計算機アーキテクチャ 137－3
ハイパフォーマンス 80－3
コンピューティング
（2000．3．2）

# 命令畳み込み，データ投機および再利用技術を用いたJava仮想マシンの高速化

重田大助[†] 小川洋平[††] 山田克樹[†]
中島康彦[†††] 富田眞治[†]

Javaバイトコードを直接実行する場合に，命令畳み込み，データ投機およびデータ再利用などの手法によってどの程度の高速化が可能かについて考察した．命令畳み込みはスタックマシンの特徴による命令数の増加を抑える手法である．効果を定量的に測定するために，命令畳み込みを考慮に入れたプロセッサの構成を仮定し，このプロセッサ上で，命令の実行結果を予測するデータ投機，命令の実行結果を再利用するデータ再利用の効果を測定した結果，データ再利用の手法がデータ投機の手法よりも効果的であることがわかった．

# High Speed Java Bytecode Execution Using Instruction Folding, Data Value Speculation and Data Value Reuse

Daisuke Shigeta,[†] Youhei Ogawa,[††] Katsuki Yamada,[†]
Yasuhiko Nakashima[†††] and Shinji Tomita[†]

This paper describes the effectiveness of Instruction Folding, Data Value Speculation and Data Value Reuse against the execution of Java bytecodes. For a quantitative evaluation, we assumed a micro architecture equipped with some special purpose tables and facilities for Instruction Folding which can reduce the number of dynamic steps of instructions. On this execution model, we measured the effectiveness of Data Value Speculation which predicts the results of instructions, and that of Data Value Reuse which reuses the result of instructions. As a result, we show that Data Value Reuse is more efficient than Data Value Speculation.

## 1. はじめに

Javaバイトコードの実行速度を高速化するための手法として，命令畳み込み，データ投機およびデータ再利用などの手法が考えられる．本稿ではこれらの方式の特徴について述べ，ハードウェアによる実装を仮定した場合の効果について述べる．2章において，Java仮想マシンの特徴について述べ，3章において，バイトコードの命令出現頻度に基づく高速化手法の検討を行う．4章において，命令畳み込みを考慮したプロセッサ構成を仮定し，5章において，このプロセッサ上でのデータ投機およびデータ再利用の手法および動作概要について述べる．また，6章において，各高速化手法の効果を測定し，7章において考察を行う．

† 京都大学大学院情報学研究科通信情報システム専攻
Division of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
†† 京都大学工学部情報学科
Department of Information Science, Faculty of Engineering, Kyoto University
††† 京都大学大学院経済学研究科
Graduate School of Economics, Kyoto University

## 2. Java仮想マシン

Java言語は，並列駆動が可能なクラスを基本としたオブジェクト指向型プログラミング言語であり，可能な限り実装に依存する部分がないように設計されている．Java言語により記述されたプログラムはJavaバイトコードに変換することにより，様々な実行環境において実行することができる．実行環境がプラットフォームに依存せず，しかも安全性が高いという特徴により，今日ではJava言語はインターネット上で広く使用されており，今後ますます普及していくものと思われる．

さて，Java仮想マシン(以下JVM)は，特定のファイル・フォーマット，すなわちクラスファイル・フォーマットの情報のみに基づいて動作する．クラスファイルとは，JVMの命令(バイトコード)とシンボル・テーブル，ならびに他の付随的な情報を保持したものである．

JVMが使用するデータ域は図1のように大きく3つの部分から成る．オペランド・スタックおよびロー

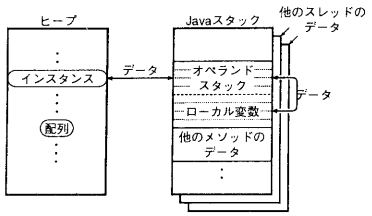


図1　JVM のデータ域

カル変数は各メソッド毎に固有であり，他のメソッドから直接参照することはできない．また，ヒープは各メソッドだけでなく，各スレッド間で共有される．

**オペランド・スタック**：JVM の大半の命令は，現在実行中のメソッドのオペランド・スタックから値を取得，演算し，同一のオペランド・スタックに結果を返す処理を行う．オペランド・スタックはメソッドに引数を渡し，返り値を受け取るためにも使われる．

**ローカル変数**：現在実行しているメソッド内でのみ使用する値を保持している．またメソッド呼び出しの引数はこの領域を用いて受け取る．

**ヒープ**：実行時にクラス・インスタンスおよび配列の割り当てを行うためのデータ域である．

JVM の命令セットは，ローカル変数のロード/ストア，演算，型変換，オブジェクトの操作，オペランド・スタックの管理，分岐，メソッドの呼び出しおよびリターン，例外のスロー，同期などの命令を含む．

JVM は，型付けが強く，また実行時に実行されるメソッドが決定されるという特徴を持っている．このため，Java 言語からバイトコードを生成する Java コンパイラは，コンパイル時の最適化を十分に行うことができない．また JVM はスタックマシンであるため，オペランド・スタック上の値のみが操作可能となっている．つまり，ローカル変数上の値は一度スタック上に値を移動しなければ，扱うことができない．このようにバイトコードを記述通りに実行すると，不必要なデータの操作が発生するため，実行の高速化が難しいと言える．

## 3. 高速化手法

まず，バイトコード実行時における命令出現頻度の調査を行った．

表 1に，SPEC JVM 98[1]において，実際に実行された命令の内訳を示す．この結果より，メソッドの呼び出し，および，ヒープの読み出し回数が比較的多いことがわかる．これらの命令は，オペランド・スタック上のオブジェクトのリファレンスや，プログラムに記述されたコンスタント・プールへのインデックスを元にして，実行するメソッドを決定する操作，および，参照するフィールドのアドレスを決定する操作を含む．これらの操作には，一般的に多くの処理時間を必要とする．

また連続する 2 命令の出現頻度を調査した結果，ローカル変数からスタックへ積んだ値を次の命令で使用する頻度が高いことが判明した．バイトコードとしてはこのような命令列である．けれども，実行の際にローカル変数上の値をスタックに積む操作を省略することができれば，高速化することが可能である．

さて，これらの特徴から，我々は以下のような手法が高速化に寄与するのではないかと予測した．

- メソッド呼び出しおよびヒープ読み出し時に必要となる対象アドレスの計算は，前回の計算値を保持しておき，再利用することにより省略する．
- ローカル変数をスタックへ積んでから使用するのではなく，直接ローカル変数の値を用いることにより，スタックへ積む操作を省略する．同様に，スタック上で行った演算の直後にローカル変数へ書き込む場合にも，直接ローカル変数へ書き込むことにより，演算結果をスタックへ積み，取り除く操作を省略する．このような手順を以下では命令畳み込みと呼ぶ[2]．

## 4. Java プロセッサ

### 4.1 命令畳み込みを考慮に入れた基本構成

本章では，前章において述べた基本的な高速化手法を実現するためのプロセッサ構成について説明する．

ヒープ読み出しやメソッド呼び出し時に必要となるアドレス変換の高速化は，以前の結果を変換表に登録しておくことにより行う．必要となる変換表は以下の 3 つである．

(1) **オブジェクト変換表**：オブジェクトリファレンスを検索キーとし，各オブジェクトの先頭アドレスおよび属性を保持する．

(2) **フィールド変換表**：コンスタント・プールへのインデックスを検索キーとし，各フィールドのオフセットを保持する．

(3) **メソッド変換表**：コンスタント・プールへのインデックスを検索キーとし，各メソッドの先頭アドレスおよび属性を保持する．

ローカル変数上の値を直接参照するために，高速アクセスが可能なローカルレジスタを設け，ローカル変数間の演算をレジスタ間演算とする．しかし，無限個のローカルレジスタを用意することはできず，レジスタに格納できないものは，主記憶に保持する．主記憶上のローカル変数の値を使用して演算を行う場合には，これらの値を一度保持しておくレジスタが必要となる．この場合にはオペランド・スタックのスタックレジスタを介して実行を行う．スタックレジスタの数は，多くとも 1 命令が 1 度に使うスタックのエントリ数だけあれば良い．また，ローカル変数の主記憶アクセスの高速化にはローカルキャッシュを用意する．同様に，

| 命令の種類 | compress | jess | db | javac | mpegaudio | mtrt |
|---|---|---|---|---|---|---|
| ローカル変数の読み出し | 32.5 | 37.3 | 40.5 | 35.4 | 32.8 | 33.4 |
| ヒープの読み出し | 19.4 | 20.7 | 25.1 | 17.9 | 20.5 | 17.4 |
| 演算，型変換，スタック操作 | 15.9 | 1.8 | 6.8 | 6.3 | 16.5 | 8.4 |
| 条件分岐 | 6.1 | 10.9 | 8.1 | 9.3 | 3.3 | 3.6 |
| 定数のプッシュ | 7.2 | 5.6 | 1.3 | 5.3 | 12.8 | 5.8 |
| メソッド呼び出し | 1.8 | 6.5 | 3.5 | 7.2 | 1.0 | 13.1 |
| メソッドリターン | 1.8 | 6.3 | 3.5 | 6.1 | 0.9 | 13.1 |
| ローカル変数への書き込み | 9.1 | 5.4 | 6.9 | 4.5 | 6.3 | 1.7 |
| ヒープへの書き込み | 4.7 | 1.5 | 1.1 | 4.5 | 3.3 | 2.4 |
| 無条件分岐 | 0.4 | 0.9 | 1.1 | 1.3 | 0.4 | 0.2 |
| その他 | 1.0 | 3.0 | 1.9 | 2.2 | 2.2 | 0.7 |

表1　命令の出現頻度 (単位は%)

オペランド・スタックに対しても高速化のためスタックキャッシュを用意する．

上に述べた点を考慮に入れて仮定したプロセッサ構成を図2に示す．

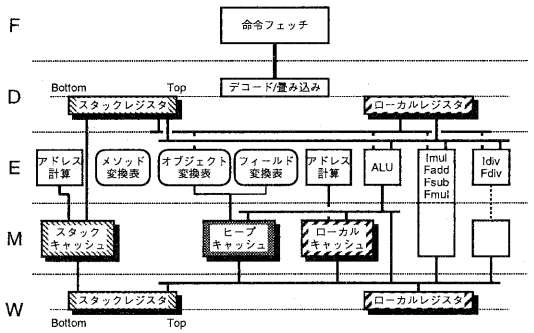


図2　Java プロセッサの構成

このプロセッサのパイプラインはF,D,E,M,W の5ステージから構成されている．ローカル変数のうち，ローカル変数番号の小さい順に一定個数をレジスタ上に保持することにより，オペランド・スタックを介さずに直接参照することを可能としている．一方，レジスタに対応付けられなかったローカル変数はMステージにおいてローカルキャッシュから読み出しを行い，一旦スタックレジスタ上に格納した後，演算を行う．スタックレジスタはDステージにおいて読み出す．オペランドスタックのトップに対するプッシュ/ポップに伴って必要となる，スタック・ボトムからキャッシュへのデータ退避，および，キャッシュからスタック・ボトムへのデータ供給は，必要に応じて自動的にプロセッサ内部で処理されると仮定した．前述の3つの変換表はEステージ，キャッシュはMステージにおいて各々参照される．

ローカルレジスタ上の値が更新される毎に，キャッシュへの書き戻しを行うと実行速度の低下をまねく．これを避けるため，キャッシュへの書き戻しはメソッドの呼び出し時に行うこととした．またメソッドからのリターン時には，キャッシュに退避したレジスタの値を復元する．この他，実行するメソッドを切り替える時には，引数および返り値の受け渡しを行う必要がある．この操作には，スタックレジスタとローカルレジスタの間でのデータの移動が必要となる．また，メソッド呼び出し，リターン時には，プロセッサの内部情報の退避，復元を行う必要がある．

演算器は，乗算および浮動小数点数の加減算については，データ依存がある場合に2サイクルを要するパイプライン動作，除算はデータ長に応じて16ないし30サイクルを要する非パイプライン動作と仮定した．これ以外の演算は，1サイクルのパイプライン動作を仮定した．データ依存によるパイプラインハザードの削減のため，各ステージ間にデータバイパス機構を備えている．

### 4.2　各機構の詳細

| プログラム名 | 4 | 8 | 16 | ∞ |
|---|---|---|---|---|
| compress | 30.7 | 37.8 | 41.6 | 41.6 |
| jess | 20.7 | 23.6 | 24.0 | 24.0 |
| db | 19.2 | 19.6 | 19.9 | 19.9 |
| javac | 22.6 | 26.4 | 26.4 | 26.4 |
| mpegaudio | 24.3 | 29.7 | 40.1 | 40.1 |
| mtrt | 24.6 | 25.9 | 25.9 | 25.9 |

表2　レジスタ数を変化させた場合の，命令畳み込みによって削減可能な動的ステップ数の割合 (単位は%)

図2の構成では，ローカルレジスタに格納することができるローカル変数の数が，命令畳み込みの効果に影響を与える．表2にローカルレジスタ数を変化させた場合に，命令畳み込みによって削減可能な動的ステップ数の割合を示す．レジスタ数を8とした場合に，レジスタ数を無限と仮定した時と，ほぼ同じ効果が得られることがわかる．

また，各変換表のうちフィールド変換表およびメソッド変換表は，表3に示すとおり，高々300程度のエントリ数があれば十分であるのに対し，オブジェクト変換表は，極めて多くのエントリを必要とする．これは，メソッドおよびクラスは静的に数が決定されるのに対し，オブジェクトは動的に数が決定されるからであると考えられる．このことから，プロセッサ上では，

| プログラム名 | オブジェクト変換表 | フィールド変換表 | メソッド変換表 |
|---|---|---|---|
| compress | 2229 | 112 | 199 |
| jess | 80163 | 148 | 223 |
| db | 135977 | 111 | 208 |
| javac | 94920 | 167 | 267 |
| mpegaudio | 4477 | 223 | 226 |
| mtrt | 503672 | 128 | 225 |

表 3　変換対を全て登録するのに必要なエントリ数(単位は個)

フィールドのオフセットおよびメソッドのオフセットを求めるには，全エントリを収容できるハードウェアを用意し，変換対が変換表にない場合のレイテンシは無視できると仮定する．

これに対しオブジェクト変換表は，全エントリを登録するだけのハードウェアを用意することが非現実的であるため，現実的なエントリ数と効果との関係を探る必要がある．そこで変換表のエントリ数と，変換表上にオブジェクトの情報が存在する割合を調査した結果，4096 エントリに対してヒット率が約 93.7% と 100%には届かなかった．これはオブジェクトの寿命が短いものが多いためと考えられる．以上のことから，オブジェクト変換表の参照オーバヘッドは，無視することができないと言える．

| プログラム名 | 16K | 64K |
|---|---|---|
| compress | 87.8 | 95.9 |
| jess | 85.8 | 91.1 |
| db | 79.2 | 82.8 |
| javac | 88.0 | 91.9 |
| mpegaudio | 95.1 | 97.0 |
| mtrt | 82.8 | 88.5 |

表 4　ヒープキャッシュのヒット率(単位は%)

ローカルキャッシュおよびスタックキャッシュは，参照される範囲が現在実行中のメソッドで扱うものだけに限定される．SPEC JVM 98 において，1 つのメソッドを実行中に参照されるローカル変数とオペランド・スタックの最大数は 31 および 13 であった．このため局所性が非常に高く，エントリ数の少ないキャッシュでもヒット率は極めて高い．一方，ヒープキャッシュの構成をダイレクトマップ，ラインサイズを 64 バイトと仮定し，容量が 16K バイトと 64K バイトの場合において，ヒット率を測定した結果を表 4に示す．

以上のことから，本プロセッサ上において主に考慮すべきパイプラインハザードとして，1) オブジェクト変換表上に求める値が存在しなかった場合；2) ヒープキャッシュがミスヒットして主記憶アクセスを行う場合；3)M ステージにおいて生産した値を次命令の E ステージにおいて消費する場合；4) メソッドの呼び出しおよびリターン；5) 多くのサイクル数を要する演算；の 5 つを取り挙げることとした．

## 5. データ投機とデータ再利用

### 5.1 データ投機

データ投機とは，命令の完了を待たずにその命令が出力する結果を予測し，予測値を用いて後続の命令の実行を開始する手法である[4]．ただし，後から予測されたデータが正しくないことが判明した時に，予測に基づいて変更したメモリの状態を戻す必要があり，予測を行った時点でのメモリの状態を保存しておく機構が必要となる．

本稿では，スーパスカラ実行を仮定せず，まずは図 2の構成の範囲内におけるデータ投機の効果について調査した．データ投機により高速化が可能なのは，前述のパイプラインハザードが発生する場合であり，具体的には，1) ヒープを参照する命令；2) ローカルキャッシュからの読み出し結果を直後に使用する命令；3) 整数乗除算，浮動小数点演算命令；の 3 つの命令が対象となる．メソッドの呼び出しおよびリターンについてはデータ投機の対象としていない．これらの命令は，M ステージにおいて値を生産するために，次の命令の E ステージにおいて値を必要とする場合には，パイプラインハザードが生じる．ところで，キャッシュへのアクセスを伴うローカル変数の読み出しおよびヒープの読み出しは，主記憶アクセスが必要となる場合には，次の命令において E ステージで値を使用しない場合でも，パイプラインハザードの原因となりうる．ただし，ローカル変数の読み出しに関しては，キャッシュのヒット率が十分に高いので，キャッシュミスの場合はデータ投機の対象とはしない．一方，ヒープアクセス時に必要となるオブジェクト変換表の参照において，求める値が変換表上に存在しない場合のパイプラインハザードについては，前述のように発生頻度を無視することができないため，データ投機の対象とする．

データ予測の手法には，最も簡単な機構である Last Value Prediction[4]を仮定している．この手法で予測される値は，前回その命令が生産した値と同じ値である．

### 5.2 データ再利用

データ再利用とは，実行結果を保存しておき，再度同じ入力データを用いて実行する場合に，実行結果を再利用することにより実行を省略し高速化する手法である[5)6)]．データ再利用はデータ投機とは異なり，再利用する値は必ず正しいものでなければならない．使用する値が有効であるかの判断を，データ投機では使用後に行うのに対して，データ再利用では使用前に行う．

データ再利用においては，入力データが正しいかどうかの判断を行うために，どの命令からどの命令までを単位として再利用を適用するのかを決定する必要がある．データ再利用の単位としては，メソッド内の複数の命令列を単位とする方法と，メソッドを単位と

する方法が考えられる．メソッド内の命令列を単位とする場合には，命令をスキップする際にパイプラインハザードが生じる．このためスキップする命令数が少ない場合には効果が得られない．一方，メソッドを単位とする場合には，メソッドの呼び出し，および，リターンに伴うパイプラインハザードを削減することが可能になる．そこで本稿では，メソッドを単位とすることにした．

メソッドを単位としてデータ再利用を行う場合，再利用が可能かどうかの判断は，メソッドが呼び出されてからリターンするまでに行われたメモリリード全てについて，参照したアドレスと値が以前実行した場合と同じであるかどうかを調べることにより行う．またデータ再利用が可能である場合には，呼び出しからリターンまでに行われた全てのメモリライトを以前と同様に実行する．図3に，データ再利用のために保持する情報を示す．

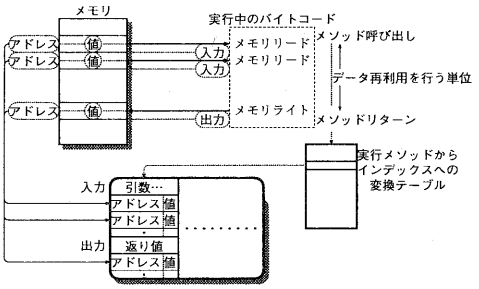


図3　データ再利用に必要なデータ

## 6. 測定条件および結果

評価にはSPEC JVM 98[1]を用いた．このベンチマークには，s1, s10, s100と3段階の実行サイズの目安が存在する．測定にはKaffeを用いた．なお，Kaffe上ではSPEC JVM 98のjackが動作しなかったため，評価の対象から外した．

まず，プログラムサイズ毎に命令畳み込みによって削減することができた動的ステップ数の割合を表5に示す．

次にデータ投機の効果について示す．パイプラインハザードは，今回データ投機の対象とした命令，および，メソッドの呼び出し，リターン命令のみで発生するものとする．命令畳み込みの効果とは独立して測定するために，ローカルレジスタ数は0と仮定し，全てのローカル変数はローカルキャッシュ上に存在すると仮定する．ローカルキャッシュからの読み出し結果を直後の命令で使用する際のペナルティは1サイクルと仮定する．ヒープキャッシュの構成は，ダイレクトマップ，ラインサイズ64バイト，容量64Kバイトと仮定し，オブジェクト変換表は4096エントリと仮定する．ヒープキャッシュのミスヒットのペナルティ，および，オブジェクト変換表のミスヒットのペナルティはともに20サイクルと仮定し，単精度除算の実行時に生じるパイプラインハザードは16サイクル，倍精度除算の実行時に生じるパイプラインハザードは30サイクルと仮定する．メソッド呼び出し，リターン時に必要な内部状態レジスタの退避および復元には10サイクルを要するものとし，また，退避および復元の必要となるローカル変数のサイズは8であるとする．ローカル変数は1サイクルにより退避および復元が可能であるとする．これらを合計すると，1回のメソッド呼び出しおよびリターンに要するサイクル数はそれぞれ18となる．また，予測が外れた時のペナルティは0と仮定する．これらの仮定のもとで，データ投機により削減することができたサイクル数を表6に示す．この結果によると，データ投機の手法により3.8%～29.1%のサイクルを削減することが可能である．

最後にデータ再利用の効果を表7示す．登録可能な総エントリ数は32768と仮定した．メソッド呼び出しおよびリターンに要するサイクル数の仮定は，データ投機の評価で用いた値を使用する．また，ヒープとの比較はメソッド呼び出しに先立っては行わず，メソッド呼び出しとオーバラップして実行するものとし，1サイクルにつき1つのヒープ上の値との比較が可能であると仮定する．つまり，ヒープ上の値と比較する回数が増加すれば，データ再利用の効果は少なくなるものとする．この値を用い，さらに，パイプラインハザードがメソッドの呼び出し，リターン時以外に一切生じないものと仮定すると，表7に示すように，0.1%から47.1%のサイクルを削減することが可能である．

## 7. 考　　察

スタックマシンは値を操作するために，一旦スタックへデータを複写する．スタックマシンのこの特徴は，実行する命令数の増加につながるものである．命令畳み込みの手法は，この特徴に基づく命令数の増加を抑えるものである．調査の結果も，実際に効果があることを裏付けている．また，命令畳み込みの手法は，他の手法との組み合わせにおいても問題なく適用できるものである．

データ投機の手法は，予測が正しくなかった場合に，メモリの状態を予測を行った時点の状態にまで戻す機構が必要となるために複雑なハードウェアが必要となり，実際にはより多くのペナルティを要すると考えられる．このことを考えると，表6の数値は魅力的であるとは言い難い．

データ再利用の手法は，ベンチマークごとに結果にばらつきが存在する．これは，ベンチマークプログラムのソースコードの記述方法に大きく由来していると

| サイズ | compress | jess | db | javac | mpegaudio | mtrt |
|---|---|---|---|---|---|---|
| s1 | 41.6 | 24.0 | 19.9 | 26.4 | 40.1 | 25.9 |
| s10 | 41.8 | 31.3 | 20.2 | 27.8 | 40.3 | 25.1 |
| s100 | 42.0 | 29.6 | 22.3 | 27.6 | 40.3 | 23.4 |

表 5　命令畳み込みにより削減可能な動的ステップ数の割合 (単位は%)

| サイズ | compress | jess | db | javac | mpegaudio | mtrt |
|---|---|---|---|---|---|---|
| s1 | 26.8 | 15.3 | 9.7 | 7.8 | 25.4 | 7.8 |
| s10 | 29.1 | 17.9 | 14.2 | 12.1 | 27.9 | 6.6 |
| s100 | 28.6 | 13.6 | 16.4 | 14.9 | — | 3.8 |

表 6　データ投機により削減可能なサイクル数の割合 (単位は%)

| サイズ | compress | jess | db | javac | mpegaudio | mtrt |
|---|---|---|---|---|---|---|
| s1 | 11.1 | 26.5 | 10.8 | 4.70 | 10.4 | 47.1 |
| s10 | 7.89 | 44.9 | 14.3 | 15.0 | 7.58 | 1.10 |
| s100 | 12.0 | 44.3 | 14.3 | 16.8 | 10.1 | 0.0979 |

表 7　データ再利用により削減可能なサイクル数の割合 (単位は%)

考えている．他のオブジェクトの内部変数を直接参照したり，単純なデータ構造であるためにクラスとして定義しない，などの記述を行うと，今回採用した方針でのデータ再利用が難しくなる．しかし，このような記述の方法はオブジェクト指向を意識したプログラミングではなく，Java を使用する目的に合ったものとは言い難い．オブジェクト指向を意識してプログラムの記述を行う場合に，本プロセッサおよびデータ再利用の手法はより高い性能を発揮すると考えている．mtrt は，表 1の結果よりメソッド呼び出しが多く，データ再利用に適したプログラムであると考えられる．しかし，浮動小数点数を扱うため，ベンチマークサイズが大きくなるに従い，実行時のメソッドの入力データの取り得る値の種類が膨大となり，データ再利用の性質上登録可能なエントリ数が制限され，その効果が得られなかったものと考えられる．

データ投機とデータ再利用の手法を同時に適用することは不可能である．データ投機を用いると，ある時点での正しいメモリの値を知ることが不可能となるため，データ再利用が可能であるかどうかの判断ができないからである．この 2 つの手法を，測定結果をもとに比較すると，データ再利用はデータ投機よりも効果的であると考えられるが，その効果には偏りがあり，むしろ，Java の目的に沿ったプログラムであるほどより高い効果が得られると考えている．

## 8. ま と め

本稿では，バイトコードの命令出現頻度およびスタックマシンの特徴をもとに，高速化の手法について検討を行った．特に，命令畳み込みおよび変換表の機構を導入したプロセッサの構成を仮定した．その上でデータ投機およびデータ再利用を行った結果，それぞれの方式の特徴に応じた効果があることが判明した．データ再利用の手法はデータ投機よりも効果的であると考えられるものの，その効果には偏りが見られた．今後は，命令の発行多重度の増加によるデータ投機手法の性能向上，高速化手法を組み合わせた場合の総合的評価をすすめる予定である．

**謝辞**

本研究に御協力いただきましたオムロン株式会社の宮田佳昭氏および財団法人京都高度技術研究所の神原弘之氏に感謝します．

## 参 考 文 献

1) SPEC JVM 98 VERSION 1.03, the Standard Performance Evaluation Corporation, (1998).
2) L.-R.Ton, L.-C.Chang, M.-F.Kao, H.-M.Tseng, S.-S.Shang, R.-L.Ma, D.-C.Wang and C.-P.Chung: Instruction Folding in Java Processor, 1997 International Conference on Parallel and Distributed Systems, (1997).
3) 木田 裕之, 木村 晋二, 高木 一義, あべ松 竜盛, 渡辺勝正: 再構成可能部を持つ java プロセッサ, 電子情報通信学会総合大会, pp. 442-443(1999).
4) M.H.Lipasti and J.P.Shen: Exceeding the Dataflow Limit via Value Prediction, 29th International Symposium on Microarchitecture, pp. 226-237(1996).
5) A.Sodani and G.S.Sohi: Dynamic Instruction Reuse, 24th Annual International Symposium on Computer Architecture, pp. 194-205(1997).
6) A.González, J.Tubella and C.Molina: Trace-Level Reuse, 1999 International Conference on Parallel Processing (Hardcover), (1999).