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[†1] 小川 洋平[†2] 山田 克樹[†3]
中島 康彦[†4] 富田 眞治[†3]

Javaバイトコードをハードウェアにより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，より多くの記憶域を必要とするものの，データ再利用の手法がデータ投機の手法よりも有望であることが分かった．

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

DAISUKE SHIGETA,[†1] YOUHEI OGAWA,[†2] KATSUKI YAMADA,[†3]
YASUHIKO NAKASHIMA[†4] and SHINJI TOMITA[†3]

This paper describes the effectiveness of Instruction Folding, Data Value Speculation and Data Value Reuse against the hardware 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言語は，SmalltalkやC++に代表されるオブジェクト指向プログラミング言語に分類される．しかし，ネットワークコンピューティングを設計思想の中心に据えている点が従来の言語と大きく異なる点である[1]．特に，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安全に実行することができる．1）実行環境がプラットフォームに依存しない；2）Java言語から得られるJavaバイトコードのサイズはC++から得られる実行形式ファイルよりもきわめて小さい；3）実行時の安全性が高い；という特徴[13]は，近年急速に普及しつつあるインターネット上において，より高速かつ高機能なサービスを提供するという要求に合致するものである．このため，Java言語およびJavaバイトコードはインターネット上で広く利用されており，今後ますます普及していくものと思われる．

Javaバイトコードの実行環境であるJava仮想マシン（以下，JVM）の実装には，大きく以下の3つの方式がある．

インタプリタ方式 ソフトウェアによりバイトコードを逐次解釈実行する方式である．実行に必要なメモリ量は少ないものの実行速度が遅い．

静的/動的命令変換方式 高速化のためにバイトコー

†1 シャープ株式会社通信システム事業本部モバイルシステム事業部第1技術部
Engineering Department I, Mobile Systems Division, Communication Systems Group, SHARP Corporation
†2 日本IBM株式会社サービス事業ITS事業部ネットワークサービス
Network Service, Integrated Technology Service Division, Service Department, IBM-JAPAN Corporation
†3 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
†4 京都大学大学院経済学研究科
Graduate School of Economics, Kyoto University

ドをネイティブコードに変換してから実行する方式である．おおまかに，静的変換は全体を変換してから実行する方式，動的変換は必要に応じて部分的な変換を行いながら実行を進める方式である．最適化処理およびネイティブコードの格納のために十分なメモリを確保できる場合にきわめて有効である．

ハードウェア直接実行方式 ハードウェアによりバイトコードを逐次解釈実行する方式である．高速性を追求しつつも，メモリ量や消費電力に対する制約がある場合に有効である．

本稿は，機器制御のためにJVMを組み込み，ネットワークを経由して制御用データおよび制御プログラム自身を送り込むようなシステムへの適用を想定している．メモリ量や消費電力に制約があるために，静的/動的命令変換方式を採用することができない一方，可能な限り高速化を図る必要がある状況では，ハードウェア直接実行方式を採用する必要がある．

さて，JVMはスタックマシンであるため，一般的なRISCプロセッサのように命令レベル並列性をバイトコード列から直接抽出し，複数のバイトコードを並列実行することにより高速化を図ることは難しい．すなわち，バイトコードの実行に要するサイクル数を減らす，あるいは，実行すべきバイトコードそのものを減らす必要がある．前者のためには命令畳み込み，また後者のためには，データ投機およびデータ再利用の適用が考えられる．

ただし，データ投機はデータ再利用に比べて，予測が外れた場合に再実行するためのハードウェア機構やペナルティサイクルというコストがかかることから，データ再利用よりもきわめて高い性能を得られる見込みがなければ，実際に採用することは難しい．

本稿では，まず3章において，JVMの一般的特徴について述べる．次に，4章におけるバイトコードに関する調査結果をもとに，5章において，データ投機とデータ再利用の効果を定量的に比較するために，有限個のレジスタ，有限容量のキャッシュを有する現実的な5段パイプライン構造を仮定する．この上に，バイトコードの特徴に合ったハードウェア機構を追加することにより，データ投機とデータ再利用を適用しない範囲において高性能を得ることを目指す．以上の準備を行ったうえで，6章において，本プロセッサに対するデータ投機およびデータ再利用の適用手法について述べる．7章において，各高速化手法の効果を測定した後，8章において考察を行う．

2. 関連研究

高級言語やオブジェクト指向プログラミング言語により記述されたプログラムの高速実行に関しては，多くの研究が行われている[4]．

LISPやPrologは，スタックアーキテクチャによく適合する言語である．1980年代には，これら高級言語の高速実行に適したスタックアーキテクチャに関する研究が活発であった．MITのAI研が開発したLispマシンCADR[3]は，スタックの先頭部分を保持する2つのメモリ（4Kバイトと128バイト）を並列アクセス可能なキャッシュとして利用していた．NTT電気通信研究所のELIS[5]は，1986年に商用化されたLispマシンである．128Kバイトのスタックメモリと3組のスタックトップレジスタを備えていた．

このほか，スタックアーキテクチャの最適化に関する研究として，Symbolics社やLISP Machines社のLispマシン，理化学研究所と東京大学が設計・開発したLispマシンFLATS[2]，第五世代コンピュータ研究開発プロジェクトのPrologマシンPSI[8]があげられる．

NTTのTAO/SILENT[6],[7]では，リエゾン（命令畳み込み），メソッド検索や変数管理を高速化するハードウェアハッシュ関数，自動バイトコードキュー，スタックポインタに基づく自動キャッシングレジスタなどの手法が検討されており，Javaバイトコードとの比較が行われている．

Javaバイトコードにおける命令畳み込みに関しては，4命令を畳み込む条件下において約38%の命令を削減できるとする報告がある[12]．ただし，本報告ではスタックの内容を保持する内部レジスタ数を無限個と仮定しており，また，実用的なバイトコードにおいて高い命令出現頻度を占めるヒープ参照を高速化するための考慮がなされていない．Sun MicrosystemsのpicoJava[13],[14]も同様に，内部レジスタを設け命令畳み込みを行うアークテクチャを採用しており，約28%の命令を削減できるとしているものの，内部レジスタ数が64個と多いことが，一般的なRISCプロセッサに対する優位性の主張を弱めている．本稿では，スタックレジスタとローカルレジスタを合計して24個という小さなレジスタ空間でも十分な効果が得られることを述べる．

データ投機[15],[16]およびデータ再利用[17]~[20]に関しては近年さかんに研究が行われている．しかし，バイトコードの実行に対して適用した研究成果は，まだ報告されていない．我々は，バイトコードの特徴である，

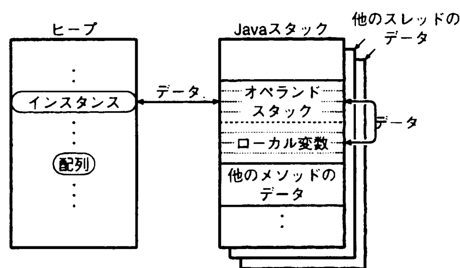

図1 JVMのデータ域
Fig. 1 Data area of JVM.

1）演算およびロード結果が必ずスタックトップに格納されること；2）各命令に関連する記憶域がオペランド・スタック，ローカル変数，ヒープ領域のいずれであるかがオペコードにより容易に区別できること；に注目した．すなわち，命令実行結果が多くの汎用レジスタに格納され，また，オペコードだけではロード結果が局所変数であるか大域変数であるかの区別が難しい一般的なRISCプロセッサに比べ，データ投機およびデータ再利用を適用するための機構を単純化することができると考えた．

3. Java仮想マシン

JVMは，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，バイトコード，シンボル・テーブル，および，他の付随的な情報を保持したものである[9]．JVMが使用するデータ域は**図1**のように大きく3つの部分からなる．

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，結果を返す処理を行う．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当てを行うためのデータ域である．各スレッド間で共有される．

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，オブジェクトの生成と操作，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．バイトコードを記述どおりに実行すると，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いえる．

4. 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 JVM98[10]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分かる．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．

次に，SPEC JVM98のうちjessを実行したときの，連続する2命令の出現頻度を**表2**に示す．このうち項番1および3は，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を示している．項番1の例は，参照するインスタンス変数を持つオブジェクトへのリファレンスを，また，項番3の例は，参照する配列のインデックスを，各々ローカル変数からスタックへと積んでいる．バイトコードとしてはこのような命令列となるけれども，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，以下の基本的な手法が高速化に寄与することは容易に推測することができる．

- 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．
- 命令畳み込み[12]を適用する．具体的には，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．

5. Javaプロセッサの構成

5.1 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

表1 命令の出現頻度（単位は%）
Table 1 Percentage of occurrences.

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
コンスタント・プールの読み出し	0.0	0.9	0.0	0.2	0.5	0.1
その他	1.0	2.1	1.9	2.0	1.7	0.6

表2 連続する2命令の出現頻度
Table 2 Occurrences of combination.

項番	命令の組合せ	頻度（%）
1	ローカル変数上のアドレス読み出し → インスタンス変数読み出し	13.3
2	インスタンス変数読み出し → ローカル変数の整数読み出し	4.4
3	ローカル変数の整数読み出し → 配列上のリファレンス読み出し	4.2
4	ローカル変数上のアドレス読み出し → ローカル変数上のアドレス読み出し	3.7
5	整数値比較条件分岐 → 整数定数プッシュ	2.8
6	ローカル変数へのアドレス書き込み → ローカル変数上のアドレス読み出し	2.2
7	インスタンス変数読み出し → 整数値比較条件分岐	2.0
:	:	:

3つである．

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．

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．

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・スタックに対応する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よいことから，8本程度で十分である．また，ローカル変数に関する主記憶アクセス高速化のためにローカルキャッシュを用意する．同様に，オペランド・スタックに関してスタックキャッシュを用意する．以上に述べた点を考慮し仮定したプロセッサ構成を**図2**に示す．

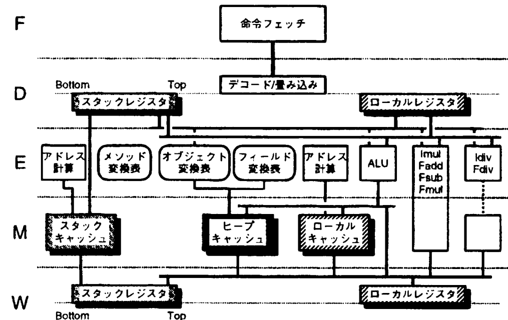

図2 Javaプロセッサの構成
Fig. 2 Structure of Java processor.

本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，いったん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/ポップにともなって必要となる，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ごとに，ローカル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．このほか，実行するメソッドを切り替えるときには，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，一般的な RISC プロセッサと同様，乗算および浮動小数点加減算については，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．

たとえば，図 2 の構成において，

(1) ローカル変数 A を読み出し，スタックへ積む，
(2) ローカル変数 B を読み出し，スタックへ積む，
(3) 加算を行い，結果をスタックへ積む，
(4) スタックトップをローカル変数 C へ書き込む，

の 4 命令を実行する場合を考えてみる．この 4 命令で扱うすべてのローカル変数がローカルレジスタ上に存在する場合には，これらの命令列はプロセッサ内部において 1 命令として実行する．すなわち，D ステージにおいてローカル変数を読み出し，E ステージにおいて加算を行う．演算結果は，W ステージにおいてローカルレジスタへ書き込む．

5.2 各機構の詳細

5.2.1 ローカルレジスタ

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 3** 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 16 とした場合に，レジスタ数を無限と仮定したときと同じ効果が得られることが分かる．また，レジスタ数を 8 とした場合，compress および mpegaudio において命令畳み込みの効果が若干減少するものの，全体として深刻な性能低下には至らないことが分かる．

表 3 レジスタ数を変化させた場合の，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%）
Table 3 Percentage of foldable bytecodes.

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 4 変換対をすべて登録するのに必要なエントリ数（単位は個）
Table 4 Required entries to hold all translation.

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 5 オブジェクト変換表の構成とヒット率の関係（単位は%）
Table 5 Hit ratio of object translation table.

	ウェイ数		
エントリ数	1	2	4
256	79.1	–	–
512	79.4	87.0	–
1024	83.2	87.3	91.0
2048	–	90.2	91.5
4096	–	–	93.7

5.2.2 変 換 表

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 4** に示すとおり，たかだか 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，きわ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ためと考えられる．このことから，本プロセッサ上では，フィールドのオフセットおよびメソッドのオフセットに関して，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，**表 5** に示すように，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

表6 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（単位は個）
Table 6 Maximum number of local variables and operand stack entries.

プログラム名	ローカル変数	オペランド・スタック
compress	24	13
jess	24	13
db	24	13
javac	24	13
mpegaudio	31	13
mtrt	24	13

表7 ヒープキャッシュのヒット率（単位は%）
Table 7 Hit ratio of heap cache.

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

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いえる．

5.2.3 キャッシュ

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．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・スタックの最大数を**表6**に示す．このように局所性が非常に高いため，エントリ数の少ないキャッシュでもヒット率はきわめて高いといえる．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7**に示す．キャッシュ容量やウェイ数を増加することによりさらなるヒット率の向上が期待できるものの，容量64Kバイトの場合でもヒット率が82.8%から97.0%であり，平均91.2%と90%を超えている．また，測定対象プログラムが異なるため単純比較はできないものの，文献15)に示されるように，一般的なRISCプロセッサにおけるロード命令の出現頻度が40%程度であるのに対し，表1に示すように，ヒープ読み出し命令の出現頻度は平均20.2%と少ない．以上のことから，性能評価時には容量64Kバイトを仮定し，ヒープキャッシュのミスペナルティを考慮することとした．

5.3 パイプラインハザードの要因

5.2節に述べたことから，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つを取りあげることとした．

6. データ投機とデータ再利用

6.1 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[15),16)]．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ときに，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，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を参照する命令；3）整数乗除算および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ともな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高いため，キャッシュミスに関するデータ投機の効果はないものとする．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[15)]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．図2において，データ投機を行った場合の動作は以下のようになる．

(1) Dステージにおいてその命令を予測の対象とするかどうかを判断する．

(2) 予測の対象とする命令である場合には，前回の実行結果をEステージにおいて取り出し，次の命令へフォワードする．並行して，予測した値が正しいかどうかの判断のため，命令の実行を開始する．

(3) 実際に実行した結果が予測値と異なる場合には，予測値を用いた実行結果を破棄し，正しい値を使用して以降の命令を再実行する．また，次に実行する場合に備え，Wステージにおいて正しい値を登録する．予測した値が正しい場合には処理を続行する．

6.2 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[17)~20)]．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ともな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すべてについて，参照したアドレスおよび値が以前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すべ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．次に，図2における動作を以下に示す．

(1) Fステージにおいて取得した命令にメソッド呼び出しが存在する場合，データ再利用が可能か否かの判断をメソッド呼び出しに先行して開始する．まず比較するメモリアドレスを得るために，メソッド呼び出し命令のオペランドから，登録してある記憶表へのインデックスを求める．

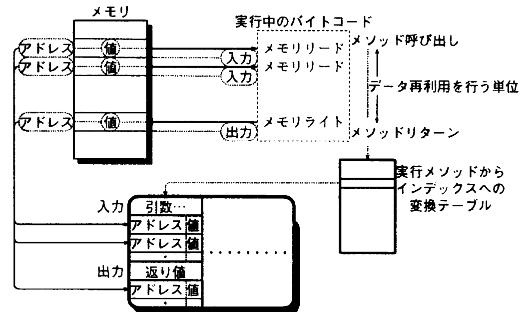

図3 データ再利用のための記憶表
Fig. 3 Structure of reuse table.

(2) データ再利用が可能か否かの判断は，以前実行したときに行ったメモリリードの結果と現在のメモリの値との比較により行う．現在のメモリ上の値のうちヒープ上に存在するものは，リードが1サイクルでは終了しない．そこで，メソッド呼び出しに先立ちリードを行い，以前のリード値とを比較する．

(3) (2)の比較結果が等しい場合，メソッド呼び出し命令のMステージ☆において残りのメモリ上の値の比較を行う．具体的には，メソッド呼び出しまでに比較が完了しなかったヒープ上の値，および，メソッド呼び出しの引数であるスタック上の値である．図2の構成では，メソッド呼び出しはパイプラインハザードをともなうため，これらの処理はオーバヘッドとはならない．

(4) (3)の結果，再利用が可能であると判断した場合には，メソッド呼び出しのWステージにおいて，以前実行した時点における実行結果をメモリに書き込む．メソッド呼び出しの次の命令が返り値を使用する場合には，同時に次の命令へと返り値をフォワードする．

(5) 再利用が不可能である場合には，実際にメソッドを呼び出し，実行結果を登録する．

ただし，メソッド内においてネイティブメソッドが呼び出された場合，登録中のメソッドすべてについて登録を取り消している．これは，ネイティブメソッドにおいて入出力などが発生する可能性があり，正しいデータ再利用を保証することができないためである．

7. 測定条件および結果

評価にはKaffe1.0b4[11)]およびSPEC JVM98を用いた．このベンチマークには，s1，s10，s100と3段階の実行サイズが存在する．なお，Kaffe1.0b4上では

☆ 直前の命令の結果が確定するのが最も遅い場合，その命令のMステージ（メソッド呼び出しのEステージ）に該当するため．

表8 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%）
Table 8 Percentage of foldable bytecodes.

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9 データ投機の対象となるバイトコードの比率（単位は%）
Table 9 Percentage of bytecodes for speculation.

サイズ	compress	jess	db	javac	mpegaudio	mtrt
s1	39.7	37.6	33.0	33.9	35.9	35.0
s10	39.4	47.2	46.4	35.9	36.2	35.6
s100	39.9	40.6	50.7	35.3	35.9	35.4

表10 予測が正しい確率（単位は%）
Table 10 Hit ratio of speculation.

サイズ	compress	jess	db	javac	mpegaudio	mtrt
s1	65.2	55.2	66.0	60.7	50.8	50.6
s10	66.9	62.3	53.2	58.2	49.2	37.5
s100	65.9	55.6	54.6	59.0	50.8	24.3

表11 データ投機により削減可能なサイクル数の割合（単位は%）
Table 11 Eliminable cycles on data value speculation.

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	28.3	3.8

jackが動作しなかったため，評価の対象から外した．

まず，ローカルレジスタ数を16とした場合に，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8**に示す．全体の算術平均は30.5%である．

次にデータ投機の効果について示す．まず，全実行命令のうちデータ投機の対象となる命令が何命令存在するかを**表9**に示す．全体の算術平均は38.5%である．

表10は，データ投機の対象とした命令に対し，予測が正しかった命令の割合を示している．算術平均は54.8%である．

パイプラインハザードは，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，リターン命令においてのみ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，すべ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，ヒープキャッシュは容量64Kバイト，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，単精度除算および倍精度除算の実行時に生じるパイプラインハザードは各々16，30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，また，退避および復元に必要となるローカル変数の個数は8とし，1サイクルにより1個の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，予測が外れたときのペナルティは0と仮定する．一方，Last Value Predictionを適用するためには，バイトコードの各1バイトごとに8バイトの演算結果を記憶する値予測表が必要となる．得られる効果の上限を求めるために，値予測表のエントリ数については無制限と仮定した．

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11**に示す．この結果によると，データ投機の手法により3.8%から29.1%のサイクルを削減することが可能である．なお，バイトコードの静的サイズは，compressが最も小さく56Kバイト，javacが最も大きく150Kバイトであることから，同じ結果を得るために最低限必要な値予測表の大きさは1.2Mバイト（8 × 150Kバイト）となる．

表 12　データ再利用（32768 エントリ）により削減可能な命令数およびメソッド数の割合（単位は%）
Table 12　Eliminable bytecodes and methods on data value reuse (32768 entries).

サイズ	compress	jess	db	javac	mpegaudio	mtrt
s1	3.86 (23.5)	14.0 (33.1)	8.14 (12.7)	2.40 (5.90)	2.47 (35.2)	17.1 (58.0)
s10	2.65 (18.0)	24.4 (58.3)	5.76 (25.9)	5.94 (20.2)	1.69 (31.0)	0.482 (1.31)
s100	4.16 (25.3)	26.0 (54.0)	5.44 (24.1)	7.32 (22.1)	2.33 (35.3)	0.0515 (0.111)

表 13　データ再利用（32768 エントリ）により削減可能なサイクル数の割合（単位は%）
Table 13　Eliminable cycles on data value reuse (32768 entries).

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 14　データ再利用（4096 エントリ）により削減可能なサイクル数の割合（単位は%）
Table 14　Eliminable cycles on data value reuse (4096 entries).

サイズ	compress	jess	db	javac	mpegaudio	mtrt
s1	6.77	24.4	10.5	4.46	9.93	46.1
s10	5.87	44.9	13.3	13.9	7.12	1.17
s100	7.38	42.3	14.0	13.4	9.65	0.0963

ただし，表 9 に示したように，データ投機の対象となるバイトコードの比率は半分以下である．効率の良い入れ替えアルゴリズムを適用することにより，値予測表のエントリ数を大幅に圧縮することが可能であると考えている．

最後に**表 12** にデータ再利用の効果を示す．表の上段は，全実行命令数に対する，データ再利用により実行不要となる命令数の比である．表の下段は，全呼び出しメソッド数に対する，データ再利用により実行不要となるメソッド数の比である．記憶表に対して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およびリターン時以外にはいっさい生じないものと仮定すると，**表 13** に示すように，記憶表の構成が 32768 エントリの場合，0.1%から 47.1%のサイクルを削減することが可能である．なお**表 14** に示すように，4096 エントリの場合では compress について若干の性能低下が見られた．記憶表の各エントリには 352 バイトを必要としたため，4096 エントリの場合，記憶表の大きさは 1.4 M バイトとなる．大幅な圧縮の可能性がある値予測表に比べて，データ再利用を効果的に行うために必要な記憶表はきわめて大きいといえる．

8. 考　　察

表 8 の結果から，SPEC JVM98 では，スタックレジスタが 8 本，ローカルレジスタが 16 本という比較的小さな構成でも，命令畳み込みにより平均 30.5%の動的ステップ数を削減できることが明らかになった．

一方，データ投機については，表 11 の数値は魅力的であるとはいい難い．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ことが予想されるためである．

ところで文献 15) において，本稿がデータ投機の対象としたバイトコードと同様，複数サイクルを要する RISC 命令についてデータ局所性（Figure 3）を命令出現頻度（Table 2）により加重平均した値を求めると，約 54.9%となる．命令セットおよび測定対象プログラムがまったく異なるため単純比較はできないものの，表 10 の算術平均（54.8%）とほぼ同じであることはきわめて興味深い．データ投機におけるヒット率に関しては，特にバイトコードが優れているとはいえないことが明らかになった．ただし，冒頭において述べ

たように，一般的なRISCプロセッサよりも単純な機構によりデータ投機を実現できる場合，バイトコードに対してデータ投機を適用する意味があるといえる．

データ再利用の効果については，ベンチマークごとに大きなばらつきが生じることが明らかになった．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依存するためと考えている．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いい難い．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の結果から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，ベンチマークのサイズが大きくなるに従い，メソッドの入力データ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．

さて，データ投機とデータ再利用を同時に適用することについて考察してみる．データ投機は入力を予測して演算を開始するため，演算結果が誤っている可能性がある．したがって，データ投機によって得られた結果に対してデータ再利用を適用すると，再利用結果を無効化する可能性が生じ，無効化の必要がないというデータ再利用本来の利点が失われる．この不都合は，データ再利用の入力のみを投機的に求めることから生じる．一方，入力と演算結果の両方を投機的に求め，データ再利用における表構造に対して正しいエントリをあらかじめ登録する手法を確立することができれば，一般的なデータ投機において問題となる予測ミス時のペナルティは発生せず，データ再利用におけるヒット率の増減問題として単純化することができると考えている．このような手法の確立は今後の課題である．

9. 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基本構成について詳細な検討を行った．そのうえでデータ投機およびデータ再利用を適用した結果，前者における予測値のヒット率およびサイクル数削減効果に関しては，特にバイトコードが優れているとはいえないこと，一方，後者については，オブジェクト指向を意識したプログラムにおいて良い効果が得られることが判明した．データ投機に対し，データ再利用の手法は，効果に偏りが見られるものの，より効果的であると考えている．ただし，後者のために必要となる記憶域の大きさは，前者に比べてきわめて大きいことから，現実のプロセッサに対して適用するためには，さらなる工夫が必要であるといえる．今後は，データ再利用の具体的な実装方式に関する研究，および，前述した，再利用のための表構造データをあらかじめ投機的手法により求めるための研究をすすめる予定である．

謝辞　本研究にご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) 青山幹雄：オブジェクト指向プログラミング言語の進化—Smalltalkから Javaへ至る道程，情報処理，Vol.41, No.1, pp.93–95 (2000).
2) Goto, E., et al.: Design of a Lisp chip into a system for military AI, *Electronics*, pp.95–96 (1987).
3) Greenblatt, R., et al.: *The LISP Machine in Interactive Programming Environments*, Barstow, D.R. (Ed.), McGraw-Hill (1986).
4) Koopman, P.J.: *Stack Computers:the new wave*, Ellis Horwood Ltd., Chichester, England (1989).
5) Takeuchi, I., et al.: A concurrent multiple-paradigm list processor TAO/ELIS, *Proc. 1987 Fall Joint Computer Conference – Exploring Technology: Today and Tomorrow*, pp.167–74 (1987).
6) 山崎憲一，天海良治，竹内郁雄，吉田雅治：TAO/SILENTのバイトコード実行方式，第1回プログラミングや応用のシステムに関するワークショップ（*SPA'98*）論文集 (1998).
7) 吉田雅治ほか：記号処理カーネル SILENTのハードウェア構成，情報処理学会計算機アーキテクチャ研究会報告，Vol.114, No.3, pp.17–24 (1995).
8) 金田悠紀夫：Prologマシン，森北出版 (1992).
9) リンドホルム，T., イェリン，F., 野崎（訳）：The Java 仮想マシン仕様，アジソン・ウェスレイ・パブリッシャーズ・ジャパン (1997).
10) SPEC JVM98 VERSION 1.03, the Standard Performance Evaluation Corporation (1998).
11) Kaffe.org: Welcome to Kaffe. http://www.kaffe.org/
12) Ton, L.R., et al.: Instruction Folding in Java

Processor, *1997 International Conference on Parallel and Distributed Systems* (1997).
13) McGhan, H. and O'Connor, M.: PicoJava: A Direct Execution Engine for Java bytecode, *IEEE Computer* (Oct. 1998).
14) O'Connor, J.M. and Tremblay, M.: PicoJava-I: The Java Virtual Machine in Hardware, *IEEE MICRO* (Mar./Apr. 1997).
15) Lipasti, M.H. and Shen, J.P.: Exceeding the Dataflow Limit via Value Prediction, *29th International Symposium on Microarchitecture*, pp.226–237 (1996).
16) Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction using Hybrid Predictors, *30th Annual International Symposium on Microarchitecture* (1997).
17) Sodani, A. and Sohi, G.S.: Understanding the Differences Between Value Prediction and Instruction Reuse, *31st Annual ACM/IEEE International Symposium on Microarchitecture* (1998).
18) Sodani, A. and Sohi, G.S.: Dynamic Instruction Reuse, *24th Annual International Symposium on Computer Architecture*, pp.194–205 (1997).
19) González, A., Tubella, J. and Molina, C.: Trace-Level Reuse, *1999 International Conference on Parallel Processing* (1999).
20) Huang, J. and Lilja, D.J.: Exploiting Basic Block Value Locality with Block Reuse, *5th International Symposium on High Performance Computer Architecture* (1999).

(平成12年2月7日受付)
(平成12年6月2日採録)

重田　大助

1975年生．1998年京都大学工学部情報工学科卒業．2000年同大学院情報学研究科通信情報システム専攻修士課程修了．同年シャープ（株）に入社．現在，同社通信システム事業本部モバイルシステム事業部第1技術部に勤務．

小川　洋平

1976年生．2000年京都大学工学部情報学科卒業．同年日本IBM（株）入社．現在，同社サービス事業ITS事業部ネットワークサービスに勤務．主に大規模なネットワークシステムの構築に従事．

山田　克樹（学生会員）

1975年生．1999年京都大学工学部情報学科卒業．同年同大学院情報学研究科通信情報システム専攻入学，現在に至る．次世代の計算機アーキテクチャ，Javaプロセッサ，効率的なJVMの実現等の研究に興味を持つ．

中島　康彦（正会員）

1963年生．1986年京都大学工学部情報工学科卒業．1988年同大学院修士課程修了．同年富士通（株）入社．スーパコンピュータVPPシリーズのVLIW型CPU，Mアーキテクチャ・命令エミュレーション，高速CMOS回路等に関する研究開発に従事．博士（工学）．1999年京都大学総合情報メディアセンター助手．同年同大学大学院経済学研究科助教授，現在に至る．計算機アーキテクチャに興味を持つ．IEEECS，ACM各会員．

富田　眞治（正会員）

1945年生．1968年京都大学工学部電子工学科卒業．1973年同大学院博士課程修了．工学博士．同年京都大学工学部情報工学教室助手．1978年同助教授．1986年九州大学大学院総合理工学研究科教授，1991年京都大学工学部教授，1998年同大学院情報学研究科教授，現在に至る．計算機アーキテクチャ，並列処理システム等に興味を持つ．著書「並列計算機構成論」「計算機システム工学」「並列処理マシン」「コンピュータアーキテクチャI」等．電子情報通信学会，IEEE，ACM各会員．平成7，8年度，10，11年度本会理事．