システムソフトウェアと
オペレーティング・システム 81－26
（１９９８．５．７）

Javaメソッドとの混在実行可能なバイトコードコンパイラの実装と評価

中村 寿彦　臼井 和敏　中本 幸一

NEC ソフトウェアデザイン研究所

E-mail:{tosihiko, usui, nakamoto}@ccs.mt.nec.co.jp

概要

Javaが広く使われるようになってきた。特に、最近、工業用コンピュータや携帯電話など組込み機器に、Javaの実行環境を備える動きが広まりつつある。しかし、実用的に使用するにあたって、実行速度の遅さが問題となっている。

組込み機器向けという観点から、メモリ使用量を押さえることと、Javaのポータビリティをなるべく損なわないということを念頭に置きつつ、実行速度問題の解決策として、我々は、実行のボトルネックとなるクラスファイルのみを事前にコンパイルし、実行時、この部分をネイティブコードで実行し、その他のクラスファイルは、そのままで実行するという混在実行方式を実装した。

実装の後、Java単独で実行する場合との比較評価を行なった。その結果、この方式で実行速度向上に効果をあげるがあることが分かった。

Byte Code Compiler executed with Java Method: Implementation and Evaluation

Toshihiko NAKAMURA,　Kazutoshi USUI
Yukikazu NAKAMOTO

Software Design Laboratories, NEC Corporation

E-mail:{tosihiko, usui, nakamoto}@ccs.mt.nec.co.jp

abstract

Java has been used widely. Particularly, Java execution environment starts to be prepare with embedded appliances, such as industrial computers, mobile phone. But It becomes problem that Java programs are executed too slowly for practical use.

As this solution for embedded appliances, we implemented the byte compiler which adopting pre-compile method to limit amount of memory and which is easy to use.

We evaluate this byte compiler by comparing with Java method execution. As the result of this evaluation, we realized that it is effective to reduction the time of execution.

1 はじめに

Java[1]が広く使われるようになってきた。特に、最近、工業用コンピュータや携帯電話などの組込み機器に、Javaの実行環境を備える動きが広まりつつある[1]。組み込み機器向けにJavaの実行環境が使われる理由としては、

- 一度コンパイルして出来たクラスファイルは、Java実行環境さえ用意すれば、CPU、OSの種類を問わず実行できる
- ネットワークを介してクラスファイルの取得、実行が可能である

などの点をあげることが出来る。

このような優れた特徴を持ってはいるが、実用的に使用するにあたっては、いくつかの問題点が指摘されている。その中でも、特に重大な問題は、C言語などのコンパイル方式と比較した実行速度に関することである。実行速度の遅さは、Javaが出現した当初から指摘され、その対策として、さまざまな方式が考えられてきた。それらを大きく分類すると、次の2つに分類できる。

- 実行時にクラスファイルネイティブコードにコンパイルし、実行する方式。Just In Timeコンパイラ(以下、JITと略記)と呼ばれている[2]。
- 実行前にコンパイルする方式。クラスファイルを入力とするバイトコードコンパイラと、Javaソースを入力とするネイティブコンパイラがある。

パーソナルコンピュータや、ワークステーションなどのメモリを比較的多く積んだマシンの場合、実行時にコンパイルする方式でも問題ない。しかし、組み込み機器の場合、実時間性や、メモリ量をいかに少なく出来るかということも重要なことである。JIT方式では、実行時にコンパイラ分のメモリを余分に必要とするので、事前コンパイルの方式をとらざるを得ない。しかし、現状存在する事前コンパイルの方式では、実行するべきコードをすべてコンパイルし、ネイティブコードに置き換える必要があり、Javaのポータビリティを損なうことにつながる。

そこで、我々は、事前コンパイル方式を採用しつつも、Javaのポータビリティをなるべく損なわないような方式として、実行のボトルネックとなるクラスファイルのみを事前にコンパイルし、実行時、この部分をネイティブコードで実行し、その他のクラスファイルは、そのままで実行するという混在実行方式のバイトコードコンパイラを実装した。

実装したバイトコードコンパイラは、NECのEWS4800上で動作し、アセンブラを出力する。アセンブルし、生成したネイティブコードは、MIPSのR4000系アーキテクチャのEWS上、または、WebPanelと呼ばれる組み込み機器上で実行できる。これらの機器を使って、実際にJava単独で実行する場合と比較評価を行なった。

以下では、第2章で、我々の実装した混在実行方式の概要を説明し、第3章で、その評価結果を紹介し、最後に評価結果から考えられる今後の方針を述べる。

2 コンパイラ、実行環境の実装

2.1 コンパイラ

JavaVMは、スタックマシンアーキテクチャを採用しており、演算や、メソッド間の引数、戻り値も、スタックを利用してい

[1]Javaは米国およびその他の国における米国Sun-Microsystems, Inc.の登録商標です

る [3]。

バイトコードコンパイラの場合、実行中にコンパイルするJITとは異なり、コンパイルにある程度のリソースを使うことが出来るため、コンパイル時に、Javaのスタックをエミュレートして、命令のツリー構造を作成する。このツリー構造を用いて、レジスタを割り付け、コードを生成している。例を用いたツリー生成の概要を図1に示す。

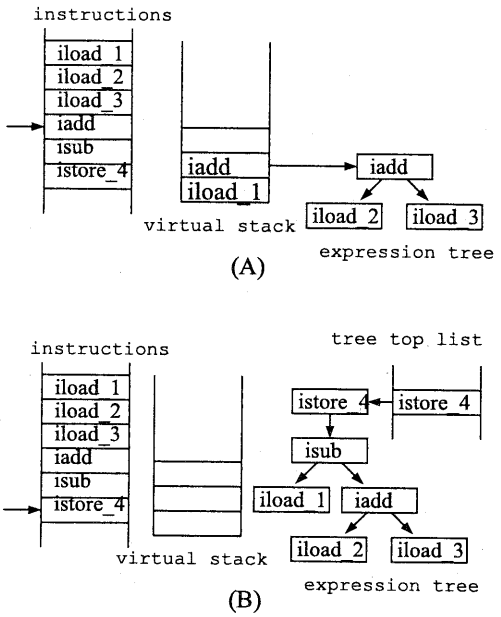

図 1: スタックエミュレートによるツリー作成の概要

2.2 混在実行方式

我々の実装したバイトコードコンパイラは、コンパイルドメソッドと、Javaメソッドが混在して実行できる点に特徴がある。その概要を図2に示す。

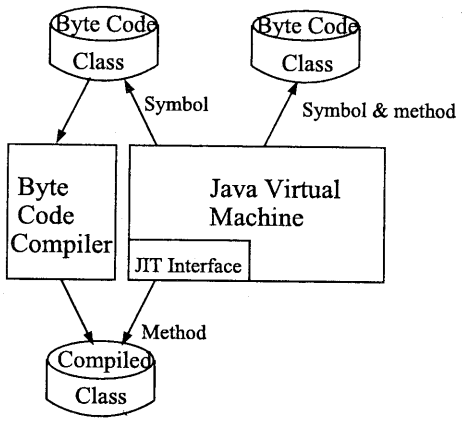

図 2: バイトコードコンパイラの概要

このバイトコードコンパイラは、Javaのクラスファイルを入力とし、クラスファイル単位にアセンブリコードを出力する。出力したコードは、市販のアセンブラを用いてロードモジュールにする。実行時には、このロードモジュールとクラスファイルがJava仮想マシン (以下、JavaVMと略記) を通じて読み込まれ、実行される。

混在実行のために必要となるJavaVMとコンパイルドメソッドとのインタフェースには、JIT用に用意されたAPI(JIT Compiler API)[4] を利用している。

また、クラス変数などのシンボル類に関しては、コンパイルしたクラスに関しても、もとのクラスファイルをロードし、すべてJavaVMで管理する方式を採用した。これは、混在実行により、シンボル類がJavaメソッドからもアクセスされる可能性があるからである。元のクラスファイルには、JavaVM本体のコンパイルドコード属

性と同じ属性をつけることで、バイトコード、メソッド変数などの不必要なものをロードしないようにできる。これにより、メモリの浪費を防ぐことが出来る。

以下の2節では、この混在実行方式のなかで処理速度に影響を与えるメソッド呼び出しとシンボル管理を取り上げ、詳細を述べる。

2.3 メソッド呼び出し

混在実行により、Javaメソッドからコンパイルドメソッドの呼び出し、逆に、コンパイルドメソッドからJavaメソッドの呼び出しが必要となる。これらには、それぞれ、JIT Compiler APIのフック関数や、JNI Interface[5]が利用できる。しかし、JNI Interfaceに関しては、処理速度の問題があるため、代わりにJIT Compiler APIが提供する関数群を使用した。

メソッド呼び出しで問題となるのが、引数、戻り値の管理方式の違いである。Javaでは、Javaのスタックを使って引数の受け渡しをしており[6]、Javaプログラム同士では、高速に引数が渡せるようになっている。

我々のバイトコードコンパイラでは、ネイティブメソッドで扱うフレームをJavaのスタックと同じ形式で扱い、引数、戻り値の引き渡しに該当領域をブロックごとコピーする方式を採用した。これにより、Javaメソッドとコンパイルドメソッドの切り替えによるオーバヘッドを最小限に押さえることができた。

2.4 シンボル管理

混在実行を実現させるため、クラス変数などのシンボル類、オブジェクト類に関しては、すべてJavaVMで管理する方式を採用している。Javaでは、実行時、そのシンボルへの最初のアクセス時に、シンボル解決が行われ、シンボル管理テーブルのIDと、実際のアドレスが結び付けられる。2度目の呼び出し以降、シンボル管理テーブル自体に解決処理を省略する機構があるが、JavaVMでは、より高速化するために、バイトコード命令を書き換えて、解決処理を省略するという方法を取っている。

コンパイルドコードでも同様の解決処理が必要となるが、シンボル管理テーブルの解決処理は、JavaVMが処理することになり、解決処理が終了していても、その呼び出しによるオーバヘッドが生じることになる。そこで、シンボル管理テーブルをコンパイルドコード内にも用意し、最初の参照で解決処理を行なった時点で、コンパイルドコード内のシンボル管理テーブルにも結果を書き込むようにする。また、コンパイル時に決定できる定数などは、コンパイル時にシンボル管理テーブルに書き込む。これにより、シンボル管理をJavaVMに任せることによるオーバヘッドを最小限に押さえている。

3 評価

評価には、我々のターゲットとした環境で動作するJITコンパイラを用意できなかったため、JITコンパイラとの違いをつかめるような評価プログラムを自作することで、事前コンパイル、混在実行という手法の評価を行なった。

また、他のシステムと速度面での比較をするために、一般に流通しているベンチマークプログラムであるCaffeineMarkEmbeddedBenchMarkを使った評価も行なった。

3.1 評価環境

実装したバイトコードコンパイラは、NECのEWS4800上で動作し、アセンブラを出力する。アセンブルし、生成したネイティ

ブコードは、MIPS R4000系アーキテクチャのNEC EWS4800上、または、Web-Panelと呼ばれる組み込み機器上で動作する。

今回の評価には、EWS4800を使用した。この環境を、表1にまとめる。

表 1: 評価環境

マシン	EWS4800/360AD
CPU	MIPS R4400SC 150MHz
メモリ	96MB
OS	UP-UX/V(Rel 4.2 MP)
JavaVM	JDK1.1.3(EWS)

3.2 プログラムによる評価

評価プログラムは、我々のバイトコードコンパイラの特徴を把握するために、Javaで、ある同一の命令だけを複数個ならべ、それをループで多数回まわすというプログラムを作成した。このプログラムをコンパイルし、コンパイルドクラスの形にして実行したものと、Javaのバイトコードのまま実行したものの実行時間を比較し、何倍速くなっているかを計測した。コンパイルしたクラスは、特に明記した場合を除き、計測対象のメソッドを含むクラスのみである。つまり、システムクラスなど、標準のクラスはコンパイルしていない。

作成したプログラムは以下の通りである。

加算演算: 2変数を加算し、別変数に代入する演算

配列アクセス: 配列への書込み

メソッド呼出 1: クラス外のコンパイルドメソッドを呼び出す

メソッド呼出 2: クラス外のJavaメソッドを呼び出す。

オブジェクト NEW1: コンパイルしたクラスのロードを伴うNEW

オブジェクト NEW2: Javaクラスのロードを伴うNEW

このプログラムを用いた評価結果を表2にまとめる。

表 2: プログラムによる評価の結果

	Caller Method		
Program	Java	Compiled	Times
加算演算	8855	216	41.0
配列アクセス	14448	441	32.8
Method 呼出 1	95	162	1/1.71
Method 呼出 2	98	521	1/5.32
Object NEW1	3082	25474	1/8.26
Object NEW2	796	21126	1/26.5

表2の結果から、演算命令や配列アクセスなど、コンパイルドメソッド内での閉じた処理に関しては、かなりの高速化が望めることが分かる。それに対して、Javaオブジェクトの NEW や Javaメソッドの呼び出しなど、コンパイルドメソッドから他のクラスへ制御を移す処理が、Javaメソッドからの場合に比べかなり遅くなっていることが分かる。

3.3 CaffeineMark による評価

CaffeineMark[2]は、Javaのベンチマークプログラムの中で、最もよく使われているものの一つである[7]。ベンチマークプログ

[2]CaffeineMark は Pendragon Software の登録商標です

ラムとして、このCaffeineMarkを取り上げ、我々のバイトコードコンパイラから生成したネイティブメソッドを使って測定した。ここで、コンパイル対象としたのは、CaffeineMarkEmbeddedBenchmarkのパッケージに入っているクラスファイルのみで、システムクラスなどその他のクラスはコンパイルしていない。

評価結果は表3にまとめる。

表 3: CaffeineMarkEmbeddedBenchmarkによる評価の結果

Tests	Java	Compiled	Times
Sieve	69	283	4.10
Loop	76	416	5.47
Logic	75	1135	15.1
String	68	55	1/1.23
Float	73	376	5.15
Method	71	119	1.67

表3の結果で、特にString Testの結果がコンパイル前よりも悪くなっていることが目立つ。また、Method Testに関しても、2倍以内の高速化しか達成できていない。これらは、プログラムによる評価の結果から判断すると、String TestやMethod TestがStringクラスAPIの呼び出しやメソッドの呼び出しを多用しているためであると考えることができる。

このような場合には、呼び出すクラスの方をコンパイルすることで、さらなる効果が得られると考えている。

4 結論と課題

混在実行方式のバイトコードコンパイラは、Javaのバイトコードとコンパイルドクラスが共存して実行することができ、JITコンパイラに匹敵する程度の使いやすさがあると考えている。

処理速度面では、通常の演算など、APIクラスを呼び出さないメソッドに関しては、評価の結果からも、大幅な処理速度向上が見込めることが分かった。しかし、Javaメソッド呼び出しを多用するようなメソッドでは、逆に処理速度の低下する可能性があることも分かった。

今後は、コンパイルドメソッドからJavaバイトコードを呼び出す際の負荷をなるべく減らすような改良をすると同時に、ボトルネックとなるメソッドを抽出し、そのメソッドを高速化するためにどの部分をコンパイルすればよいかを解析する方法を確立させる必要があると考えている。

参考文献

[1] 富森 博幸、臼井 和敏、則本 英保、千嶋 博、高木 淳司、菊池 康之：組込み向けJava実行環境JEANS, NEC技報, Vol. 51, No. 11, pp. 161–165 (1998).

[2] 志村 浩也、木村 康則：Java JITコンパイラの試作, 情報処理学会研究報告, No. 96-ARC-120, pp. 37–42 (1996).

[3] Tim Lindholm, F. Y.: *The Java Virtual Machine Specification*, Addison-Wesley Publishers Japan, Ltd (1997).

[4] Yellin, F.: *The Java Native Code API*, Sun Microsystems, Inc (1996).

[5] *Java Native Interface Specification Release1.1* (1997).

[6] Tim Lindholm, F. Y.: Java Runtime Internals (1997).

[7] CaffeineMark, *http://www.pendragon-software.com/pendragon/cm3.*