平成一二年三月一日発行（毎月一日一回発行）

經濟論叢

第165巻　第3号

Java 仮想マシンの高速化の可能性……中　島　康　彦　1

リスクをともなう社会での協力の進化的形成…藤　山　英　樹　29

自社の株式を対象とした売建
プット・オプション取引における
会計問題（2）……池　田　幸　典　47

カレツキの開発経済学（2）……山　本　英　司　68

現代欧州重電市場の構造変化と企業再編……岸　田　未　来　83

学　会　記　事

平成12年3月

京都大學經濟學會

経済論叢（京都大学）第165巻第3号，2000年3月

Java 仮想マシンの高速化の可能性

――データ投機とデータ再利用技術――

中　島　康　彦

概　要

今から約30年前に最初のマイクロプロセッサが開発されて以来，5年ごとに約10倍という脅威的なペースを保ちながら，プロセッサの処理性能が向上してきている。このような高速化は，半導体技術の進歩，および，コンパイラ技術に裏付けされたプロセッサアーキテクチャの進歩という両輪により成し遂げられてきたものである。後者に関しては，時間的並列性を利用したパイプライン化技術，空間的並列性を利用したスーパスカラや VLIW[1] などの命令レベル並列化技術を経て，現在では，演算結果そのものを予測して投機的に実行を行うデータ投機技術や，さらに，過去の演算結果を再利用するデータ再利用技術に関する研究が行われている段階である。データ投機とは，先行命令の実行が完了するのを待たずに，実行結果の予測に基づいて後続命令の実行を開始し，あとから検算を行う高速化技術，また，データ再利用とは，ある命令群に対する入力データが過去の入力データと一致した場合に，過去に記憶した出力データをそのまま使うことにより，入力データから出力データを得るための計算手順を省略する高速化技術である。本稿では，最近注目されている Java 実行環境に対して，データ投機やデータ再利用技術を適用した場合の効果について定量的に評価を行った。あるプロセッサ構成を仮定し，SPEC JVM98 に含まれるベンチマークプログラムを用いて測定した結果，全体に対する削減可能なサ

1) 〔8〕，〔17〕，〔18〕。

イクル数の割合は，データ投機の場合3.8%から29.1% (平均17.0)，データ再利用の場合0.1%から47.1% (平均16.6%) であった。このことから，予測ミス時に多くのペナルティが生じるデータ投機よりも，ペナルティが生じないデータ再利用のほうが有望であることがわかった。

以上の結果を元に，再利用表 (Reuse Buffer，以下 RB と略する) の挙動に関する調査を行ったところ，RB ヒット率が20.4%～76.0%とかなり高い数値を示すこと，また，再利用可否判定に用いる引数およびヒープ上データの個数が平均して各々3個未満と極めて少ないことが明らかになった。さらに，可否判定のためのデータ量が少ないことを利用して，投機的手法によりメソッドの引数を予測し，RB のエントリを投機的に用意する機構を提案する。本機構により，一般的なデータ投機において問題となるキャンセル時のペナルティを発生させることなく，投機的実行による高速化を図ることができると考えている。

I 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[2]。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極めて小さい，③ 実行時の安全性が高い，という特徴[3]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。

2) [15]。
3) [7]，[9]。

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極めて有効である。

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。前者のためにはデータ投機，また後者のためには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極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，第Ⅲ章において，JVMの一般的特徴について述べる。次に第Ⅳ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上で，第Ⅴ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。第Ⅵ章において，各高速化手法の効果を測定した後，第Ⅶ章以降において分析および考察を行う。

II 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。マサチューセッツ工科大学MITのAI研が開発したLispマシンCADR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。

この他，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[6]，第五世代コンピュータ研究開発プロジェクトの

4) [5]。
5) [13]。
6) [2]。

Prolog マシン PSI[7] が挙げられる。

NTT の TAO/SILENT[8] 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。

データ投機[9] およびデータ再利用[10] に関しては近年盛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。本論文では，バイトコードの特徴である，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。

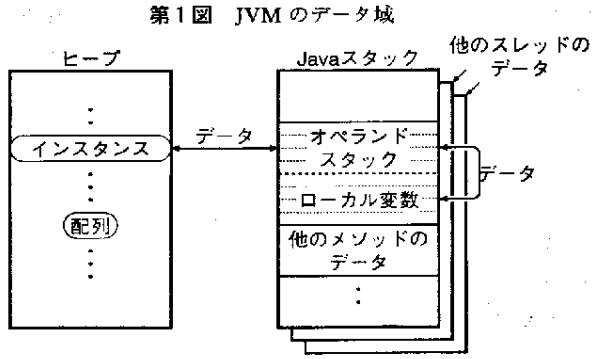
III　Java 仮想マシンと命令出現頻度

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[11]。JVM が使用するデータ域は第 1 図のように大きく 3 つの部分から成る。

オペランド・スタック　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

7) [16]。
8) [19]，[20]。
9) [6]，[14]。
10) [1]，[3]，[10]，[11]。
11) [21]。

第1図 JVMのデータ域



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。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。

第1表に，SPEC JVM98[12]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

12) [12]。

第1表　命令の出現頻度　(単位：%)

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

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。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高速化を図ることができると考えられる。

IV　Java プロセッサの構成

1　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。

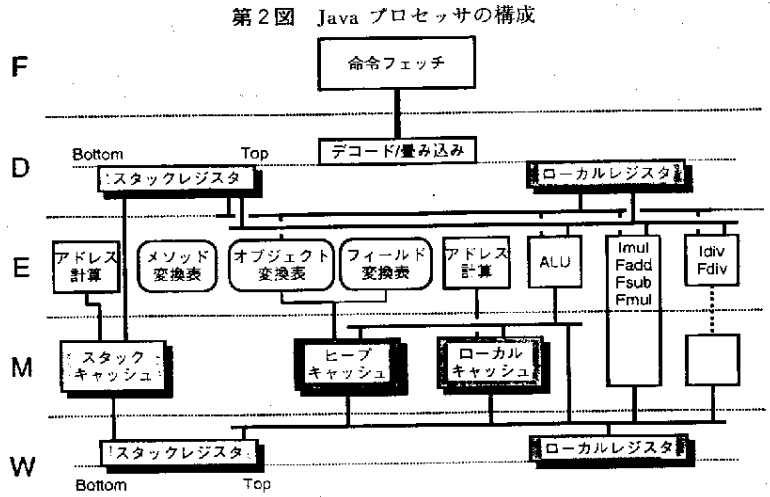
オブジェクト変換表　オブジェクト・リファレンスを検索キーとし，各オブジェクトの先頭アドレスおよび属性を保持する。

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良いことから，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第2図に示す。

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，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。

第2図　Java プロセッサの構成



ローカルレジスタ上の値が更新される毎に，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，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。

2 各機構の詳細

(1) 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，第2表に示すとおり，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，第3表に示すように，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。

(2) 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第4表に示す。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極めて高いと言える。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第5表に示す。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%を越えている。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

第2表 変換対を全て登録するのに必要なエントリ数 (単位：個)

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

第3表 オブジェクト変換表の構成とヒット率の関係 (単位：%)

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

第4表 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：個)

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

第5表 ヒープキャッシュのヒット率 (単位：%)

プログラム名	16 K	64 K
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

のの，文献〔7〕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，第1表に示すように，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。

3 パイプラインハザードの要因

第Ⅱ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つを取り挙げることとした。

V データ投機とデータ再利用

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。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，第2図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高いため，キャッシュミスに関するデータ投機の効果は無いものとする。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[13] 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。第 2 図において，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。

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

13) [6]。

法である。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。

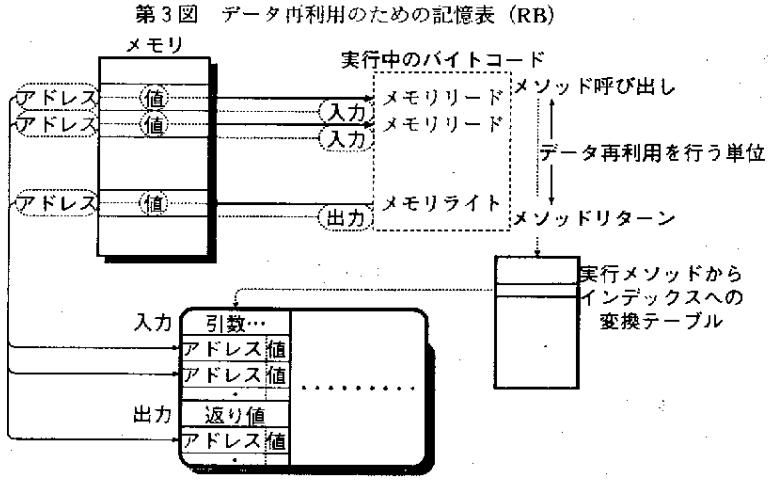
データ再利用においては，入力データが正しいかどうかの判断を行うために，どの命令からどの命令までを単位として再利用を適用するのかを決定する必要がある。データ再利用の単位としては，メソッド内の複数の命令列を単位とする方法と，メソッドを単位とする方法が考えられる。メソッド内の命令列を単位とする場合には，命令をスキップする際にパイプラインハザードが生じる。このためスキップする命令数が少ない場合には効果が得られない。一方，メソッドを単位とする場合には，メソッドの呼び出しおよびリターンに伴うパイプラインハザードを削減することが可能になる。そこで本稿では，メソッドを単位とすることにした。

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全てのメモリライトを以前と同様に実行する。第3図および第6表にRBの構造および諸元を示す。RBの各エントリは，過去にメソッドに渡された引数，メソッドが読み出したヒープ上データ（配列，フィールド，スタティック）の各アドレスおよび内容，書き込んだヒープ上データの各アドレスおよび内容，返り値を保持している。以下に，第2図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（RB）へのインデックスを求める。

(2) データ再利用が可能か否かの判断は，以前実行した時に行ったメモリ

第3図　データ再利用のための記憶表（RB）



第6表　RB の諸元

RB の総エントリ数	32768
同時に比較を行うエントリ数の上限	128
引数比較の上限	8
配列比較の上限	4
配列格納の上限	2
フィールド比較の上限	4
フィールド格納の上限	2
スタティック比較の上限	2
スタティック格納の上限	1

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[14] に

14) 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。

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。第2図の構成では，メソッド呼び出しはパイプラインハザードを伴うため，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，登録中のメソッド全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等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。

VI 測定条件および結果

評価にはKaffe1.0b4[15]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jackが動作しなかったため，評価の対象から外した。

まずデータ投機の効果について示す。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第7表に示す。全体の算術平均は38.5%である。

第8表は，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。ローカルキャッ

15) [4]。

第7表 データ投機の対象となるバイトコードの比率 (単位：%)

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

第8表 予測が正しい確率 (単位：%)

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

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，予測が外れた時のペナルティは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バイト毎に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第9表に示す。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

第9表 データ投機により削減可能なサイクル数の割合 (単位：%)

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

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。ただし，第7表に示したように，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。効率の良い入れ換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。

最後に第10表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。RB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一切生じないものと仮定すると，第11表に示すように，RB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。なお第12表に示すように，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。RB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，RBの大きさは1.4M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RBは極めて大きいと言える。

第10表 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：%）

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)

第11表 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：%）

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

第12表 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：%）

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

VII 考　　察

データ投機については，第9表の数値は魅力的であるとは言い難い。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〔7〕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（Table 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全く異なるため単純比較はできないものの，第8表の算術平均（54.8%）とほぼ同じであることは極めて興味深い。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言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言える。

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，第1表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。

VIII メソッド毎のRBヒット率の分析

さらに，現状のRBの挙動を分析する。各メソッド呼び出しに注目したRBの挙動を第13表および第14表に示す。RB参照回数は，メソッド呼び出しが第6表の条件を満たしRBに最低1エントリが登録されたためにRBの参照が行われた回数，RB参照率は全メソッド呼び出し回数に対するRB参照回数の割合である。RBヒット回数は，再利用が可能であった回数，RBヒット率は

第13表　メソッド呼び出しに注目した分析（s1）

	compress	jess	db	javac	mpegaudio	mtrt
メソッド呼び出し	17359045	579805	105180	629304	1136453	6044166
RB 参照回数	15538113	355057	52220	181516	842789	4608397
RB 参照率（%）	89.5	61.2	49.6	28.8	74.2	76.2
RB ヒット回数	4194462	192576	13920	37036	401213	3500838
RB ヒット率（%）	27.0	54.2	26.7	20.4	47.6	76.0
引数比較平均（個）	2.00	1.33	1.21	0.91	1.57	1.13
ヒープ比較平均（個）	2.00	1.41	1.57	0.79	0.86	1.13

第14表　メソッド呼び出しに注目した分析（s10）

	compress	jess	db	javac	mpegaudio	mtrt
メソッド呼び出し	18198757	6024455	1867579	4493950	9300517	24391637
RB 参照回数	15582153	4964854	1119574	1959292	7273108	862944
RB 参照率（%）	85.6	82.4	59.9	43.6	78.2	3.54
RB ヒット回数	3331209	3511692	464445	912528	2892786	317969
RB ヒット率（%）	21.4	70.7	41.5	46.6	39.8	36.8
引数比較平均（個）	2.00	1.91	1.58	1.13	1.59	0.77
ヒープ比較平均（個）	2.00	1.16	2.32	0.75	0.85	0.80

RB 参照回数に対する RB ヒット回数の割合である。また，引数比較平均は，ヒット時の引数の平均個数，ヒープ比較平均は，同じくヒープ上データの平均個数である。

RB ヒット率が20.4%～76.0%とかなり高い数値を示していることは興味深い。また，RB に登録可能な引数およびヒープ上データが最大 8 個および10（4＋4＋2）個であるのに対し，実際に必要となるのは各々平均 3 個未満と極めて少ないことがわかる。

RB ヒット率が低いメソッド呼び出しに先立ち，引数を予測し，正しい実行結果を予め RB に登録することができれば，RB ヒット率を上げたり，より小さな RB でもヒット率を維持できる可能性がある。比較の対象となる引数の個数が比較的少ないことが判明したために，引数を予測することが現実味を帯

びてくる。

次の段階として，メソッドの引数を予測する余地があるかどうかについて調査した。出現頻度が極めて高い一方，RB ヒット率が極めて低いメソッドが引数予測の候補として適当である。このようなメソッドを探索するために，jess と db を対象として，メソッド毎の RB 参照回数とヒット回数，および，メソッドに引数を加えた比較パターン毎の RB 参照回数とヒット回数を第4図，第5図，第6図，第7図に示す。

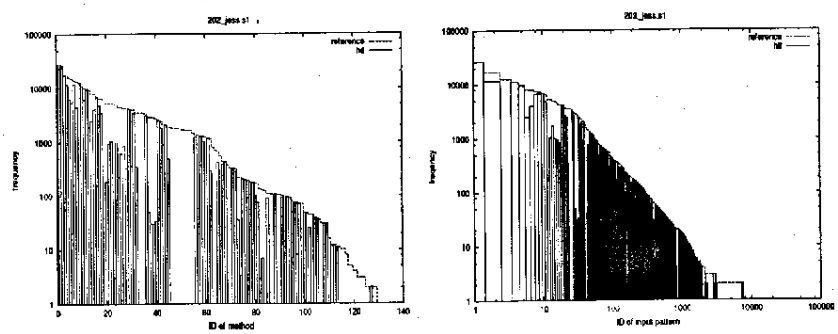
各図の左側のグラフは，RB 参照回数の多い順に，X 軸方向のメソッドを並べ替え，RB 参照回数およびヒット回数を対数表示したものである。また右側のグラフは，メソッドに引数を加えた比較パターンを X 軸方向とし，X 軸についても対数表示したものである。

第4図と第5図を比較すると，jess (s1) に比べて jess (s10) はわずか数個のメソッドの RB 参照回数および RB ヒット率が極めて高いために，全体の RB ヒット率を押し上げていることがわかる。jess (s10) の比較パターンを見ると，RB 参照回数の多いパターンのほとんどが RB にヒットしている。これに対して jess (s1) では，上位のパターンにヒット率が低いものが比較的多く含まれている。さらに調査した結果，jess (s1) においてヒット率の低いパターンは，jess 自身の入力データに直接関係するものであることがわかった。問題サイズが小さいために入力に関する処理の比率がかえって高くなると考えられる。

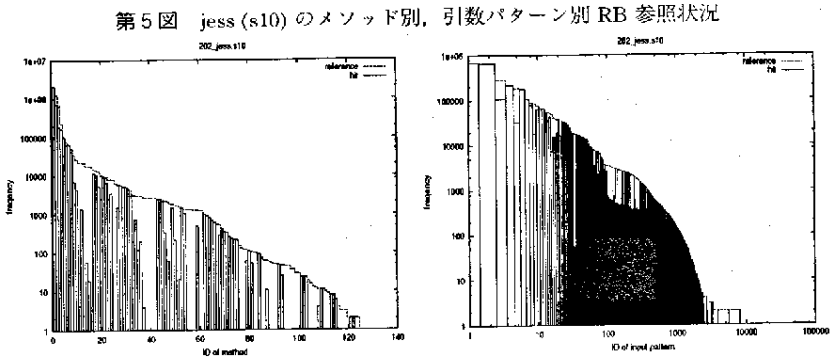
この傾向は第6図および第7図の比較においてより顕著である。db (s1) では入力データに直接関与するメソッドが上位の大部分を占めるために，RB ヒット率が低い。db (s10) では RB ヒット率の高いメソッドが上位を占めるために，全体の RB ヒット率が高くなっている。グラフを記載していない javac についても同様の調査結果が得られている。

現在の調査範囲では，出現頻度が極めて高い一方，RB ヒット率が極めて低いようなメソッドのほとんどは，残念ながら，プログラムの入力に直接影響を

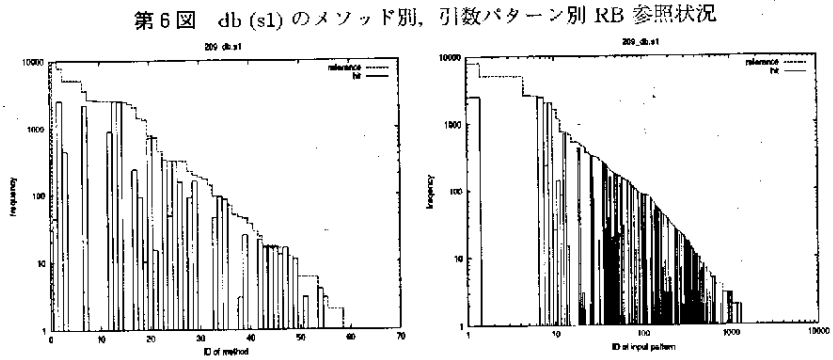
第 4 図　jess (s1) のメソッド別，引数パターン別 RB 参照状況



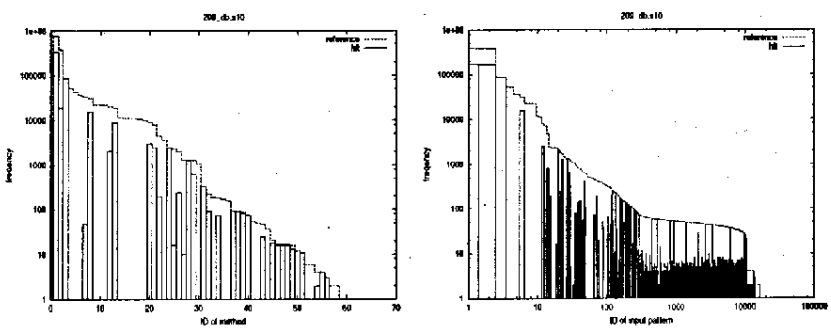
第 5 図　jess (s10) のメソッド別，引数パターン別 RB 参照状況



第 6 図　db (s1) のメソッド別，引数パターン別 RB 参照状況



第7図 db (s10) のメソッド別，引数パターン別 RB 参照状況



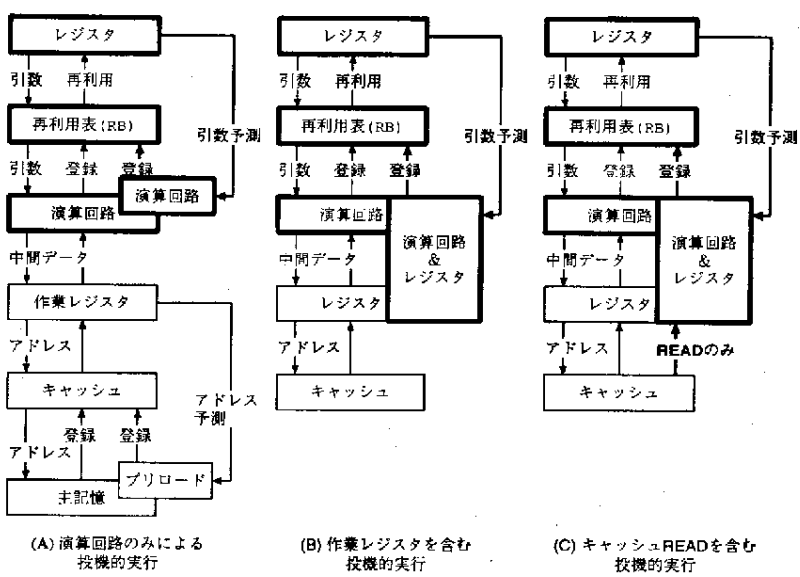
受ける引数を用いている。従って，現時点では投機的実行により RB ヒット率を格段に向上させることは難しい。他のベンチマークプログラムについても詳細に調査する必要がある。

IX RB への投機的プリロードによるデータ再利用の高速化

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。従って，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，RB に対して正しいエントリを予め登録（プリロード）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，データ再利用において RB ヒット率を上げたり，より小さな RB でも高いヒット率を維持できる可能性がある。

jess の実行の際，出現頻度がそれほど高くないものの，引数がほぼ単調に増加するメソッドが Kaffe1.0b4 内部のライブラリ中に見つかっている。また，mtrt において出現頻度が高いメソッドのいくつかは，ヒープ上の同一フィー

第8図　RB へのプリロード機構



ルドを連続して数回参照し，次のフィールド位置へ移動することを繰り返している。引数が単調に変化する場合には，過去の演算結果が再利用されることはなく，変化直後の RB ヒット率は0である。一方，単調変化を予測して引数および実行結果を RB に登録しておくことができれば，ヒット率をさらに高めることができる。第8図に，このような投機的手法を適用するためのハードウェア構成を例示する。(A)は，アドレス予測によるキャッシュへのプリロードとの類似性を示した図である。多くのサイクル数を要する単一演算においてソースデータに局所性が存在する場合には，通常の演算回路とは別に，予測したソースデータに対して演算を行う演算回路を用意すればよい。単一演算では不十分である場合には，(B)のように作業レジスタを追加することが考えられる。さらに，ヒープの内容も必要とする場合には，(C)のようにキャッシュからの読み出しパスの追加が必要となる。投機的演算結果をキャッシュではなく RB

にのみ書き込むことにより，引数の予測が外れた場合でも投機的実行のキャンセルを不要とすることができる。

X 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，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。その上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言えないこと，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，前者に比べて極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言える。また本稿では，RB の挙動を調査した。再利用可否の判定のためのデータ量が少ないことから，さらに，引数および実行結果を投機的手法により求めるデータ再利用の可能性について考察を行った。今後は，引数を効率良く予測するための具体的な方法，および，第8図の(A)から(C)の各構成におけるコストと効果について，定量的な評価を進める予定である。

謝辞 なお，本研究の一部は文部省科学研究費補助金（基盤研究(B)(2)課題番号12480072 ならびに 132558027）による。

参考文献

〔1〕 González, A., Tubella, J., Molina, C., "Trace-Level Reuse," *1999 International Conference on Parallel Processing*, 1999.

〔2〕 Goto, E., et al., "Design of a Lisp Chip into a System for Military AI," *Electronics*, pp. 95-96, 1987.

[3] Huang, J., Lilja, D. J., "Exploiting Basic Block Value Locality with Block Reuse," *The Fifth International Symposium on High Performance Computer Architecture*, 1999.

[4] Kaffe. org : Welcome to Kaffe, http://www.kaffe.org/

[5] Koopman, P. J., *Stack Computers : the New Wave*, Ellis Horwood Ltd., Chichester, England, 1989.

[6] Lipasti, M. H., Shen, J. P., "Exceeding the Dataflow Limit via Value Prediction," *29th International Symposium on Microarchitecture*, pp. 226-237, 1996.

[7] McGhan, H., O'Connor, M., "PicoJava : A Direct Execution Engine for Java bytecode," *IEEE Computer*, Oct., 1998.

[8] Nakashima Y., Kitamura T., Tamura H., Takiuchi M., Miura K., "Scalar Processor of the VPP500 Parallel Supercomputer," *Proceedings of 9th ACM Int. Conf. of Supercomputing*, Jul., 1995, pp. 348-356.

[9] O'Connor, J. M., Tremblay, M., "PicoJava-I : The Java Virtual Machine in Hardware," *IEEE MICRO*, March/April, 1997.

[10] Sodani, A., Sohi, G. S., "Understanding the Differences between Value Prediction and Instruction Reuse," *31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[11] Sodani, A., Sohi, G. S., "Dynamic Instruction Reuse," *24th Annual International Symposium on Computer Architecture*, pp. 194-205, 1997.

[12] SPEC JVM98 VERSION 1.03, the Standard Performance Evaluation Corporation, 1998.

[13] Takeuchi, I., et al., "A Concurrent Multiple-paradigm List Processor TAO/ELIS," *Proceedings 1987 Fall Joint Computer Conference — Exploring Technology : Today and Tomorrow*, pp. 167-174, 1987.

[14] Wang, K., Franklin, M., "Highly Accurate Data Value Prediction using Hybrid Predictors," *30th Annual International Symposium on Microarchitecture*, 1997.

[15] 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, 2000年, 93-95ページ。

[16] 金田悠紀夫『Prolog マシン』森北出版, 1992年。

[17] 中島康彦, 北村俊明, 田村秀夫, 滝内政昭「VPP500 スカラプロセサの特徴」『情報処理学会研究報告』ARC-104-17, 1994年1月, 129-136ページ。

[18] 中島康彦, 大野優人, 竹部好正「VPP500 スカラプロセサの性能」『情報処理学会論文誌』Vol. 38, No. 4, 1997年, 863-872ページ。

[19] 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年。

[20] 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, 1995年，17-24ページ。

[21] 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年。