって、計測コード手法のような詳細なプロファイルでは必要ない。呼出し頻度の高いメソッドは呼出し回数の多さから、サンプリングによるプロファイリングでも検出されやすい。

インタプリタとプロファイラの間を関係を図 3 に示す。

STEP2. において、プロファイラは、スレッドが呼び出していたメソッドを決定するために、スレッド内に存在する Java スタック内で一番上位にあるフレーム¹、即ちカレントフレームを読み出して、フレーム内に格納されたメソッド情報をプロファイルする。ここで、カレントフレームが呼び出しているメソッドが、ホットスポットの判定対

¹フレームはメソッドの起動ごとに生成され、自身のローカル変数配列、自身のオペランドスタック、メソッドのクラスに対する実行時コンスタントプールへの参照が保持される

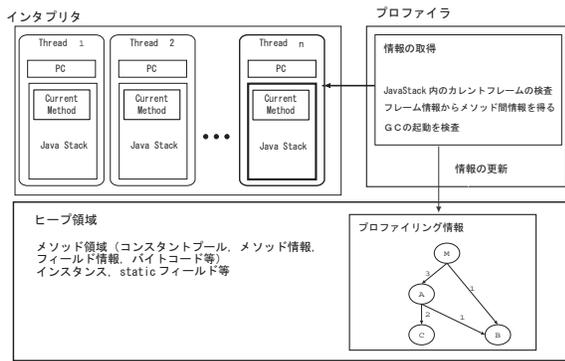


図 3: インタプリタとプロファイラの関係 .

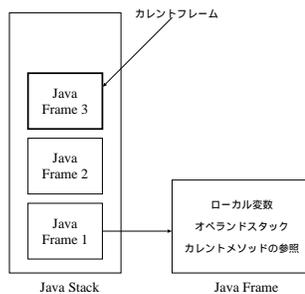


図 4: Java スタックと Java フレームの様子 .

象となる . スレッド , Java スタック , Java フレームの関係を図 4 に示す .

また , STEP5. においてプロファイラはメソッド間の情報を得るために , Java スタック内のフレームを読み出して , メソッドの相互関係をプロファイルする . メソッドの相互関係より , メソッド単位のホットパスが決定できる .

メソッドのサンプリング回数と呼出し元メソッドの情報を格納するために , グラフ構造を用いてプロファイルデータを格納する . グラフ構造を用いて , プロファイル情報を格納する様子を図 5 に示す . STEP3,STEP6 において , プロファイル情報にメソッドを追加する場合 , メソッドのコードサイズを記録する . コードサイズはメソッドのバイトコードの長さである . それによってコンパイル時の生成ネイティブコード量の見積もる .

プロファイル結果は Java 仮想マシン終了後に , ファイルとして保存される . 次回起動時に , プロファイル結果を読み込んで , コンパイルを行なう . コンパイル後もプロファイリングは継続し , プロファイルデータの更新を続ける .

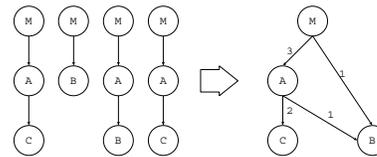


図 5: メソッド間情報とメソッド起動割合を格納するのに用いるデータ構造 .

4.3 ヒープ領域のプロファイリング

ホットスポットはコンパイラによって , バイトコードからネイティブコードへとコンパイルされる . ネイティブコードを格納するための領域を , Java 仮想マシン内のヒープ領域から , 新たに確保する必要がある . ヒープ領域が消費されて , 使用できるヒープ領域が少なくなった場合 , Java 仮想マシンはガーベッジコレクションを起動する . ガーベッジコレクションはヒープ領域から , 再利用できるメモリ領域を見つけ出す処理である . ガーベッジコレクションが発生すると , Java 仮想マシンは割り込みを受けつけることが出来なくなるため , ガーベッジコレクションが頻繁に起動される事は好ましくない . ガーベッジコレクションの起動回数が増えた場合 , ユーザーイベント等の割り込みの応答性が悪くなる . また , アプリケーションで必要とするヒープ領域を確保できない場合 , アプリケーションの実行が不可能になる .

ガーベッジコレクションの起動回数をできるだけ少なくして , より多くのヒープ領域を使用できる事が望ましい . ガーベッジコレクションの起動回数は , 実行アプリケーションによって依存し , 実行アプリケーションによって , 使用するヒープ領域も依存する . そこで , ガーベッジコレクション起動後毎に , プロファイラに以下の情報を取得させる .

- 使用可能ヒープ領域

Java 仮想マシンにおけるヒープ領域は一定サイズのセルによって管理されている . セルには使用可能なフラグが付加されており , ガーベッジコレクション時にこのフラグによって , Java 仮想マシンはセルの解放・保持を行なう . ガーベッジコレクション終了後に使用可能ヒープ領域は使用可能なセルの合計によって求める . プロファイラは使用可能ヒープ領域の最小値 H_{min} を保持する . プロファイラは , ガーベッジコレクションの起動毎に取得する使用可能ヒープ領域 H が , H_{min} よりも少ない場合 , $H_{min} = H$ とする .

STEP1. プロファイル情報から、一定数以上呼ばれたメソッドの集合 MH を生成。

STEP2. 集合 MH をプロファイルのヒット数によって、優先順位をつける。

STEP3. 優先順位の上位のメソッドから、コンパイルをする。

STEP4. コンパイルによって生成されるネイティブコード長の合計が、割当てヒープ量よりも小さければ、STEP.3へ

STEP5. 生成されたネイティブコードを割り当てる。

図 6: ホットメソッド決定アルゴリズム。

5 ホットスポットのコンパイル

コンパイラはプロファイラによって得られた情報から、ホットスポットとなるメソッドを決定する。ホットメソッドの決定アルゴリズムを図 6 に示す。

ホットメソッドを決定するために、プロファイルによって得られた全メソッド数を N 、メソッドのサンプル数を s_i とする。

全サンプル数 S は以下の式で求められる。

$$S = \sum_{i=1}^n s_i \quad (1)$$

あるメソッド m_i がホットスポットであるとは、そのメソッドが以下の式を満たすこととする。

$$\frac{h_i}{S} > T_{HotMethod} \quad (2)$$

ここで、 $T_{HotMethod}$ はホットメソッドと判定するための閾値である。上式を満たすメソッド m_i を全てコンパイルするのではなく、上式を満たすメソッドの内、ヒット数 h_i の値が高い順に優先順位をつける。

メソッド m_i をコンパイルした時のネイティブコード長を nc_i とする。 h_i の値が高いメソッドから順にコンパイルを行い、その合計が T_{Heap} を越えた時に、コンパイルを終了する。ネイティブコードによって、合計が越えたメソッドはバイトコードのままとする。 T_{Heap} は生成されたネイティブメソッドを格納するために使用可能なヒープ領域の量を表す。 T_{Heap} は 4.3 節にて述べた H_{min} によって以下のように表わされる。

$$T_{Heap} = H_{min} \times H_{rate} \quad (3)$$

H_{rate} はヒープ領域の使用割合を表わす。 H_{rate} の値はアプリケーションによって動的に求めることが困難であるため、計算機実験によって値を決定する。

コンパイラは Java 仮想マシンの起動時に、プロファイラによって得られた情報を元にコンパイルし、生成されたネイティブコードをヒープ領域に

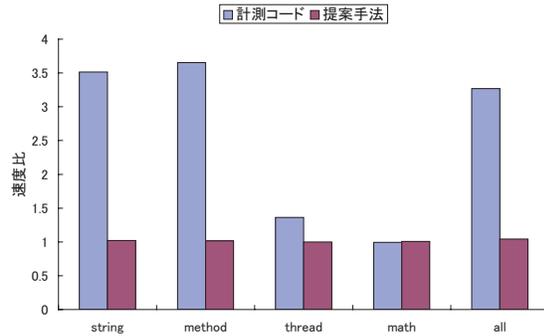


図 7: プロファイラの時間的オーバーヘッド。

格納し、メソッドとのリンクを行なう。プロファイル情報は、Java 仮想マシンの終了時に、ファイルとして保存され、アプリケーション毎に作成される。

6 計算機実験結果

提案手法は C 言語によって実装し、提案組込み機器を対象とした、Java 仮想マシンである KVM(K-Virtual-Machine) 上で動作する。計算機実験は、Pentium 4 2.66 GHz、メモリ 1024 MB、OS は Debian Linux kernel 2.4.19、gcc 2.95.4 最適化レベル O2 の環境で行なった。また、KVM は CLDC1.0.4 の Reference Implementation を使用した。

プロファイラには、以下に挙げるベンチマークプログラムを入力した。

- 算術演算
- メソッドテスト (小規模メソッド呼出し)
- 文字列計算
- マルチスレッド

算術計算プログラムはカオス計算を行なうプログラムである。算術計算プログラムはメソッドが 6 つ程度で構成される、ホットスポットが特定しやすいプログラムである。メソッドテストプログラムはコードサイズが 10 数バイト程度の小規模のメソッドを、30 程度で構成されるプログラムであり、算術計算プログラムと比較して、ホットスポットが特定しにくいプログラムである。文字列計算プログラムは、Java API における java.lang.String クラスのメソッドを呼び出して文字列演算メソッドを多用するプログラムである。文字列計算プログラムは文字列型の生成を繰り返すので、ガーベッジコレクションが発生しやすいプログラムである。

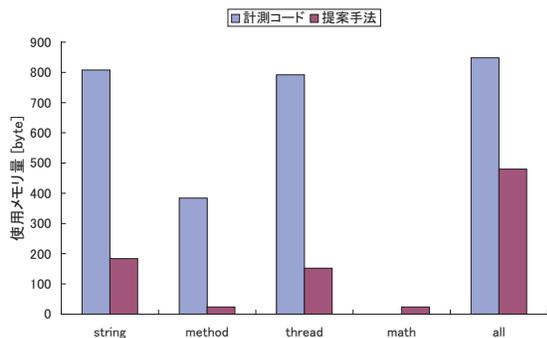


図 8: プロファイラが使用するメモリ量 .

6.1 プロファイラによるオーバーヘッド

提案手法における時間的オーバーヘッドを図 7 に、空間的オーバーヘッドを図 8 に示す。提案手法との比較のため、計測コードによるプロファイル結果も示す。図中の速度比とは、元の KVM との実行速度の比を表わす。ここで、提案手法の結果はコンパイルによるネイティブコード実行を含まない。コンパイルによる高速化をせずに、提案プロファイラのオーバーヘッドのみを測定している。

時間的オーバーヘッド

図 7 の結果から、計測コードによる手法は平均で約 2 倍程度のオーバーヘッドになるのに対して、提案手法は元の KVM の約 3% 程度のオーバーヘッドで動作する。

計測コードによる手法は、メソッドの起動ごとにプロファイリングされるため、提案手法の方がオーバーヘッドが少ない結果となった。ただし、メソッドの呼出し回数が少ない算術計算プログラムに関しては、計測コードによる手法が良い結果となっている。

空間的オーバーヘッド

図 8 の結果から、提案手法は計測コードによる手法よりも少ないメモリ量で、プロファイリングが可能である。特にマルチスレッドプログラムに関しては、計測コードによる手法の 5 分の 1 程度のメモリ量で動作する。20k バイト程度の規模のアプリケーションのプロファイルに使用するメモリ量は、500 バイト程度である。

6.2 プロファイラの評価

ネイティブコード割当てによるガーベッジコレクションの変化の様子を図 9 に示す。プロファイラによって得られた、ホットスポットをコンパイルした場合の実行速度比とすべてのメソッドをコンパイルした場合の実行速度比を図 10 に示す。比

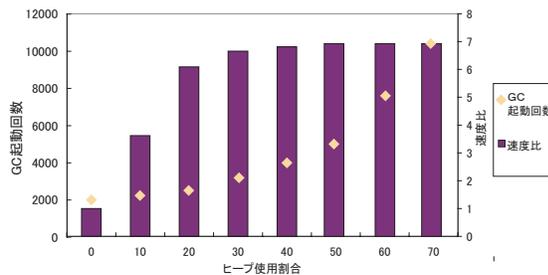


図 9: ガーベッジコレクションの変化 .

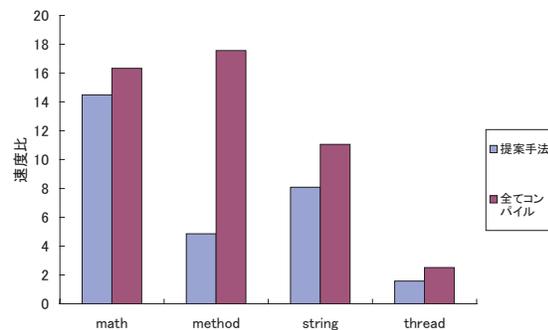


図 10: コンパイルによる速度向上 .

較の対象は、アプリケーションをインタプリタ実行した結果である。

ガーベッジコレクションの変化

図 9 からヒープ領域の使用割合に比例して、ガーベッジコレクションの起動回数が増えることが分かる。また、実行速度はヒープ領域の使用割合が高くなるに従って向上するが、ヒープ領域の使用割合が 30% を越えるあたりから収束していくことが分かる。

この結果より、ネイティブコード割当てのためのヒープ領域は、使用可能なヒープ領域の 30% 程度が望ましいと言える。30% を越えてネイティブコードを割当てると、実行速度の向上はあるものの、ガーベッジコレクションの起動回数が多くなり、ガーベッジコレクションによるオーバーヘッドが増えていくと思われる。

実行速度

計算機実験において、図 6 中の割当てヒープ量は、使用可能なヒープ領域の 30% を割り当てた。各ベンチマークにおいて、コンパイルしたメソッド数、プロファイラによって決定された、メソッドのネイティブコード量、およびすべてのメソッドをコンパイルした場合のネイティブコード量を表 1 に示す。ひとつのメソッドから生成されるネイティブコード量は、そのメソッドを構成するバ

表 1: ネイティブコード

ベンチマーク	コンパイル数	ホットスポット のネイティブ コード量 [byte]	全ネイティブコード量 [byte]
算術演算	1	1356	1702
メソッドテスト	8	6028	44280
文字列計算	6	10120	28704
マルチスレッド	4	12100	20402

イトコード量の約 10 倍から 20 倍程度であった。

算術計算は約 14.5 倍の高速化に成功している。高速化の度合いでは、4 つのベンチマークの中で最も高い数値である。これは、算術計算はホットスポットが推定しやすいプログラムであるためである。そのため、全てコンパイルして実行した結果と比較して、大差のない結果となっている。メソッドテストでは約 5 倍の高速化という結果になった。全てコンパイルした結果と比較すると、4 つのベンチマークの中で最も低い数値である。これは、メソッドテストは多くの小規模なメソッド呼出しで構成されるアプリケーションであるため、ホットメソッドを決定するのが困難であるためである。ただし、全てコンパイルしたアプリケーションはネイティブコード量が、多量であるため、ヒープの割当てが出来ずに、仮想マシンが異常終了した。文字列計算は約 8.1 倍の高速化に成功している。全てコンパイルして実行した結果の約 11 倍の実行速度と比較して、大差のない結果となっている。マルチスレッドは約 1.5 倍の高速化に成功している。高速化の度合いは低い結果となっているが、全てコンパイルした結果も約 2.5 倍であるため、ホットメソッドを検出している。

提案手法はプロファイルの時間的、空間的オーバーヘッドが微少であり、ヒープ領域のプロファイリングによってガーベッジコレクションを抑え高速化していることが確認できた。

7 おわりに

本稿では、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を提案した。今後の課題として、メソッドを基本ブロックに分割し、基本ブロックを対象にプロファイリング、コンパイルを検討することが挙げられる。謝辞 本研究を進めるにあたり、有用な議論、討論をいただいた株式会社ルネサス テクノロジ 中屋雅夫氏、三菱電機株式会社 坂本守氏、高橋克英氏に感謝いたします。

参考文献

- [1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. T. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Wehl, "Continuous profiling: Where have all the cycles gone?," *ACM Trans. Computer Systems*, Vol.15, No.4, pp. 357–390, November 1997.
- [2] 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, June 2001.
- [3] M. G. Bruke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. JSerrano, V. C. Serrano, H. Srinivasan, and J. Whaley, "The jalapeno optimizing compiler for Java," in *Proc. ACM SIGPLAN Java Grande*, pp. 129–141, June 1999.
- [4] M. Cierniak, G. Y. Lueh, and J. M. Stichnoth, "Practicing JUDO: Java under dynamic optimizations," in *Proc. ACM SIGPLAN Programming Language Design and Implementation*, pp. 166–179, 2000.
- [5] Sun Microsystems, *The CLDC HotSpot Implementation Virtual Machine*, http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf, 2003.
- [6] T. Sunagawa, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a Java just-in-time compiler," in *Proc. ACM SIGPLAN Object Oriented Programming Systems Languages and Applications*, pp. 180–194, 2001.
- [7] 高橋 克英, 清原 良三, 坂本 守, "携帯端末向け Java の高速化手法の検討," *情報研報*, MBL 022-011, October, 2002.
- [8] J. Whaley, "A portable sampling-based profiler for Java virtual machines," in *Proc. ACM SIGPLAN Java Grande*, pp. 78–87, 2000.