システムソフトウェアと
オペレーティング・システム 77－5
マルチメディア通信と分散処理 87－5
（1998. 2. 26）

高速かつ移植性の高いJava Virtual Machineの設計と実装

河合 栄治, 砂原 秀樹, 湊 小太郎

奈良先端科学技術大学院大学

概要

Java言語は、バイトコードを仮想機械JVMによって解釈実行することにより、プラットフォーム独立な実行環境を可能にした。しかし、JVMは基本的にインタプリタであり、ソフトウエアによるバイトコードの解釈実行が必要であるため、実行速度が問題となる。また、Javaの利点である、プラットフォーム独立という性質はJVMの移植によって初めて達成される。本研究では、実行速度と移植性の向上を目的とし、JVMを設計および実装した。

The design and implementation of high performance and portable Java Virtual Machine

Eiji Kawai, Hideki Sunahara, Kotaro Minato

Nara Institute of Science and Technology

Abstract

Java is an architecture independent environment which enables us to execute programs on various platforms without recompiling. Currently, excecution speed of Java programs is too slow, because the byte code is interpreted by Java Virtual Machine(JVM) at its execution time. The advantages of architecture independent Java applications are achieved when many platforms have their own JVMs. In this paper, we design and implement a new JVM, considering the issues of the execution speed and the portability.

1 背景

近年、分散環境への要望が大きくなってきている。この分散環境を構築する上で、大きな問題となるのがプラットフォームの混在である。Java言語およびその実行環境は、ソースコードをコンパイルして得られたバイトコードを仮想機械Java Virtual Machine(JVM)によって解釈実行する形式を採用することにより、プラットフォーム独立な実行環境を可能にし、分散環境構築の可能性を広げた。

Javaを用いてアプリケーションを開発する利点には次のものがある。

- 移植の作業が不要
- プラットフォーム毎のコンパイル、パッケージングが不要
- シンプルなオブジェクト指向言語である
- マルチスレッドプログラミングが可能
- Java RMIなどの分散処理が可能

以上の利点をもっているJavaであるが、欠点もある。

- 実行速度が遅い
- Java 実行環境の移植が必要

本研究では実行速度の向上と移植性の向上を目標とし、仕様 [1] に基づき JVM の設計および実装を行った。

2 Java の問題点 –速度–

Java の実行速度を低下させる要因には以下のものが挙げられる。

- インタプリタ言語であること
- オブジェクト指向言語であること
- JVM がスタックマシンアーキテクチャであること

本章では、それぞれについて考察する。

2.1 実行速度の問題：インタプリタ

Java はバイトコードを実行するため高速であると言われている。バイトコードとは、一種のオブジェクトコードであり、仮想機械 JVM をターゲットとした最適化まではコンパイルの時点で完了している。

しかしながら、基本的にはインタプリタであり、ソフトウエアで実装された JVM がバイトコードを解釈実行していかなければならない。

実際には、以下の手順をバイトコード中の一つ一つのオペコードに対して行わなければならない。

1. オペコードを読みだし、
2. そのオペコードの処理内容を実装したサブルーチンを呼び出し、
3. 実行する

そのため、C 言語のようにネイティブコードへコンパイルされる言語で記述されたプログラムと比べて実行速度が劣る。

2.2 実行速度の問題：オブジェクト指向

Java はオブジェクト指向の言語である。

一般に、オブジェクト指向言語で記述されたプログラムは、動的な束縛によるオーバーヘッドが生じる。

2.2.1 1 クラスにつき 1 ファイル

オブジェクト指向では、データとそれに関するルーチンをオブジェクトに抽象化するが、Java においてはその抽象化の単位となるクラス毎にファイルを用意している。これは、クラスへの初めてのアクセスがあると、必ずファイルからデータを読み出さなければならないことを意味する。

2.2.2 静的で強い型付け

Java では、オブジェクト指向における動的な束縛を極力排除するために、静的で強い型付けを行っている。つまり、コンパイル時の型チェックを厳密に行い、型情報をバイトコードに埋め込んでいる。このために、変数探索やメソッド探索において探索範囲の大幅な縮小が可能で、負荷を軽減している。しかしながら、動的束縛を行わなければならないことには変わりはなく、オーバーヘッドとなる。

2.3 実行速度の問題：スタックマシン

JVM はスタックマシンアーキテクチャを採用している。

レジスタマシンでは、データを高速なレジスタに読み込み、演算結果をレジスタに書き出す。スタックマシンでは、レジスタの存在を仮定せず、演算結果は全てスタックに積まれる。スタックはそのアクセスの原理がレジスタのモデルと異なるため、レジスタとの対応をとることが難しい。そのため、プラットフォームによってはスタックを主記憶上に配置することになり、演算の度にメモリへの書き出しが行われることになる。また、データを読み込むとそのデータはスタックから取り除かれるため、複製という作業が必要となる場合もある。

以上のような欠点をスタックマシンアーキテクチャは持っているが、利点も持っている。まず、レジスタを仮定しないため、レジスタの少ないプラットフォーム上でも仮想機械の構築が可能である。さらに、スタックトップを暗黙のオペランドとして使用することが可能となるので、コード中のオペランドの数を減らすことができ、コードサイズを小さくすることができる。実際に Java の大半のオペコードはオペランドをとらないものである。

この性質は、ネットワークからダウンロードしてプログラムを実行するアプレットの様な形態に適している。

3 Javaの問題点 –移植性–

Javaはプラットフォーム独立な実行環境を提供する。しかし、そのためにはJavaバイトコードを解釈実行するJVMとクラスライブラリが各プラットフォームに移植されなければならない。

3.1 クラスライブラリ

クラスライブラリの大半は、Java言語で記述されている。しかしながら、JVMはプリミティブな演算とメソッド呼び出し、スタック操作程度しかサポートしていない。そのため、JVMが処理できないI/Oなどの機能は個別にC言語等で記述され、ネイティブコードで実装される。これらは、移植性が(ソースレベルにおいても)低い。

また、オペレーティングシステム(OS)によってはネイティブコードの動的なロードができない場合がある。このようなOSでは、実行開始時に全ての参照を解決するか、JVMに埋め込みで実装するしかない。

3.2 データ型

C言語等の移植性で問題になるものの一つにデータのサイズがある。Javaではこれを固定することによって、この問題を回避している。

しかしながら、スタックのサイズは各マシンアーキテクチャのワードサイズとなっており、参照型のデータも各アーキテクチャのメモリ空間のサイズ(通常ワードサイズ)となる。64-bitアーキテクチャのマシンでは、1ワードに32-bitデータを2つもしくは64-bitデータを1つ格納でき、スタックポインタの操作等が32-bitアーキテクチャのものと微妙に異なるため、別々に実装する必要がある。

4 JVMの設計および実装

本研究では、JVMの仕様[1]に基づき、JVMを設計・実装した。

JVMの仕様は比較的自由度が高く設定されている。JVMができなければならないのは、次のことだけであるとあるとされている。

- クラスファイルの読み込み
- JVMコードの意味論を正確に実装する

そのため、実装者はそれぞれの設計方針にしたがってJVMを実装することができる。

今回は、C言語を用いて32-bit Unix OSをターゲットに、高速性と移植性の両立を目標に、JVMを設計、実装した。

4.1 データ構造

今回実装したデータ構造は主に次の3種類に分類できる。

- クラス
- スレッド(スタックおよびフレーム)
- インスタンスオブジェクト

4.1.1 クラス

今回実装したJVMにおいては、クラスファイルのデータ構造を解体することなく、ほぼそのままの形で利用している。つまり、クラスファイルのフォーマットとほぼ同じデータ構造に、若干の管理用データおよびクラス変数保持用メモリ空間を追加したものとなっている。

クラスのデータの管理は、クラス名をキーとしたハッシュテーブルで管理している。

4.1.2 スレッド

JVMはスレッドにスタックを割り当てる。要求に応じた柔軟なメモリ利用を実現するため、非連続領域のリストを一つのスタックとして用いている(図1参照)。

本実装では、以下の構造体を定義した。

javaスタック構造体
: スタック=スレッドの意味論を実現するためのデータで、スレッド実行に必要な管理データを格納する。具体的にはスレッドID、プログラムカウンタ、スタックポインタ、他の構造体への参照など。

スタックデータ構造体
: 断片のリンクリストとなっているスタック領域のそれぞれに一つ用いる。実際のスタック領域を管理するためのデータを格納する。スタック領域は必要に応じて動的に非連続領域を確保するため、それぞれの断片に一つ割り当てる。

フレーム構造体

メソッド呼び出し毎に生成されるフレームを管理するデータで、実行に必要なローカル変数、各種オブジェクトへの参照、オペランドスタックを格納する。さらに、上位データ構造への参照とメソッド呼び出し時の状態保存のためのプログラムカウンタを格納する。

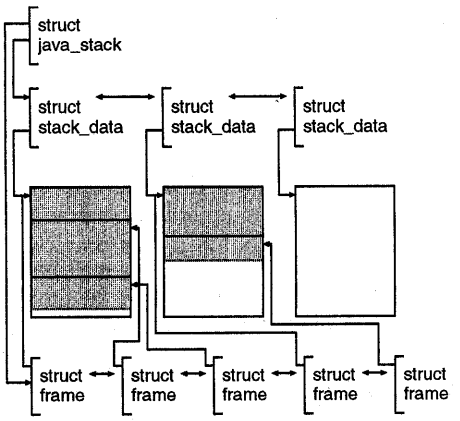

図 1: スレッドのデータ構造

4.1.3 インスタンスオブジェクト

Java におけるインスタンスはメモリ管理の観点から以下のものに分類できる。

- 配列
- 文字列 (java.lang.String のインスタンス)
- それ以外のインスタンス

配列のクラスは配列の構成要素のデータ型と、次元数によって定まるが、動的に定義されるものであり、クラスライブラリでは定義されていない。また、言語仕様上では多次元配列を扱えるようになっているが、直接はサポートせずに配列への参照の配列で多次元配列を実現している。

文字列は、一度生成されると変更されないので、同じ内容の文字列には同じインスタンスオブジェクトを用いる。そのため、検索が必要となるので保持する文字列をキーとするハッシュテーブルで管理している。

これらのインスタンスは参照型という一つのデータ型で参照できるよう、以下のデータを保持する。

- オブジェクトの種類を表すフラグおよび記述子
- インスタンス変数
- モニタ用データ
- 参照カウント

4.2 高速化

JVM の実行速度の問題点に対して、それぞれ対策を行った。

4.2.1 デコード

JVM では仮想機械の形式を取っているため、オペコード一つ一つに対してそれぞれのルーチンの実行が必要である。

それぞれのオペコードはバイトコードの名の通り 1 バイト幅であり、それゆえ計 256 個のオペコードが定義可能である。そのうち、0(0x00) から 201(0xc9) までが JVM の仕様として定義されている[1]。

このバイトコードからいかにして目的のコードを呼び出すかであるが、ここでは 2 つの手法がある。

- それぞれのオペコードに対して処理内容をサブルーチンで実装し、オペコード順 (0, 1, ... , 255) にサブルーチンへの参照をテーブルに保持しておく。各オペコードを 8-bit 符号なし整数のインデックスとして、サブルーチンを呼び出す。
  - 利点：デコードが不要となる
  - 欠点：オペコード実行時に、関数呼び出しのオーバーヘッドが生じる
- 全てのオペコードに対する処理内容を一つのサブルーチン内に実装し、各オペコードをデコードし、目的のコードへ移動する。
  - 利点：関数呼び出しのオーバーヘッドがない
  - 欠点：デコードが必要

[1] 186(0xba は使用されていない)

プログラムの構造化の観点からは、それぞれをサブルーチンで実装することが望ましいが、実行速度が重要である。

コンパイラ依存であるが、C言語のswitch文は、今回のような連続した値をとるcase文を並べた場合、各オペコードをオフセットにしてジャンプする命令に変換する。そのため、実質上デコードは行われず、オペコードを直接オフセットとしてジャンプするだけで済む。本実装では、各オペコードはブロックで実装し、switch文でデコードする方法を取った。

4.2.2 各オペコードの実装

現代のアーキテクチャはプログラムの実行においてメモリアクセスがボトルネックになることが多い。キャッシュシステムやレジスタを有効活用し、各オペコードの実行速度を向上させる手法は次の通りである。

- 実行命令数の削減
- データ参照の局所化

オペコードはその処理中に各種データを参照するが、それらを参照するために用いる間接アドレッシングはできるだけ少ない方が望ましい。

各オペコードが参照するデータはそれぞれ違うが、次のものは頻繁に参照される。

- スタック構造体へのポインタ
- フレーム構造体へのポインタ
- スタックポインタ
- プログラムカウンタ

本実装では、これらのデータへの参照を保持するポインタをVM中に一組用意し、実行中のスレッド、メソッドに関してはこのポインタを用いてスタック、フレームにアクセスしている。こうすることで、上位構造体から間接アドレッシングで参照する必要がなくなる。これらのポインタは、スレッド切替えや、メソッド呼び出しなど退避する必要がある時に本来の構造体へ書き戻す実装となっている。

4.2.3 コードの動的書き換え

オブジェクト指向では、動的束縛によるオーバーヘッドが問題となり、さまざまな高速化技法が提案されている。[3]

Javaでは、一度解決した参照をキャッシュしておいて、次回からは高速なデータ参照を実現している。Sunの実装では、これらは_quickオペコードへのダイナミックな書き換えで実現しており、本実装でもそのうちのいくつかを実装した。

クラス変数、クラスメソッドの参照は静的なものであるために、一度解決したものは保存しておけば次回から高速な参照が可能である。そのために、本実装では呼び出し側のコンスタントプールに解決済みの参照を保持するエントリを設けている。オペコードgetstaticを例にとると、これを初めて実行した時はカレントクラスのコンスタントプールを参照し、目的のクラス、変数名、型の情報を取得しその後必要ならそのクラスをロード・初期化を行い、変数の値を取得する。そして、オペコードをgetstatic_quickに変更し、カレントクラスのコンスタントプールのエントリに、目的の変数が格納されているアドレスを保持する。次回からは、getstatic_quickが呼び出され、これはカレントクラスのコンスタントプールのエントリに格納されているアドレスから、直接変数の値を取得しスタックに積む。こうすることで、大幅な性能向上が計測された。

4.3 移植性

4.3.1 エンディアン

プラットフォームによってエンディアンが異なるため、クラスファイルの読み込みには注意が必要である。本実装ではクラスファイルをメモリ内に読み込む時点でデコードし、その後はそれぞれのアーキテクチャの方式でデータを扱っている。

4.3.2 ワードサイズ

本実装では、32-bitプラットフォームをターゲットとしている。ワードサイズ(スタックのサイズ)を32-bitとしているため、現在のところワードサイズが32-bitのプラットフォームでしか動作していない。64-bitプラットフォームでは、スタックを参照型を除いて64-bit中32-bitのみ使用すれば動作させることができるが、メモリの利用効率が悪い。

4.3.3 稼働実績

クラスライブラリが未実装であるため実行可能な処理は少ないが、以下のプラットフォームでの動作を確認した。

- FreeBSD 2.2.5R(Pentium)
- Solaris 2.5(Sparc)
- SunOS 4.1.4(Sparc)
- IRIX 5.3(MIPS R4600)
- NEWS 6.1.2(MIPS R4000)
- OSF/1 2.0(Alpha)

ほぼ、ANSI-C にしたがって記述しているため、ANSI-C コンパイラの備わっている 32-bit のアーキテクチャなら、ほとんど修正することなく動くものと考えられる (OSF/1 は 64-bit スタック中 32-bit のみ使用)。

5 評価

本実装の実行速度を評価するために、for 文によるループ (参照データはローカル変数のみ) とクラスメソッドの呼び出しを行うプログラムをそれぞれ作成した。

用いたプラットフォームは以下の通り。

- OS : FreeBSD-2.2.5R, AMD K6-166,64M
- CC : gcc version 2.7.2.1

この環境で、作成した本研究の JVM と Sun Microsystems の JDK の移植版である FreeBSD 用 JDK1.0.2 および JRE1.1.5 に付属の JVM で実行時間を測定した。

実行時間の測定には csh の組み込みコマンド time を用いて測定した (単位は秒)。

表中の java が JDK1.0.2 の JVM、jre が JRE1.1.5 の JVM、java_kawai が本実装の JVM。

- for ループ

	1000 万回	1 億回
java	6.770	66.162
jre	4.093	39.218
java_portable	6.998	68.383

- クラスメソッド呼び出し

	100 万回	1000 万回
java	2.290	21.288
jre	1.375	12.065
java_portable	2.367	23.535

ほぼ、JDK1.0.2 付属の JVM と同等の性能が得られたといってよい。本研究の JVM はそのまま多くのプラットフォームで稼働することを併せて考えると目的は達成されたと考えられる。

6 今後の課題

6.1 さらなる最適化

今回の実装では、スレッドに関する各種ポインタ用に変数を用意し、通常はその変数を通じて直接データにアクセスすることで間接参照を減らし、メモリアクセスの局所化を計った。スレッドの他に複雑なデータ構造を持つものにクラスがあり、このデータを参照するオペコードも多く、同じ手法で高速化が期待できる。しかし、レジスタが少ないもしくはキャッシュの小さなアーキテクチャでは、実行速度が低下する可能性もある。

6.2 JIT

JIT の実装は今後不可欠となってくると考えられる。しかしながら、JIT は移植性が乏しく、ダイナミックロードが必要となるなど問題点もある。さらに、分散環境で必要となるオブジェクトの状態保存が難しい。こうした問題点を考慮した JIT の設計が望まれる。

7 まとめ

本研究では、Java 言語の高速化と移植性の向上について考察した。実際に JVM を仕様に基づき、いくつかの技術について有用であることを検証した。今後の課題としては、JIT の実装が挙げられる。

参考文献

[1] Tim Lindholm, Frank Yellin: The Java Virtual Machine Specification Addison-Wesley, 1997

[2] Jon Meyer, Troy Downing: Java Virtual Machine O'Reilly, 1997

[3] 小野寺 民也: オブジェクト指向言語におけるメッセージ送信の高速化技法 Journal of IPSJ Vol.38 No.4 Apr. 1997