計算機アーキテクチャ 125－13
（1997. 8. 21）

Java Virtual Machine の静的・動的解析

渡辺 健司†、金田 正一‡、大津山 公平†

† 会津大学コンピュータ理工学部
{s1022080, kohei-o}@u-aizu.ac.jp
‡ 富士通AMD

あらまし： Java 言語はインターネットユーザの急増と共に現在急速に普及しつつある。Java は仮想実行環境である Java Virtual Machine(JVM)によって実行されるが、これは通常インタプリタもしくは JIT コンパイラとして実現されている。また、最近ではハードウェアとして JVM を実現する Java チップも発表されている。本研究では Java クラスファイルを静的・動的に解析することで、JVM をハードウェアで実現する際の効率の良いアーキテクチャについて考える。

Static and Dynamic Analysis of Java Virtual Machine

Kenji Watanabe†, Shouichi Kaneda‡, Kohei Otsuyama†

† School of Computer Science and Engineering, University of Aizu
‡ Fujitsu AMD

Abstract: Java language has been widely used as the number of internet users has grown rapidly. Java is executed with virtual machine environment called Java Virtual Machine(JVM). Usually JVM is realized as interpreter or Just In Time(JIT) compiler, Java chip which can realize JVM with hardware is already announced.

In this paper, we analyzed the behavior of Java class file using static and dynamic way and discuss about suitable configuration of Java chip. We specially focused on the cache which can contain operand stack data.

1 はじめに

Java は Sun Microsystems が提唱した新しいプログラミング言語である。Java はアプレットとして WWW 上で動的な情報を提供することが出来るため、インターネットの普及とともに急速に普及しつつある。Java ソースはコンパイルされ、バイトコードと呼ばれるクラスファイルとなる。このファイルは Java Virtual Machine(JVM)と呼ばれる仮想実行環境で実行される。JVM は通常インタプリタや JIT コンパイラによって実現されるが、picoJava と呼ばれる、Java バイトコードをハードウェアで実行する Java チップも発表されている[1]。また、最近では論理合成による Java チップ作成例も報告されている[2]。本研究では、クラスファイルやクラスファイルを実行した時の命令トレースか

らJVMの静的[3]・動的解析を行う。出現頻度の高い命令や、スタックキャッシュサイズなどの解析から、JVMをハードウェアで実現する際の効率の良いアーキテクチャについて考える。

2 Java Virtual Machine

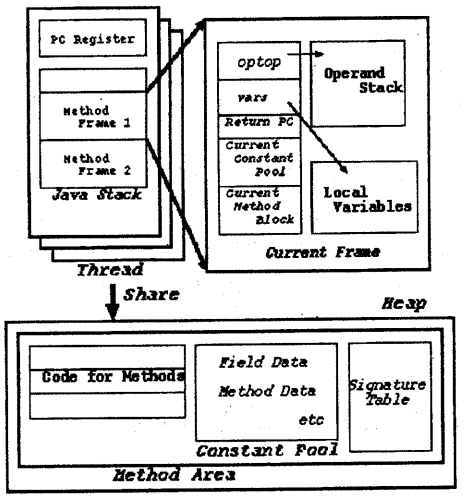

図1：Java Virtual Machineの構成

2.1 Java Virtual Machineの構造

JVMはJavaのバイトコードを実行するための仮想マシン環境であり、スタック・アーキテクチャを採用している。(図1参照)

2.2 データ型

Javaがサポートしているデータ型は、プリミティブ型とリファレンス型の2つである[4]。プリミティブ型には、1バイトのbyte、2バイトのshort, char、4バイトのint, float、8バイトのlong, doubleがある。リファレンス型は、クラス、インターフェイス、配列の参照に使われる4バイトのデータ型である。

2.3 Javaスタック

JVMの各スレッドは、プライベートなJavaスタックを持っており、JVMフレームに保持されている。

Javaスタックは、メソッドの状態やパラメータ等を保持している。

2.4 オペランドスタック

JVMでは、データはオペランドスタックと呼ばれるスタックに保持される。このスタックの1ワードは4バイト(32ビット)固定長である。1バイトのbyte、2バイトのshort, char型はスタック上では4バイトに拡張される。なおbyte, short型は符号の拡張も行われる。8バイトのlong, double型は2ワードに分けて保持される。

殆どの命令は、このスタックを対象に演算などを行う。

2.5 レジスタ

JVMでは、pc, optop, vars, frameというレジスタを持っている[5]。

pcレジスタはスレッド内で現在実行中のコード番地を指している。optopレジスタはオペランドスタックのスタックトップを指し、varsレジスタはローカル変数のベースアドレスを保持している。また、frameレジスタは実行環境の位置を指している。

2.6 ヒープ

メモリとしてヒープが存在し、ヒープの情報は全てのスレッド間で共有される。ヒープの一部としてメソッドエリアが含まれ、コード、コンスタントプールの実体やオブジェクトの型を与える情報であるシグネチャーテーブルが置かれる。

2.7 コンスタントプール

ソースファイルをコンパイルすることによりバイトコード(クラスファイル)中に、そのクラスに静的に用意されるべき情報が集まった可変長の配列が生成される。これをコンスタントプールと呼ぶ。コンスタントプールは1バイトのタグと可変長のデータで構成された配列で、コンスタントプールは、ヒープ内のメソッドエリアに配置され、参照にはインデックスによってアクセスする。

後述する命令の中には、クラス、インターフェイス、フィールドの型を調べるために、3回のコンスタントプールへのアクセスが必要なもののある。これは、情報が間接的に格納されているためである。

2.8 命令セット

JVMの機械語はクラスファイル内のメソッドのコードアトリビュートに格納されている。Javaの命令は殆どが固定長で、最小命令長が1バイト、最大命令長は不定となっている。Sunの報告[1]では、平均命令長は1.8バイト程度となっている。可変長

命令は、tableswitch, lookupswitch の２種類があり、これらは多方向分岐命令である。

今回の命令の分類は、ロード・ストア命令、算術演算命令、型変換命令、オブジェクト操作命令、スタック操作命令、コントロール命令、メソッド操作命令、その他（例外処理、同期、NOP など）の８種類とした。

また、invokevirtual 命令のようにオブジェクトの型や引数の数を調べるためにコンスタントプールに数回アクセスするものがある。これらの命令は、そのままだと再度実行する際に、再びコンスタントプールにアクセスすることになる。 そこで、これらの命令が実行された後に、命令を書き換えて、処理速度を向上させることができる。これらの命令をまとめて QUICK 命令と呼ぶ。QUICK 命令になる命令は１６個ある。

3 命令トレーサー

動的解析に用いる命令トレースはトレーサーによって採取される。今回は、UNIX 上で動作する Java インタプリタである kaffe (version 0.8.4) を改良し、命令トレースを採取した。命令トレースには、命令の種類、スタックの深さ、ローカル変数の数、コード長、可変長命令の命令長などの情報が含まれる。

4 解析プログラム

JVM の解析を行うために、静的・動的（トレース）の２つのプログラムを作成した。

静的解析プログラムは、クラスファイルを読み込んで、コンスタントプールや各メソッドのコードなどを静的に解析するものである。

動的解析プログラムは、トレーサによって作成された命令トレースから解析を行い、平均スタックサイズ、平均命令長、命令毎出現頻度、QUICK 命令の使用率などを表示させるものである。

5 解析に使用したデータ

解析に使用したアプリケーションとアプレットは次のものである。

1. cat: ＵＮＩＸの cat コマンド。
2. wc: ＵＮＩＸの wc コマンド。
3. LinpackJava: ベンチマーク。
4. ThreeDimensionalArray: JVM での３次元配列のシミュレーションを行うアプレット。

以降、LinpackJava と ThreeDimensional-Array をそれぞれ Linpack、3DASim と略記する。

データの諸元を表１に示す。

表 1：解析データの命令数

	Static	Dynamic
cat	222	19659897
wc	336	7030082
Linpack	1084	8032193
3DASim	4793	49309836

6 静的解析結果

最大スタックサイズ(max_stack)、最大ローカル変数サイズ(max_locals)、メソッドのコード配列サイズ(code_length)、及び、平均・最大命令長(Instruction length)の静的解析結果を表２に示す。

図２、図３は、命令長別出現頻度、及び、命令の種類別出現頻度についての結果である。

表 2：スタック、ローカル変数、コード配列サイズと命令長の静的解析結果

	max_stack [word]			max_locals [word]			code_length [byte]			Instruction length [byte]	
	Min	Avg.	Max	Min	Avg.	Max	Min	Avg.	Max	Avg.	Max
cat	1	2.00	4	0	3.00	9	5	87.67	311	2.37	75
wc	1	2.25	5	1	3.75	6	5	98.12	250	2.34	43
Linpack	2	6.21	12	1	10.36	32	11	135.71	480	1.75	4
3DASim	0	3.56	7	0	2.69	14	1	93.11	2996	1.94	323

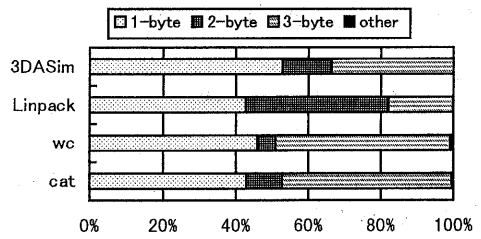

図2:命令長別出現頻度(静的)

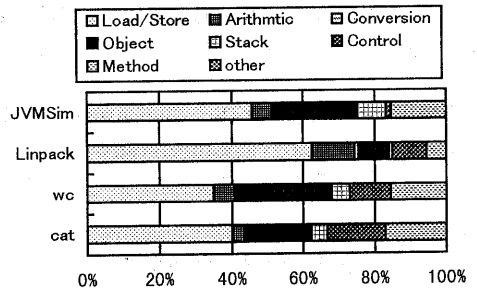

図3:命令の種類別出現頻度(静的)

図4は、命令累積出現頻度で横軸に命令の頻度順にとり、縦軸に累積出現頻度をとったものである。

命令数が40-60程度で出現頻度が100%になっている。

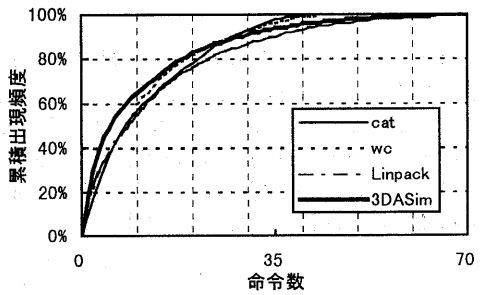

図4:静的解析による命令累積出現頻度

7 動的解析結果

最大スタックサイズ(max_stack)、最大ローカル変数サイズ(max_locals)、メソッドのコード配列サイズ(code_length)、及び、平均・最大命令長(Instruction length)の動的解析結果を表3に示す。また、命令長別出現頻度、及び、命令の種類別出現頻度については、それぞれ図5、図6に解析結果を示す。

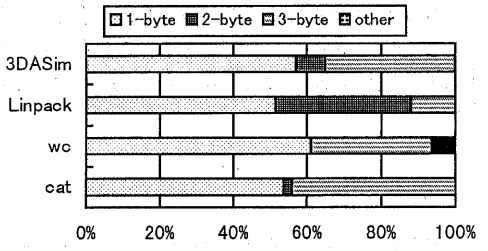

図5:命令長別出現頻度(動的)

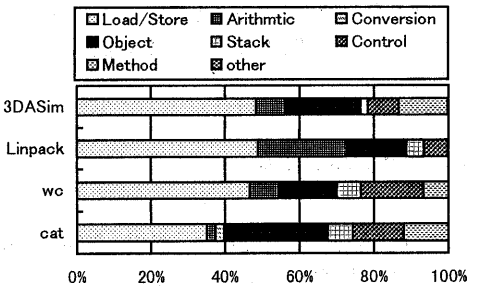

図6:命令の種類別出現頻度(動的)

図7はメソッドとネイティブメソッドの呼び出し回数の割合である。Javaはオープンプラットフォームな言語であるが、プラットフォームに依存し

表3:スタック、ローカル変数、コード配列サイズと命令長の動的解析結果

	max_stack [word]			max_locals [word]			code_length [byte]			Instruction length [byte]	
	Min	Avg.	Max	Min	Avg.	Max	Min	Avg.	Max	Avg.	Max
cat	0	3.45	7	0	2.00	16	1	37.93	2695	1.90	3
wc	0	3.17	7	0	2.40	16	1	49.46	2695	3.74	34
Linpack	0	6.35	12	0	7.83	32	1	76.39	480	1.61	4
3DASim	0	3.35	10	0	2.79	17	1	41.01	2996	1.83	323

たものを作成するためにネイティブメソッドが用意されている。ネイティブメソッドは Java 以外の言語（Cやアセンブリ言語など）で記述されている。

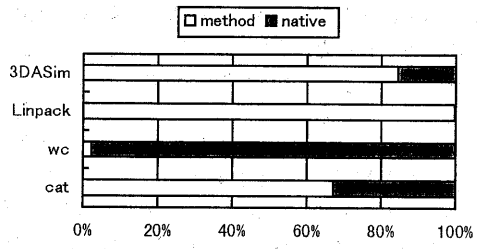

図7：メソッドとネイティブメソッドの呼び出し回数の割合

表 4 は QUICK 命令の種類を分類したものである。QUICK 命令の実行回数を QUICK、命令を書換えて最適化した（QUICK 命令に書き換えた）回数を pre_QUICK とした時のそれぞれの割合である。Linpack の QUICK の割合が低いのは、ループ中に QUICK 命令に書き換えられる命令（invoke 系の命令など）が殆どないためである。

表 4：pre_QUICK と QUICK の回数と割合

	non_ QUICK	QUICK	QUICK の割合
cat	524	5968401	99.99%
wc	605	1548012	99.96%
Linpack	168	71773	99.77%
3DASim	5850	11301536	99.95%

図 8 は、動的解析による命令累積出現頻度である。命令数が 20-50 程度で出現頻度が 100%になっている。wc, cat, Linpack の最初の立ち上がりが急峻であるが、これはループにより使用される命令が限定されている可能性が高い。

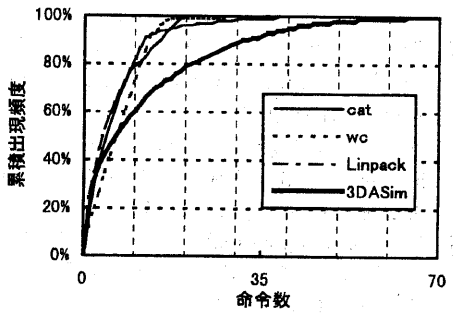

図8：動的解析による命令累積出現頻度

8 解析結果の考察

以上の静的・動的解析結果から次のことがわかった。

8.1 平均スタックサイズ

静的と動的解析での違いはそれほどなく、平均スタックサイズは 2-4 ワード程度なので、少量のレジスタもしくはキャッシュを使用することで高速化されることが期待される。

8.2 ローカル変数サイズ

スタックサイズより静的と動的解析で差が大きく、プログラムによってかなりの差がある。これは、静的解析はクラスライブラリの中まで解析を行っていないためである。

サイズは最大でも 30-40 ワード程度であるため、キャッシュを使用することで高速化が期待され、以下で述べる Load/Store 命令の高速化が望める。

8.3 平均命令長

動的解析で静的より一般的に小さくなることがわかった。つまり、ループ中では比較的命令長の短い命令を使用している。wc を除けば Sun の報告どうりの 1.8 バイト程度になり、命令数が同じだと仮定すると普通の 32-bit プロッセサに比べコードサイズが半分程度になる。

8.4 命令の種類別出現頻度

静的・動的解析の両方で Load/Store 命令が 4 割近くを占めている。このため、レジスタやキャッシュによる高速化が極めて重要である。また、メソッド操作命令のうち、invoke 系の命令は引数の個数を調べるためコンスタントプール内を 3 回アクセスしなければならず、かなりの影響が出ると思われる。コンスタントプールの効率的な割当てが必要である。

8.5 QUICK 命令

QUICK 命令を使うことで、99.9%以上の invoke 命令や他の効率の悪い命令を QUICK 命令に書き換えることができる。QUICK 命令を書き換えると、コンスタントプールへのアクセス回数が 1 回になり命令実行が速くなる。

8.6 命令累積出現頻度

動的解析結果は、静的なものに比べ最初の立ち上がりが急峻である。これは、静的解析ではループの回数を解析出来ないためである。いずれにしても60 種類程度の命令でほぼ 100%に達するので、ハードウェア実装時の目安となる。

9 オペランドスタックのシミュレーション

オペランドスタックにレジスタやキャッシュを用いた場合、どの程度のサイズが必要かを調査するため、トレースを用いてシミュレーションを行った結果が図9である。

このシミュレーションでは、オペランドスタック（以降、スタックと記述する）へ1ワード単位でアクセスし、スタックが溢れたらメモリ上のスタック（以降、メモリと記述する）へ1ワード単位でストアする。また、スタックが空になったら、メモリから1ワード単位でロードすると仮定する。

一般にスタックサイズとメモリアクセスの関係は、スタックサイズが大きくなるにつれメモリアクセスが指数的に減少する[6]。

Linack 以外の結果は殆ど等しく、スタックサイズが4ワードでメモリへのアクセス割合がほぼ0となる。また、Linpack では、8ワードでほぼ0となる。Linpack 以外は int 型（1ワード）のデータを頻繁に使用するが、Linpack では double 型（2ワード）も使用している。このことから、スタックサイズは、頻繁に使用する型の大きさの少なくとも2倍から4倍程度必要であると推測出来る。

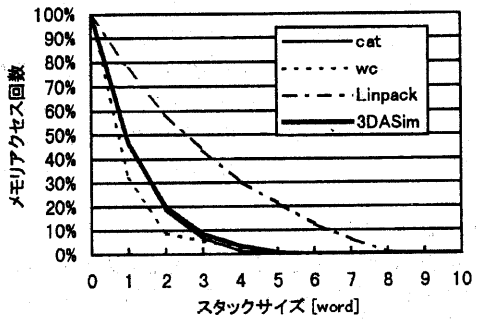

図9：スタックサイズとメモリアクセス回数の関係

10 終わりに

JVM をハードウェア化するための予備調査として命令の種類別出現頻度、命令別出現頻度、スタックの深さ、ローカル変数の数、コード長、命令長などをクラスファイルから直接解析する静的方法と、命令トレースから解析する動的方法にて解析した。

命令ではスタックに対するロード・ストア命令が多い事が判明した。これらはメモリアクセスを伴うものであるので、キャッシュが必須である。しかしながら、スタックやローカル変数の大きさはさほど大きくないことからキャッシュはかなり小さくて済むと期待される。実際にオペランドスタックのシミュレーションの結果から4～8ワード程度のスタックキャッシュがあれば良いと言えるだろう。

また、命令累積出現頻度から見ると 60 命令程でほぼ 100%になるので、ハードウェア化する時の目安になるであろう。

なお、この実験では 4 種類のデータしか解析を行っていないので、今後はもっと多くのデータにて解析を行いたい。また、Java スタックや、ローカル変数も含めたスタックシミュレーションも行いたい。

参考文献

[1] "picoJava™: A Hardware Implementation of the Java Virtual Machine", *Hot Chips VIII*, Aug. 19-20, 1996, pp.131-144.

[2] 清水 尚彦, 青柳 真佐樹: "Java Virtual Machine アーキテクチャの実装に関する研究：TRAJA プロジェクト", 情報処理学会研究報告, ARC-124-4, pp.19-20, 1997.

[3] Shouichi Kaneda: "Static Analysis of Java Class File", graduation thesis of the University of Aizu, Mar. 1, 1997.

[4] Tim Lindholm and Frank Yellin: "The Java™ Virtual Machine Specification", *Addison Wesley*, 1996.

[5] "The Java Virtual Machine Specification Release 1.0 Beta DRAFT", *Sun Microsystems*, Aug. 21, 1995.

[6] Philip J. Koopman, Jr.: "スタックコンピュータ: CISC/RISC とスタックアーキテクチャ", *共立出版*, 1994, pp.153-155.